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 D1.2
Full ABS Modeling Framework

Due date of deliverable: (T24)
Actual submission date: 1st March 2011

Start date of the project: 1st March 2009
Duration: 48 months

Organisation name of lead contractor for this deliverable: KUL

Final version

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

This document summarizes deliverable D1.2 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](http://www.hats-project.eu).

The report contains the definition of the full ABS language, an abstract executable modeling language intended for modeling and analysis of software product families. Three parts concerning the language are reported:

**Full Language Framework:** We specify the full ABS language by providing concrete syntax and semantics. The full ABS language is based on a core concurrent object-oriented language with purely functional data types and concurrent objects groups of asynchronously communicating objects. We describe a behavioural interface specification framework for specifying the behavioural properties of ABS models. We detail four language extensions which work in harmony to model software product line variability. These express high level variability in terms of a feature model, along with software components called delta modules for implementing the corresponding variability, and languages for configuring product lines and for selecting products. Finally, we present our approach to platform modeling and configuration, whereby abstract models of low-level variability due to the target deployment scenarios can be expressed and modeled.

**Examples:** The concrete syntax of the language is defined by EBNF notation and the parsers. We illustrate the language in this document by way of formal semantics and example, which are also meant to convey the pragmatics of the modeling language.

**Tools:** We describe the intended tool chain dealing with the full ABS language, and the current state of the implementation.

**List of Authors**

Dave Clarke (KUL)
Stijn de Gouw (CWI)
Reiner Hähnle (CTH)
Einar Broch Johnsen (UIO)
Ilham W. Kurnia (UKL)
Radu Muschevici (KUL)
Bjarte M. Østvold (NR)
José Proença (KUL)
Ina Schaefer (CTH)
Jan Schäfer (UKL)
Martin Steffen (UIO)
Arild B. Torjusen (NR)
Contents

1 Introduction

2 Rationale

2.1 Motivation for an Abstract Behavioral Modeling Language

2.2 Design Choices for the Full ABS

2.3 Relation to the Requirement Elicitation of Task 5.1

3 The ABS Language

3.1 Notation

3.2 Lexical Structure

3.2.1 Line Terminators and White Spaces

3.2.2 Comments

3.2.3 Identifiers

3.2.4 Keywords

3.2.5 Literals

3.2.6 Separators

3.2.7 Operators

3.3 Names and Types

3.3.1 Names

3.3.2 Types

3.4 Algebraic Data Types

3.4.1 Parametric Data Types

3.4.2 Predefined Data Types

3.4.3 N-ary Constructors

3.4.4 Abstract Data Types

3.5 Type Synonyms

3.6 Functions

3.6.1 Parametric Functions

3.7 Pure Expressions

3.7.1 Let Expressions

3.7.2 Data Type Constructor Expressions

3.7.3 Function Applications

3.7.4 Case Expressions / Pattern Matching

3.7.5 Operator Expressions

3.8 Expression With Side Effects

3.8.1 New Expression

3.8.2 Synchronous Call Expression

3.8.3 Asynchronous Call Expression

3.8.4 Get Expression

3.9 Statements
HATS Deliverable D1.2

Full ABS Modeling Framework

3.9.1 Block .................................................. 27
3.9.2 If Statement ........................................... 27
3.9.3 While Statement ....................................... 28
3.9.4 Variable Declaration Statements ..................... 28
3.9.5 Assign Statement ..................................... 28
3.9.6 Await Statement ..................................... 29
3.9.7 Suspend Statement ................................... 29
3.9.8 Skip Statement ....................................... 29
3.9.9 Assert Statement ..................................... 29
3.9.10 Return Statement ................................... 30
3.9.11 Expression Statement ................................ 30
3.10 Classes and Interfaces ................................. 30
  3.10.1 Interfaces ........................................... 30
  3.10.2 Classes ............................................. 31
3.11 Annotations ............................................. 33
  3.11.1 Type Annotations ................................... 33
3.12 Modules ................................................ 33
  3.12.1 Exporting ............................................ 34
  3.12.2 Importing .......................................... 34
  3.12.3 Exporting Imported Names ......................... 35
3.13 Model .................................................. 36
3.14 Related Work ........................................... 36

4 Behavioral Interface Specification Framework ........................................ 40
  4.1 Message Specification Framework ..................... 40
  4.2 Specification Techniques ................................ 45
    4.2.1 Assertion Language ................................ 45
    4.2.2 Attribute Grammar Specification Language ...... 46
    4.2.3 Interaction Pattern Specification Language ..... 49
  4.3 Related Work ........................................... 52

5 Variability Modeling ......................................... 54
  5.1 Feature modeling ........................................ 55
    5.1.1 Concrete Syntax .................................... 55
    5.1.2 Abstract Syntax .................................... 57
    5.1.3 Semantics ........................................... 58
  5.2 Delta Modeling ........................................... 59
    5.2.1 Syntax .............................................. 60
    5.2.2 Formal Semantics .................................. 64
  5.3 Product Line Configuration ............................ 66
    5.3.1 Syntax .............................................. 66
    5.3.2 Semantics .......................................... 68
  5.4 Product Selection ....................................... 70
    5.4.1 Syntax .............................................. 70
    5.4.2 Semantics .......................................... 71
  5.5 Product Generation ..................................... 72
  5.6 Related Work ........................................... 73
## Platform Models and Configuration

6.1 Deployment Components .......................................................... 76
6.2 Example .................................................................................... 77
   6.2.1 Core Application ............................................................... 77
   6.2.2 Deployment Components ..................................................... 78
   6.2.3 Feature Model ................................................................. 79
   6.2.4 Delta Modules ................................................................... 79
   6.2.5 Configuration .................................................................... 80
   6.2.6 Product Selection .............................................................. 80
6.3 Related Work ........................................................................... 81

## Tool Support and Integration

7.1 The HATS Framework ............................................................... 83
7.2 The ABS Compiler Front End .................................................... 84
   7.2.1 Compiler Technology ......................................................... 84
   7.2.2 Using the Compiler Front End ............................................. 85
7.3 Maude Back End ....................................................................... 86
   7.3.1 Running models using Maude ............................................. 87
7.4 Java Back End .......................................................................... 87
   7.4.1 Using the Java Backend .................................................... 88
7.5 Core ABS Code generation ....................................................... 91
7.6 ABS Integrated Development Environment ............................ 91
   7.6.1 Debug Perspective ........................................................... 92
   7.6.2 Emacs Mode ................................................................. 93
7.7 Documentation ......................................................................... 93
7.8 Running Examples ................................................................... 95

## Summary

8 ........................................................................................................ 96

## Bibliography

9 ....................................................................................................... 98

## Glossary

10 .................................................................................................... 109

## A ABS Standard Library

11 ..................................................................................................... 111

## B Syntax of Core ABS

12 ..................................................................................................... 119
   B.1 A Language for Abstract Behavioral Specification .................. 119
      B.1.1 Abstract Syntax ............................................................... 119

## C Data type and functional sub-language

13 ..................................................................................................... 122
   C.1 Syntax .......................................................... 122
   C.2 Typing ............................................................................... 123
      C.2.1 Well-formed Contexts .................................................... 124
      C.2.2 Kinding ..................................................................... 125
      C.2.3 Type system ............................................................... 125
   C.3 Semantics .......................................................................... 128
   C.4 Predefined types ............................................................... 130
Chapter 1

Introduction

The HATS Deliverable 1.2 is presented in one part, namely, D1.2 Report on the Full ABS Language, which reports on the Full ABS language. The HATS Description of Work (DoW) describes our intentions for the full ABS. Language constructs for expressing feature and platform models are developed along the lines of standard feature modeling languages. Feature models can be equipped with configuration parameters (attributes), contain a simple description of deployment possibilities, and refer to behavioural ABS components. In addition, we describe techniques to close the gap between the abstract level of feature and attribute specification languages and the concrete level of executable models, by capturing the relationship between the feature model and the underlying behavioural specification. We focus on the specification of behavioural models of features and techniques for feature integration, by combining components based on variation points. Techniques for invasive composition are presented based on delta modules, which provide a means for adapting a code based on a given feature selection. The language features presented work together to describe full software product lines. In addition, the core ABS language inherently supports concurrency and distribution, and components, which are encapsulated by interfaces and allow a flexible communication model, form units of computation. Furthermore, as a foundation for a formal method, the full ABS will have a formally defined semantics.

The full ABS is based on four specialised language extensions, sitting atop the core ABS language, depicted in Figure 1.1. The first extension, the micro textual variability language (µTVL), based on TVL [39, 53], expresses variability at the level of feature models. The second extension, the Delta Modeling Language (DML), based on delta modeling [155], expresses the code-level variability required for software product lines. Feature models and delta modules are independent entities that do not refer to each other. The third extension, the Product Line Configuration Language (CL), links a feature model and delta modules together, and forms the top level specification of an entire product line. The fourth extension, the Product Selection Language (PSL), expresses products by providing feature and attribute selection along with initialisation code for the product. These languages extensions combine to describe an entire product line, where the behaviour is written using core ABS. A specific product selection results in a collection of applicable delta modules, which are applied to a core ABS program and result in another core ABS program corresponding to the selected product. After the translation into core ABS, the language extension no longer play a rôle. Thus one writes entire product lines in the combination of all language extensions, whereas single

<table>
<thead>
<tr>
<th>Language</th>
<th>Rôle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Micro Textual Variability Language (µTVL)</td>
<td>Feature models, attributes and constraints on them</td>
</tr>
<tr>
<td>Delta Modeling Language (DML)</td>
<td>Modifications to core behavioural modules</td>
</tr>
<tr>
<td>Product Line Configuration Language (CL)</td>
<td>Links features and delta modules, configures delta modules with attributes</td>
</tr>
<tr>
<td>Product Selection Language (PSL)</td>
<td>Feature and attributes selections plus product initialisation block</td>
</tr>
<tr>
<td>Core ABS</td>
<td>Specifies core behavioural modules (independent of extensions)</td>
</tr>
</tbody>
</table>

Figure 1.1: Full ABS Language
products/applications are written in core ABS, and the full language compiles to core ABS.

For the core ABS, we propose a syntax and, as a formal foundation for the language, a type system (core only) based on interface types and an operational semantics. A comprehensive type system for the full language will be the result of Task 2.4. We have adopted a Java-like syntax for the core ABS language. The operational semantics of the core ABS language forms the basis for a prototype language interpreter, defined in Maude [54]. This interpreter supports the simulation of models in the core ABS, as defined by the operational semantics.

A tool chain transforming models expressed in the syntax of the full ABS language into the run-time syntax of the interpreter has been developed, though several experimental features have not fully been completed. See Chapter 7 for details.

The core ABS language is an executable, class-based object-oriented language. However, for simplicity, the core ABS does not support class inheritance. An important motivation for the ABS language is that it should address abstract behavioral modeling. This is realized in the language by abstracting from a certain range of implementation choices (which may otherwise be explicit or implicit in a less abstract model). The ABS language is based on the concurrency model of Creol [107, 64], which uses asynchronous method calls and underspecified local scheduling. In particular, any method can be called either synchronously or asynchronously, and this may be a run-time decision of the caller. The local scheduling may depend on the availability of replies to method calls, or on the explicit release of control which makes it straightforward to model objects which combine active and reactive behavior. Furthermore, object references are always typed by interface and not by class. The interface exports a subset of the methods defined in a class, making these methods available to other objects. All communication between objects is in the form of method calls.

Another important abstraction inherited from Creol is the use of algebraic data types for local state inside objects and a side effect-free expression language (even though statements generally may have side effects in ABS). From the perspective of modeling, this has important consequences in that objects should be understood as high level units of computation. It also means that the modeler need not be concerned with the choice of implementation data structures for, e.g., a list, a set or a map. In this respect, we go beyond the Creol language such that the core ABS language supports user-defined abstract data types and case expressions over these types.

Furthermore, we introduce a concept of component into the ABS language based on the CoBox model [159, 160]. Our components, called concurrent object groups, consist of one or more ABS objects which share the computation resource; i.e., there can be at most one activity running inside the group. Thus, the computation inside the group combines a thread-like notion of activity which supports local re-entrance for synchronous calls, with the underspecified scheduling of asynchronous calls.

In this deliverable, we present the full ABS language. This consists of a proposed Java-like syntax, a type system, and an operational semantics for the core ABS language; a language extension µTVL for expressing feature models and its semantics; a language extension DML for expressing variability at the code level in terms of delta modules, and its semantics; a language extension CL for configuring a product line by connecting a feature model with delta modules, and its semantics; a language extension PSL for expressing a software product for a given product line as a set of selected features and their attributes, and its semantics; and a prototype tool chain for checking the validity of feature models, for determining the validity of feature selections, for determining and applying the relevant delta modules to generate core ABS, and compilation from source syntax to the simulation environment. To describe the dynamic behavior of components we also introduce a behavioral specification framework, addressing specification of behavioral properties of components (Objective 8). This forms the basis for refinement and abstraction (Task 2.6), correctness (Task 4.3) and development of tool support for run-time checking and static verification by means of the theorem provers such as KeY (Task 1.3).

This deliverable is organized as follows. Chapter 2 lists and explains the major design decisions for ABS and connects them to the relevant high-level methodological requirements that had been elicited in Deliverable 5.1 [66]. Chapter 3 contains a description and user guide of the ABS language. Chapter 4 describes the behavioural interface specification framework of the ABS language. Chapter 5 describes the four language extensions for expressing variability. Chapter 6 describes how full ABS (plus an additional future language extension) can be used to provide models of the underlying software/hardware platform, using variability modeling techniques to describe the range of configurations. Chapter 7 shows the current and planned stages of development of the tool chain for the ABS language. Finally, Chapter 8 summarizes the work presented in this report. Syntax, semantics and type system of the core
ABS language have been presented in the HATS Deliverable 1.1a [67], but we include them here in the appendices for reference. Appendix A contains the standard ABS library, included by default when compiling core ABS programs. Appendix B introduces the core ABS language and proposes an abstract syntax. The core ABS language consists of two parts, a functional part and an imperative part. The functional part is presented in Appendix C and the imperative part in Appendix D. The language descriptions were already presented in Deliverable 1.1a [67], and are included here with minor changes for completeness. Appendix E presents an example of a peer-to-peer network node as a model in the core ABS. Fragments of this example are used throughout the entire document. The remaining three appendices include papers published in the context of this deliverable. Appendix F introduces JCoBox, used as the basis for the Java translation described in Section 7.4; Appendix G describes techniques on based attribute grammars for checking assertion in Java, used in Chapter 4; and Appendix H introduces a technique for specifying interaction patterns of components used in the same chapter. Moreover, related work is discussed in each relevant chapter.

We emphasize that the development of the ABS modeling framework is not finalized with this deliverable. At this time, we provide a “full” set of features that allows ABS to be used, as intended, in modeling and specification of highly parametric and adaptable software systems, specifically, software product lines. In the remaining time of the project we expect that the ABS language features will be further refined and extended, for example, based on feedback from several projected case studies. We also envisage that a suitable notion of architectural components will be defined and implemented (Tasks 2.4, 3.1).

Please note that while this report contains small examples that are intended to illustrate the various language features, it is beyond the scope of the current document to provide an extended case study for the full language features. A larger case study involving the ABS variability modeling features will be the part of Deliverable 5.3 “Evaluation of Modeling” which is due in project month 36.

Conventions in this document

The document is intended as a reference (for the further development) of the full ABS language. It serves therefore at least two purposes. One is as a guideline and specification for the “implementers”, specifying for instance the abstract syntax, the operational rules, and the type checker for core ABS and the syntax and semantics of the remaining languages that make up the full ABS. Another purpose is to illustrate the manner in which concrete programs are fed into the compiler front end. The abstract syntax is captured by context free grammars in EBNF notation. By convention, when showing listings in sans serif font, we intend to illustrate the concrete programming syntax.

We intend the (formal) definitions mostly based on the abstract syntax to constitute a kind of core calculus. It is, of course, not really formally defined what constitutes a core calculus (as opposed to a programming language) but one guiding principle is: each “semantic” principle or feature should be represented exactly once in the grammar. Of course, from the user perspective, more luxury might be wished, but that is dealt with by “syntactic sugar”. More details are given of the four languages for expressing variability, as these are smaller in scale than the core ABS language.
Chapter 2

Rationale

We first list, explain, and motivate the main design choices made for the ABS language, and then we relate these design choices to the high-level requirements described in Deliverable 5.1 [66].

2.1 Motivation for an Abstract Behavioral Modeling Language

The main rationale of the abstract behavioral modeling language is to fill the gap between highly abstract, generic, and often only structural modeling languages such as UML [139], B [4], and ASM [37], and highly specific ones such as JML [43] or Spec# [28]. In addition, ABS aims to have constructs for reasoning about highly adaptable systems such as software product lines. For example, UML does not offer a coherent view on communication and concurrency, as different standard notations assume either synchronous or asynchronous communication. The integration of these two basic forms of communication within a UML framework is highly complex [62]. Specification formalisms close to programming languages typically inherit the idiosyncrasies and problematic design decisions of their host languages (which were not designed for verification). For example, Java’s concurrency model is widely considered to be impractical for the design of modular, concurrent systems, but JML has to follow it. It seems clear that proof systems for multi-threaded Java programs are inherently complicated and do not scale to the verification of real programs [2]. Source code-level specification formalisms have the additional problem that too many design decisions are already taken and, therefore, are not useful at the design stage. They also have the problem that their host languages are determinant whereas non-determinism is an essential abstraction mechanism in design of adaptable systems. Furthermore, these modeling languages offer little or no support for specifying software product lines.

The ABS modeling language aims to fill this gap between structural modeling languages and implementation-close formalisms. In order to make ABS easy to use for programmers, we position ABS within the object-oriented paradigm and model control flow using familiar imperative structures. However, the ABS language supports abstractions which are not supported in implementation languages, in particular by means of functional data types, flexible concurrency and communication constructs, and cooperative scheduling. These abstractions make ABS models configurable. In addition the full ABS language contains facilities for expressing variability, such as feature models and “code” level variability via delta modules, along with means for linking these two abstractions together.

This deliverable presents the full ABS language. Below, in Section 2.2, we elaborate on design choices guiding the design of the language and then, in Section 2.3 we discuss how these choices relate to the high-level requirements of Deliverable 5.1 [66].

2.2 Design Choices for the Full ABS

This section discusses some of the main design choices underlying the design of the Full ABS language. Initial points reflect the design of the Core ABS language, and later points cover design choices for the variability part of the language.

Object-Oriented: The core ABS language is class-based in the sense that a program is given as a set of classes.
However, it does not feature code reuse via inheritance. Structuring mechanisms such as class inheritance have intentionally been excluded from the core ABS language. Following the Description of Work (DoW), code reuse, evolution, and variability is not based on standard class inheritance. Instead, the notion of delta module from delta modeling is exploited to perform the invasive composition required for modeling software product lines.

**Concurrency and composition:** The language features a concurrency model based on concurrent objects, asynchronous method calls, and futures. Asynchronous method calls may be understood as triggers of concurrent activities. Concurrent objects may be composed into concurrent object groups (COGs), based on the idea of coboxes [160] (see Appendix F). COGs generalize the concurrency model of Creol from single concurrent objects to concurrent groups of objects. COGs can be regarded as object-oriented runtime components, which have their own heap of objects and which solely communicate via asynchronous method calls. The behavior of a COG is represented by cooperative multi-tasking, as introduced in Creol. Cooperative multi-tasking guarantees data-race freedom inside a COG and enables the safe combination of active and reactive behavior. In addition, sequential object-oriented programming can be modeled by a COG that only has a single task.

**Strongly typed:** The language is strongly typed. The object-oriented part uses a nominal type system, where the roles of classes and of interfaces are separate. Interfaces are types. Classes are not types. A class may implement a number of interfaces, and an interface may be implemented by a number of classes. Even if we do not support class inheritance (code reuse between classes), we have (nominal) subtyping on interfaces. We do support interface inheritance, which defines the subtype relation in the core ABS language.

**Data types:** Beside the object-oriented part, the core language supports user defined data-types with (non-higher-order) functions and pattern matching. This functional sublanguage of ABS is largely orthogonal to the object-oriented part and is intended to model data. As such data is immutable, it can safely be exchanged between COGs. It can also be used as part of the assertion language. In addition, using functional data types to realize most internal data structures of COGs will simplify the specification and verification of COGs. The semantic value of data expressions can be underspecified. This is widely considered to be a simple and adequate technique for dealing with partiality in specifications.

The combination of a functional representation of data and concurrent imperative control structures based on concurrent objects suggests a development methodology for ABS models, in which the initial focus of design and verification is on the interaction between high-level concurrent objects. The gradual (and partly automated) replacement of functionally represented data with imperative structures decomposes the high-level concurrent object into a concurrent object group in order to narrow the gap between the model and a target object-oriented programming language. The realization of concurrent object groups in Java is reported in [160].

**Non-deterministic:** The language contains non-deterministic constructs; in particular, the outcome of executing concurrency primitives is not deterministic. While underspecification is used to realize abstraction on data, non-deterministic execution semantics is the prerequisite for abstracting behavior. As ABS is a modeling language we do not want to make any a priori assumptions about, for example, scheduling.

**Executable:** Underspecification and non-determinism do not preclude executability: an unknown value is still a value and the outcome of a non-deterministic statement is a set of possible successor states from which one can be picked in simulation and visualization. An executable semantics allows the application of simulation-based analysis techniques at an early stage of design and is the prerequisite for our program logic which is based on symbolic execution. Symbolic execution matches underspecification and non-determinism very well: unknown values are symbolic values and symbolic execution is non-deterministic by nature even for deterministic target languages.

**Feasible proof theory:** The concurrency and composition primitives are carefully designed in such a way that a proof theory with practically feasible proof search can be developed. While this will be the objective of Task 2.5 we made sure that it is possible. (In [8, 9] a symbolic execution engine with histories for Creol, the basis for the ABS concurrency model, is presented. For concurrent object groups, one can adapt proof techniques originally
developed for universe types \[165\]. A first version for a program logic for ABS is part of HATS Deliverable 2.5 “Verification of Behavioral Properties” \[69\].

**Layered design:** To achieve maximal modularity and extensibility of the ABS language we decided a *layered architecture* with clearly defined tiers as depicted in Figure 2.1. The ABS language is separated into two clear components, namely the core ABS language, and the variability modeling languages.

**Separation of concerns:** Insomuch as is possible, languages for expressing the core behaviour, the abstract feature models, behavioural variability, and configuration of software product lines have been kept separate. This is realised for example by the fact that the core ABS can be used independently to express systems. Feature models are specified without any reference to behaviour, and delta modules are expressed without reference to features. To further enhance this separation, delta modules for expressing variability of code are separate from feature modeling artifacts. All ingredients are tied together using product line configurations, from which products can be extracted using product specifications.

Platform models, for example, are realised by separating configuration parameters, specified using attributes in feature models, from the deployment components (Deliverable 2.1 “Configuration deployment”) which implement the resource specific behaviour of the underlying low-level models.

**Variability Modeling**

| Feature Modeling  
| **(μTVL)** | Delta Modeling  
| **(DML)** |  
| Product Line Configuration  
| **(CL)** |  
| Product Selection  
| **(PSL)** |

**Core ABS**

- Behavioral Interface Language
- Assertion Language
- Compositionality
- Concurrency model
  - CoreCreol
- Object model
- Side-effect free expressions
- ADT

**2.3 Relation to the Requirement Elicitation of Task 5.1**

This section relates the design decisions for the ABS language to the high-level requirements of the Requirements Elicitation of Task 5.1, described in Deliverable 5.1 \[66\].

Although this document provides a description of the Full ABS language, it does not represent the totality of the language, methodology and tool developments of the HATS project. Several requirements are consequently not yet addressed and will be addressed by other work tasks. The requirement concerning resource guarantees (MR16) is addressed in Deliverables 4.2 and 4.4. Several requirements concerning the HATS methodology (MR1, MR2, MR14) are addressed in deliverable 1.1B. Concerning the Full ABS language, the following requirements are relevant.
**MR3 (Testing reusable artifacts)** Delta modules, components (COGs), and concurrent objects permit independent design and high deal of cohesion and hence are more readily testable independently. In particular, delta modules are written to be independent of any feature model or software product line, so can be tested outside of the context in which they are intended to be used. Testing is covered in more detail in Task 2.3.

**MR4 (Providing language support for PLE)** The languages described in Chapter 5 are designed specifically to support the modeling of software product lines. These languages allow the description of feature models independently of the underlying implementation, to maintain the appropriate high level of abstraction required for feature modeling and the specification of possible configurations of the product line. The delta modeling approach is based on a core architecture which implements typically one of the basic products in the product line. Adding more features results in applying different deltas to inject more code into the core, either by adding classes or by invasively modifying classes, or by merely specifying details of how the product is configured based on the chosen feature and attribute selections.

Delta modules provide the ability to customise not only objects, but their behaviour through configuration parameters specified in feature models, and their interfaces, as interfaces may be adapted in the same way as classes are. Any product-specific artifacts can be implemented either using classes or deltas, configured in such a way that they are only included in the product if the appropriate criteria specified in the configuration file are satisfied by the feature selection.

**MR5 (Specifying reusable contracts)** Delta modeling and feature modeling provide a convenient way to model variability. However, deltas do not exhibit any behavior on their own: deltas are combined using syntactical operations on for example (parts of) classes, and it is these classes which exhibit behavior (semantics). The behavior of the classes is specified using the techniques described in Chapter 4. Let \( C \) be a class with behavioral specification \( S \), and let \( B(C) = S \) be the behavior of the class \( C \). The semantical characterization of a delta \( \delta \) on classes can be seen as a delta \( \delta' \) on specifications iff \( \delta' \) satisfies the reuse contract: \( B(\delta(C)) = \delta'(S) \) which justifies reuse of the contract \( S \) and class \( C \). The semantical characterization of deltas and reuse contracts with deltas will be explored further in the context of Deliverable 2.4 (Types for variability) and Deliverable 2.6 (Refinement and abstraction).

**MR6 (Defining reusable artifacts and variation points)** Components (COGs), classes, and delta modules all specify reusable artifacts. Variation points are specified via a combination of delta modules, which specify the actual changes and where they occur, feature models, which specify the space of variability, and the product line configuration language, which links these ingredients. The specific details of the interdependencies between these artifacts are explicitly specified but need to be checked using type checking and conflict detection, which will be addressed in Deliverable 2.4 (Types for variability).

**MR11 (Learnability).** Learnability is supported by the ABS language by building on language constructs well-known from mainstream languages. The ABS language is based on the standard object-oriented concepts of classes and interfaces with a nominal type system. A standard statement language with conditional statements, while statements and standard method calls, will allow average programmers to quickly use the language. Nevertheless, the ABS language also introduces new concepts, not known by average programmers who are only familiar with mainstream object-oriented languages. These are the functional data-type part and the new concurrency model. The functional data-type part of the language is based on well-known concepts from functional languages such as ML. Functional language concepts have lately also been integrated into object-oriented languages like Scala, for example, and are thus already known to a larger programmer audience. Even though it will require a bit of learning effort for some programmers, it should be straightforward for programmers who already know functional concepts.

The concurrency model of ABS reflects how concurrency is perceived in the everyday world, and also how it is usually perceived by modellers. Unlike most mainstream programming languages, the ABS language is not based on the concept of threads working on a shared state (a notion which stems from a sequential understanding of programming). The thread model is known to be very hard to understand, and requires expert programmers. In

\[\text{http://www.scala-lang.org}\]
particular, the thread-model is very error-prone due to data-races and deadlocks. The concurrency model of the ABS language is based on the concept of isolated object-oriented components (COGs), which share no state and communicate via (asynchronous) method calls. It will require some learning effort for programmers who are used to thread-concurrency to write software in this new model. However, once the model is understood, we are convinced that it is much easier to write correct concurrent programs in the ABS concurrency model than in the standard thread model.

Finally, the part of the ABS aimed at specifying software product lines consists of four small languages. The most complicated, $\mu$TVL, has the advantage of being the most familiar to developers already using feature models (albeit in textual not visual form). Delta module specifications differ little from ordinary core ABS modules, and thus the language will be straightforward to use. The remaining two languages are very small and pose no learnability concerns.

MR12 (Usability), MR13 (Reducing manual effort), and MR18 (Integrated Environment Support). Realizing usability is not a central objective of the present deliverable, but the ABS language is already supported by an early prototype of a source code editor which is integrated into the widely used Eclipse IDE, presented in Section 7.6. Tool support will constantly be improved during the project and integrated into the Eclipse IDE (addressing MR18). All tools for ABS work on a common AST representation, which will allow for a seamless interaction of the different tools. One of the major goals of HATS is the reduction of manual effort by automation with tools. Requirement MR13 is thus directly addressed by the tool suite of ABS.

MR17 (Protocol analysis) The Behavioral Interface Specification Framework presented in Chapter 4 provides the means for specifying the behavior of concurrent objects at a high level of abstraction using interaction patterns. Concurrent objects, in particular when used in conjunction with concurrent object groups, provide an adequate framework for modeling protocols. The specifications form the basis for other tasks (Tasks 1.3 “Analysis” and 4.3 “Correctness”) to development appropriate analysis and verification techniques.

MR20 (ABS Extensibility) The ABS language permits the possibility of specifying extension or variation points, as described in points MR4 and MR6 above. The whole approach is based on having a core model, written in core ABS, and separate customisations, specified using the remaining languages. Customisation can be achieved within a given product line by varying the configuration parameters (attributes) when providing a feature selection in PSL. Further customisation is possible by adding additional features to the feature model and writing the corresponding delta modules. This requires no modification of the core architecture and thus can be performed independently of the developers of the application core.

MR21 (Service orientation), MR22 (Middleware Abstraction), MR23 (Architectural Style) While not specifically designed for service-orientation, it has been shown that concurrent objects provide an adequate abstraction for modeling service oriented computation [52]. This is particularly the case when one considers the functional data type layer to be analogous to the immutable XML messages sent, instead of the usual object-oriented computational model whereby mutable object references are passed between objects. That message passing between concurrent objects can be both asynchronous and synchronous captures two primary modes of communication in the service oriented setting. Service composition can be seen as analogous to the composition of concurrent objects, in particular coordination via the use of futures.

The ABS language operates at a higher level of abstraction than code, and can abstract away from the specifics of the concrete middleware layer. Deployment components (Deliverable 2.1 “Configuration Deployment”) can be used to express deployment-specific characteristics of this middleware when performance criteria need to be modeled. Similarly, the ABS language is flexible enough to model different architectural styles and to specify their behavioural properties, as discussed in Chapter 4 and performance-related properties (Deliverable 2.1). Provided that a sufficiently versatile and coherent core architecture has been designed, delta modules offer the flexibility to change the underlying architectural style.
Chapter 3

The ABS Language

This chapter describes the core ABS language as it is implemented in the ABS tools. The ABS language is a class-based object-oriented language that features algebraic data types and side effect-free functions. Syntactically, the ABS language tries to be as close as possible to the Java language so that programmers that are used to Java can easily use the ABS language without much learning effort.

3.1 Notation

In this chapter we often present the concrete syntax of the ABS language. To do so we use BNF with the following denotations.

- $[x]$ denotes zero or one occurrences of $x$. The same notation is used to represent an optional element in the formal system. In effect $[x]$ corresponds to either nothing or an element of $x$.
- $x^*$ denotes zero or more occurrences of $x$. Note that in formal semantics the notation $\bar{x}$ is used to represent an explicit sequence of elements $x_1, \ldots, x_n$; similarly $x : T$ represents $x_1 : T_1, \ldots, x_n : T_n$, following Pierce.
- $x^+$ denotes one or one occurrences of $x$.
- $x | y$ means one of either $x$ or $y$.
- $[: :]$ denotes the POSIX character class $x$.
- Text in monospace denote terminal symbols.
- Text in italics denote non-terminals in the grammar.

3.2 Lexical Structure

This section describes the lexical structure of the ABS language. ABS programs are written in Unicode.

3.2.1 Line Terminators and White Spaces

Line terminators and white spaces are defined as in Java.

Syntax:

---

1 Backus-Naur Form
2 http://www.unicode.org
3.2.2 Comments

Comments are code fragments that are completely ignored and have no semantics in the ABS language. ABS supports two styles of comments: end-of-line comments and traditional comments.

3.2.2.1 End-Of-Line Comments

An end-of-line comment is a code fragment that starts with two slashes, e.g., `// text`. All text that follows `//` until the end of the line is treated as a comment.

Example:

```
// this is a comment
module A;  // this is also a comment
```

3.2.2.2 Traditional Comments

A traditional comment is a code fragment that is enclosed in /* */, e.g., `/* this is a comment */`. Nested traditional comments are not possible.

Example:

```
/* this is a multiline
comment */
```

3.2.3 Identifiers

ABS distinguishes identifier and type identifier. They differ in the first character, which must be a lower-case character for identifiers and an upper-case character for type identifiers.

Syntax:

- `Identifier ::= [:lower:] ([:alpha:] | [:digit:] | _)*`
- `TypeId ::= [:upper:] ([:alpha:] | [:digit:] | _)*`

3.2.4 Keywords

The following words are keywords in the ABS language and are not regarded as identifiers.

```
adds after assert await builtin case
cog core class data def delta
else export features from get hasField
hasInterface hasMethod if implements import in
interface let modifies module new null
product productline removes return skip suspend
this type when while
```


3.2.5 Literals

A literal is a textual representation of a value. ABS supports three kinds of literals, integer literals, string literals, and the null literal.

Syntax:

\[
\begin{align*}
\text{Literal} & ::= \text{IntLiteral} \mid \text{StringLiteral} \mid \text{NullLiteral} \\
\text{IntLiteral} & ::= 0 \mid [1-9][0-9]^* \\
\text{StringLiteral} & ::= " \text{StringCharacter}^* " \\
\text{NullLiteral} & ::= \text{null}
\end{align*}
\]

Where a StringCharacter is defined as in the Java language [87, p. 28]

3.2.6 Separators

The following characters are separators:

( ) { } [ ] , ; :

3.2.7 Operators

The following tokens are operators:

|| && == != < > <= >= + - * / % ~ &

3.3 Names and Types

3.3.1 Names

A name in ABS can either be a simple identifier as described above, or can be qualified with a type name, which represents a module.

Syntax:

\[
\begin{align*}
\text{TypeName} & ::= \text{TypeId} (, \text{TypeId})^* \\
\text{Name} & ::= \text{Identifier} \mid \text{TypeName} . \text{Name}
\end{align*}
\]

Examples for syntactically valid names are: head, x, ABS.StdLib.tail. Examples for type names are: Unit, X, ABS.StdLib.Map.

3.3.2 Types

Types in ABS are either plain type names or can have type arguments.

Syntax:

\[
\begin{align*}
\text{Type} & ::= \text{TypeName} [\text{TypeArgs}] \\
\text{TypeArgs} & ::= < \text{TypeList} > \\
\text{TypeList} & ::= \text{Type} (, \text{Type})^*
\end{align*}
\]

Where TypeName can refer to a data type, an interface, a type synonym, and a type parameter. Note that classes cannot be used as types in ABS. In addition, only parametric data types can have type arguments. Examples for syntactically valid types are: Bool, ABS.StdLib.Int.List<Bool>, ABS.StdLib.Map<Int,Bool>.

17
3.4 Algebraic Data Types

Algebraic Data Types make it possible to describe data in an immutable way. In contrast to objects, data types do not have an identify and cannot be mutated. This makes reasoning about data types much simpler than about objects. Data types are built by using Data Type Constructors (or constructors for short), which describe the possible values of a data type.

Syntax:

```plaintext
DataTypeDecl ::= data TypeId [TypeParams] [ = DataConstrList ] ;
TypeParams  ::= < TypeId (, TypeId)* >
DataConstrList ::= DataConstr ( | DataConstr)*
DataConstr  ::= TypeId [ [TypeList] ]
```

Example:

```plaintext
data IntList = NoInt | Cons(Int, IntList);
data Bool = True | False;
```

3.4.1 Parametric Data Types

Parametric Data Types are useful to define general-purpose data types, such as lists, sets or maps. Parametric data types are declared like normal data types but have an additional type parameter section inside broken brackets (< >) after the data type name.

Example:

```plaintext
data List<A> = Nil | Cons(A, List<A>);
```

3.4.2 Predefined Data Types

The following data types are predefined:

- **data Bool = True | False;**. The boolean type with constructors True and False and the usual Boolean infix and prefix operators.
- **data Unit = Unit;**. The unit type with only one constructor Unit (for methods without return values).
- **data Int;**. An arbitrary integer (Z) for which values are constructed by using integer literals and arithmetic expressions.
- **data String;**. A string for which values are constructed by using string literals and operators.
- **data Fut<T>;**. Representing a future. A future cannot be explicitly constructed, but it is the result of an asynchronous method call. The value of a future an only be obtained by using the get expression (Sec. 3.8.4).
- **data List<A> = Nil | Cons(A, List<A>);**, with constructors Nil and Cons(A, List<A>). This predefined data type is used for implementing arbitrary n-ary constructors (see below).

A complete list of predefined data types is contained in Appendix A which lists the ABS Standard Library.
### 3.4.3 N-ary Constructors

For data types of arbitrary size, like lists, maps and sets, it is undesirable having to write them down in the form of nested constructor expressions. For this purpose, ABS provides a special syntax for *n-ary constructors*, which are transformed into constructor expressions via a user-supplied function.

**Example:**

```abs
data Set<A> = EmptySet | Insert(A, Set<A>);
def Set<A> set<A>(List<A> l) =
  case l {
    Nil => EmptySet;
    Cons(hd, tl) => Insert(hd, set(tl));
  } ;
{
  Set<Int> s = set[1, 2, 3];
}
```

An expression `type[parameters*]` is transformed into a literal by handing it to a function named `type` which takes one parameter of type `List` and returns an expression of type `Type`. (It is desirable, although not currently enforced, that `type` and `Type` are the same word, just with different capitalization.)

### 3.4.4 Abstract Data Types

Using the module system (cf. Sec. 3.12) it is possible to define *abstract data types*. For an abstract data type, only the functions that operate on them are known to the client, but not its constructors. This can be easily realized in ABS by putting such a data type in its own module and by only exporting the data type and its functions, without exporting the constructors.

### 3.5 Type Synonyms

*Type Synonyms* define synonyms for otherwise defined types. Type synonyms start with an uppercase letter.

**Syntax:**

```
TypeSynDecl ::=
  TypeId = TypeName ;
```

**Example:**

```
type Filename = String
type Filenames = Set<Filename>
type Servername = String
type Packet = String
type File = List<Packet>
type Catalog = List<Pair<Servername,Filenames>>
```
3.6 Functions

*Functions* in ABS define names for parametrized data expressions. A Function in ABS is always side effect-free, which means that it cannot manipulate the Heap.

**Syntax:**

\[
\begin{align*}
\text{FunctionDecl} & ::= \text{def Type Identifier} \left[ < \text{TypeIdList} > \right] ( \text{ParamList} ) = \text{FunBody} ; \\
\text{FunBody} & ::= \text{builtin} | \text{PureExp} \\
\text{TypeIdList} & ::= \text{TypeId} (, \text{TypeId})*
\end{align*}
\]

**Example:**

```plaintext
def Int length(IntList list) =
    case list {
    Nil => 0;
    Cons(n, ls) => 1 + length(ls);
    }
```

3.6.1 Parametric Functions

*Parametric Functions* serve to work with parametric data types in a general way. For example, given a list of any type, a parametric function `head` can return the first element, regardless of its type. Parametric functions are defined like normal functions but have an additional type parameter section inside broken brackets (`<>`) after the function name.

**Example:**

```plaintext
def A head<A>(List<A> list) =
    case list {
    Cons(x, xs) => x;
    }
```

3.7 Pure Expressions

*Pure Expressions* are side effect-free expressions. This means that these expressions cannot modify the Heap.

**Syntax:**
3.7.1 Let Expressions

Let Expressions bind variable names to pure expressions.

Syntax:

\[
\text{LetExp} ::= \text{let} \ ( \text{Param} ) = \text{PureExp} \ \text{in} \ \text{PureExp}
\]

Example:

\[
\text{let} \ (\text{Bool} \ x) = \text{True} \ \text{in} \ \neg x
\]

3.7.2 Data Type Constructor Expressions

Data Type Constructor Expressions are expressions that create data type values by using data type constructors. Note that for data type constructors that have no parameters the parentheses are optional.

Syntax:

\[
\text{DataConstrExp} ::= \text{TypeName} \\
| \text{TypeName} \ ( \ [\text{PureExpList}] )
\]

Example:

\[
\text{True} \\
\text{Cons(True, Nil)} \\
\text{ABS.StdLib.Nil}
\]
3.7.3 Function Applications

*Function Applications* apply functions to arguments.

**Syntax:**

\[
FnAppExp ::= Name \left[ PureExpList \right]
\]

**Example:**

\[
\begin{align*}
tail(\text{Cons(True, Nil)}) \\
\text{ABS.StdLib.head(list)}
\end{align*}
\]

3.7.4 Case Expressions / Pattern Matching

ABS supports pattern matching by the *Case Expression*. It takes an expression as first argument, which a series of patterns is matched against. When a pattern matches, the corresponding expression on the right hand side is evaluated.

**Syntax:**

\[
\begin{align*}
\text{CaseExp} & := \text{case PureExp } \{ \text{CaseBranch}^* \} \\
\text{CaseBranch} & := \text{Pattern } \Rightarrow \text{PureExp} \\
\text{Pattern} & ::= \text{Identifier} \\
& \quad | \text{Literal} \\
& \quad | \text{ConstrPattern} \\
& \quad | \_ \\
\text{ConstrPattern} & ::= \text{TypeName} \left[ \left( \text{PatternList} \right) \right] \\
\text{PatternList} & ::= \text{Pattern} \left( , \text{Pattern} \right)^*
\end{align*}
\]

3.7.4.1 Patterns

There are five different kinds of patterns available in ABS:

- Pattern Variables (e.g., \(x\), where \(x\) is not bound yet)
- Bound Variables (e.g., \(x\), where \(x\) is bound)
- Literal Patterns (e.g., \(5\))
- Data Constructor Patterns (e.g., \(\text{Cons(Nil, x)}\))
- Underscore Pattern (\(\_\))

**Pattern Variables.** Pattern variables are simply unbound variables. Like the underscore pattern, these variables match every value, but, in addition, bind the variable to the matched value. The bound variable can then be used in the right-hand-side expression of the corresponding branch. Typically, pattern variables are used inside of data constructor patterns to extract values from data constructors. For example:

```haskell
def A fromJust<A>(Maybe<A> a) = 
  case a { 
    Just(x) => x; 
  };
```
3.7.4.2 Type Checking

A case expression is type-correct if and only if all its expressions and all its branches are type-correct and the right-hand side of all branches have a common super type. This common super type is also the type of the overall case expression.

A branch (a pattern and its expression) is type-correct if its pattern and its right-hand side expression are type-correct. A pattern is type-correct if it can match the corresponding case expression.
### 3.7.5 Operator Expressions

ABS has a number of predefined operators which can be used to form *Operator Expressions*.

**Syntax:**

\[
\begin{align*}
\text{OperatorExp} & ::= \text{UnaryExp} \mid \text{BinaryExp} \\
\text{UnaryExp} & ::= \text{UnaryOp PureExp} \\
\text{UnaryOp} & ::= - | \neg \\
\text{BinaryExp} & ::= \text{PureExp BinaryOp PureExp} \\
\text{BinaryOp} & ::= == \mid != \mid < \mid <= \mid > \mid >= \mid + \mid - \mid * \mid / \mid %
\end{align*}
\]

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
<th>Associativity</th>
<th>Argument types</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>e1</td>
<td></td>
<td>e2</td>
<td>logical or</td>
<td>left</td>
</tr>
<tr>
<td>e1 &amp;&amp; e2</td>
<td>logical and</td>
<td>left</td>
<td>Bool, Bool</td>
<td>Bool</td>
</tr>
<tr>
<td>e1 == e2</td>
<td>equality</td>
<td>left</td>
<td>compatible</td>
<td>Bool</td>
</tr>
<tr>
<td>e1 != e2</td>
<td>inequality</td>
<td>left</td>
<td>compatible</td>
<td>Bool</td>
</tr>
<tr>
<td>e1 &lt; e2</td>
<td>less than</td>
<td>left</td>
<td>Int, Int</td>
<td>Bool</td>
</tr>
<tr>
<td>e1 &lt;= e2</td>
<td>less than or equal to</td>
<td>left</td>
<td>Int, Int</td>
<td>Bool</td>
</tr>
<tr>
<td>e1 &gt; e2</td>
<td>greater than</td>
<td>left</td>
<td>Int, Int</td>
<td>Bool</td>
</tr>
<tr>
<td>e1 &gt;= e2</td>
<td>greater than or equal to</td>
<td>left</td>
<td>Int, Int</td>
<td>Bool</td>
</tr>
<tr>
<td>e1 + e2</td>
<td>concatenation</td>
<td>left</td>
<td>String, String</td>
<td>String</td>
</tr>
<tr>
<td>e1 + e2</td>
<td>addition</td>
<td>left</td>
<td>Int, Int</td>
<td>Int</td>
</tr>
<tr>
<td>e1 - e2</td>
<td>subtraction</td>
<td>left</td>
<td>Int, Int</td>
<td>Int</td>
</tr>
<tr>
<td>e1 * e2</td>
<td>multiplication</td>
<td>left</td>
<td>Int, Int</td>
<td>Int</td>
</tr>
<tr>
<td>e1 / e2</td>
<td>division</td>
<td>left</td>
<td>Int, Int</td>
<td>Int</td>
</tr>
<tr>
<td>e1 % e2</td>
<td>modulo</td>
<td>left</td>
<td>Int, Int</td>
<td>Int</td>
</tr>
<tr>
<td>~ e</td>
<td>logical negation</td>
<td>right</td>
<td>Bool</td>
<td>Bool</td>
</tr>
<tr>
<td>- e</td>
<td>integer negation</td>
<td>right</td>
<td>Int</td>
<td>Int</td>
</tr>
</tbody>
</table>

Table 3.1: Operator expressions, grouped according to precedence from low to high.

Table [3.1](#) describes the meaning as well as the associativity and the precedence of the different operators. They are grouped according to precedence, as indicated by horizontal rules, from low precedence to high precedence.

### 3.8 Expression With Side Effects

Beside pure expressions, ABS has expressions with side effects. However, these expressions are defined in such a way that they can only have a single side effect. This means that subexpressions of expressions can only be pure expressions again. This restriction simplifies the reasoning about ABS expressions.
3.8.1 New Expression

A New Expression creates a new object from a class name and a list of arguments. In ABS objects can be created in two different ways. Either they are created in the current COG, using the standard new expression, or they are created in a new COG by using the new cog expression.

Syntax:

\[
\begin{align*}
Exp &::= \text{PureExp} \mid \text{EffExp} \\
\text{EffExp} &::= \text{NewExp} \mid \text{SyncCall} \mid \text{AsyncCall} \mid \text{GetExp}
\end{align*}
\]

Example:

\[
\begin{align*}
\text{new } \text{Foo}(5) \\
\text{new cog } \text{Bar}()
\end{align*}
\]

3.8.1.1 Standard Object Creation

When using the standard new expression, the new object is created in the current COG, i.e., the COG of the current receiver object. Figure 3.1 illustrates this by showing two different runtime states, one before the creation of an object b and one after its creation.

3.8.1.2 COG Object Creation

The concurrency model of ABS is based on the notion of COGs (cf. Section D.2). An ABS system at runtime is a set of concurrently running COGs. A COGs can be seen as an isolated subsystem, which has its own state (an object-heap) and its own internal behavior. COGs are created implicitly when creating a new object by using the new cog expression. Figure 3.2 illustrates this by showing two different runtime states, one before the creation of an object b using the new cog expression and one after its creation. In the second runtime state, two COGs exists.
3.8.2 Synchronous Call Expression

A Synchronous Call consists of a target expression, a method name, and a list of argument expressions.

Syntax:

\[
SyncCall ::= PureExp \ . \ Identifier \ ( \ PureExpList )
\]

Example:

\[
Bool b = x.m(5);
\]

3.8.3 Asynchronous Call Expression

An Asynchronous Call consists of a target expression, a method name, and a list of argument expressions. Instead of directly invoking the method, an asynchronous method call creates a new task in the target COG, which is executed asynchronously. This means that the calling task proceeds independently after the call, without waiting for the result (cf. Section D.2). The result of an asynchronous method call is a future (Fut<V>), which can be used by the calling task to later obtain the result of the method call. That future is resolved by the task that has been created in the target COG to execute the method.

Syntax:

\[
AsyncCall ::= PureExp \ ! \ Identifier \ ( \ PureExpList )
\]

Example:

\[
Fut<Bool> f = x!m(5);
\]

3.8.4 Get Expression

A Get Expression is used to obtain the value from a future. The current task is blocked until the value of the future is available, i.e., until the future has been resolved. No other task in the COG can be activated in the meantime (cf. Section D.2).

Syntax:

\[
GetExp ::= PureExp \ . \ get
\]
Example:

```java
Bool b = f.get;
```

### 3.9 Statements

In contrast to expressions, *Statements* in ABS are not evaluated to a value. If one wants to assign a value to statements it would be the `Unit` value.

**Syntax:**

\[
\text{Statement} ::= \text{CompoundStmt} \\
    | \text{VarDeclStmt} \\
    | \text{AssignStmt} \\
    | \text{AwaitStmt} \\
    | \text{SuspendStmt} \\
    | \text{SkipStmt} \\
    | \text{AssertStmt} \\
    | \text{ReturnStmt} \\
    | \text{ExpStmt}
\]

\[
\text{CompoundStmt} ::= \text{Block} \\
    | \text{IfStmt} \\
    | \text{WhileStmt}
\]

#### 3.9.1 Block

A block consists of a sequence of statements and defines a name scope for variables.

**Syntax:**

\[
\text{Block} ::= \{ \text{Statement}^* \}
\]

#### 3.9.2 If Statement

**Syntax:**

\[
\text{IfStmt} ::= \text{if (PureExp) Stmt [else Stmt]}
\]

**Example:**

```java
if (5 < x) {
    y = 6;
} else {
    y = 7;
}
```
if (True)
  x = 5;

3.9.3 While Statement

Syntax:
while (PureExp) Stmt

Example:

while (x < 5)
  x = x + 1;

3.9.4 Variable Declaration Statements

A variable declaration statement is used to declare variables.

Syntax:
VarDeclStmt ::= TypeName Identifier [= Exp] ;

A variable has an optional initialization expression for defining the initial value of the variable. The initialization expression is mandatory for variables of data types. It can be left out only for variables of reference types, in which case the variable is initialized with null.

Example:

Bool b = True;

3.9.5 Assign Statement

The Assign Statement assigns a value to a variable or a field.

Syntax:
AssignStmt ::= Variable = PureExp ;
  | FieldAccess = PureExp ;

Example:

this.f = True;
  x = 5;
3.9.6 Await Statement

Await Statements suspend the current task until the given guard is true (cf. Section [D.2]). While the task is suspended, other tasks within the same COG can be activated. Await statements are also called scheduling points, because they are the only source positions, where a task may become suspended and other tasks of the same COG can be activated.

Syntax:

\[
\text{AwaitStmt} ::= \text{await Guard ;}
\]
\[
\text{Guard} ::= \text{ClaimGuard} \\
| \text{PureExp} \\
| \text{Guard 
& Guard}
\]
\[
\text{ClaimGuard} ::= \text{Variable ?} \\
| \text{FieldAccess ?}
\]

Example:

```javascript
Fut<Bool> f = x!m();
await f?;
await this.x == True;
await f? & this.y > 5;
```

3.9.7 Suspend Statement

A Suspend Statement is just syntactic sugar for an Await Statement with a True guard.

Syntax:

\[
\text{SuspendStmt} ::= \text{suspend ;}
\]

Example:

```javascript
suspend;
```

3.9.8 Skip Statement

The Skip Statement is a statement that does nothing.

Syntax:

\[
\text{SkipStmt} ::= \text{skip ;}
\]

3.9.9 Assert Statement

An Assert Statement is a statement for asserting certain conditions.
Syntax:

\[
\text{AssertStmt} ::= \text{assert PureExp ;}
\]

Example:

\texttt{assert } x \texttt{! = null;}

### 3.9.10 Return Statement

A Return Statement defines the return value of a method. A return statement can only appear as a last statement in a method body.

Syntax:

\[
\text{ReturnStmt} ::= \text{return PureExp ;}
\]

Example:

\texttt{return } x;

### 3.9.11 Expression Statement

An Expression Statement is a statement that only consists of a single expression. Such statements are only executed for the effect of the expression.

Syntax:

\[
\text{ExpStmt} ::= \text{Exp ;}
\]

Example:

\texttt{new } C(x);

### 3.10 Classes and Interfaces

Objects in ABS are built from classes, which implement interfaces. Only interfaces can be used as types in ABS.

#### 3.10.1 Interfaces

Interfaces in ABS are similar to interfaces in Java. They have a name, which defines a nominal type, and they can extend arbitrary many other interfaces. The interface body consists of a list of method signature declarations. Method names start with a lowercase letter.
Syntax:

\[
\begin{align*}
\text{InterfaceDecl} & ::= \text{interface } \text{TypeId} \ [\text{extends } \text{TypeName (, TypeName\*)}] \ \{ \text{MethSigList} \} \\
\text{MethSigList} & ::= [\text{MethSig}, \text{MethSig}] \\
\text{MethSig} & ::= \text{Type Identifier ( ParamList ) ;} \\
\text{ParamList} & ::= [\text{Param (, Param)*}] \\
\text{Param} & ::= \text{Type Identifier}
\end{align*}
\]

The interfaces in the example below represent a database system, providing functionality to store and retrieve files, and a node of a peer-to-peer file sharing system. Each node of a peer-to-peer system plays both the role of a server and a client. The data types are defined in the ABS standard library, included in Appendix A, and the remainder types are type synonyms included in Section 3.5. The full details of this example are presented in Appendix E.

Example:

```plaintext
interface DB {
    File getFile(Filename fId)
    Int getLength(Filename fId)
    Unit storeFile(Filename fId, File file)
    Filenames listFiles()
}

interface Client {
    List<Pair<Server,Filenames>> availFiles(List<Server> sList)
    Unit reqFile(Server sId, Filename fId)
}

interface Server {
    Filenames inquire()
    Int getLength(Filename fId)
    Packet getPack(Filename fId, Int pNbr)
}

interface Peer extends Client, Server {
    List<Server> getNeighbors()
}
```

3.10.2 Classes

Like in typical class-based languages, classes in ABS are used to create objects. Classes can implement an arbitrary number of interfaces. ABS does not support inheritance, as code reuse in ABS is realized by delta modules. Classes do not have constructors in ABS but instead have class parameters and an optional init block. Class parameters actually define additional fields of the class that can be used like any other declared field.
Syntax:

\[
\text{ClassDecl} ::= \text{class TypeId } [\text{ ParamList }] [\text{ implements TypeName (, TypeName)*}] \\
\{ [\text{FieldDeclList}] [\text{ Block }] [\text{ MethDeclList}] \}
\]

\[
\text{FieldDeclList} ::= \text{FieldDecl (, FieldDecl)*}
\]

\[
\text{FieldDecl} ::= \text{TypeId Identifier [= PureExp];}
\]

\[
\text{MethDeclList} ::= \text{MethDecl (, MethDecl)*}
\]

\[
\text{MethDecl} ::= \text{Type Identifier ( ParamList ) Block}
\]

We continue peer-to-peer example with an implementation of the DB interface, and the signature of a node that implements the Peer interface. The full implementation of a node is presented in Appendix E.

Example:

```java
class DataBase(Map<Filename, File> db) implements DB {
  File getFile(Filename fId) {
    return lookup(db, fId);
  }
  Int getLength(Filename fId) {
    return length(lookup(db, fId));
  }
  Unit storeFile(Filename fId, File file) {
    db = insert(Pair(fId, file), db);
  }
  Filenames listFiles() {
    return keys(db);
  }
}
class Node(DB db, Peer admin, Filename file) implements Peer {
  Catalog catalog;
  List<Server> myNeighbors;
  // implementation...
}
```

3.10.2.1 Active Classes

A class can be active or passive. Active classes start an activity “on their own upon creation, passive classes only react to incoming method calls.

A class is active if and only if it has a run method:

Example:

```java
Unit run() {
  // active behavior ...
}
```
The run method is called after object initialization.

### 3.11 Annotations

ABS supports annotations to enrich an ABS model with additional information, for example, to realize pluggable type systems. Annotations can appear before any declaration and type usage in ABS programs (which is not given in the grammar definitions, to improve readability).

**Syntax:**

\[
\text{Annotation} \ ::= \ [\text{TypeName} : \text{PureExp}]
\]

**Example:**

```
[LocationType:Near] Peer p;
[Far] Network n;
List<[Near] Peer> peers = Nil;
```

#### 3.11.1 Type Annotations

ABS has a predefined “meta-annotation `TypeAnnotation` to declare annotations to be Type Annotations. Data types that are annotated with that annotation are specially treated by the ABS compiler to support an easier implementation of pluggable type systems.

**Example:**

```
[TypeAnnotation]
data LocationType = Far | Near | Somewhere | Infer;
```

### 3.12 Modules

For name spacing, code structuring, and code hiding purposes, ABS offers a module system. The module system of ABS is very similar to that of Haskell [110]. It uses, however, a different syntax that is similar to that of Java [87] and Python.
Syntax:

\[
\begin{align*}
\text{ModuleDecl} & ::= \text{module } \text{TypeName} \ [\text{ExportList} \ [\text{ImportList}] \ \text{Decl}^{*} \ [\text{Block}] \\
\text{ExportList} & ::= \text{Export} \ (, \ \text{Export})^{*} \\
\text{ImportList} & ::= \text{Import} \ (, \ \text{Import})^{*} \\
\text{Export} & ::= \text{export AnyNameList [from TypeName]} \ ; \\
& \quad | \text{export * [from TypeName]} ; \\
\text{Import} & ::= \text{import AnyNameList [from TypeName]} ; \\
& \quad | \text{import * from TypeName} ; \\
\text{AnyNameList} & ::= \text{AnyName} \ [, \ \text{AnyName}] \\
\text{AnyName} & ::= \text{Name} \ | \ \text{TypeName} \\
\text{Decl} & ::= \text{FunDecl} \ | \ \text{TypeSynDecl} \ | \ \text{DataTypeDecl} \\
& \quad | \ \text{InterfaceDecl} \ | \ \text{ClassDecl} \ | \ \text{DeltaDecl}
\end{align*}
\]

A module with name MyModule is declared by writing

\[
\begin{align*}
\text{module} & \ \text{MyModule};
\end{align*}
\]

This declaration introduces a new module name MyModule which can be used to qualify names. All declarations which follow this statement belong to the module MyModule. A module name is a type name and must always start with an upper case letter.

### 3.12.1 Exporting

By default, modules do not export any names. In order to make names of a module usable to other modules, the names have to be exported. Exporting is done by writing one or several exports after the module declaration. For example, to export a data type and a data constructor, one can write something like this:

\[
\begin{align*}
\text{module} & \ \text{Drinks}; \\
\text{export} & \ \text{Drink, Milk}; \\
\text{data} & \ \text{Drink} = \text{Milk} \ | \ \text{Water};
\end{align*}
\]

Note that in this example, the data constructor Water is not exported, and can thus not be used by other modules. By only exporting the data type without any of its constructors one can realize abstract data types.

### 3.12.1.1 Exporting Everything

Sometimes one wants to export everything from a module. In that case one can write:

\[
\begin{align*}
\text{export} & \ *,
\end{align*}
\]

In this case, all names that are defined in the module are exported, in particular, this means that imported names are not exported.

### 3.12.2 Importing

In order to use exported names of a module in another module, the names have to be imported. After the list of export statements follows an optional list of imports, which are used to import names from other modules. For example, to write a module that imports the Drink data type of the module Drinks one can write something like this:

\[
\begin{align*}
\text{module} & \ \text{Bar}; \\
\text{import} & \ \text{Drinks.Drink};
\end{align*}
\]

After a name has been imported, it can be used inside the module in a fully qualified way.
3.12.2.1 Unqualified Importing

To use a name from another module in an unqualified way one has to use *unqualified imports*. For example, to use the \texttt{Milk} data constructor inside the \texttt{Bar} module, without having to qualify it with the \texttt{Drinks} module each time, one uses the unqualified import statement:

\begin{verbatim}
module Bar;
import Drinks.Drink;
import Milk from Drinks;
\end{verbatim}

Note that this kind of import also imports the qualified names. So in this example the names \texttt{Milk} and \texttt{Drinks.Milk} can be used inside the module \texttt{Bar}.

To use all exported names from another module in an unqualified way one can write:

\begin{verbatim}
import * from SomeModule;
\end{verbatim}

3.12.3 Exporting Imported Names

It is possible to export names that have been imported. For example,

\begin{verbatim}
module Bar;
export Drink;
import * from Drinks;
\end{verbatim}

exports data type \texttt{Drink} that has been imported from \texttt{Drinks}.

To export all names imported from a certain module one can write:

\begin{verbatim}
export * from SomeModule;
\end{verbatim}

In this case, all names that have been imported from module \texttt{SomeModule} are exported. For example,

\begin{verbatim}
module Bar;
export * from Drinks;
import * from Drinks;
\end{verbatim}

exports all names that are exported by module \texttt{Drinks}.

However, in this example:

\begin{verbatim}
module Bar;
export * from Drinks;
import Drink from Drinks;
\end{verbatim}

only \texttt{Drink} is exported as this is the only name imported from module \texttt{Drinks}. Note: only names that are visible in a module can be exported by that module.

To only export some names from a certain module one can write, for example:

\begin{verbatim}
module Bar;
export Drink from Drinks;
import * from Drinks;
\end{verbatim}

only exports \texttt{Drink} from module \texttt{Drinks}.
3.13 Model

A Model in ABS represents a type-closed set of Modules. A Module defines a set of declarations and an optional Main Block. Modules reside in Compilation Units, which are typically represented by files ending with .abs. A Model is thus set of Compilation Units.

Syntax:

\[
\text{Model ::= CompilationUnit}^* \\
\text{CompilationUnit ::= ModuleDecl}^*
\]

3.14 Related Work

The objective of ABS is to situate itself between design-level notations, foundational calculi, and programming languages. The concurrent object model of ABS based on asynchronous communication and a separation of concern between communication and synchronization is part of a trend in programming languages today, due to the increasing focus on distributed systems. For example, the recent programming language Go (http://golang.org, promoted by Google) shares in its design some similarities with ABS: a nominal type system, interfaces (but no inheritance), concurrency with message passing and non-blocking receive. Similarly, the Actor extension of the Scala language provides support for asynchronous messages and futures [91]. Erlang [23] also supports the Actor model, with a non-blocking send operation, but neither of these languages provide the cooperative scheduling supported in ABS. Below, we briefly compare the mechanisms proposed in Core ABS to related work in the areas of design-level notations, foundational calculi, and programming languages.

Foundational calculi. The ABS process concept is inspired by notions from process algebra [131, 100]. In fact, future variables as used in ABS resemble channels in process algebra, with operations for sending, receiving, and polling. Process algebra is usually based on synchronous communication. In contrast to the asynchronous π-calculus [101], which encodes asynchronous communication in a synchronous framework by dummy processes, our communication model is truly asynchronous and without channels: message overtaking may occur. Furthermore, ABS differs from process algebra in its integration of processes as tasks in an object-oriented setting using method activations, including active and passive object behavior, and self reference rather than channels. In formalisms based on process algebra the operation of returning a result is not directly supported, but typically encoded as sending a message on a fresh return channel [146, 172, 153]. This provides a unique reference to a call, similar to the values bound to ABS future variables at runtime.

Object calculi such as the ς-calculus [1] and its concurrent extension [86] aim at a direct expression of object-oriented features, supporting, e.g., the return of result values, but asynchronous invocation of methods is not addressed. This also applies to Obliq [44], a programming language based on similar primitives which targets distributed concurrent objects. The concurrent object calculus of [70] provides both synchronous and asynchronous invocation of methods. In contrast to ABS, return values are discarded when methods are invoked asynchronously and the two ways of invoking a method have different semantics.

The internal concurrency model of concurrent objects in ABS stems from the intra-object cooperative scheduling introduced in Creol [107] and may be compared to monitors [99] or to thread pools executing on a single processor, with a shared state space given by the object attributes. In contrast to monitors, explicit signaling is avoided. In contrast to thread pools, processor release is explicit. The activation of suspended processes is non-deterministically handled by an unspecified scheduler. Consequently, intra-object concurrency in ABS is similar to the interleaving semantics of concurrent process languages [71, 17], where each ABS process resembles a series of guarded atomic actions (discarding local process variables). In contrast to monitors, sufficient signaling is ensured at the semantic level, which significantly simplifies reasoning [61]. Internal reasoning control is facilitated by the non-preemptive cooperative scheduling; i.e., preemption occurs at explicitly declared release points, at which class invariants are expected to hold [72].
Design-level notations. Integrated formal methods that combine state-based object-oriented structuring languages such as Object-Z and B with process algebras such as CSP and CCS exploit process algebra to express channel communication and synchronization [163][78]. In this vein of work, TCOZ [125] addresses asynchronous communication explicitly through actuators and sensors which represent the local channel ends of asynchronous channels, making global information unnecessary. However, channel-based communication in integrated approaches based on process algebra fixes the communication medium and disallows message overtaking. Finally, the high-level integration of asynchronous and synchronous communication in ABS, in which a method may be invoked in both ways (suspending or blocking), and the organization of pending processes and interleaving at release points within objects seem hard to be captured naturally in process algebra and integrated approaches which fix the communication structure.

Maude’s inherent object concept [126][54] represents an object’s state as a subconfiguration, as we have done in this report, but in contrast to our approach object behavior is captured directly by rewrite rules. Both Actor-style asynchronous messages and synchronous transitions (rewrite rules which involve more than one object) are allowed, which makes Maude’s object model very flexible. However, asynchronous method calls and processor release points as proposed in this paper are hard to be represented directly using Maude’s proposed object model.

There are at least three main differences between the design rationale of ABS as compared to UML [139]: first, ABS concentrates on the precise specification of behavior in a concurrent setting. While some UML diagram types also allow to specify behavior (e.g., activity or state diagrams), the level of specification is much more abstract and does not provide a precise semantics for concurrency or language constructs that support encapsulation of components. The language OCL [177], used for specifying constraints or assertions in UML, does not support concurrency or inheritance. Second, ABS has a uniform, formal, executable semantics whereas UML has only various partial formal semantics for some diagram types that are mostly not in sync with the latest versions of the language. Third, UML has been conceived as a collection of different notations with unified graphical elements, whereas ABS is designed as a homogeneous language with one abstract syntax. For example, UML offers asynchronous event communication and synchronous method invocation but does not integrate these, resulting in significantly more complex formalizations [62] than ours. To facilitate the developer’s task and reduce the risk of errors, implicit control structures combined with asynchronous method calls as proposed in ABS seem more attractive, allowing a higher level of abstraction in the language.

Other abstract languages such as feature description languages are essentially structural. Formalisms such as the B method [2] or Abstract State Machines (ASMs) [38] allow specification of behavior, but they are based on very low-level (in a programming language sense), generic concepts (set theory in the case of B, one-place updates of sorted algebras in the case of ASM). This leaves the modeling of complex concurrent behavior and object composition up to the user of the language and makes it very tedious to model non-trivial systems.

Programming languages. Many object-oriented languages offer constructs for concurrency; surveys are given in [144][42]. A common approach has been to keep activity (threads) and objects distinct, as done in Hybrid [136] and Java [87]. These languages rely on the tightly synchronized RMI model of method calls, forcing the calling method instance to block while waiting for the reply to a call. Verification considerations suggest that methods should be serialized [41], which would block all activity in the calling object. Closely related are method calls based on the rendezvous concept in languages where objects encapsulate activity threads, such as Ada [17] and POOL-T [15].

For distributed systems, with potential delays and even loss of communication, activity threads as well as the tight synchronization of the RMI model seem less desirable. Hybrid offers delegation as an explicit construct to (temporarily) branch an activity thread. Clearly, asynchronous method calls may be seen as a form of delegation. Asynchronous method calls can be implemented in, e.g., Java by explicitly creating new threads to handle calls [56]. In ABS, polling for replies to asynchronous calls is handled at the level of the operational semantics: no active loop is needed to poll for replies to delegated activity.

Languages based on the Actor model [7][9][27][120] take asynchronous messages as the communication primitive, focusing on loosely coupled processes with less synchronization. This makes Actor languages conceptually attractive for distributed programming. The interpretation of method calls as asynchronous messages has lead to the notion of future variables which may be found in languages such as ABCL [180], Argus [121], ConcurrentSmalltalk [179], Eiffel/ [45][47], CJava [56], and in the Join calculus [81] based languages Polyphonic C [33] and Join Java [102]. Our communication model is also based on asynchronous messages and the proposed asynchronous method calls resemble
programming with future variables, but the explicit processor release points in ABS further extend this approach to
asynchrony with additional flexibility.

ABS is based on the model of concurrent object groups communicating via message passing instead of shared
state. Typically, object-oriented approaches that are based on message passing are based on the concept of active
objects, where the unit of concurrency are single objects and not groups of objects as in ABS. Examples are ABCL/1
[181], POOL2 [16], Eiffel/ [45], [47], and Creol [107]. Hybrid [135] introduced the concept of domains, which group
active objects. However, communication in Hybrid is via synchronous method calls. ASP [40] groups objects into so-
called activities. Activities, however, have only one distinguished object, the active object, which can be referenced by
other activities. Other objects of an activity are passive and deep-copied between activities. ABS allows to reference
any object of a COG from other COGs. Instead of using passive objects to transfer data between COGs, ABS uses
functional data-types. The ASP concurrency model is implemented in ProActive [25].

The E programming language [130] introduces a concurrency model called communicating event loops. The
unit of concurrency in E is a vat, which hosts a group of objects. All objects of a vat can be referenced by other vats,
equally to COGs. Computations inside a vat, however, are only executed by a single thread of control. This leads to an
event-based programming model, where the control flows has to be spread over event handlers. The E programming
model can be simulated in ABS by never suspending or yielding a task. The E programming model has been lately
adopted by AmbientTalk [169]. Like ASP, AmbientTalk also integrates a notion of passive objects called isolates.

The idea of using cooperative multitasking for concurrency inside of active objects stems from Creol [107]. In
addition, combining cooperative multitasking with futures, was also pioneered by Creol [64]. Creol, however, has
single objects as the unit of concurrency, which does not allow for multiple service objects. The Creol model corre-
sponds to concurrent object groups with only single objects in ABS. In Symbian OS [133] active objects are scheduled
coopertively within the same active scheduler, which are thread-local. Each active object only has a single thread
of control. Symbian OS also shows how to combine active objects with GUI programming. Kilim [164] allows to
schedule tasks cooperatively if the assigned scheduler is configured to be single-threaded. Rodriguez and Rossetto
[151] combine cooperative multitasking with asynchronous RMI in the Lua language.

Thorn [36] is an actor approach, which combines message sending and asynchronous method calls. Unlike ABS,
Thorn does not support multiple tasks within a process. However, Thorn has a special splitSync construct, which can be
used to solve the problems of the single-threading of Thorn components in some cases. Only methods declared to
be asynchronous can be invoked in an asynchronous way, where in ABS, any method can be invoked asynchronously.

X10 [49] is a novel object-oriented language targeting non-uniform cluster computing. X10 has special support
for explicitly partitioning the heap by a concept called places. Inside a place, multiple activities can run concurrently.
Places in X10 are similar to COGs in the sense that they partition the object heap into groups of objects. However,
places in X10 cannot be dynamically created and their number has to be fixed when a program is started. In addition,
activities inside places are scheduled pre-emptively, where ABS uses cooperative multitasking inside COGs.

Futures. The notion of futures used in ABS stems from Creol [64]. Futures were devised as a simple means
for expressing concurrency in a manner that reduced the dependency on latency by enabling synchronization at the
latest possible time. Futures were discovered by Baker and Hewitt in the 70s [26], and later rediscovered by Liskov
and Shrir as Promises [121] and by Halstead in the context of MultiLisp [92]. Futures appear in languages like
Alice [152], Oz-Mozart [17], Concurrent ML [150], C++ [119] and Java [178], often as libraries. Futures in these
languages are essentially the same as in our language.

All implementations associate a future with the asynchronous execution of an expression in a new thread. The
future is a placeholder object which is immediately returned to the calling site. From the perspective of the calling
site, this placeholder is a read-only structure [134]. In some systems, this placeholder can be explicitly manipulated
by the programmer in order to write the resulting data. In many implementations of futures, the placeholder can be
accessed in both modes (CML, Alice, Java, C++, etc), though typically the design is such that both interfaces are
presented separately — one to the caller and one to the callee. The calculus λ(fut) [134] formalizes this distinction.
Programming with promises explicitly is quite low-level, so our language ties writing the resulting value with method
call return.

Futures can either be transparent or non-transparent. Transparent futures cannot be explicitly manipulated, the
type of the future is the same as the expected result, and accesses made to the future transparently access the result
stored in the future, possibly after waiting (e.g., in Multilisp). Non-transparent futures have a separate type to denote the future (e.g., !T is a future of type T), and future objects can be manipulated (e.g., in CML, Alice, Java, C++, and our language). In addition, futures can also be dealt with lazily to give the effect of call-by-need computation, by delaying the invocation of the asynchronous computation until the moment when the future is accessed (e.g., in Alice).

Flanagan and Felleisen [80] present different semantic models of futures at various levels of abstraction in terms of an abstract machine. Their goal was to enable optimizations and program analyses. Their language was purely functional in contrast to ours, which is an imperative, object-oriented language.

Caromel, Henrio, and Serpett [46] present an imperative, asynchronous object calculus with transparent futures. Their active objects may have internal passive objects which can be passed between active objects by first deep copying the entire (passive) object graph. We do not provide this feature, which is orthogonal to the issue discussed in this paper. To manage the complexity of reasoning about distributed and concurrent systems, they restrict the language to ensure that reduction is confluent and deterministic, whereas our focus is on preserving object invariants. No proof theory is presented for their calculus.

Actor systems [5] are concurrent processes which communicate exclusively through asynchronous messages. An actor encapsulates its fields, procedures that manipulate the state, and a single thread of control. Our objects are similar to actors, except that our methods return values which are managed by futures, and control can be released at specific points during a method execution. Messages to actors return no result and run to completion before another message can be handled. The lack of return makes programming with actors cumbersome.

Reasoning. In previous work, we have shown that the notion of futures adopted in ABS have particularly nice properties for verification, in contrast to shared variables in general [64]: concurrent objects with futures basically allow local reasoning similar to reasoning for sequential programs. Proof systems for actor languages exist [73], but these require explicit structures in the proof rules for reasoning about message queues, which our proof theory avoids. Previous work by UIO [72] on the verification of asynchronous method calls was performed in a language without first-class futures. The paper took a transformational approach by encoding the language into a sequential language with a non-deterministic assignment operator. However, the Hoare rules described only the custom semantics. Various proof systems for monitors exist [84, 99]. Our approach is distinct as we present a novel model of an object that maintains multiple local invariants monitoring its release points and a global invariant that describes its interaction with the other objects via futures. The model is formalized and has a sound and complete proof theory. Initial work on integrating this proof system into the KeY prover [31] has been presented in [8, 9].
Chapter 4

Behavioral Interface Specification Framework

In this chapter we introduce the behavioral interface specification framework and techniques based on this framework for the ABS, allowing for the specification of behavioral properties of components (Objective 8). The specification language forms the basis for static verification (Task 2.5), refinement and abstraction (Task 2.6), correctness (Task 4.3) and development of tool support for both run-time checking and static verification by means of the theorem provers such as KeY (Task 1.3).

As the notion of components in the ABS is not yet fully developed, in the framework presented here the Concurrent Object Groups (COG) [67] form the basic unit of run-time components. The foundations of semantics of such object-oriented components are based on the notion of a history of observable events [60]. A behavioral specification gives a set of legal histories a component is allowed to exhibit in response to environmental stimuli.

To have enough information to describe the behavior of a component, the characteristics of the messages passing between COGs need to be extracted from the Core ABS semantics. Three types of messages which influence the behavior of COGs are the COG creation messages, the asynchronous method calls and the resolution of futures. As the environment and a COG can access each other only through the COG’s provided and required public interfaces, the messages can be extracted from the list of interfaces a component supports. These messages then act as the alphabet for specifying the behavior of the component.

As specifying the set of legal histories by enumerating the set content is not a feasible approach, we need to produce a specification technique that enables compact behavioral specifications. As different techniques may fit better in different situations, we instead provide a generic framework and show how different techniques can be fitted into this framework. Here we provide techniques based on attribute grammars [65] and interaction patterns [118]. Grammars allow to specify protocol-oriented properties in a declarative and highly convenient manner, while the attributes focus on data-oriented properties. Interaction patterns are a light-weight technique for specifying interactions of a component in terms of pairs of incoming and outgoing messages and their associated pre- and post-conditions.

The remaining part of this chapter is organized as follows. In Section 4.1 we describe the generic framework for specifying a component. This section elaborates on the necessary ingredients to denote the behavior of a component without showing irrelevant implementation details. Section 4.2 provides an informal description how the generic framework can be instantiated through two example specification techniques: attribute grammars (Section 4.2.2) and interaction patterns (Section 4.2.3). The chapter closes with a discussion on related work.

4.1 Message Specification Framework

The message specification framework described in this section comes from the need of having common ground to apply various specification techniques for specifying the observable behavior of a COG. The specifications need to be related to the ABS programs, and the aim of this section is to identify parts of the core ABS that are relevant for specification techniques.

The first part that stands out in ABS is that a COG is inherently a runtime entity. In an ABS program, there are no specific classes that indicate that it is a starting point of a COG, and a COG is created when the construction of a new object is appended with the information that says that it forms a new COG (cf. new cog $C(\sigma_p)$). If we want to
say that a COG satisfies a certain specification \( S \), then this has information to be encoded directly at that spot through an extension of the ABS syntax (e.g., \texttt{new cog satisfies \( SC(cog) \))}. The specification is then written somewhere else, although additional annotations may be embedded into the program to support the specification. This extension can be integrated to the notion of component currently under development in Tasks 2.4 and 3.1.

The second part is that communication between a COG and its environment (which as a whole we consider as a system) is done through interfaces. This means that a specification of a COG should base itself on the interfaces used in the ABS program. It is not enough to specify the behavior of each interface and then call the specification of a COG as the collection of the specifications of the interfaces the COG uses, as this approach leads to an underspecified COG. For example, if a COG uses two external object references, \( a \) and \( b \), of the same interface type \( I \) and the COG should call the method \( m \) of \( a \) before calling the method \( m \) of \( b \), then simply stating that the COG allows interaction through \( I \) is not enough to restrict the call order. Therefore it is appropriate to describe the COG’s behavior by looking at the COG as a combination of interfaces that needs to be specified as a whole.

Armed with this analysis, we propose a generic specification framework which is based on possible messages generated by the COG’s interfaces. Figure 4.1 provides the abstract syntax of the generic specification framework. A component begins with the list of interfaces used by the COG, which are the interfaces that will be used in the ABS program. We reuse the constructs for defining the interfaces from the core ABS abstract syntax (see Appendix B), namely an interface \( I \) has (possibly none) some method signatures and may extend other interfaces. Each method signature \( M \) consists of a return type, a method name, and a (possibly empty) list of typed arguments. The types considered here are interface types \( I \), data type \( D \), and future type \( Fut(T) \). The COG specification consists of its name \( S \), the (at least one) interfaces/services the COG provides to its environment (as indicated by \texttt{in interface \( I \)}), the interfaces/services the COG requires from its environment (\texttt{out interface \( I \)}). In our formalization, we use Interface, \( \text{Interface}_{\text{in}} \subseteq \text{Interface} \) and \( \text{Interface}_{\text{out}} \subseteq \text{Interface} \) to represent the set of all interfaces defined in the specification, the required interfaces and the provided interfaces, respectively. Note that \( \text{Interface}_{\text{in}} \) and \( \text{Interface}_{\text{out}} \) do not form partitions of Interface as it is possible for an interface to be used both as provided and required interfaces in a COG specification. The methods contained in the interfaces act as a template for the messages that can come in and go out of the COG. The observable behavior of a COG then can be characterized by these messages.

In ABS, the following operations translate to messages that may appear on the boundary of a COG.

- The new COG creation expressions, which determine the initial state of the COG;
- The asynchronous call expressions, which describe the interaction between an object within the COG and an object outside the COG;
- The return statements, which resolve the value of future objects.

The asynchronous call expressions and the return statements are asymmetric because of the use of futures. An asynchronous call means that the caller wants its callee to process the caller’s request while the caller possibly does something else in between. When this callee finishes processing the call, the callee does not return the value back to the caller (in fact the callee does not know who the caller is, unless this information is explicitly given as one of the call parameters), but instead resolve the value of the future. This resolution is done once, but as the reference to the futures can be passed around, there can be more than one request (also known as \texttt{claim [3]})) to access the value of the resolved future. These requests are not visible anymore as it is assumed that the value of a resolved future is stored in some global collection, accessible to any object that has a reference to the future. Other operations, such as synchronous calls, do not generate messages, as they operate only within a COG.

Figure 4.1: Abstract syntax for component specification

\[
\begin{align*}
CS & := \text{In}^{*} \text{Spec} & \text{component specification} \\
\text{In} & := \text{interface } I \{ \text{extends } I^+ \} \{ M_s \} & \text{interface declaration} \\
M_s & := T \ m((T\ x)^*) & \text{method signature} \\
T & := I \ | \ D \ | \ Fut(T) & \text{types} \\
\text{Spec} & := \text{spec}\ S \{ \text{in interface } I^+; \text{out interface } I^+; \text{specification} \} & \text{specification declaration}
\end{align*}
\]
We formalize the notions of messages and histories as follows. First we define the notation we use, and then describe the set of messages that may appear on the boundary of a COG. Using the message set, we construct the history as a sequence of messages. A set of histories defines the allowed observable behavior of a COG.

Let \( \text{Obj} \) be the set of object identities that exist within the system. To handle futures, which in ABS are first class entities, the set of futures are distinguished into \( \text{Obj}_{\text{fut}} \subseteq \text{Obj} \). Let \( \text{Mtd} \) be the set of methods, and \( \text{Mtd}_{\text{cog}} \subseteq \text{Mtd} \) be the set of methods extracted from \text{Interface}. To accommodate the need for describing a new COG creation, we assume that \( \text{Mtd} \) also includes constructors. This extension will be clearly indicated in the messages. Another approach is to use the factory pattern \[82\] to represent the constructors, which restricts the object’s usage through the interface return type. Let \( \text{Value} \) be the set of values which encompasses all possible parameter values in actual method calls and return values, including \( \text{Obj} \). We assume \( \text{Unit} \) to be a part of \( \text{Value} \) to allow capturing the return of a method whose return type is \( \text{Unit} \).

To characterize the history, we use the sequence type \( \text{Seq}(T) \). An empty sequence is denoted by \( \varepsilon \). We write a sequence in the usual way: \( s = [e_1, e_2, \ldots] \). In the following, we use the concatenation function \( _{-}^{-} : \text{Seq}(T) \times \text{Seq}(T) \rightarrow \text{Seq}(T) \) which adds the second sequence to the end of the first sequence.

**Definition 4.1.1 (Messages).** Let \( o \in \text{Obj}, f \in \text{Obj}_{\text{fut}}, m \in \text{Mtd}, v \in \text{Value}, \overline{v} \in \text{List}(\text{Value}) \). The set of messages \( \text{Msg} \) is defined as the union of the following sets of messages:

- The set \( \text{CCM} \) of COG creation messages \( o : \text{new cog } C(\overline{v}) \). This kind of message denotes the creation of a new object \( o \) in a new COG. The choice of interface type of \( o \) is left to the specification.

- The set \( \text{ACM} \) of asynchronous call messages \( f : o!m(\overline{v}) \). This denotes calling the method \( m \) of the object \( o \) called with parameters \( \overline{v} \) and returns a future \( f \).

- The set \( \text{RM} \) of resolve messages \( f : \text{resolve } v \). This denotes the resolution of the future \( f \) with the value \( v \).

Information can be extracted from the messages by the following functions. The function \( \text{par} : \text{CCM} \cup \text{ACM} \rightarrow \text{List}(\text{Value}) \) extracts the parameter values from a COG creation or an asynchronous call message. The function \( \text{par}_{\text{Obj}} \) refines \( \text{par} \) to extract the set of objects from the method/constructor parameters. The function \( \text{future} : \text{ACM} \cup \text{RM} \rightarrow \text{Obj}_{\text{fut}} \) extracts the future object from an asynchronous call or a resolve message. The function \( \text{method} : \text{ACM} \rightarrow \text{Mtd} \) extracts the method from an asynchronous call. The function \( \text{callee} : \text{ACM} \rightarrow \text{Obj} \) extracts the callee of an asynchronous call. The function \( \text{resVal} : \text{RM} \rightarrow \text{Value} \) extracts the value of a future resolution. The function \( \text{initObj} : \text{CCM} \rightarrow \text{Obj} \) extracts the first object created in the new COG.

**Remark 4.1.2 (Non-uniform message format).** In, e.g., \[106, 167\] the notion of messages in a concurrent object setting is defined uniformly as \( \langle \text{sender}, \text{receiver}, \text{method}, \text{parameters} \rangle \). In the ABS case, the luxury of knowing the sender of a message is not available. Furthermore, instead of sending the result of an asynchronous call back to its sender, the return value is used to resolve the value of the corresponding future object. Using a non-uniform message notion allows the resulting specification to be closer to the ABS programs, which may ease the verification efforts later on.

**Remark 4.1.3 (Splitting the messages).** The message model above can be refined by introducing to the COG creation and asynchronous call messages the notion of message emission and reception (as in, e.g., interface automata \[63\] or behavioral protocols of SOFA \[147\]). This means that we distinguish between when a message is sent by a COG and when it is received by the corresponding COG. To keep the presentation simple and keeping with the goal of specifying a component, we avoid doing so, and consider that when a message appears in the history, it has been received.

Using the messages defined above, we are now ready to define the histories. We first define the possible histories of a system (i.e., the COG and the whole environment), then we concentrate on the possible histories that portray the interaction between the COG and its environment, excluding the interaction between COGs in the environment. Note that, as will be explained later, we may still need to include the resolution message produced by a COG of the environment that is not interacting directly with the COG whose behavior is to be specified. As not all COG histories coincide with the ABS operational semantics, we describe what it means for a COG history to be well-formed.
Definition 4.1.4 (Histories). A history of a system \( h \) is a finite sequence of type \( \text{Seq}(\text{Msg}) \). The set of histories is a prefix-closed subset of \( \text{Msg}^* \), where \( ^* \) is the Kleene star operator.

The history of a COG contains only messages whose callees/initial objects/futures/parameters/return values are in the set of objects \( \text{Obj}_{\text{cog}} \subseteq \text{Obj} \) to reflect which objects lie on the boundary of the COG. In other words, \( \text{Obj}_{\text{cog}} \) contains all objects residing in the COG that are exposed to the COG’s environment and all objects residing in the COG’s environment that are exposed to the COG. Based on this, \( \text{Obj}_{\text{cog}} \) is partitioned into internal and external sets of objects \( \text{Obj}_{\text{in}} \) and \( \text{Obj}_{\text{ext}} \), respectively. The set of available messages \( \text{Msg}_{\text{cog}} \subseteq \text{Msg} \) are messages which are built from \( \text{Obj}_{\text{cog}} \) only, so the history of a COG contains only events that are generated by that COG.

Definition 4.1.5 (COG Histories). A history of a COG is a finite sequence of type \( \text{Seq}(\text{Msg}_{\text{cog}}) \).

As we deal with the behavioral specification of a COG, the possible COG histories generated by a specification must be well-formed with respect to the ABS semantics. This means that the knowledge growth of exposed objects from both the COG and its environment follows from the ABS semantics (recall the dynamic nature of the COG). Complicating the matter is the future resolution messages, as explained in the following example.

Example 4.1.6. Figure 4.2 shows an ABS program with three classes, \( A \), \( B \), and \( C \). Let’s assume that instances of \( A \), \( B \), and \( C \) are in different COGs, and we want to specify the behavior of the COG containing an instance of \( C \). It is possible that when that instance receives an asynchronous call \( m(f) \), the future \( f \) already resolved. In such case, the resolution message appears in the history before the call, and therefore before the future \( f \) comes on to the boundary the instance of \( C \). In order for a specification to derive the value the instance of \( C \) should return, we need to allow the possibility of the corresponding future resolution message to appear within a well-formed history even before the asynchronous call is made.

To define well-formedness of a COG history, we define a recursive predicate \( \text{wf} \) which processes a history \( h \) message per message starting from the head of the sequence. The main intuition is that the predicate keeps separate tracks of object references that have been passed by the COG to the environment and by the environment to the COG. Whenever an object reference which is passed by the COG to the environment is not within the set of known external object references, we assume that that object is created within the COG (as local object creation is not observable). Furthermore, for each future resolution message present in the history, we store the pair of future and its resolved value, in order to enrich the knowledge of exposed objects (as claiming a future is not observable). When the whole history is fully processed, then the accumulated knowledge about the exposed objects is contained within the assumed object sets \( \text{Obj}_{\text{in}} \) and \( \text{Obj}_{\text{ext}} \). To simplify matters, we assume the type correctness of the messages within the history, uniqueness of resolution messages for every future and freshness of newly created objects, and also the initial object of the COG is given.
Definition 4.1.7 (Well-formed COG Histories). The well-formedness predicate \( w_{f} : \text{Seq}(\text{Msg}_{\text{COG}}) \times \text{Obj}_{\text{in}} \times \text{Set}(\text{Obj}) \times \text{Set}(\text{Pair}(\text{Obj}_{\text{in}}, \text{Value})) \rightarrow \text{Bool} \) is defined as follows.

\[
\begin{align*}
w_{f}(\varepsilon, & \ldots, O_{1}, O_{2}, \ldots) \overset{\text{def}}{=} O_{1} \subseteq \text{Obj}_{\text{in}} \wedge O_{2} \subseteq \text{Obj}_{\text{ext}} \\
w_{f}(\{ o : \text{new cog} C(\forall) \}^{-} h, o', O_{1}, O_{2}, P) & \overset{\text{def}}{=} \\
& \begin{cases} 
w(h, o', O_{1} \cup \{ o \}, \text{res}(\text{par}_{\text{Obj}}(\forall), O_{1}, O_{2}, P), P) & \text{if } o = o' \\
\emptyset \in O_{1} \wedge w(h, o', O_{1} \cup (\text{par}_{\text{Obj}}(\forall) \setminus (O_{1} \cup O_{2})), O_{2} \cup \{ o \}, P) & \text{if } o \neq o' 
\end{cases} \\
w_{f}(\{ f : \text{a!m}(\forall) \}^{-} h, o', O_{1}, O_{2}, P) & \overset{\text{def}}{=} \\
& \begin{cases} 
\text{false} & \text{if } o' \not\in O_{1} \vee o \not\in O_{1} \cup O_{2} \\
w(h, o', O_{1} \cup \{ f \}, \text{res}(\text{par}_{\text{Obj}}(\forall), O_{1}, O_{2}, P), P) & \text{if } o' \in O_{1} \wedge o \in O_{1} \\
w(h, o', O_{1} \cup (\text{par}_{\text{Obj}}(\forall) \setminus (O_{1} \cup O_{2})), O_{2} \cup \{ f \}, P) & \text{if } o' \in O_{1} \wedge o \in O_{2} 
\end{cases} \\
w_{f}(\{ f : \text{resolve v} \}^{-} h, o, O_{1}, O_{2}, P) & \overset{\text{def}}{=} \\
& \begin{cases} 
w(h, o, O_{1}, O_{2}, P \cup \{ (f, v) \}) & \text{if } f \in O_{1} \wedge (v \not\in \text{Obj} \vee v \not\in O_{2}) \\
w(h, o, O_{1}, O_{2}, P \cup \{ (f, v) \}) & \text{if } f \in O_{2} \wedge (v \not\in \text{Obj} \vee v \not\in O_{1}) \\
w(h, o, O_{1}, O_{2}, P \cup \{ (f, v) \}) & \text{if } f \in O_{2} \wedge (v \not\in \text{Obj} \vee v \not\in O_{1}) \\
w(h, o, O_{1}, O_{2}, P \cup \{ (f, v) \}) & \text{if } f \not\in O_{1} \cup O_{2} 
\end{cases} 
\end{align*}
\]

where the future resolution function \( \text{res} : \text{Set}(\text{Obj}) \times \text{Set}(\text{Obj}) \times \text{Set}(\text{Obj}) \times \text{Set}(\text{Pair}(\text{Obj}_{\text{in}}, \text{Value})) \rightarrow \text{Set}(\text{Obj}) \) is defined by

\[
\text{res}(\emptyset, \ldots, O_{2}, \ldots) \overset{\text{def}}{=} O_{2} \\
\text{res}(\{ o \} \cup O_{1}, O_{2}, P) & \overset{\text{def}}{=} \\
& \begin{cases} 
\text{res}(O, O_{1}, O_{2}, P) & o \in O_{1} \\
\{ o \} \cup \text{res}(O, O_{1}, O_{2}, P) & o \not\in O_{1} \wedge o \not\in \text{Obj}_{\text{in}} \\
\{ o \} \cup \text{res}(O, O_{1}, O_{2}, P) & o \not\in O_{1} \wedge o \in \text{Obj}_{\text{in}} \wedge (\langle o, v \rangle \in P \implies v \not\in \text{Obj})' \\
\{ o \} \cup \text{res}(\{ v \} \cup O_{1}, O_{2}, P) & o \not\in O_{1} \wedge o \in \text{Obj}_{\text{in}} \wedge \langle o, v \rangle \in P \wedge v \in \text{Obj} 
\end{cases} 
\]

A COG history \( h \) with the initial object of \( o \in \text{Obj}_{\text{in}} \) is called well-formed when the predicate \( w(h, o, \emptyset, \emptyset, \emptyset) \) holds.

The well-formedness predicate takes into account the COG history, the initial object of the COG, and from then on builds up two sets of objects exposed on the boundary of the COG, the set of internal objects \( O_{1} \) and the set of external objects \( O_{2} \). The third set \( P \) captures the information of resolved futures. An underscore means that the corresponding parameter is not important.

The base case of this recursive predicate is when the history is an empty sequence. At this point, the accumulated knowledge of exposed objects (\( O_{1} \) and \( O_{2} \)) is compared to the internal object set \( \text{Obj}_{\text{in}} \) and external object set \( \text{Obj}_{\text{ext}} \).

When a creation message is encountered, the predicate determines whether this is the first creation message in the history. If it is the first, then it should return the expected initial object, which is recorded to the set of internal objects. The parameters of the constructor are scrutinized before joined to the set of external objects exposed to the COG, in order to resolve future values within the parameter. The resolution is done recursively, through the future resolution function \( \text{res} \), because the value of a future may be another future. Note that the resolution is based on exposed internal and external objects and previous resolve messages. If another creation message has occurred before, then the creation message is expected to come from the COG, with the constructor parameters filled with objects known to the COG up to that point. Object parameters which are not yet part of the boundary are assumed to come from the COG. Recall that internal object creation is not observable.

In the case of an asynchronous call message, the COG creation message must appear before (marked by the initial object \( o' \) being part of the exposed internal objects) and the callee must be part of the known boundary objects. If the callee is an internal object, the message is an incoming call and we add to the set of external objects the resolved parameters. Otherwise, the message is an outgoing call, which may open the possibility of new exposed internal objects. In both cases, we note the resulting future accordingly.

The resolve message is the most complex case, simply because we need to distinguish whether the resolved value is an object or not. For the former, the resolved value is put into the corresponding set of internal or external objects,
and the resolution pair is noted down in the third set. Similar to the creation message and the incoming asynchronous calls, resolving the value of an external future requires the \texttt{res} function.

The \texttt{res} function updates the set of external objects \(O_2\) by the resolved value of futures contained in the first parameter of the function \(O\). The base case is when \(O\) is empty, in which case no update is done to \(O_2\). Otherwise, we check each element \(o\) of \(O\), based on the following cases. If \(o\) is an internal object, then we ignore it, or else we use it to update \(O_2\). If \(o\) is not a future, then no resolution is needed. The same applies if \(o\) is a future and its resolution either has not been made or is not an object. If the resolution of \(o\) is yet another object, then we add the resolved object to \(O\).

\textbf{Example 4.1.8.} Referring to the previous example (Example 4.1.6), the following two COG histories are well-formed.

\[
\begin{align*}
\text{[} & o : \text{new cog } C(), f_1 : o!m(f), f : \text{resolve 1}, f_1 : \text{resolve 1}] \\
\text{[} & f : \text{resolve 1}, o : \text{new cog } C(), f_1 : o!m(f), f_1 : \text{resolve 1}] \\
\end{align*}
\]

In this section, the message specification framework has been described. In particular, the framework deals with the common structure a specification should have, namely the required and provided interfaces of the COG, what is considered as messages, which is the main building block of a COG’s specification, and when a history of a COG follows the ABS semantics. In the following section, we see how this framework is instantiated by means of two specification techniques.

\section{Specification Techniques}

A behavioral specification of a component could be as simple as a set of possible histories of messages coming in and going out of the boundary of the component. However, describing the behavior of a component in such a manner is not only cumbersome, but also impossible as the set is likely to be infinite because of our domain of eternal systems. Therefore, we apply various specification techniques to achieve compact specifications that describe exactly what a component can and cannot do.

The specification techniques presented in this section are based on an assertion language, which is described in Section 4.2.1. This language is similar to what is presented in Deliverable 1.1A [67, Section 3.1.3], except instead of using quantifiers, we allow the use of pure methods in the expressions. This assertion language is used in the specification language to make a connection between the knowledge a component has gathered prior to sending out a message and that message itself.

In Section 4.2.2 the attribute grammar technique is presented, while in Section 4.2.3 the interaction pattern technique is presented. The generic framework described in the previous section enables the possibility to apply other specification techniques.

\subsection{Assertion Language}

Together with programming to interfaces [82], the design by contract principle [127] is one of the main approaches to mastering the complexity of software today. A contract is a formal specification of the behavior of an interface in terms of \textit{invariants}, \textit{preconditions} and \textit{postconditions}. In a JML-like style we add primitive constructs to the ABS to specify preconditions of methods (with the \texttt{@ requires} keyword, followed by an assertion), postconditions (\texttt{@ ensures}) and invariants (\texttt{@ invariant}), thus the full ABS natively supports design by contract.

The ABS has a built-in data type \texttt{Bool} (see Appendix A for its definition), and each type has a built-in equality predicate \texttt{==} which results in a \texttt{Bool}.

Assertions are pure expressions \(e_p\) of type \texttt{Bool}. Appendix B shows a grammar describing side-effect free expressions. Note that we do not have quantifiers in our assertions. Quantifiers can be replaced by a function using skolemization. If one replaces an existentially quantified Boolean term \(t\) by a function, the definition of the function is a witness of \(t\). This constructive approach to quantifiers accommodates automated run-time assertion checking.

As the history is an instance of a normal ABS class, we may refer to attributes of the history in pure assertions. How the history is used in assertions is explained further in Section 4.2.2.3.
4.2.2 Attribute Grammar Specification Language

We introduce a formal modeling language for the specification of properties of sequences of messages at an abstraction level which coincides with that of the ABS and which lends itself easily to automated verification of such properties at run-time. Naively using data structures like lists to represent such sequences has proven in practice to lead to very complicated assertions: a systematic approach is needed.

We represent message sequences as words of a language generated by a grammar. Grammars as such allow in a declarative and highly convenient manner to describe the protocol structure of the messages. However, the question now rises how to describe the data flow of a message sequence. We propose here a formal modeling language for the specification of communication histories which describes sequences of messages in terms of attribute grammars. We show how attribute grammars provide a powerful separation of concerns between high-level protocol-oriented properties, which focus on the kind of messages sent and received, and data-oriented properties, which focus on the actual data communicated. Exploiting this separation of concerns, attribute grammars allow the specification of user-defined abstractions of message sequences in an elegant and systematic manner. We illustrate the main concepts with a running example of a stack. A more elaborate case study will be part of Deliverable 5.3 “Evaluation of Modeling”.

4.2.2.1 Communication View

In Definition 4.1.1 it was pointed out that three kinds of messages are to be used in the behavioral description of components: COG creation messages, asynchronous method calls and returns of such calls. This gives rise to several questions: are all such messages relevant for the behavioral specification? Do we also allow incomplete specifications, which focus on a specific property? What is the relation between the messages and attribute grammars? To address these questions, we introduce the concept of a communication view. A communication view is a general mechanism attached to a specification of a COG which specifies a mapping between messages sent/received by that COG and (abstract) terminals of a grammar. We represent this mapping as a set of pairs, as shown in the syntax (which extends that of the ABS) of a communication view in Figure 4.3.

\[
\begin{align*}
V & ::= \text{view } v \{ P \} \quad \text{communication view} \\
E & ::= \text{call | resolve} \quad \text{Asynchronous calls and future resolutions} \\
S & ::= I_{\text{in}}.m | I_{\text{out}}.m \quad \text{method names in interfaces} \\
M & ::= E,S | \text{new C} \quad \text{messages: calls/resolutions and creations} \\
P & ::= (M \ t)^* \quad \text{(message M terminal t) pairs}
\end{align*}
\]

Figure 4.3: Syntax of Communication Views.

The first component in a pair identifies a message, the second component is the corresponding terminal. This allows us to abstract from irrelevant messages (by projection: a message does not appear as first component in the list), identify distinct messages (by giving two different messages the same name), and use intuitive user-specified names for the selected messages. In general the modeling framework supports multiple communication views for a given specification \(S\) (Figure 4.1) which allow the developer to focus on the different behavioral aspects of the COG, using intuitive names for the selected messages. This naturally leads to support for incomplete specifications. As an example, a communication view “stackhist” for the interface of a stack is given in Figure 4.4. This view abstracts from calls to “pop” and returns of “push”.

Note that messages consist of several attributes: the identity of the called object, the method name, parameter values, etc. As a (concrete instance of a) terminal is used to represent a message, a terminal cannot be something as simple as a string: the values of the attributes must be recorded somehow. Each abstract terminal corresponds to a class implemented in the ABS. The instance variables of this class correspond to the attributes of the message, and getters are used to retrieve the value of the instance variables.
4.2.2.2 Context-free grammars

It was described in the previous section that each message corresponds to a concrete terminal (a token). Thus the history of a concurrent object group, as formalized in Definition 4.1.4, can also be represented as a list of instances of terminals. The abstract protocol behavior of a concurrent object group can now be described by a context-free grammar, in which instances of terminals correspond to messages and the actual protocol is defined by the productions of this grammar. In Figure 4.5 a grammar describing a stack is shown. The distinguished non-terminal ‘START’ designates the start symbol of any grammar.

\[
\begin{align*}
START & ::= S \\
S & ::= push S \\
& | SS \\
& | B \\
B & ::= push B pop \\
& | \epsilon
\end{align*}
\]

Figure 4.5: Abstract Stack Behavior.

The language generated by the grammar restricts the set of legal histories of the COG. Note that in general the ongoing behavior of a concurrent object group requires prefix-closed grammars. This implies that the grammar can be seen as defining an invariant property of the history: valid histories are words of the language generated by the grammar, and consequently every time the history is updated (after a message occurs), the history must still be a word of this language. We do not force the use of a fixed grammar to express specifications, but instead allow the use of any grammar provided it is context-free. Two context-free languages may be equal (in the sense that they contain exactly the same words), but can be generated by two completely different grammars. The user is free to choose the most convenient structure of the grammar, which results in a very flexible approach. This would not have been the case had we used for example, regular expressions, where there are several fixed predefined operations (+, *, concatenation) from which to build up other regular expressions. Though in general it is undecidable whether two context-free grammars are equivalent, this is decidable for the restriction to regular grammars. A method for deciding language inclusion of two grammars is important for establishing behavioral equivalence and formalizing behavioral subtyping based on histories (along the lines of [79]).

4.2.2.3 Attribute Grammars

We have used context-free grammars to specify the high-level protocol behavior of concurrent object groups. This gives rise to the question how to describe properties of the data-flow of the history. We have identified attribute grammars [65] as providing a powerful and convenient way to specify behavioral properties of both control-flow and data-flow of histories. In particular, the attributes in the attribute grammar are used to define properties of the data actually communicated by the messages, such as for example the actual parameters in a sequence of asynchronous method calls. As described above, each terminal has a number of built-in attributes. For an asynchronous call, fut records the future, callee the object identity of the callee, and each formal parameter with name v corresponds to an
attribute $v$ of the terminal which captures the value of the actual parameter that was passed to the method. Further, each non-terminal has a number of user-defined attributes which may be both synthesized and inherited. To define attributes of non-terminals we use the side effect-free data type language of ABS. User-defined synthesized attributes of non-terminals (which are defined in terms of the attributes of the direct descendants of the non-terminal) can refer to the user-defined attributes of its “child non-terminals”, as well as to the built-in attributes of its “child terminals”. It is important to note that for a meaningful specification, we must restrict our grammars to disallow dereferencing the objects contained in built-in attributes, as these refer to the current heap, and not the past one in which the message actually occurred. The non-terminal “start” does not have inherited attributes, and the synthesized attributes of “start” conceptually correspond to data-flow properties of the entire history. Figure 4.6 shows an extension of the context-free grammar of Figure 4.5 with attributes. We use subscripts $S_1, S_2$ to distinguish different occurrences of the same non-terminal $S$.

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

Figure 4.6: Attribute Grammar Stack Behavior.

The attributes in the stack grammar capture the items which were pushed but not yet popped. Therefore, the stack attribute of “start” corresponds to the current content of the stack. Note that only the non-terminal “start” is extended with attributes, because “B” describes balanced stacks, and balanced stacks are empty (e.g., all pushed items were popped).

In order to specify the behavioral contract of methods, we allow referring to the history in pre- and postconditions. Specifically, since synthesized attributes of the non-terminal “start” are properties of the whole history, we use an observer function “history.x()” to refer to the value of the synthesized attribute “x” of the non-terminal “start” in the attribute grammar with communication view “history”. As an example we specify the behavior of “push” and “pop” in Figure 4.7. In the assertions we use the logical variable $z$ to refer to the contents of the stack before push/pop were invoked (with type List<Int>, as defined in the ABS Standard Library in Appendix A).

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

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

Figure 4.7: Contracts for the Stack Interface.

### 4.2.2.4 Run-time Assertion Checking

Attribute grammars are generally used to assign meaning (semantic values) to words of a language. A parser of an attribute grammar takes as input a stream of tokens, corresponding to a word of the language, and outputs an abstract
syntax tree annotated with the values of the synthesized and inherited attributes. This, together with the observation that the history can be viewed as a stream of tokens, forms the basis of our approach to run-time assertion checking. In our tool suite we use ANTLR [141], a popular parser generator: ANTLR takes an attribute grammar as input and outputs a parser. The attributes in the attribute grammar are defined using standard Java code. As the first step, we transform our attribute grammar (in which attributes are defined using the ABS language) to an ANTLR grammar. The Java back end is used to transform the ABS attribute definitions to standard Java code and ANTLR generates Java code for a parser of the given attribute grammar. Second, we augment the program under test with updates to the history: every time a message (asynchronous call, COG creation, future resolution) is sent/received the history is updated. When the history is updated, it is immediately parsed by the parser created by ANTLR. Parse errors correspond to violations of the high-level protocol specified by the productions of the grammar. In this case an assertion failure occurs.

After parsing succeeds successfully (and the values of the attributes in the abstract syntax tree have been computed), assertions over the history attributes (of the “start” non-terminal) can be checked in the same way as any other normal assertion (similar to what is done in the Core ABS). As proof of concept, we have implemented a run-time assertion checker for Java which integrates with JML in [65].

4.2.3 Interaction Pattern Specification Language

Interaction patterns are an assertion-based specification technique to describe the black box, protocol behavior of an object group component, both its control and data flows, by characterizing incoming and outgoing messages in a state-based manner (a preliminary write-up of this technique is available in [118]). The main idea is to extend the popular contract-based specification techniques such as Eiffel [127], JML [43] and Spec# [28] by describing the restrictions of and changes in the state of a component after a series of incoming and outgoing message exchanges, instead of just describing them at the entrance of a method call and its return. In this way, all possible interaction patterns between the component and its environment are captured. By relating states that fulfill postconditions of an interaction and states that fulfill preconditions of another interaction, the set of allowed histories can be obtained.

To apply this specification technique to the ABS setting, we take the COG as the object group component, the state as the program store of the COG and the messages described in Section 4.1 as the foundation for the specification. A behavioral specification of a COG consists of the state description and interaction patterns based on the state. Taking the idea of tasks from the description of COG in Appendix D, the interaction patterns of a COG are divided into tasks. Each task has its own local state and contains interaction elements which consist of pairs of lists of incoming and outgoing messages, the pre-, frame, and postconditions of the state (both global to the COG and local to the task). For simplicity, exceptional behaviors are not considered. The incoming messages are the messages the COG receives before producing outgoing messages. Each interaction element illustrates what an active task within the COG does before giving up control to other tasks within the COG (i.e., allowing other tasks to proceed).

\[
\begin{align*}
\text{Specification} &::= (T \ x = e_p)^* ; \text{Task}' \\
\text{Task} &::= \text{task Tname} \{ (T \ x = e_p)^* ; IE^* \} \quad \text{task definition} \\
\text{IE} &::= \text{in Msg^* cog out Msg^* cog requires Assert modifies x^* ensures Assert} \quad \text{interaction element}
\end{align*}
\]

Figure 4.8: Interaction Pattern Specification Abstract Syntax.

The abstract syntax of interaction pattern specification is given in Figure 4.8. Note that the format given here is only to illustrate the approach and still subject to further development. The specification format follows closely to the description we gave above. The specification starts by describing state variables which act as the global state of the COG, which can be used later on by any task, along with their initial values (using the pure expressions $e_p$—usually we would like this initial values to default values, such as null or 0 or false). The types of the state variables are derived from the given interfaces and also built-in data types as defined in Appendix A. This is followed by the task definitions. Each task definition starts with a number of local state variables which form the local state which can be used only by the task, along with their initial values, followed by the interaction elements. Each interaction element states the accepted incoming messages and a precondition based on the global and local state, the frame condition of the state, and the outgoing messages that will be sent as a response and the postcondition of the state.
The incoming messages are annotated using the keyword _in_, the outgoing messages _out_, the precondition _requires_, the frame condition _modifies_ and the postcondition _ensures_. We use the messages extracted from the interfaces as defined in Definition 4.1.1 and the assertions from Section 4.2.1. Only one incoming asynchronous call or COG creation message can be used within a task definition, although its occurrence may be in multiple interaction elements within that task to handle different cases based on the precondition assertion. When the set of incoming or outgoing messages is empty, then the COG is expected to receive or emit no messages, respectively, within that interaction. Empty pre-/postconditions are replaced with a `true` assertion, while empty frame conditions mean that the state is not modified.

The semantics of a specification S written using the interaction pattern technique is based on particular characteristics of the COG histories generated from S. The main intuition behind this characterization is similar to the semantics of a COG. A COG history generated from S can be divided to subhistories (which are subsequences), each of which represents the semantics of a task. A task is started when the COG receives an incoming asynchronous call or COG creation message (obviously, the first task will be triggered by the COG creation message). When a task is in control, it processes the appropriate interaction element with the arrival of the expected incoming message(s) and the satisfied precondition assertion. During this process, other tasks may not interfere with it. This means that the COG outputs without interruption all outgoing messages specified in the appropriate interaction element, and the COG global and task local states are updated according to the interaction element’s frame condition and postcondition assertions. After that, another task can take turn processing incoming messages. The task ends when the outgoing messages the COG outputs include a resolve message. The histories generated from S are well-formed because the outgoing messages and the state of the COG are based on the information found within the incoming messages.

As an example, we describe the card payment task of the cash desk PC component from the Trading System case study presented in Deliverable 5.1 [66]. When a client pays the goods using a card and enters the pin, the PC forwards the request for validating a card to the bank. If the validation is not successful, a notification is shown on the cash box. Otherwise, the PC attempts to proceed with the payment. Should the payment be unsuccessful (e.g., because the account does not have enough money), then the cause is shown on the cash box. Otherwise, the cash box gives a success sale notification, the inventory stock of the store is updated, and the bill for the customer is printed. In a way, the cash desk PC resembles a subject of the observer pattern [82], whereas the cash box, inventory stock, and the bill printer take the role of observers.

```java
interface CardEventReceiver {
    Unit sendCreditInfoAndPin(CreditInfo creditInfo, Int pin);
}

interface CashDeskPC extends CardEventReceiver, ... {
}

interface Bank {
    Maybe<TransactionID> validateCard(CreditInfo creditInfo, Int pin);
    DebitResult debitCard(TransactionID transactionID, Money money);
}

interface CashBox {
    Unit show(String s);
}

interface Inventory {
    Unit accountSale(...);
}

interface Printer {
    Unit print(...);
}
```

Figure 4.9: Cash Desk PC Interfaces.

In Figure 4.9, the relevant interfaces and methods for describing the card payment task are shown. The ellipses hide details unnecessary for discussion. The cash desk PC interface offers to handle card events, while the bank
interface allows a card to be validated and used to pay certain amounts of money. The cash box, inventory and printer interfaces have relevant methods to accept notification from the cash desk PC.

The specification of the card payment task is shown in Figure 4.10. It consists of a COG global state declaration and a task specification payByCard. The global state consists of internal objects: pc, state, and amount, and references to external objects: cashBox, printer, inventory, and bank.

The task specification consists of the task local state declaration, which in this case used to memorize relevant future objects, and five interaction elements. The first interaction element accepts an incoming call to use the card as payment method complete with the card information and the associated PIN. This request is forwarded to the bank. The call is accepted only if the cash desk PC is in the payment mode, and such an interaction causes the future object information to be stored in the local task state. After this interaction is made, the control is given up to other tasks within the COG. The second and third interaction elements distinguish the result of the card validation process. The underscore denotes that we do not care about the value for that part of the message. The third interaction element, which describes a successful validation, emits another request to the bank to debit the account for a certain amount of money. The fromJust function is taken from the data type Maybe (as defined in the ABS Standard Library in Appendix A), which extracts the transaction ID from mtid. The fourth and fifth interaction elements deal with the result of debiting the account. In particular, the fifth interaction element shows the multiple outgoing messages which notify the cash box, the inventory stock, the printer of the successful purchase. This interaction element also resets the state of the cash desk PC back to the initial state.

Linking the interaction elements together produces possible traces which are subhistories of the COG’s histories. For example, the first, third and fifth interaction elements illustrate the high-level protocol of a successful card payment between the cash desk PC and its environment. A subset of histories generated by the complete specification include the following subhistory which states a successful card payment.

\[
\begin{align*}
& f: \text{pc!sendCreditInfoAndPin(ci, pin)}, \text{ftid: bank!validateCard(ci, pin)}, \text{ftid: resolve mtid}, \\
& \text{fdr: bank!debitCard(fromJust(mtid), amount)}, \text{fdr: resolve dr, } \_\text{: cashBox!show(toString(dr))}, \\
& \_\text{: inventory!accountSale(_)}, \_\text{: printer!print(_)}, \_\text{: cashBox!show("Sale succeeded..."), f: resolve Unit}
\end{align*}
\]

The example above shows how interaction patterns can be applied to describe the behavior of a COG. This specification technique describes each task a COG needs to handle in a stepwise manner using the interaction elements and the global model state of the COG. The technique itself still undergoes development to address some limitations related to the future objects. More precisely, as noted in Section 4.1 that the value of a future object may have been resolved prior to its reference being passed on to the COG via the method parameter of an asynchronous call or the result of a resolve message. As this resolution may occur arbitrarily before the future object reference is obtained by the COG, the generated subhistories are subsequences of the allowed histories of the COG, only when the resolve messages are filtered out from the subhistories.

The other problem is the incapability of the presented technique to specify a forced synchronization with the use of get as illustrated in the following code snippet.

```plaintext
...  
f1 = a!m();  
x = f1.get();  
f2 = a!n(x);  
y = f2.get();  
...
```

In the code snippet above, two synchronizations between this COG and the COG containing the object referred to by a are forced. The described interaction pattern technique is not enough to describe these forced synchronizations, as the incoming messages of an interaction element is received before the outgoing messages are dispatched. One way to handle this problem is by giving the messages extra labels within the sets of incoming and outgoing messages to denote the expected ordering, but we are keen to explore other possibilities which does not involve extra labelings.
4.3 Related Work

The message specification framework given in this chapter follows closely the work by Dovland, Johnsen & Owe [72], where they describe the message framework for the Creol language. The basis of a component there is active objects, whereas here we consider COGs. Furthermore, the work does not consider the use of futures.

The framework presented in this chapter bears similarities to component architectural specification frameworks, such as SOFA [147], Acme [83], Rapide [124], and Wright [13]: the behavior of a component is described through possible messages extracted from the interfaces a component provides and requires, and the behavior protocol based on the messages. The main differences are that there the notion of a component is defined statically and the data flow issue is rarely addressed, as the component configuration is the focus. Furthermore, communication is done via ports, rather than objects, such that when the data flow is considered, the user must encode the data in primitive data types, such as integers and strings, and later on let the program logic handle it.
spec CashDeskSpec {
  in interface CashDeskPC;
  out interface Bank, CashBox, Inventory, Printer;

  CashDeskPC pc = null;
  CashDeskPCState state = null;
  Money amount = null;
  CashBox cashBox = null;
  Printer printer = null;
  Inventory inventory = null;
  Bank bank = null;

  task payByCard {
    Fut<Maybe<TransactionID>> ftid = null;
    Fut<DebitResult> fdr = null;
    Fut<Unit> f = null;

    in fut: pc!sendCreditInfoAndPin(ci, pin)
    out fut2: bank!validateCard(ci, pin)
    requires state == PAYING
    modifies f, ftid
    ensures f == fut ∧ ftid == fut2

    in ftid: resolve Nothing
    out _: cashBox!show("ID_NOT_VALID"),
        f: resolve Unit
    requires state == PAYING

    in ftid: resolve mtid
    out fut: bank!debitCard(fromJust(mtid), amount)
    requires mtid == Nothing ∧ state == PAYING
    modifies fdr
    ensures fdr == fut

    in fdr: resolve dr
    out _: cashBox!show(toString(dr)),
        f: resolve Unit
    requires dr != DEBIT_OK ∧ state == PAYING

    in fdr: resolve dr
    out _: cashBox!show(toString(dr)),
        _: inventory!accountSale(_),
        _: printer!print(_),
        _: cashBox!show("Sale succeeded...")
        f: resolve Unit
    requires dr == DEBIT_OK
    modifies state
    ensures state == INIT
  }
}
Chapter 5

Variability Modeling

Software product line engineering (SPLE) [148] addresses the development of complex systems with a high degree of variability. A software product line is ultimately regarded as a set of software products that share a number of core properties, and differ on other aspects. These properties or aspects are the smallest unit of variability, and are known as features. At the highest level of abstraction, variability is expressed in terms of a feature model, usually depicted using feature diagrams [111, 30]. Figure 5.1 illustrates the workflow in the development of software using an SPLE approach. Given a feature model representing valid combinations of features and a set of software artifacts associated to these features, the final product is built by selecting the desired features and combining the corresponding artifacts.

![Feature Model Diagram](image)

Figure 5.1: Stages of Product Line Development.

This chapter explains the facilities provided by the full ABS framework to support the development of software product lines. Section 5.1 shows how feature models are specified using μTVL, a text-based feature modeling language based on TVL [39, 53]. Variable code artifacts are specified using delta modules which derive from delta-oriented product line development [158, 154, 155, 157, 156], as described in Section 5.2. In delta-oriented product line development, a product is described by a core (initial) code-base, together with a sequence of transformations to this core (additions, removals, or modifications). Section 5.3 describes how feature model and code artifacts are connected using configurations, which associate features to deltas. Section 5.4 show how the individual products of a product line are defined based on feature selections, and how they are obtained from or checked for inclusion in feature models. Finally, the full process of generating a software product from a software product line specification using our framework is explained in Section 5.5. Related work is presented in Section 5.6.

Remark 5.0.1. In contrast to the semantics for the core ABS language, which is a rewriting logic-style operational semantics, the semantics for the variability modeling language fragments is more model-theoretic in style. This approach was chosen primarily in order to directly provide the most straightforward semantics. It also reflects the implementation in the current version of the compiler, which is based on flattening delta modules. Thus the present semantics of delta modules is as a translation from core ABS models to core ABS models.
An alternative variant of the compiler that preserves the delta modules in the Maude interpreter has also been implemented. It maintains the collection of all delta modules at run-time and enables run-time modification of the set of applicable deltas, thereby facilitating dynamic evolution. This run-time could serve as the basis of a more uniform semantics. It is however significantly more complex and still being experimented with.

5.1 Feature modeling

This section introduces the µTVL text-based feature modeling language, pronounced either ‘micro textual variability language’ or simply ‘mu tee vee ell’, an extended subset of TVL [39, 53]. TVL was developed at the University of Namur, Belgium to serve as a reference language for specifying feature models. It is textual as opposed to diagrammatic, and aims to be scalable, concise, modular, and comprehensive, and thus serves as a suitable starting point for our purposes. A feature model is represented textually as a tree of nested features, each with a collection of boolean or integer attributes. Additional cross-tree dependencies can also be expressed in the feature model.

µTVL is designed to be deliberately smaller than TVL in order to capture the essential feature modeling requirements and to simplify the manipulation of feature models. The simplification also meant that a number of semantic constraints imposed by TVL can be reduced to syntactic constraints. µTVL enables a feature model with multiple roots (hence, multiple trees) to express orthogonal variability [148], which is useful for expressing application models or simply language roots (hence, multiple trees) to express orthogonal variability [148], which is useful for expressing application models.

µTVL features can only be extended (in FeatureExtension clauses) by adding new constraints, but not by introducing new features. Note that even though TVL syntax is used (with a few variations), the entire tool-set for µTVL has been developed from scratch to integrate with the ABS language tool-set environment.

We start by describing the syntax of µTVL, followed by its semantics, together with a small running example.

5.1.1 Concrete Syntax

The grammar of µTVL is given in Figure 5.2. Assume the presence of two global sets: FID of feature names and AID of attribute names.

```
Model ::=(root FeatureDecl)* FeatureExtension*
FeatureDecl ::=FID [(|Group| AttributeDecl* Constraint* )]
FeatureExtension ::= extension FID { AttributeDecl* Constraint* }
Group ::=group Cardinality { [opt] FeatureDecl, ([opt] FeatureDecl)* }
Cardinality ::=allof | onef | \[ n1 .. n2 | [n1 .. 2] | Group}
AttributeDecl ::=Int AID ; | Int AID in\[ Limit .. Limit ) ; | Bool AID ;
Limit ::= n | +
Constraint ::=Expr ; | ifin: Expr ; | ifout: Expr ; | require: FID ; | exclude: FID ;
Expr ::=true | false | n | FID | AID | FID.AID | UnOp Expr | Expr BinOp Expr | ( Expr )
UnOp ::= ! | -
BinOp ::= | | | & | | > | < | <= | >= | >= | + | - | * | / |
```

Figure 5.2: Grammar of µTVL; n ranges over Integers.

Attributes and values in µTVL range either over integers or over booleans.

The Model clause specifies a number of “orthogonal” root feature models along with a number of extensions which specify additional constraints, typically cross-tree dependencies. A feature model may be separated into different files. The FeatureDecl clause specifies the details of a given feature, firstly by giving it a name (FID), followed by a specification of any sub-features, the feature’s attributes and any relevant constraints. The FeatureExtension clause
specifies additional constraints and attributes for a feature. This is particularly useful for specifying constraints that do not fit into the tree structure given by the root feature model. The Group clause specifies the sub-features of a feature. This consists of a specification of the cardinality of the group, plus a number of possibly optional sub-features. The Cardinality clause describes the number of elements of a group that may appear in a result, where all means that all elements of the group must appear, and one means that one element must appear. \([n_1 \ldots n_2]\) specify the range of values on the number of elements of the group. These can be bounded below and above or unbounded above (\(\ast\)). The AttributeDecl clause specifies both integer (bounded or unbounded) and boolean attributes of features. The Limit clause is used to specify the bounds, where \(n\) is some integer and \(\ast\) indicates that that attribute is unbounded below and/or above.

The Constraint clause specifies constraints on the presence of features and on attributes. An ifin constraint is only applicable when the current feature is selected. Similarly, an ifout constraint is only applicable when the current feature is not selected. An include clause specifies that the current feature requires some other feature, whereas exclude expresses the mutual incompatibility between the current feature and some other feature. The Expr clause ultimately expresses a boolean constraint over the presence of features and attribute values. Features are referred to by identity (FID). Attributes are referred to either using an unqualified name (AID), for in scope attributes, or using a qualified name (FID.AID) for attributes of other features. Unary operators UnOp are logical negation (!) and integer negation (-). Binary operators BinOP are logical or (||), logical and (&&), logical implication (\(\rightarrow\)), logical equivalence (\(\leftrightarrow\)), equality on integer and boolean attributes and values (==), inequality (!=), greater than, less than, greater than or equal to and less than or equal to on integers (>, <, >=, <=), and plus, minus, times, div and mod on integers (+, -, *, /, %).

For a feature model to be valid the following syntactic and semantic constraints must hold:

- arguments to binary and unary operators type check in the standard way;
- attributes must be declared before being used;
- feature names must be unique, that is, each feature can be declared only once;
- attribute names are unique per feature, meaning that one cannot, for example, have English.time twice even at different types, even though A.att and B.att may certainly coexist; and
- zero or one instances of each feature can be present in the ultimate model—this means that cardinality specifies not that features can be multiply instantiated, rather it specifies the number of selections that can be made for a choice.

Example 5.1.1. The following is a feature model of the MultiLingualHelloWorld product line, which describes software that outputs “Hello World” in multiple languages some number of times.

```plaintext
root MultiLingualHelloWorld {
  group allof {
    Language {
      group oneof { English, Dutch, French, German }
    },
    opt Repeat {
      Int times in [0..1000];
      times > 0;
    }
  }
}
extension English {
  ifin: Repeat ->
    (Repeat.times >= 2 && Repeat.times <= 5);
}
```
The MultiLingualHelloWorld product line in the example above has two main features, Language and Repeat, under the root feature and joined with the allof combinator. The Language feature requires one out of four possible features: English, Dutch, French, or German. The Repeat feature has no associated group of features, and it has an attribute times which ranges between 0 and 1000, with an added condition that it must be strictly greater than 0 (for illustration purposes). Extensions are constraints that can be added to existing features. In this example an extension for the English feature is given. When the English and the Repeat features are present, the attribute times must be between 2 and 5, inclusive.

This feature model can be depicted as a feature diagram using standard notations [59], as shown in Figure 5.3. While µTVL is text-based, such feature diagrams can easily be generated; and generating µTVL descriptions of feature diagrams should also be straightforward.

Recall the example of a peer-to-peer file sharing system, introduced in Section 3.10, and explained in detail in Appendix E. Each node in the peer-to-peer system can be extended with two extra functionalities: a node can prevent the transfer of undesired files, and, more specifically, of content not suitable for younger people.

Example 5.1.2. Below is a feature model of the Peer2Peer product line, which describes variations of the functionality of nodes in a peer-to-peer system. Nodes can be basic nodes, extended with a general file filter, or extended with a parental control filter. The parental control filter feature requires the general file filter feature.

```plaintext
root Peer2Peer {
  group oneof {
    Basic,
    Extended {
      group [0..2] {
        FileFilter,
        ParentalControl { require: FileFilter; }
      }
    }
  }
}
```

5.1.2 Abstract Syntax

The abstract syntax for µTVL programs is presented in Figure 5.4, where $f \in FID$, $a \in AID$, and $n \in Int$. The translation from concrete to abstract syntax is straightforward, and hence omitted. (See implementation for details.)
Local attribute names are expanded to fully qualified names. Bounds are placed on all integer attributes. Feature extensions are treated the same way as features, so these are unified in the abstract syntax. The semantics of \(\mu\)TVL is given by the solutions of the integer constraints derived by feature models from this syntax tree, presented in the next subsection.

### 5.1.3 Semantics

The semantics of a feature model in \(\mu\)TVL is defined by translation into constraints over integers whose solutions correspond to valid feature and attribute selections. Boolean variables are treated as integers in the standard manner: 0 corresponds to false, and 1 to true. The function \(\llbracket \cdot \rrbracket\) encoding feature model \(M\) as an integer constraint is given in Figure 5.5. Within the context of a given feature \(f\), function \(\llbracket \cdot \rrbracket_f\) translates constraints relative to that feature.

\[
\begin{align*}
M & := F^* & \text{feature model} \\
F & := f [G] A^* C^* & \text{feature (extension)} \\
G & := c N^* & \text{group} \\
N & := \text{opt } F | \text{mand } F & \text{feature node} \\
c & := \text{alof } \min n | \text{rng } n n & \text{cardinality} \\
A & := f.a T & \text{attribute declaration} \\
T & := \text{bool } \text{int } L \ L & \text{type and domain} \\
L & := * | n & \text{domain limit}
\end{align*}
\]

\[
\begin{align*}
\llbracket M \rrbracket & = \llbracket T \rrbracket \\
\llbracket f [G] A^* C^* \rrbracket & = (0 \leq f \leq 1) \land \llbracket G \rrbracket_f \land \llbracket A \rrbracket_f \land \llbracket C \rrbracket_f \\
\llbracket c N^* \rrbracket & = \text{tree}(f, N) \land \sum N = \#N \land \llbracket N \rrbracket \\
\llbracket \text{alof } N \rrbracket_f & = \text{tree}(f, N) \land n \leq \sum N \land \llbracket N \rrbracket \\
\llbracket \text{rng } n_1 n_2 \rrbracket_f & = \text{tree}(f, N) \land n_1 \leq \sum N \leq n_2 \land \llbracket N \rrbracket \\
\llbracket \text{opt } (f [G] A C) \rrbracket & = f \rightarrow f^\dagger \land \llbracket f [G] A C \rrbracket \\
\llbracket \text{mand } F \rrbracket & = \llbracket F \rrbracket \\
\llbracket f.a \text{ int } L_1 L_2 \rrbracket & = \text{val}_\text{min}(L_1) \leq f.a \leq \text{val}_\text{max}(L_2) \\
\llbracket f.a \text{ bool } \rrbracket & = 0 \leq f.a \leq 1 \\
\llbracket \text{ifin } e \rrbracket_f & = f \rightarrow [e] \\
\llbracket \text{ifout } e \rrbracket_f & = \neg f \rightarrow [e] \\
\llbracket \text{require } f' \rrbracket_f & = f \rightarrow f' \\
\llbracket \text{exclude } f' \rrbracket_f & = \neg (f \land f') \\
\llbracket e \rrbracket & = \phi_e \\
\llbracket \text{opt } (f _-_ _) \rrbracket & = f^\dagger \\
\llbracket \text{mand } (f _-_ _) \rrbracket & = f \\
\llbracket \text{val}_i(n) \rrbracket & = n \\
\llbracket \text{val}_\text{min}(*) \rrbracket & = \text{MIN} \\
\llbracket \text{val}_\text{max}(*) \rrbracket & = \text{MAX}
\end{align*}
\]

Figure 5.5: Semantics of \(\mu\)TVL.

In the translation, \(f^\dagger\) is a unique name based on name \(f\). It is used when dealing with cardinalities involving optional features. In such cases \(f^\dagger\) can freely be set to 1 to count the optional feature, even when it is absent. For example, when dealing with an allof constraint, it is required that all children are present; some may however be
optional, so as far as the allOf constraint is concerned, optional children are counted, though the corresponding features may not be included. Expressions e are encoded into constraints, denoted $\phi_e$. Their encoding is straightforward and therefore omitted (see [53]). Boolean operations are mapped to a conjunctive set of integer operations over the values 0 and 1 where, for example, $a \rightarrow b$ becomes $a \leq b$. Finally, we assume a lower bound $MIN$ and an upper bound $MAX$ on the values of integer variables.

Given a feature model $FM$ in $\mu$TVL, the set of solutions of the integer constraints $[FM]$ provides semantics for $FM$. Such a solution will specify values for all attributes even when the corresponding feature is not selected. Such assignments should have no effect.

The semantics also enforces that each feature is selected either zero or one times, in spite of cardinality conditions which may appear to allow more instances of a feature. Cardinality conditions specify the number of selected subfeatures from a group. Note that optional features can only appear under the allOf cardinality; otherwise there would be a fragile interaction between cardinality conditions and optional features [32].

The semantics of feature models are illustrated below using our running example MultiLingualHelloWorld.

**Example 5.1.3.** Below is the encoding into integer constraints of the Hello World feature model introduced in Example 5.1.1.

$$
0 \leq \text{MultiLingualHelloWorld} \leq 1 \land \\
\text{Language} \rightarrow \text{MultiLingualHelloWorld} \land \text{Repeat}^\dagger \rightarrow \text{MultiLingualHelloWorld} \land \\
\text{Language} + \text{Repeat}^\dagger = 2 \land \\
0 \leq \text{Language} \leq 1 \land \\
\text{English} \rightarrow \text{Language} \land \text{Dutch} \rightarrow \text{Language} \land \text{French} \rightarrow \text{Language} \land \text{German} \rightarrow \text{Language} \land \\
1 \leq \text{English} + \text{Dutch} + \text{French} + \text{German} \leq 1 \land \\
0 \leq \text{English} \leq 1 \land 0 \leq \text{Dutch} \leq 1 \land 0 \leq \text{French} \leq 1 \land 0 \leq \text{German} \leq 1 \land \\
0 \leq \text{Repeat}^\dagger \leq 1 \land \\
\text{Repeat} \rightarrow \text{Repeat}^\dagger \land \\
0 \leq \text{Repeat} \leq 1 \land 0 \leq \text{Repeat.times} \leq 1000 \land \text{Repeat.times} > 0 \land \\
\text{English} \rightarrow (\text{Repeat} \rightarrow (\text{Repeat.times} \geq 2 \land \text{Repeat.times} \leq 5)).
$$

Every declaration of a new feature or an attribute $x$ is converted into a constraint of type $min \leq x \leq max$, and in the case of booleans and feature names $min = 0$ and $max = 1$. The tree structure of the feature model is captured by implications between the children and their parents, as shown in the second line of Example 5.1.3. The optional feature Repeat is split into two variables: Repeat and Repeat$^\dagger$. The latter is used only to address the cardinality of the parent MultiLingualHelloWorld, and they are connected by the implication Repeat $\rightarrow$ Repeat$^\dagger$, similar to how child features are related to their parent. Cardinalities are encoded as constraints that add the 0-1-integer value of the feature variables and check whether they belong to a specific domain, as shown in the third and seventh line of our example. Constraints over attributes are simply interpreted as integer constraints.

**Remark 5.1.4.** The semantics of $\mu$TVL differs little in essence from the semantics of TVL presented in the literature [39, 33]. The main difference is that our translation is uniformly into integer constraints, which are defined inductively on the abstract syntax of $\mu$TVL models. In contrast the abstract syntax of TVL is presented as a collection of operations for extracting information out of the parse tree expressed as a graph.

### 5.2 Delta Modeling

Delta-oriented programming was introduced by Schaefer et al. [155, 157, 156] as a novel programming language approach for developing software-based product lines, and as a direct alternative to feature-oriented programming [29]. Both approaches aim at automatically generating software products for a given valid collection of features, providing flexible and modular techniques to build different products that share functionality or code. In feature-oriented programming software modules are associated to features, and building a product consists of combining the modules for
a feature selection. In delta-oriented programming [155], application conditions, conditions over the set of features and their attributes, are associated with modules of program modifications (add, remove, or modify code), called delta modules. The collection of applicable delta modules is given by the application conditions that are true for a particular feature and attribute selection. By not associating the delta modules directly with features, a degree of flexibility is obtained, resulting in better reuse of code and the ability to resolve conflicts caused by deltas modifying the code base in incompatible ways [51]. The flexibility offers benefits in the management of the evolution of product lines, by allowing versions to be implemented using software deltas.

The implementation of a software product line in delta-oriented programming [155] is divided into a core model and a set of delta modules (or deltas). The core model consists of the classes that implement a complete product of the corresponding product line. Delta modules describe how to change the core model to obtain new products, as depicted in Figure 5.6. The choice of which delta modules to apply is based on the selection of desired features for the final product. Schaefer et al. described and implemented delta-oriented programming for Java [155], introducing the programming language DELTJAVA. This language has strongly influenced our design, though we further separate deltas from features by moving application conditions out of deltas and into a product line configuration language, as pursued in [157, 156]. Delta modeling is included in the ABS language to implement variability at the source code level of abstraction.

5.2.1 Syntax

Figure 5.7 specifies the ABS syntax related to delta modeling. Nonterminals written in purple (gray) refer to core ABS symbols defined in Chapter 3.

The DeltaDecl clause specifies the syntax of delta modules, which consists of an unique identifier, a list of parameters, and a body containing a sequence of class and interface modifiers. The ClassOrInterfaceModifier clause describes the syntax of modifications at the level of classes and interfaces. Such a modification can add a class or interface declaration, modify an existing class or interface, or remove a class or interface. The ImplementsModifiers clause describes how to modify the interfaces a class implements or an interface extends, either by adding new or removing existing interfaces.

The Modifier clause specifies the modifications that can occur within a class or interface body. These include (where relevant) adding and removing fields and method signatures (from interfaces), and modifying methods, which amounts to replacing a method with a new one, but enabling the original method to be called using the original keyword.

In contrast to delta modules presented in the literature [155-157], delta modules in the HATS ABS language can be parameterised both by attribute values, which ultimately flow from the feature model selection, and by class names, to enable the application of a single delta in more than one circumstance. The pair Type Identifier is a core ABS clause corresponding to the syntax of such parameters, which are regular type-parameter name pairs. Finally, the HasCondition describes constraints on class arguments to which a delta may be applied. These constraints consist of descriptions of the methods and fields such a class implements and any interfaces it is expected to have.

Remark 5.2.1. At present only a subset of the previous language constructs have been implemented in the tool-set. There is not yet support for delta modules parameterised over classes. In addition, delta modules are not type checked. This challenging issue is being addressed in Task 2.4 “Types for Variability”.

Example 5.2.2. Following is the ABS Implementation of the Hello World product line with the feature model shown in Example 5.1.1. Delta modules are used to specify variable behaviour.
interface Greeting {
    String sayHello();
}

class Greeter implements Greeting {
    String sayHello() {
        return "Hello world";
    }
}
class Application {
    Unit run() {
        Greeting bob;
        bob = new Greeter();
        String s = "";
        s = bob.sayHello();
    }
}
delta De {
    modifies class Greeter {
        modifies String sayHello() {
            return "Hallo welt";
        }
    }
}
delta Nl {
    modifies class Greeter {

Figure 5.7: ABS Grammar: Delta Modules.
The example above has seven blocks of code. The first three—the interface Greeting and the classes Greeter and Application—form the core module of the ABS implementation, and are written in the core ABS language. The other four blocks correspond to different delta modules: De, Nl, Fr, and Rpt. The delta module De has a single class modifier for Greeter, which in turn has a single method modifier. This method modifier replaces the method sayHello to return the German text “Hallo welt”. Delta modules Nl and Fr are similar. The delta module Rpt has a single parameter for the number of times that the hello string should be repeated. As the previous delta modules, it replaces the method sayHello() inside Greeter with new ABS code. However, the method being replaced remains accessible via the special method call original().

Example 5.2.3. Recall the modules in Example 5.2.2. The diagram below exemplifies the result of applying the delta module De to the core ABS in this example. The single modifier fragment replaces the method sayHello() with a different body.

Example 5.2.4. The following diagram illustrates both the use of parameters and of the original keyword. A parameterised delta, such as Rpt, must have its arguments for its specified before it can be applied. The arguments
are substituted into the body of the delta module prior to application. The aim of original is to enable the method being replaced to be called from the delta module that replaces it. This is implemented by renaming the original method, and replacing the call via keyword original with a call to the renamed method.

```java
class Greeter {
    String sayHello() {
        return "Hello";
    }
}
```

The semantics of calling original() as shown in the above example are essentially the same as Super() from feature-oriented programming [29], and proceed from context-oriented programming [98], and similar to ordinary super calls in standard object-oriented languages, as well as the around advice from aspect-oriented programming [116], except without quantification.

**Example 5.2.5.** The following code fragment demonstrates a delta module parameterised by both a class argument and a delta argument.

```java
delta Gen(Cl hasMethod String sayHello(), String hello) {
    modifies Cl {
        modifies String sayHello() {
            return hello;
        }
    }
}
```

The delta module Gen is a generalisation of De, NL, and Fr and illustrates the use of delta module (class) parameters. The first parameter binds a class that has the method sayHello() to the class variable Cl, and the second parameter binds a string to the variable hello. The rest of the delta module body is interpreted as expected.

The following diagram illustrates the application of the parameterised delta module Gen, with arguments Greeter and "Hallo welt", which has the same effect as delta module De.

```java
interface Greeting {
    String sayHello();
}
class Greeter implements Greeting {
    String sayHello() {
        return "Hello world";
    }
}
```

Note that string parameters to delta modules are not implemented in the compiler, but can be added without any difficulty.

Recall the peer-to-peer system used as example in Section 3.10 and in Section 5.1.1. The full details of this example are presented in Appendix E.
Example 5.2.6. The core product of the peer-to-peer system was defined without any file filter functionality (see Figure E.1). File filter and parental control functionality can be added using the following delta modules.

```abs
delta DFileFilter()
  modifies class Node {
    modifies Server findServer(Filename fId, Catalog catalog) {
      if (isValid(fId)) { original(fId, catalog); } 
      else { return null; }
    }
    adds Bool isValid(Filename fId) {
      return True;
    }
  }

delta DParentalControl() {
  modifies class Node {
    modifies Bool isValid(Filename fId) {
      return (~isSubstring("xxx", fId));
    }
  }
}
```

The delta DFileFilter modifies the findServer method within the Node class by adding a method isValid, which is used to validate a filename before requesting the search for a server. The DParentalControl delta simply overwrites the isValid method with a new one that accepts only filenames not containing the string “xxx”.

5.2.2 Formal Semantics

Given a core ABS program $P$ and a delta module $\Delta$, a new core ABS program $P_{\Delta}$ can be constructed by applying $\Delta$ to $P$. Thus the construction of a product from the core application and a collection of delta modules is achieved by successively applying each delta module in turn. This section presents a formal semantics of delta modules based on the more abstract presentation of Clarke et al. [51]. That work also describes the composition of delta modules with each other, which is essential for reasoning about conflicting delta modules, but this feature is elided from the current presentation. ABS programs, classes and delta modules will be represented in terms of finite maps from identifiers to the corresponding contents of the program, class, or delta module, in order to more cleanly present the semantics. The semantics only describes the modifications of methods; dealing with fields and so forth is a straightforward extension. Parameters are omitted. These will be treated when dealing with configurations in Section 5.3.

Let $\text{Identifier}$ be the set of identifiers, let $\text{MethBody}$ be the set of method bodies, including the parameter and return types, and let $\text{MethBodyWrap}$ be the set of method bodies with an explicit call to $\text{original}$. In the following domains, Replace, Update, and Remove are used to tag the various branches of sum data types. Finally, let $\text{Error}$ denote that an error has occurred. Errors occur if one attempts to wrap a method that is not present. Other irregularities, such as attempting to update a class that is not present, can be given a sensible semantics, so long as no wrapping occurs.
Thus a program is a map from class names to classes, which themselves are collections of named method bodies. A delta module is a map from class names to delta bodies, which consist of three different types of modification: Replace either adds or replaces the class with the specified contents; Update modifies a class in place, where the three elements within an update clause correspond to replacing a method with a new body from \( \text{MethBody} \), wrapping the method with a body from \( \text{MethBodyWrap} \) or removing the method; and finally, Remove denotes the removal of the class.

\[ \text{Program} = \text{Identifier} \to \text{ClassBody} \]
\[ \text{ClassBody} = \text{Identifier} \to \text{MethBody} \]
\[ \text{Delta} = \text{Identifier} \to \text{DeltaBody} \]
\[ \text{DeltaBody} = \text{Replace} (\text{Identifier} \to \text{MethBody}) \]
\[ \cup \text{ Update} (\text{Identifier} \to (\text{MethBody} \cup \text{MethBodyWrap} \cup \text{Remove})) \]
\[ \cup \text{ Remove} \]

**Notation 5.2.7.** Let \( f : X \to Y \) denote a partial function from \( X \) to \( Y \). If \( f(x) \) is undefined for \( x \in X \), write \( f(x) = \bot \), where \( \bot \notin Y \). For set \( A \), let \( A_\bot \) denote \( A \cup \{ \bot \} \), where \( \bot \notin A \). We freely shift between partial functions \( X \to Y \) and functions \( X \to Y_\bot \), define the lifting of \( \circ \) to partial functions over index set \( I \) as

\[
\circ : (I \to A) \times (I \to B) \to (I \to C)
\]
\[
(f \circ g)(i) = f(i) \circ g(i), \quad \text{where } i \in I
\]

Given a class update \( f : \text{Identifier} \to (\text{MethBody} \cup \text{MethBodyWrap} \cup \text{Remove}) \), we define the function \( f^* : \text{Identifier} \to (\text{MethBody} \cup \text{Error}) \) as follows. For \( i \in \text{Identifier} :\]

\[
f^*(i) = \begin{cases} 
\bot & \text{if } f(i) = \text{Remove} \\
 f(i) & \text{if } f(i) \in \text{MethBody} \\
 \text{Error} & \text{if } f(i) \in \text{MethBodyWrap}.
\end{cases}
\]

**Notation 5.2.8.** In the following definition, the notation \( w[ \ ] \) denotes a wrapper method from \( \text{MethBodyWrap} \), where the hole \( [ \ ] \) denotes that the original method is unknown. Notation \( w[b] \) denotes the wrapping of method body \( b \) with wrapper \( w \), thus the original call can be successfully bound. The resulting method \( w[b] \) is considered to be an element of \( \text{MethBody} \).

**Definition 5.2.9** (Delta module application). The application of a delta module to a program is specified by the following functions:

\[
\text{apply} : \text{Delta} \times \text{Program} \to \text{Program}
\]

\[
\text{apply}(d, p) = d \circ_c p
\]

where

\[
\circ_c : \text{DeltaBody}_\bot \times \text{ClassBody}_\bot \to \text{ClassBody}_\bot
\]

\[
\bot \circ_c x = x
\]
\[
\text{Remove} \circ_c \bot = \bot
\]
\[
(\text{Update } f) \circ_c \bot = f^*
\]
\[
(\text{Update } f) \circ_c h = f \circ_m h
\]
\[
(\text{Replace } g) \circ_c \bot = g
\]

65
and

\(⊙_m : (\text{MethBody} ⊔ \text{MethBodyWrap} ⊔ \text{Remove})⊥ × \text{MethBody}_⊥ \rightarrow (\text{MethBody} ⊔ \text{Error})⊥\)

\[\downarrow ⊕_m x = x\]

\(\text{Remove} ⊕_m _x = \bot\)

\(m ⊕_m _x = m, \quad \text{where } m ∈ \text{MethBody}\)

\(w[ ] ⊕_m _z = \text{Error}, \quad \text{where } w[ ] ∈ \text{MethBodyWrap}\)

\(w[ ] ⊕_m b = w[b], \quad \text{where } w[ ] ∈ \text{MethBodyWrap}\).

**Remark 5.2.10.** An alternative semantics would define \((\text{Update } f) ⊕_c _x \rightarrow \text{Error}\), capturing the attempt to update a class that it not present.

**Notation 5.2.11.** If \(m ∈ \text{Identifier}\) then \(w[m]\) denotes the wrapper with each call to \(\text{original}\) replaced by a call to \(m\).

In the implementation the method body is not inlined, as the process above suggests. Instead, if the resulting class \(C\) has an element \(m ↦→ w[b] ∈ C\), the following post-processing steps should be performed before applying another delta module:

1. generate a fresh method name \(m′ ≠ C\),
2. remove \(m ↦→ w[b]\) from \(C\), and
3. add \(m ↦→ w[m′]\) and \(m′ ↦→ b\) to \(C\).

Example 5.2.4 illustrated this process with concrete code. The modified method in that example is \(\text{sayHello()}\). Before replacing the method, it was renamed to a fresh name such as \(\text{original\_sayHello()}\). The new method was then added to the class, with its body modified so that \(\text{original}\) is replaced by the renamed method’s name \(\text{original\_sayHello}\). The stipulation that the method name is fresh is required in the case that multiple delta modules are applied to the same class, each wrapping the same method. In such a case, the first renaming would result in method \(\text{original\_sayHello}\), for example, the second in name \(\text{original\_sayHello2}\), and so forth. In addition to being fresh, the name should also be obfuscated to avoid polluting the space of sensible method names, and thus avoid it accidentally being called or overridden.

### 5.3 Product Line Configuration

This section describes the product line configuration language (CL) which links feature models specified in \(\mu\)TVL (Section 5.1) with delta modules (Section 5.2), as indicated in Figure 5.8, to provide a specification of the variability in a product line. This approach is similar to the product line specification proposed in delta-oriented programming [157, 156].

A product line configuration consists of a set of features assumed to exist, a set of core features, and a set of *delta clauses*. Each delta clause specifies a delta and the conditions required for its application, called *application conditions*. These conditions contain (1) propositional formulas over the set of known features and attributes, and (2) a partial ordering relation with respect to other deltas. When the propositional formula holds for a given product, the delta is said to be active. The partial ordering is given by indicating which deltas, when active, should be applied before it.

#### 5.3.1 Syntax

The syntax of the product line configuration language is given in Figure 5.9.

The *Configuration* clause specifies the name of the product line, the set of features it implements, and the set of delta modules used to implement those features. The feature names are included so that certain simple self-consistency checks can be performed. The *Features* clause describes which features the product line refers to. The
Figure 5.8: Application of a Product Line Configuration.

Figure 5.9: Product Line Configuration Grammar.

$Configuration ::= productline TypeId \{ Features ; DeltaClauses \}$

$Features ::= features FID (, FID)*$

$DeltaClauses ::= DeltaClause (, DeltaClause)*$

$DeltaClause ::= delta DeltaSpec [AfterCondition] [ApplicationCondition] ;$

$DeltaSpec ::= TypeName [\{ DeltaParams \}]$

$DeltaParams ::= DeltaParam (, DeltaParam)*$

$DeltaParam ::= FID | FID.AID | PureExp$

$AfterCondition ::= after Identifier (, TypeName)*$

$ApplicationCondition ::= when Expr$

$DeltaClause$ clause is used to specify each delta module, linking it to the feature model. The $DeltaSpec$ clause names a specific delta module and, optionally, specifies the parameters passed, which are feature identifiers indicating if they are present in the feature model, attributes specified in the feature model, or a constant value specified in core ABS as a $PureExp$. The $AfterCondition$ clause specifies the delta modules that the current delta module must be applied after. The $ApplicationCondition$ clause specifies an arbitrary predicate (see Figure 5.2) describing when the given delta module is included in the product line. This condition can be in terms of the presence and absence of features and feature combinations, as well as attributes of features and integer and boolean constants.

Remark 5.3.1. At present only a subset of the application conditions are implemented: our tools accept only a conjunctive sequence of features. Also the only implemented parameters types are booleans and integers.

Example 5.3.2. The Hello World product line is configured, connecting the features and attributes defined in the feature model to code.

```plaintext
productline MultiLingualHelloWorld {
  features English, German, French, Dutch, Repeat;

  delta Rpt(Repeat.times) after De, Fr, Nl when Repeat;
  delta De when German;
  delta Fr when French;
  delta Nl when Dutch;
}
```

The example configuration above first names the set of features from the feature model shown in Example 5.1.1 which are used to configure this product line. The $delta$ directives link each delta module to the feature model through an application condition ($when$ clause); in this case, a delta module is applied simply when the specified feature is selected (e.g. “De when German”). Note that there is no delta module corresponding to the feature $English$, as the core provides support for the English language as a default. In addition, $Rpt$ is configured such that it has to be
applied after all the language-specific delta modules. Rpt’s argument is the value Repeat.times, namely, the times attribute of the Repeat feature; its value (defined by product selection, see Section 5.4) is propagated from the feature model down to the Rpt delta module.

A greater range of possibilities are expressible in CL. For example, the application conditions $A \land \neg B$, $\neg A \land B$, and $A \land B$ on separate delta modules $D_A$, $D_B$, and $D_{AB}$ express the variety of situations where these delta modules are applicable, depending upon which combination of features are selected. In this case, $D_A$ is applicable when feature $A$ is selected, but $B$ is not. Similarly for $D_B$. When both features $A$ and $B$ are selected, delta module $D_{AB}$ is instead used. If the application conditions were instead $A$, $B$, and $A \land B$, respectively, then delta module $D_{AB}$ plays the role of resolving any conflicts between $D_A$ and $D_B$ (assuming that $D_{AB}$ is applied after both $D_A$ and $D_B$).

Example 5.3.3. The two delta modules for the peer-to-peer system from Example 5.2.6 are connected to the feature model from Example 5.1.2 by the following product line configuration.

```plaintext
productline Peer2Peer {
    features Basic, FileFilter, ParentalControl;
    delta DFileFilter when FileFilter;
    delta DParentalControl after DFileFilter when ParentalControl;
}
```

5.3.2 Semantics

A CL product line configuration file specifies how the feature model relates to the delta modules that are to be applied to the core model. It does so by specifying the parameters and application conditions for each delta module, along with an ordering on the delta modules.

A delta module in a configuration file is modeled by the following type:

$$ \Delta \times \text{Params} \times \text{AppCondition} $$

where $\Delta$ is the semantic domain of delta module bodies, defined in Section 5.2.2,

$$ \text{Params} = \text{Identifier} \rightarrow \text{FID} \cup (\text{FID} \times \text{AID}) \cup \text{Int} $$

models the substitution of actual parameters, which may be attributes or constants, defined in the CL script, for the formal parameters of the corresponding delta module, and $\text{AppCondition}$ is the syntactic category of application conditions. Class parameters to delta modules are not modeled.

A configuration script can be modeled as a partial order over the declared delta modules (with their parameters and application conditions), where the partial order is determined by the reflexive, transitive closure of the after clauses. This is given by the following domain, where $\text{PO}(\cdot)$ denotes the collection of all partial orders over a given set:

$$ \text{Config} = \text{PO}(\Delta \times \text{Params} \times \text{AppCondition}) $$

The semantics of a configuration script $conf \in \text{Config}$ is a function of the type

$$ [\text{conf}]_\_ : \text{ProductSelection} \rightarrow \mathcal{P}(\Delta^*) $$

which maps a product selection—the interpretation of a PSL script (see Section 5.4.2)—to the delta modules to apply, in the order they should be applied. Not that many orders may exist if the after-order is underspecified. A product selection is an assignment from feature names to true or false (1 or 0) and from attributes to values, given by the domain ProductSelection:

$$ \text{ProductSelection} = (\text{FID} \cup (\text{FID} \times \text{AID})) \rightarrow \text{Int} $$

68
We now develop the ingredients making up function \([\text{conf}]_\cdot\).

Firstly, assume that a notion of substitution exists for delta modules, respecting the scoping of variables, to replace the parameters with appropriate values:

\[
\text{Subst} = \text{Identifier} \rightarrow \text{Int} \\
\text{apply} : \Delta \times \text{Subst} \rightarrow \Delta
\]

Next, we define the composition of the parameter specifications of delta modules with a product selection, giving a mapping from formal parameters of delta modules to values (Int), which will be used to refine the delta modules with the configuration parameters specifying in the product selection:

\[
\sigma \circ p = \{ v \mapsto x \sigma \mid v \mapsto x \in p \}
\]

where \(x \sigma = \{ v \mapsto x \mid v \in \text{FID} \cup (\text{FID} \times \text{AID}) \} \) and \(\vdash \in \sigma\) is \(\vdash\) restricted to \(D'\), and \(\vdash \subseteq \text{ProductSelection} \times \text{AppCondition}\) is the satisfaction relation.

Now the function taking a product selection \(\sigma \in \text{ProductSelection}\) and giving the collection of delta modules to apply is computed as the composition of the following steps:

1. Select applicable deltas:

\[
\text{select}_\sigma : \text{Config} \rightarrow \PO(\Delta \times \text{Params})
\]

\[
\text{select}_\sigma(D, \prec) = (D', \prec|_{D'})
\]

where \(D' = \{ (d, p) \mid (d, p, \phi) \in D, \sigma \models \phi \} \) and \(\prec|_{D'}\) is \(\prec\) restricted to \(D'\), and \(\models \subseteq \text{ProductSelection} \times \text{AppCondition}\) is the satisfaction relation.

2. Specialise deltas:

\[
\text{specialise}_\sigma : \PO(\Delta \times \text{Params}) \rightarrow \PO(\Delta)
\]

\[
\text{specialise}_\sigma(D, \prec) = (D', \prec|_{D'})
\]

where \(D' = \{ \text{apply}(d, \sigma \circ p) \mid (d, P) \in D \}\)

3. Order deltas

\[
\text{order} : \PO(\Delta) \rightarrow \mathcal{P}(\Delta^*)
\]

\[
\text{order}(D, \prec) = \{ [d_1, \ldots, d_n] \mid d_1, \ldots, d_n \text{ is a linear extension of } (D, \prec) \}.
\]

Finally, the semantics of a configuration language script can be interpreted as a function

\[
\llbracket \_ \rrbracket : \text{Config} \times \text{ProductSelection} \rightarrow \mathcal{P}(\Delta^*) \\
\llbracket \text{conf} \rrbracket_\sigma = \text{order}(\text{specialise}_\sigma(\text{select}_\sigma(\text{conf}))).
\]

**Remark 5.3.4.** Note that this process may be ambiguous when multiple orderings of delta modules are possible. This should be resolved either by adding more elements to the ‘after’ order, or by introducing conflict-resolving deltas [51]. Detecting conflicts automatically is being investigated within Task 2.4 (Types for Variability).

---

1In our tools the linear extension is calculated using topological sorting for a graph with edges given by the relation, before the transitive closure.
5.4 Product Selection

A product selection needs to be specified to generate a product from a product line. A product selection is specified using the product selection language (PSL), and specifies a product by stating which features are to be included in the product and by setting any attributes of those features with concrete values. In addition, some core ABS code is provided to initialise the selected product. As depicted in Figure 5.10, a product selection is checked against a μTVL feature model for validity. It is then used by the configuration file to guide the selection and application of deltas during the generation of the final software product. This section presents the syntax for product selections, gives its semantics, and defines satisfaction of product selections with respect to feature models.

5.4.1 Syntax

Figure 5.11 specifies the grammar of the ABS product selection language.

```
Selection ::= product Ty McD ( FeatureSpecs ) { InitBlock }
FeatureSpecs ::= FeatureSpec ( , FeatureSpec )*
FeatureSpec ::= FID [ AttributeAssignments ]
AttributeAssignments ::= { AttributeAssignment ( , AttributeAssignment )* }
AttributeAssignment ::= AID = Literal
InitBlock ::= Block
```

Figure 5.11: PSL Grammar

The Selection clause specifies a product by giving it a name, by stating the features that are included in that product, and by specifying an initialisation block. The FeatureSpec clause specifies that a given feature is present, and the optional AttributeAssignment clause is used for specifying the attributes, which are assigned concrete values. The InitBlock clause specifies the initialisation block for the given product. This can be any core ABS block, but typically will be a simple call to some already present “main” method. Initialisation blocks are specified in the product selection language to enable product lines with multiple entry points to the code base.

**Example 5.4.1.** Products of the Hello World product line are defined as product selections.

```
// basic product with no deltas
product P1 (English) {
    new Application();  // this runs the application
```
In the example above we specify four products: P1, P2, P3, and P4. In the case of the product P1, the parameter English means the product consists of this feature and of the features implied by the constraints over the feature model. In this case the implied features are Language and the root MultiLingualHelloWorld, according to the model in Example 5.1.1. In P3 and P4 the parameters also include attribute values, in these cases assigning a value to the attribute times from the feature Repeat. The block of ABS code associated to each product provides its initialisation code. Every product in our example executes the main method of the class Application, which is included in the core module.

Example 5.4.2. Define the following product selections for the peer-to-peer file sharing system.

```plaintext
product P2P1 (Basic) {}
product P2P2 (FileFilter) {}
product P2P3 (ParentalControl) {}
product P2P4 (FileFilter, ParentalControl) {}
```

Each of these product selections produce a different peer-to-peer application, with the exception of P2P3—it will fail the satisfiability check with respect to the feature model. The product P2P1 yields the core module, the product P2P2 applies the delta module DFileFilter to the core module, and product P2P4 applies the delta modules DFileFilter followed by DPARENTalControl to the core module.

5.4.2 Semantics

Consider the PSL script

```plaintext
product P (Feature1 { attribute1_1 = value1_1, ...},
  Feature2 { attribute2_1 = value2_1, ... }, ...)
{ InitBlock }
```

There are two components of interest in a product selection:

- An assignment \( \sigma \in \text{ProductSelection} \) defined as follows:
  - for each Feature, \( \sigma(\text{Feature}) = 1 \)
  - for each attribute \( \sigma(\text{Feature}, \text{attribute}_{i,j}) = \text{value}_{i,j} \)
- The initialisation block.
Note that 0 and 1 are again used to represent false and true, respectively, as our constraints are entirely over integers.

The assignment is not complete in that it does not specify the values for unselected or implicitly-selected features. An example of an implicitly selected feature occurs when a leaf feature is selected, then its parents in the tree need to be selected too. In addition, the variable $f^\dagger$ introduced to count optional feature $f$ (whether or not it is present) is set to 1. Finally, values of attributes for unselected features need to be set to some arbitrary value for technical reasons, namely, because the satisfaction relation for the constraints a feature model is encoded into requires that all variables (from $FID \cup FID.AID$) appearing in the constraint be defined.

The following steps add the missing elements to an assignment. We call this the completion of the product selection. Assume that $f \in FID$ and $a \in AID$. For feature model $FM$, it is encoded as constraints given by $\psi = \llbracket FM \rrbracket$.

1. Iterate the following steps until a fixed point is reached:
   (a) If $f \in \text{dom}(\sigma)$ and $f'$ is the parent of $f$, then set $\sigma(f') = 1$.
   (b) If $f \in \text{dom}(\sigma)$ and $f^\dagger$ appears in $\psi$, then set $\sigma(f^\dagger) = 1$.
2. If $f \notin \text{dom}(\sigma)$ and $f$ appears in $\psi$, then set $\sigma(f) = 0$.
3. If $f.a \notin \text{dom}(\sigma)$ and $f.a$ appears in $\psi$, then set $\sigma(f.a) = v$, where $v$ is an arbitrary (integer) value within the range specified for $f.a$.

A product selection $\sigma$ is valid whenever for all completions $\sigma'$ it is the case that $\sigma' \models \psi$.

**Example 5.4.3.** The product $P3$ from [Example 5.4.1] results in the following initial variable assignment

$$\begin{align*}
\sigma(\text{French}) &= 1 \\
\sigma(\text{Repeat}) &= 1 \\
\sigma(\text{Repeat.times}) &= 10
\end{align*}$$

In the context of the feature model in [Example 5.1.3] The remaining variables are English, Dutch, German and MultilingualHelloWorld, which is the parent of French and Repeat, and there are no other attributes. The completion of $\sigma$ includes the following additional elements:

$$\begin{align*}
\sigma(\text{MultilingualHelloWorld}) &= 1 \\
\sigma(\text{English}) &= 0 \\
\sigma(\text{Dutch}) &= 0 \\
\sigma(\text{German}) &= 0
\end{align*}$$

Some effort shows that the resulting completed assignment $\sigma$ satisfies the constraints specified in [Example 5.1.3]. In contrast to this, the constraints would not be satisfied for product $P4$, where $\sigma(\text{English}) = 1$, $\sigma(\text{Repeat}) = 1$, and $\sigma(\text{Repeat.times}) = 6$, due to the clause $\text{English} \rightarrow (\text{Repeat} \rightarrow (\text{Repeat.times} \geq 2 \land \text{Repeat.times} \leq 5))$.

### 5.5 Product Generation

In this chapter we introduced four language extensions to core ABS: the $\mu$TVL language to represent feature models, the delta modeling language (DML) to represent delta modules, the product line configuration language (CL) to associate deltas to products and to establish the order of application of the deltas, and the product selection language (PSL) to describe the desired products. We now give a global perspective of the use of these four languages in the generation of a final software product.

Given a core ABS program $P$, a set of delta modules $\Delta$, a product line configuration $C$, a feature model $FM$, and a product selection $p$, the following steps are performed to build the final software product:

**Check** that the product selection $p$ is satisfied by the feature model $FM$, as explained in [Section 5.4.2].
Select the delta modules from $\Delta$ with valid application condition according to $p$, as described in Section 5.3.2.

Apply the deltas to the core program $P$, in the prescribed order, as described in Section 5.2.2. Add the initialisation block from the product selection—this will be the ‘main’ method.

The application of the deltas yields the final software product, which is a core ABS program.

Example 5.5.1. The generated core ABS code for the Hello World product $P_3$ follows.

```java
interface Greeting {
    String sayHello();
}

class Greeter implements Greeting {
    String original_sayHello() {
        return "Bonjour tout le monde";
    }
    String sayHello() {
        String result = "";
        Int i = 0;
        while (i < 10) {
            result = result + original_sayHello();
            i = i + 1;
        }
        return result;
    }
}

class Application {
    Unit run() {
        Greeting bob;
        bob = new Greeter();
        String s = "";
        s = bob.sayHello();
    }
}
{
    new Application();
}
```

Remark 5.5.2. At present there is no facility in the ABS compiler to regenerate the core ABS code resulting from to the application of delta modules, as in the above example. All transformations are applied to the internal AST within the compiler. See Chapter 7 for more details.

5.6 Related Work

Features at the level of a feature model are merely labels [58]. Existing approaches to integrate feature-based variability into modeling and implementation languages can be classified into two main categories [174, 114]: annotative (or negative) and compositional (or positive). As a third approach, model transformations are applied for representing variability mainly in modeling languages.

Annotations. Annotative approaches consider one model representing all products of the product line. Variant annotations, e.g., using UML stereotypes in UML models [183, 85] or presence conditions [58], define which parts of the model have to be removed to derive a concrete product model. The orthogonal variability model (OVM) proposed
in Pohl et al. [148] models the variability of product line artifacts in a separate model where links to the artifact model take the place of annotations. Similarly, decision maps in KobrA [24] define which parts of the product artifacts have to be modified for certain products. On the implementation level, annotative approaches are realized by conditional compilation, frames [182] or in the programming language Colored Featherweight Java (CFJ) [112]. In these approaches, the source code of the whole product line is marked with respect to product features on a syntactic level. In the Koala component model [170], the variability of a component architecture containing all possible components is expressed by component parametrization that is instatiated depending on the product features.

**Composition.** Compositional approaches, such as delta modeling [158, 154, 155, 157], associate model fragments with product features that are composed for a particular feature configuration. A prominent example of this approach is AHEAD [29], which can be applied on the design as well as on the implementation level. In AHEAD, a product is built by stepwise refinement of a base module with a sequence of feature modules. Design-level models can also be constructed using aspect-oriented composition techniques [94, 174, 137]. [160] apply model superposition to compose model fragments. On the programming language level, several program modularization techniques [122], such as aspects [113], framed aspects [123], mixins [162], hyperslices [167] or traits [74, 34], are used to implement features in a compositional fashion. In addition, the modularity concepts of recent languages, such as Scala [138] or NewSpeak [40], can be used to represent product features. CeasarJ [128] and Aspectual Feature Modules [18] are proposed as a combination of feature modules and aspects to modularize crosscutting concerns.

In feature-oriented software development (FOSD) [29], features are considered on the linguistic level by feature modules. Apart from Jak [29], there are various other languages using the feature-oriented paradigm, such as FeatureC++ [19], FeatureFST [20], or Prehofer’s feature-oriented Java extension [149]. In [128, 18], combinations of feature modules and aspects are considered. In [21], an algebraic representation of FOSD is presented. Feature Alloy [22] instantiates feature-oriented concepts for the formal specification language Alloy.

**Transformations.** Model transformations are used to represent product variability mainly on the artifact modeling level. The common variability language (CVF) [93] represents the variability of a base model by rules describing how modeling elements of the base model have to be substituted in order to obtain a particular product model. In [105], graph transformation rules capture artifact variability of a single kernel model comprising the commonalities of all systems. In [95], architectural variability is represented by change sets containing additions, removals or modifications of components and component connections that are applied to a base line architecture. Perrouin et al. [142] obtain a product model by model composition and subsequently refinement by model transformation.

**Delta Modeling.** The notion of program deltas was introduced by Lopez-Herrejon [122] to describe the modifications of object-oriented programs. Schaefer et al. [158] introduced delta modeling as a means to develop product line artifacts suitable for automated product derivation and implemented with frame technology [182]. In subsequent work [154], delta modeling was extended to a seamless model-based development approach for SPLs, where an initial product line representation is stepwise refined until an implementation can be generated. The conceptual ideas of delta modeling have also been applied to the programming language level in an extension of Java with core and delta modules allowing the automatic generation of Java-based product implementations [155]. In [157] and [156], a version of delta-oriented programming is proposed where products are generated only from delta modules applied to the empty product. Furthermore, in this version the application conditions and the application ordering are specified separately from the delta modules in a product line specification in order to increase the reusability of the delta modules and to enable compositional type checking.
Chapter 6

Platform Models and Configuration

This chapter addresses Platform Models and Configuration. As planned in the work description of HATS Task 1.2, we investigate abstract, declarative platform models to capture aspects of the target architecture and hardware platform. In addition, feature models are equipped with configuration parameters and contain a simple description of the deployment possibilities.

Software systems are generally written to be deployed on a wide range of highly configurable (hardware) platforms, which often exhibit very different characteristics, such as architecture, operating system, available memory, processor speed, number of cores, network bandwidth, and so forth. Such variability may also occur not as a change in the hardware platform, but as a change in the resources available to applications running on the platform. Varying any of these aspects results in a change in the behaviour of the software and/or requires that the software be changed. Changes in operating system and hardware platform often require that the software modules interfacing the operating system or hardware be changed (though platforms such as Java and virtualisation technology aim to ameliorate the need for source code variation). Increasing the available memory, network bandwidth or processor speed generally all contribute to improving the performance of the software, whereas decreasing such resources may even make it impossible for the software to run. Changing the number of cores may allow or even require a different underlying concurrency model in order to better exploit the parallel execution opportunities thereby opened. Better understanding of the consequences of such variability is essential before making costly investment into inappropriate technology.

One of the goals of the HATS project is to model the variability of deployment scenarios using the ABS language, by enabling platform models and relevant configuration parameters to be encoded in an appropriate high-level fashion (work description of HATS Task 1.2). This will enable systems to be analysed before deployment and without a priori fixing the deployment scenario. Ingredients of the full ABS language (plus experimental extensions) can be combined to construct appropriate platform models and enable them to be externally configured. These ingredients include Deployment Components (experimental feature), μTVL feature models, DML delta modules, and CL configuration scripts. The approach taken leverages the language extensions designed for expressing product line variability, thereby reducing the number of different extensions a programmer needs to master and reducing redundancy in the tool-set.

At the lowest level, Deployment Components [108, 109] are special concurrent ABS objects that model real world deployment concerns in an high-level fashion, for example, by abstracting away from specific details such as the number and speed of physical processors and modeling the consumption of resources associated with a concurrent object in a special deployment component. Deployment components are parametric in the computational resources offered and different kinds of deployment components make it possible to model different kinds of resource usage. Parameters to deployment components can represent resource bounds and resources consumption rates.

Feature models written in μTVL express variation in different deployment scenarios via features and variation in the configuration of such scenarios via attributes. Again, this is at a high level of abstraction without any behavioural ingredients. Constraints can be imposed on features to express the compatibility between different deployment scenarios. For example, two different operating systems may not be deployed together, but network devices of different capabilities may. Constraints on attributes reflect the range of valid values for such attributes and their combinations. For example, memory modules may be available only in chunks of 1GB, and cache sizes may be enforced to be $\frac{1}{4}$th the size of the main memory. Deployment scenarios would typically be expressed as a separate root feature model,
exploiting the orthogonality between application variability and deployment variability.

DML delta modules express coarse-grained variability in the deployment scenarios at the level of behavioural models. For example, one delta module may introduce a deployment component based on a particular scheduling strategy, whereas another delta module may introduce one using a different strategy. (Or alternatively, the two delta modules may modify the key method introducing the deployment component). Fine-grained variability in the deployment scenarios is expressed using configuration parameters to delta modules. These parameters will represent the resource bounds and resource consumption rates relevant for the deployment scenario being modeled.

CL configuration scripts play the role of passing the configuration parameters defined in the µTVL feature model to the delta modules (and consequently, the deployment components) that implement the corresponding low-level modeling. Such scripts will also link feature names with the delta modules and deployment components implementing the features, as usual.

Remark 6.0.1. Deployment Components are an experimental ABS feature developed in the context of Task 2.1 (Configurable Deployment Architecture) and will be reported in Deliverable 2.1 “Configuration deployment” in project month 30. The discussion in this chapter is based on the existing experimental versions of such components, which are subject to change. A consequence is that these models are only descriptive and cannot be executed with the current version of the Maude interpreter. Nevertheless, the approach of combining feature models, delta modules and deployment components remains valid.

6.1 Deployment Components

In order to capture deployment scenarios for ABS models, an experimental extension to the ABS language with deployment components [109] has been devised. Deployment components can be parametric in the amount of concurrent activity they allow within a time interval, can track memory allocation and deallocation, and, in principle, can monitor other resource usage scenarios of the concurrent objects that make up an ABS model. Focusing on the amount of concurrent execution resources available to a component, for example, can be used to determine the effect variation in such resources can have on the response time of a component. The underlying model is based on a lightweight notion of timed concurrent objects [35], extended to capture parametric concurrent activities and other resource usage between observable points in time. In order to validate and compare the timed concurrent behavior of models under restricted concurrency assumptions, the operational semantics of the experimental ABS extension has been defined using interleaving concurrency, which enables the use of Maude as a model simulator.

The first version of deployment components restrict the inherent concurrency of objects in ABS, mapping the logical concurrency to a model of physical computing resources. Deployment components abstract from the number and speed of the physical processors available to the component by a notion of concurrent resource [109]. The granularity of the time model defines the points in time when the executing system is observable. Concurrent resources may be consumed in parallel or in sequential order, which reflects the number of processors and their speeds relative to the granularity of the time intervals of the model. Thus, the logical concurrency model of the concurrent objects is controlled by their associated deployment component. Another kind of deployment component addresses memory allocation. These could be combined into a deployment component that monitors both concurrent resources and memory resources. A deployment component is parametric in the computational resources it offers to a group of dynamically created objects, which allows easy configuration of concurrent resources. Objects deployed on a component may iteratively consume resources within a time interval until the component runs out of resources or the objects are otherwise blocked.

The current experimental language extensions supporting deployment components are:

- An expression “component $D(e^*)$” which creates a deployment component of type $D$. It accepts configuration parameters for the resources it manages.
- The possibility of putting newly-created objects into a deployment component “$\text{new } C(e^*) \text{ in } dc$”
- An expression “thisComp” for returning the deployment component the current thread is operating in.
- An expression “now” for returning the present logical time with respect to the current deployment component.
• An expression “available” for returning the amount of available resources allocated to the current deployment component.

• An expression “load(e)” for returning the number of resources used in the current deployment component.

• A statement “transfer(e,e)” for dynamic resource reallocation between deployment components.

The last three features enable concurrent objects to monitor their own resource usage and perform load balancing via resource transfer between different deployment components. Wrapped appropriately within methods, this also permits external entities to monitor a collection of concurrent objects and perform load balancing.

Note that in the related publications [108, 109] the keyword “component” is used alone to generate a deployment component, as there was only one sort of deployment component. Our syntax enables the creation of different component types within one application.

When a concurrent object is created and associated with a deployment component, the concurrent object is said to reside in the deployment component. All accesses to resources by the concurrent object are monitored by the deployment component; if within a given logical time slice, the deployment component’s resources are exhausted, the concurrent objects depending on those resources to proceed must block.

Note that a deployment component is not the same thing as a concurrent object group (COG). Concurrent object groups have only one thread of control, whereas a deployment component may have many. Deployment components are concerned with resource usage management and measurement.

6.2 Example

To illustrate platform modeling we present an example modeling multiple queries to an in-memory database engine. Three different platform models are considered, modeling the concurrency of a regular multi-core processor, a memory management module, and cloud. Statistics about the behaviour of a model can be collected by running it in the Maude simulator.

6.2.1 Core Application

The core application consists of a database (type Database) which is sent queries (type Query) to perform. The database spawns a new thread to deal with each incoming query by performing a new asynchronous method call. We abstract away from any actual data processing in our query model, i.e., queries simply block the processor for some specified minimum amount of time. Executing a query thus requires both concurrency and memory resources. The deployment component on which the database is deployed manages those resources and limits computations that exceed the available resources.

```java
interface Query { Bool perform(); }
interface Database { Bool perform(Query q); }
interface Client { Unit run(); }

class Query(Int min, Int max) implements Query {
    Bool perform () {
        Time t = now;
        await now >= t + min;
        return now <= t + max;
    }
}
class Database() implements Database {
    Bool perform(Query q) {
        Fut<Bool> res = q!perform();
        await res?;
    }
}
```
The client (type `Client`) to the database periodically fires off queries to the database at a given rate. The client also expects parameters giving the minimum and maximum time taken for queries.

```java
class Client(Database db, Int min, Int max, Int rate) {
  Unit run {
    Time t = now;
    Query q = new Query(min, max);
    Fut<Bool> rc = db!perform(q);
    await now >= t + rate;
    this!run();
    await rc?;
    Bool result = rc.get;
  }
}
```

The main class sets up the deployment component and associates it with a database object. It creates a client and runs it. The methods `max()`, `min()`, `rate()` and `getComponent()` return default configuration parameters and a default deployment component. These methods are candidates to be replaced by applying delta modules, see Subsection 6.2.4 below.

```java
class Main {
  Int max() { return 10; /* default maximum query time */ }
  Int min() { return 5; /* default minimum query time */ }
  Int rate() { return 10; /* default rate of querying */ }

  DeploymentComponent getComponent() {
    return component Core(1); // Default deployment component, explained below
  }
  Unit run () {
    DeploymentComponent dc = this.getComponent();
    Database db = new Database() in dc;
    Client c = new Client(db, min(), max(), rate());
  }
}
```

### 6.2.2 Deployment Components

The following deployment components are available:

- **Core(Int resources)** — models a processor with `resources` many concurrent resources.
- **MemMan(Int memory)** — models a memory managing component where a maximum amount of memory is specified.
- **Cloud(Int memory, Int resources)** — models deployment of the application on a cloud, where the parameters specify the maximum amount of memory (in gigabytes) and the number of concurrent resources reserved by the application. This is in essence a combination of the previous two deployment components.
These deployment components would have special implementations in the Maude engine to monitor the resources following the specified policies.

### 6.2.3 Feature Model

The high level view of a platform model is expressed via a $\mu$TVL feature model. For the present example, this consists of two orthogonal feature models.

The first is Resources with features expressing the possible deployment components, MultiCore, MemoryManager, and Cloud, along with the configuration parameters and their bounds.

```plaintext
root Resources {
  group oneof {
    MultiCore { Int resources in [1 .. *]; },
    MemoryManager { Int memory in [1 .. *]; },
    Cloud {
      Int memory in [1 .. *];
      Int resources in [1 .. *];
    }
  }
}
```

The second feature model is Query expressing the minimum and maximum times, and the rate of queries sent to the model of the database. These are expressed as two features Times and Rate, along with appropriate attributes and constraints on them.

```plaintext
root Query {
  group allof {
    opt Times {
      Int min;
      Int max;
      min >= 0;
      max >= min;
    },
    opt Rate {
      Int rate in [1 .. *];
    }
  }
}
```

### 6.2.4 Delta Modules

The selected deployment component and other parameters are injected into the core code using delta modules. The first three delta modules replace the methods that create the deployment component, thereby allowing a different deployment component to be created. The last two delta modules change the methods that return the values of the configuration parameters specifying query processing times and rates. All delta modules are parameterised by the configuration parameters.

```plaintext
delta MultiCore(Int resources) {
  modifies class Main {
    modifies DeploymentComponent getComponent() {
      return component Core(resources);
    }
  }
```
6.2.5 Configuration

The following CL configuration script links the feature model with the deltas, by describing the passing of attributes from selected features into the appropriate delta modules.

```java
productline Database {
    features MultiCore, Cloud, MemoryManager, Times, Rate;

    delta Core(Core.resources) when MultiCore;
    delta MemMan(MemoryManager.size) when MemoryManager;
    delta Cloud(Cloud.memory, Cloud.resources) when Cloud;
    delta QueryTimes(Times.min, Times.max) when Times;
    delta QueryRate(Rate.rate) when Rate;
}
```

At this point, we have a complete specification of all the platform variability in the application.

6.2.6 Product Selection

In order to actually try out a particular configuration, a product selection is made using the PSL language. Below is one such example. It specifies (ultimately) that the Cloud deployment component will be used with 32G of memory and 1000 units of concurrency resources. In addition, it specifies the query rate, while leaving the default minimum and maximum query times open. This small script can be checked against the feature model above for validity.
The tool set can then be employed to generate a core ABS model, which, with the appropriate experimental extension to the Maude interpreter plugged in, can be used to simulate the specified deployment scenario of the application.

### 6.3 Related Work

Techniques and methodologies for predictions or analysis of non-functional properties are based on either measurement and modeling. Measurement-based approaches apply to existing implementations, using dedicated profiling or tracing tools like, e.g., JMeter or LoadRunner. Model-based approaches allow abstraction from specific system intricacies, but depend on parameters provided by domain experts [75]. A survey of model-based performance analysis techniques is given in [27]. Formal systems using process algebra, Petri Nets, game theory, and timed automata (e.g., [96, 48, 77, 50]) have been applied in the embedded software domain, but also to the schedulability of processes in concurrent objects [103]. The latter work complements ours as it does not consider resource restrictions on the concurrency model, but associates deadlines with method calls.

Work on modeling object-oriented systems with resource constraints is more scarce. Using the UML SPT profile for schedulability, performance and time, Petriu and Woodside [143] informally define the Core Scenario Model (CSM) to solve questions that arise in performance model building. CSM has a notion of resource context, which reflects an operation’s set of resources. CSM aims to bridge the gap between UML and techniques to generate performance models [27]. Closer to our work is M. Verhoef’s extension of VDM++ for simulation of embedded real-time systems [174], in which architectures are explicitly modeled using CPUs and buses, and resources are bound to the CPUs. However, the underlying object models and operational semantics differ. VDM++ has multi-thread concurrency, preemptive scheduling, and a strict separation of synchronous method calls and asynchronous signals, in contrast to our work with concurrent objects, cooperative scheduling, and caller decided synchronization.

Static inference of cost for sequential programming languages has recently received considerable attention. A cost analysis for Java bytecode has been developed in [10], for C++ in [89], and for functional programs in [104]. Inference of worst-case upper bounds on the memory usage of Java like programs in the presence of garbage collection (GC) has been studied in [12]. The analysis accounts for memory that is freed by garbage collection, and thus infer more tight and realistic bounds. The analysis supports several GC schemes. The analysis of [104] supports inference of memory usage, and accounts for memory freed due to destructive matching. In [115] live heap space analysis for a concurrent language has been proposed. However it uses a very limited model of shared memory. Recently, in [11] a cost analysis for X10 programs [49] has been developed. The analysis infers upper bounds on the number of maximum tasks that can be running in parallel. The concurrency primitives in the X10 language are similar to the ABS, though X10 is not based on concurrent objects.

Formal resource modeling happens mainly in the embedded domain. For example, Verhoef et al. [173] use the timed VDM++ to model processing time, schedulability and bandwidth resources for distributed embedded systems, but their approach is less general and not used for memory consumption. There is not much work combining static cost analysis and simulation to analyze resource usage. However, Künzli et al. [117] combine exact simulation and arrival curves to model processing costs, decreasing the needed simulation time by using arrival curves in their simulations to abstract from some of the components in a SystemC model of specific hardware.

Platform models can take into account factors such as the topology, network latency, middleware overhead, the gap between sending different messages, and the number of processors/modules [57]. These models are very concrete, whereas our current approach abstracts away from many of these details. Explicit platform modeling has been used in the context of model-driven architecture (MDA) [175]. These areas make the distinction between platform-specific and platform-independent models, and are not as such models of platforms, but reflect the dependency on platforms. The MDA guide defines a platform as follows [129]: “A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application
supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented.” Wegelaar and Jonkers [176] presents an approach to platform modeling in the context of MDA based on description logic ontologies. These ontologies can be seen as analogous to our feature models. Their approach is aimed at generating software rather than modeling its behaviour, and the platform models are introduced to move from platform-independent models to platform-specific models when performing model transformations. Their platform models could include different kinds of network layers. The key difference with our approach is that our models describe the behaviour directly, whereas their models are at a much higher-level of abstraction and require more transformation to be realised as code. Related approaches include [168, 14]. These do not consider performance issues as we do. Cortellessa et. al. [55] combines software models with platform models for modeling performance. Their approach avoids the intricacies of transformational approaches, such as those based on MDA. Their performance models are validated against actual applications, which we have not yet done. Their approach does not consider software product lines.
Chapter 7

Tool Support and Integration

This chapter presents the design of the tools for the ABS language and how they act in concert in the HATS tool framework. These tools include a compiler for the full ABS language, a virtual machine for testing and executing ABS models, and integration with the Eclipse IDE and Emacs editor.

7.1 The HATS Framework

This report and accompanying implementation and documentation are part of the ongoing work for building the tool support for what we call the HATS framework and its integrated prototype tool (Milestone M4 in the HATS DoW): a set of techniques and tools for developing software product lines rigorously using the ABS language.

We therefore take a look at the HATS framework from the tool perspective. Figure 7.1 gives an overview of the current HATS tools framework. In the figure, the blue parts are the tools designed as part of the HATS project. The yellow boxes represent the states of the internal data as it is processed by the tool chain. The green boxes represent external data which the tools take as input or produce as an output. The ABS compiler takes full ABS source code (an ABS model) as input, which includes feature model descriptions of software product lines (Section 5.1), ABS delta modules (Section 5.2), product line configurations (Section 5.3), product selections (Section 5.4), as well as core ABS code (Chapter 3). Several different back ends translate ABS models into different languages like Maude and Java, which allows ABS models to be executed and analyzed on these platforms.

The abstract syntax tree (AST) is the cornerstone of tool integration: it is the internal representation for ABS models, and tools will typically reason about one or more such models or produce them. All tools will work on a common representation of the AST, which has the following benefits:

- **Reduced implementation costs:** individual tools need not know about concrete model syntax and type-checking.
- **Easier integration:** when the output from one tool can be the input to another, or when several tools can cooperate to produce results in the same format.

Some of the tools that will generate and analyse our AST representation include an ABS model miner ( Deliverable 3.2 “Model Mining”), a verifier for behavioral properties ( Deliverables 2.5 “Verification of Behavioral Properties” [69] and 4.3 “Correctness”), as well as code generators. Possibly there will be additional pre-processing steps to adapt input to or output from tools. For example, an AST version of an ABS model may have to be converted to a particular logic before it can be read by a verifier. Figure 7.1 is not exhaustive; for example, we have not integrated tools for resource analysis ( Deliverable 4.2 “Resource Guarantees” [68]) or debugging ( Deliverable 2.3 “Debugging, Visualisation, and Test Generation”). A run-time assertion checker for attribute grammar specifications will also be part of the tool chain. As proof of concept an automated run-time checker has been implemented for Java [65], which will be ported to ABS (and integrated into the Java back end) in the context of Deliverable 1.3 “Analysis of ABS Models”. A detailed presentation of the final HATS tool chain will be part of Deliverable 1.5 “ABS Tool Platform and Tutorial”.

83
7.2 The ABS Compiler Front End

The role of the ABS compiler front end is to translate textual ABS models into an internal representation and check the models for syntax and semantic errors. The role of the compiler back end is to generate code for the models targeting some suitable execution or simulation environment. Figure 7.2 shows the architecture for the core ABS compiler. Note that the compiler back end has been simplified into just a code generator since optimisation is outside the scope of this deliverable.

The front end for the other extension languages included in full ABS (µTVL, DML, CL, and PSL) is also characterised by the diagram in Figure 7.2 after replacing ‘core ABS’ by a program in these languages, and discarding the code generation part.

7.2.1 Compiler Technology

We give an overview of the technology that we have chosen for the implementation of the core ABS compiler. The lexer and parser are generated using the JFlex and Beaver generators\(^1\). The JastAdd compiler compiler\(^2\) forms the basis for the AST, the semantic analyzer, and the code generator. The attribute grammar mechanism of JastAdd allows

---

\(^1\)JFlex: [http://www.jflex.de/](http://www.jflex.de/)
Figure 7.2: ABS Compiler.

the AST to be augmented during semantic analysis, in order to ease both the analysis itself and the code generation.

Dealing with Variability

The compiler front end deals with the variability encoded in ABS models of software product lines (cf. Chapter 5) in one of two ways. One way is by removing variabilities when a particular product is selected. In this case, the delta modules that correspond to the configured feature set are applied to the core ABS model by modifying the AST accordingly. The resulting flattened AST is then typechecked and used for generating the code for that specific product. The second alternative is to keep the variability information in the AST. A variability-aware back end can then exploit such an extended AST by offering dynamic reconfiguration of products. This approach is currently used by the Maude back end.

7.2.2 Using the Compiler Front End

The compiler front end is in general used as part of the ABS IDE (cf. Sec. 7.6), but it can also be used as a stand-alone tool from the command line.

Prerequisites. In the following we assume that the absfrontend.jar file exists in the current working directory. For example, the following command type-checks the ABS-file PeerToPeer.abs:

```
java -jar absfrontend.jar PeerToPeer.abs
```

In case of a type error, a corresponding error message is printed on the console. For example:

```
PeerToPeer.abs:56:3:Method getLength has not the same return type as defined in interface DataBase. Expected ABS.StdLib.Int, but found ABS.StdLib.String.
    String getLength(Filename fId) {
        ...
```
Validating product selections

Product selections are validated with respect to a given feature model using the \( \mu \)TVL tools. These tools have been included in the absfrontend.jar file. For example, the following command verifies if the product P2, which includes only the Basic feature, belongs to the feature model PeerToPeer.mtvl (cf. Section E.2):

```java
java -cp absfrontend.jar mtvl.parser.Main -c PeerToPeer.P2 PeerToPeer.abs PeerToPeer.mtvl
```

In this example, the output is simply a success message:

```
parsing ABS file: PeerToPeer.abs
parsing mTVL file: PeerToPeer.mtvl
checking solution: true
```

Executing the same command to the product P3 yields false when checking the solution, as expected. It is also possible to check if there are solutions for the PeerToPeer.mtvl feature model, and if so, to display an example of a solution, by executing the following command:

```java
java -cp absfrontend.jar mtvl.parser.Main -s PeerToPeer.mtvl
```

Flattening of full ABS models

Flattening a full ABS model means applying a sequence of delta modules to a core ABS model, in order to obtain the behaviour of a particular product. In the ABS compiler front end, the `-product=<name>` switch triggers the flattening for a given product, as shown in the example below.

```java
java -jar absfrontend.jar -product=PeerToPeer.P1 PeerToPeer.abs
```

If the application of deltas, name resolution or type checking are not successful, an appropriate message is displayed. Otherwise, no output is displayed, and the internal AST is flattened according to the product selection. The flattened AST, can then be used, for example, in the generation of Java code, as we will show in Section 7.4.1.

7.3 Maude Back End

In order to have a concrete simulator for ABS models, the operational semantics described in Section D.2 is adapted to the format of rewriting logic, which is executable on the Maude\(^4\) engine. The AST of an ABS model, created as described in Section 7.2.1 is converted into Maude terms that are then used for simulation by an interpreter written in Maude. The Maude engine was chosen because of its high level of abstraction, existing expertise in implementing interpreters on that platform, and because its output (describing final or intermediate states of model simulation) is easily amenable for parsing and further analysis and visualization by other tools. Note that this decision does not preclude other simulation or execution environments; in particular, the AST will be used to generate code for symbolic execution in KeY (Deliverables 2.5 “Verification of Behavioral Properties” [69] and 4.3 “Correctness”).

The output of Maude is a plain-text representation of the state of the entire model, comprising classes, futures, COGs and object instances with their state and process queue. This output can be used for visualization or analysis purposes.

The Maude interpreter is also the designated testbed for dynamic adaptation of ABS models. Work is under way to convert the extended AST, which includes product selections, configurations and delta declarations into Maude terms (as indicated in Figure 7.1 by the dotted arrow). Note that the Java code generator ignores the AST parts corresponding to ABS extensions. The Maude interpreter will support runtime reconfiguration of models. Implementations efforts currently underway will allow us to reconfigure a model to represent a different product of a given SPL, or to dynamically add and remove deltas from a running product. The Maude interpreter will have built-in support for the addition or removal of interfaces and classes and the modification of class structure, i.e., class methods and fields, via the Delta mechanism.

\(^4\)Maude: [http://maude.cs.uiuc.edu/](http://maude.cs.uiuc.edu/)
7.3.1 Running models using Maude

The Maude code generator takes an ABS model (possibly contained in multiple files) as input and generates a Maude file. This generated file contains Maude representations of all functions, deltas, classes and products defined in the model. (Interfaces and datatypes are not represented, since a type-correct ABS model cannot contain invalid expressions, hence they are not needed at runtime.) For example, to generate a Maude representation of the peer-to-peer model, the following command line is used:

```
java -cp absfrontend.jar abs.backend.maude.MaudeCompiler PeerToPeer.abs -o PeerToPeer.maude
```

The resulting file `PeerToPeer.maude` can be loaded into Maude. When loading, Maude expects the file `abs-interpreter.maude` to be either already loaded, in the same directory as `PeerToPeer.abs` or in a directory included in the environment variable `MAUDE_LIB`.

After loading the compiled file into Maude, the model’s main block is started by the following command:

```
rew start .
```

The result of evaluating is a “dump” of the complete system state of the model, with one Maude term for each class, cog, object, and future variable. Usually most interesting are the object states, which look like this:

```
< ob("Bob-0") : "Bob" |
  Cog: "Bob-0",
  Att: @ "alice" |-> ob("Alice-0"), @ "msg" |-> "int[7]", @ "this" |-> ob("Bob-0"),
  Pr: idle,
  PrQ: noProc,
  Lcnt: 2 >
```

Here we have an object with identity `ob("Bob-0")`, of class `Bob`, running in the cog `"Bob-0"`. It has three attributes: `this` pointing to itself, `msg` with the integer 7, and `alice` pointing to another object.

To start a model with deltas `D1` and `D2` applied, enter the following command at the Maude prompt (no commas between the delta names):

```
rew start("D1" "D2") .
```

Directly specifying Deltas is more a debugging aid, since normally these are implied by the product and product line definitions.

To start a specified product, e.g. `P2`, enter the following command at the Maude prompt:

```
rew start(P2) .
```

The Maude code generator then starts the main block of product `P2`, after applying the deltas that are needed by `P2`.

In all these cases, the resulting output is a Maude expression of the whole system state, i.e. the state of the main block and all objects that were created while running the model.

7.4 Java Back End

The Java back end takes ABS models as input and generates runnable Java code as output. It has a built-in Java compiler so that it can also directly generate JVM bytecode. The rationale behind having a Java back end is to have an execution platform for ABS models that:

1. has a high performance and scalability so that very large ABS models can be executed and tested;
2. has a high configurability for testing and applying different scheduling strategies;

5The Maude tool can be found at [http://maude.cs.uiuc.edu/download/](http://maude.cs.uiuc.edu/download/)
6Delta parameters are being implemented currently, so the syntax for starting a product with delta parameters is not fixed yet.
allows for easy observation and for integration of additional tools written in standard Java;
4. makes it possible to directly use generated code from ABS models in systems that are written in standard Java.

In contrast to the Maude back end, the Java back end cannot be easily used for analysis purposes, for example model checking. The Java back end uses a similar Java translation as is used by JCoBox [160] which proved to be very efficient. However, in contrast to JCoBox, the generated code of ABS has better support for configuring the scheduling strategies, for system observation, and debugging. The details of the Java generation is out of scope of this report and will be provided in Deliverable 1.4 “ABS Derivation and Code Generation”.

7.4.1 Using the Java Backend

The Java back end can either be used from the ABS IDE (cf. Sec. 7.6), or can be used on the command line to generate Java code.

7.4.1.1 Generating Java Code

For example, the following command generates Java code for the PeerToPeer.abs into the javagen directory:

```
java -cp absfrontend.jar abs.backend.java.JavaBackend -d javagen PeerToPeer.abs
```

If the command is successful it will generate Java source and JVM class files into the javagen directory. For this example, the following files are generated:

```
Catalog.java  findServer_f$1$3$4$6.class  Network_i.class  Node_c$9.class
Client_i.class findServer_f$1$3$4.class    Network_i.java  Node_c.class
Client_i.java findServer_f$1$3.class      Node_c$10.class  Node_c.java
DataBase_i.class findServer_f$1.class    Node_c$11.class  OurTopology_c.class
DataBase_i.java findServer_f.class      Node_c$1.class   OurTopology_c.java
DataBaseImpl_c.class findServer_f.java  Node_c$2.class   Packet.java
DataBaseImpl_c.java Main$1.class        Node_c$3.class   Peer_i.class
File.java     Main$2.class              Node_c$4.class   Peer_i.java
Filename.java Main$3.class              Node_c$5.class   Server_i.class
Filenames.java Main$4.class              Node_c$6.class   Server_i.java
findServer_f$1$2.class  Main.class      Node_c$7.class
findServer_f$1$3$4$5.class Main.java     Node_c$8.class
```

If the ABS model specifies a software product line by using the ABS extensions introduced in Chapter 5, variability is resolved prior to generating Java code. This is achieved by generating code for one specific product of the product line. On the command line, the switch `-product=<name>` is used to select a specific product. The product name needs to include the ABS module where the product was defined as a prefix. For example:

```
java -cp absfrontend.jar abs.backend.java.JavaBackend -d javagen -product=PeerToPeer.P2 PeerToPeer.abs
```

7.4.1.2 Executing Generated Java Code

The Java backend generates for every Main block that exists in an ABS model, a corresponding Main class that contains a standard Java main method. Thus, the generated Java code can be executed like any other standard Java code by using the java command. The generated Java code relies on a runtime library (included in the absfrontend.jar), which must be provided when executing the system. For example, to execute the code generated from the PeerToPeer.abs example, one can use the following command on the command line:

```
java -cp absfrontend.jar abs.backend.java.JavaBackend -d javagen -product=PeerToPeer.P2 PeerToPeer.abs
```

7.4.1.2 Executing Generated Java Code

The Java backend generates for every Main block that exists in an ABS model, a corresponding Main class that contains a standard Java main method. Thus, the generated Java code can be executed like any other standard Java code by using the java command. The generated Java code relies on a runtime library (included in the absfrontend.jar), which must be provided when executing the system. For example, to execute the code generated from the PeerToPeer.abs example, one can use the following command on the command line:

```
java -cp absfrontend.jar abs.backend.java.JavaBackend -d javagen -product=PeerToPeer.P2 PeerToPeer.abs
```

7.4.1.2 Executing Generated Java Code

The Java backend generates for every Main block that exists in an ABS model, a corresponding Main class that contains a standard Java main method. Thus, the generated Java code can be executed like any other standard Java code by using the java command. The generated Java code relies on a runtime library (included in the absfrontend.jar), which must be provided when executing the system. For example, to execute the code generated from the PeerToPeer.abs example, one can use the following command on the command line:

```
java -cp absfrontend.jar abs.backend.java.JavaBackend -d javagen -product=PeerToPeer.P2 PeerToPeer.abs
```

This paper is included in Appendix F.
As ABS does not support I/O operations, executing an ABS model on the command line does not produce any output. In this case, this command will just finish without further information. In case an `assert` statement (cf. Sec. 3.9.9) fails during the execution, a corresponding error message will be shown. For example:

```
Error in Task (1) [COG [Main] (1), Method: Main.main block]:
PeerToPeer.abs:181:3: Assertion failed
```

### 7.4.1.3 Observing ABS Systems

In order to be able to observe an ABS system, the Java back end supports a flexible observation mechanism. This mechanism allows the user to write system observers by using the Java programming language. Currently, there are three observers implemented:

1. A **Console Observer**, which prints information on the standard output.

2. A **Graphical Debugging View**, which shows the system state using a Java-based graphical user interface (see Fig. 7.3). It shows information about the object-state of the system as well as the state of all executing tasks including their source positions.

3. An **UML Sequence Chart Generator**, which generates UML sequence charts. It provides a high-level view of the communication between COGs. The sequence charts are visualized by using a modified version of the existing tool SDEdit.

Observers can be specified on the command line by using the `-Dabs.systemobserver=<observerlist>` parameter. It is possible to specify several observers, which allows the user to use multiple observers at the same time, [http://sdedit.sourceforge.net/](http://sdedit.sourceforge.net/)
e.g., visualization the state by using the graphical debugger and see the high-level communication by using the UML sequence chart generator. The following command executes the PeerToPeer example and visualizes the system by using the graphical observer. In this case, the -Dabs.debug=true option must be given to be able to observe the stepping of tasks.

```
java -cp javagen:absfrontend.jar -Dabs.debug=true \
    -Dabs.systemobserver=abs.backend.java.debugging.GraphicalDebugger PeerToPeer.Main
```

### 7.4.1.4 Debugging ABS Systems

The Java backend provides a flexible configuration mechanism to define the scheduling strategies that are used during the execution of an ABS system. This is important as ABS is a non-deterministic language and the scheduling is crucial for the execution of the system. Currently, the following scheduling strategies are implemented:

1. The **default scheduler**, which executes the system in a non-deterministic way.
2. A **random scheduler**, which also executes the system in a non-deterministically way, but uses a configurable random seed. With a given seed the system is actually executed deterministically, so that a system execution can be reproduced by providing the seed.
3. An **interactive scheduler**, which provides a graphical user interface that allows the user to make scheduling decisions manually (see Fig. 7.5).
4. A **replay scheduler**, which replays a given scheduling history. Such a history can be created by a user by using the interactive scheduler and save a manually created scheduling history to a file.

Schedulers are specified by using the command line option `-Dabs.totalscheduler=<scheduler>`. Schedulers and observers can be combined in an arbitrary way. A typical use scenario is to use the interactive scheduler in combination with the graphical debugger view:

```
java -cp javagen:absfrontend.jar -Dabs.debug=true \
    -Dabs.systemobserver=abs.backend.java.debugging.GraphicalDebugger PeerToPeer.Main
```
7.5 Core ABS Code generation

A core ABS code generator that translates the Core AST back into ABS code is currently under development. The goal is to be able to feed generated core ABS code back to the parser in order to aid debugging of full ABS models.

7.6 ABS Integrated Development Environment

ABS features an Integrated Development Environment (IDE) for writing ABS models. The IDE is realized as an Eclipse plug-in for the current Eclipse distribution version 3.6 (Helios). It can be installed by using the standard Eclipse installation routine via an update site. The ABS IDE implementation uses the ABS front end to parse and type-check ABS models. Internally it uses the type-checked AST, which it obtains from the ABS front end for further analysis and visualization purposes. A complete description of the IDE is out of scope of this report and will be provided in Deliverable 1.5 “ABS Tool Platform and Tutorial”.

Figure 7.6 shows a screen shot of the IDE. The following main features are implemented in the ABS IDE:

- Tool Integration Currently, the complete Core ABS compiler suite is integrated into the IDE. The Maude and the Java back end can be directly executed from the IDE by using corresponding buttons and run configurations. Additional HATS tools will be integrated in future versions.

- Background Type-Checking and Error Reporting ABS models are automatically type-checked whenever the user saves an ABS file. The IDE uses the ABS compiler suite to parse and type-check the ABS model and reports possible errors back by providing error reports and underlining the corresponding parts in the ABS code (see Fig. 7.6).

- Semantic Syntax Highlighting Beside the standard highlighting of keywords, the ABS IDE supports semantic syntax highlighting. This means that the syntax highlighting of words is based on their semantics. For example, the editor can distinguish between interface names and data type names and can show them in a different color.

Update site: [http://tools.hats-project.eu/update-site](http://tools.hats-project.eu/update-site)
Semantic Auto-Completion. When a user types an access operator, i.e., a dot ‘.’ or an exclamation mark ‘!’ the semantic auto-completion automatically opens a window and provides possible methods or fields (see Figure 7.7). The shown information is based on semantic information, namely the type of the target.

Outline View. The Outline View is a special window, which gives the user a structural overview over an ABS file (see Figure 7.8). The Outline View can be used to navigate in an ABS file, as clicking onto a certain element opens the corresponding declaration position. This outline of an ABS file can also be shown in the Project Explorer by unfolding a file.

Module Explorer. The Module Explorer offers a hierarchical view of the module structure of an ABS model, independent of the actual files where they are defined in, see Fig. 7.9.

Jump-To-Declaration. When being over an arbitrary name, it is possible to press F3 to get to the declaration of the name. For example, when being on an interface name, pressing F3 will open the corresponding interface declaration.

7.6.1 Debug Perspective

The ABS IDE provides an experimental debug perspective, which is going to replace the graphical debugging view as well as the interactive scheduler of the Java back end (cf. Sec. 7.4.1.3 and Sec. 7.4.1.4). It provides an interactive debugging perspective for observing and executing ABS systems, generated by the Java back end. Figure 7.10 shows a screen shot of a debugging session using the debug perspective. Compared to the graphical debugging view, the debug perspective has the advantage that it is completely integrated in Eclipse. This allows the user to use all IDE features, while debugging the system. Note that even when using the debug perspective, it is possible to use the above mentioned observers, for example, to generate an UML sequence chart while debugging the system.
7.6.2 Emacs Mode

Besides the full Eclipse-based IDE there is an additional Emacs mode that supports the user when editing ABS models using the Emacs editor. The Emacs mode supports syntax highlighting, code indentation, compilation and error localization in source code. It also aids the execution of ABS models using the Maude backend and the Emacs Maude mode.\(^\text{10}\)

The Emacs mode currently resides in the Tools/ABS-emacs/ directory of the HATS code repository; a downloadable version will be available soon.

7.7 Documentation

Documenting the ABS Tool suite is an ongoing effort, which includes the ABS language specification, ABS tools user guide, and developer documentation. The documentation is available online at [http://tools.hats-project.eu](http://tools.hats-project.eu).

\(^{10}\)Emacs Maude mode: [http://maude-mode.sourceforge.net/](http://maude-mode.sourceforge.net/) (also maintained by the HATS project)
Figure 7.10: Screen shot of a debug session in the debug perspective of the ABS IDE

Figure 7.11: The Emacs mode for ABS.
7.8 Running Examples

Concise descriptions of how to compile and run the HelloWorld and Peer2Peer examples, both via translation to Java, and in the Maude interpreter are provided in the example folder shipped with this deliverable.
Chapter 8

Summary

In this report, we have presented the Full ABS language. The language consists of a behavioural core language, core ABS, and four language extensions (μTVL, DML, CL, and PSL) for expressing variability. Together these languages can specify all the variability of a software product line, with PSL scripts specifying the eventual products to derive from the product line.

The core ABS language is class-based, object-oriented, and inherently concurrent. It supports asynchronous method calls, underspecified local scheduling, and interface types for concurrent objects, as well as a notion of components based on the concept of concurrent object groups. Furthermore, the language provides support for user-defined data types to model the internal data structures inside concurrent objects and a side-effect free functional expression language.

The feature modeling language μTVL is a subset of text-based feature modeling language TVL [39,53]. It can be used to express quite complex constraints over the valid feature combinations and their attributes, which may either be boolean- or integer-valued. Although the design of μTVL is not original, the elements of the tool set dealing with it has been re-implemented to better integrate with the ABS tool chain.

The delta modeling language DML is used to specify the variability of the underlying behavioural models in terms of modifications to core ABS classes and interfaces. These modifications can add, remove or modify implemented interfaces, fields and the methods. Delta modules are parametrised and specified independently of a feature module, and therefore may be reused in different places within the same product line and in different product lines.

The product line configuration language CL describes a product line by specifying both features and delta modules applicable to the product line. It specifies the application conditions for the applicability of delta modules based on the presence of features and attribute values, and constraints on the order in which delta modules can be applied. Finally, it specifies how configuration parameters represented as attributes in the feature model are passed into delta modules and ultimately into products derived from the product line.

The product selection language PSL is a small language for giving a feature and attribute selection, which in conjunction with a CL configuration script is used to select and configure the appropriate delta modules to in order to generate a final software product from the product line. A PSL script also provide an initialisation block, which generally is used to call the main method, but may also provide further configuration parameters.

We have provided a formalization of the full ABS language in terms of an EBNF for the concrete/abstract syntax, a basic type system using ADTs and interface types, and an operational semantics, along with specialised semantic models for the language extensions. The language has been demonstrated by means of a simple multi-lingual Hello World example and a more involved model of a peer-to-peer network node, which illustrates both the relationship between the functional sublanguage of ABS and the imperative language, and the use of asynchronous communication and synchronization, and how delta modules can be used in a non-trivial setting.

We have investigated the design of the HATS framework and developed a parser for the surface syntax of the full ABS language, a translator which takes a full ABS model (of a product line) and converts it into a core ABS model (for a specific product), and a prototype interpreter which can perform concrete simulations of Core ABS models. The completion of the tool chain for the full ABS remains to be done; specifically, the type and consistency checker of the language extensions has not yet been completed. An integrated development environment for ABS has been
implemented as a plug-in for Eclipse.

In addition to the language features for expressing software product line variability, the ABS language can also express the configuration demands coming from different deployment scenarios, along with platform models to enable modeling of these scenarios. At the highest levels of abstraction, these platform models are expressed using the above mentioned languages, whereas the low level resource modeling is done using Deployment Components, which are being developed in Task 2.1 and will be reported in Deliverable 2.1.

To describe the dynamic behavior of components we also introduced a behavioral specification framework, addressing specification of behavioral properties of components (Objective 8). This forms the basis for refinement and abstraction (Task 2.6), correctness (Task 4.3) and development of tool support for run-time checking and static verification by means of the theorem provers such as KeY (Task 1.3).

In the proposed design of the Full ABS language, we have tried to keep the language fairly simple, yet incorporate flexible mechanisms for concurrency control and composition, and the specification of product line variability, as promised in the DoW. Further language extensions and refinements will undoubtedly be necessary as we gain experience with full ABS. The inclusion of subsequent language constructs in the final ABS language should also depend on their properties with respect to the analysis of ABS models.
Bibliography


[34] Lorenzo Bettini, Ferruccio Damiani, and Ina Schaefer. Implementing Software Product Lines using Traits. In *Proc. of Object-Oriented Programming Languages and Systems (OOPS), Track of ACM SAC*, 2010. 5.6


[66] Requirement Elicitation, August 2009. Deliverable 5.1 of project FP7-231620 (HATS), available at http://www.hats-project.eu. [4, 2.1, 2.3, 4.2.3]


[74] S. Ducasse, O. Nierstrasz, N. Schärl, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. ACM TOPLAS, 28(2), 2006. [5.6]


[82] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns. Addison-Wesley, Boston, MA, January 1995. 4.1, 4.2.1, 4.2.3
[93] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. Olsen, and A. Svendsen. Adding Standardized Variability to Domain Specific Languages. In SPLC, 2008. 5.6
[95] Scott A. Hendrickson and André van der Hoek. Modeling product line architectures through change sets and relationships. In ICSE, pages 189–198, 2007. 5.6


[139] OMG. Unified modeling language, infrastructure and superstructure (version 2.2, OMG final adopted specification), 2009. 2.1, 3.14

[140] John Ousterhout. Why threads are a bad idea (for most purposes), 1996. Invited talk at the 1996 USENIX Conference. 2.3

[141] Terrence Parr. *The Definitive ANTLR Reference*. Pragmatic Bookshelf, 2007. 4.2.2.4


[155] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In Proc. of 15th Software Product Line Conference (SPLC 2010), September 2010. \[1,2,3,4,5,6\]


[160] Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In European Conference on Object-Oriented Programming (ECOOP’10), volume 6183 of Lecture Notes in Computer Science, pages 275–299. Springer-Verlag, June 2010. \[1,2,3,4,5\]


Glossary

Terms and Abbreviations

µTVL  Micro Textual Variability Language
DML  Delta Modeling Language
Delta  Synonymous with Delta Module
Delta Module  A specification of modifications to core ABS classes and interfaces
DoW  Description of Work
ABS  Abstract Behavioral Specification (Language)
ASM  Abstract State Machine
AST  Abstract Syntax Tree
CL  (Product Line) Configuration Language
Core ABS  The behavioural functional and object-oriented core of the ABS modeling language
COG  Concurrent Object Group
CSP  Communicating Sequential Processes
CCS  Calculus of Communicating Systems
Deployment Component  An abstract resource monitor for platform modeling.
Full ABS  Core ABS language plus all extensions (µTVL, DML, CL, PSL, etc)
Future  A place holder for the result of an asynchronous method call.
IDE  Integrated Development Environment
OOL  Object-Oriented (Programming) Language
KeY  A formal software development tool that aims to integrate design, implementation, formal specification, and formal verification of object-oriented software as seamlessly as possible.
PLE  Product Line Engineering, i.e., SPLE
PSL  Product Selection Language
RMI  Remote Method Invocation
SPLE  Sofware Product Line Engineering
UML  Unified Modeling Language
Meta-Variables

\[
\begin{array}{ll}
  b & \text{branches} \\
  B & \text{blocks} \\
  C & \text{class names} \\
  Cl & \text{class definition} \\
  Co & \text{constructor terms} \\
  D & \text{data type names} \\
  Dd & \text{data type declaration} \\
  e & \text{expressions} \\
  e_e & \text{expressions with side effects} \\
  e_f & \text{functional expressions} \\
  e_p & \text{pure expressions} \\
  f & \text{field names} \\
  fn & \text{function names} \\
  F & \text{function declarations} \\
  g & \text{guards} \\
  \Gamma & \text{type contexts} \\
  I & \text{interface names} \\
  In & \text{interface declarations} \\
  K & \text{kinds} \\
  m & \text{method names} \\
  M & \text{method definition} \\
  M_s & \text{method signatures} \\
  p & \text{patterns} \\
  P & \text{programs} \\
  s & \text{statements} \\
  t & \text{terms} \\
  T & \text{types} \\
  U & \text{types} \\
  v & \text{state variables / functional values} \\
  x & \text{local variables} \\
  z & \text{logical variables}
\end{array}
\]
Appendix A

ABS Standard Library

```haskell
module ABS.StdLib;
export *

data Unit = Unit;        // builtin
data String;             // builtin
data Int;                // builtin
data Bool = True | False; // builtin
data Fut<A>;             // builtin

def Bool and(Bool a, Bool b) = a && b;
def Bool not(Bool a) = ~a;

def Int max(Int a, Int b) =
    case a > b {
        True => a; False => b; 
    };

def Int abs(Int x) =
    case x > 0 {
        True => x; False => -x; 
    };

data Maybe<A> = Nothing | Just(A);

def A fromJust<A>(Maybe<A> a) = case a { Just(j) => j; };
def Bool isJust<A>(Maybe<A> a) =
    case a { Just(j) => True; Nothing => False; };

data Either<A, B> = Left(A) | Right(B);

def A left<A,B>(Either<A, B> val) =
    case val { Left(x) => x; };

def B right<A,B>(Either<A, B> val) =
    case val { Right(x) => x; };

def Bool isLeft<A,B>(Either<A, B> val) =
    case val { Left(x) => True; _ => False; };
def Bool isRight<A,B>(Either<A, B> val) = ~isLeft(val);
```
data Pair<A, B> = Pair(A, B);

def A fst<A, B>(Pair<A, B> p) = case p { Pair(s, f) => s; };
def B snd<A, B>(Pair<A, B> p) = case p { Pair(s, f) => f; };

data Triple<A, B, C> = Triple(A, B, C);
def A fstT<A, B, C>(Triple<A, B, C> p) =
  case p { Triple(s, f, g) => s; };
def B sndT<A, B, C>(Triple<A, B, C> p) =
  case p { Triple(s, f, g) => f; };
def C trd<A, B, C>(Triple<A, B, C> p) =
  case p { Triple(s, f, g) => g; };

// Sets
data Set<A> = EmptySet | Insert(A, Set<A>);
// set constructor helper
def Set<A> set<A>(List<A> l) =
  case l { Nil => EmptySet; Cons(x, xs) => Insert(x, set(xs)); }

/**
 * Returns True if set 'ss' contains element 'e', False otherwise.
 */
def Bool contains<A>(Set<A> ss, A e) =
  case ss {
    EmptySet => False;
    Insert(e, _) => True;
    Insert(_, xs) => contains(xs, e);
  };

/**
 * Returns True if set 'xs' is empty, False otherwise.
 */
def Bool emptySet<A>(Set<A> xs) = (xs == EmptySet);

/**
 * Returns the size of set 'xs'.
 */
def Int size<A>(Set<A> xs) =
  case xs {
    EmptySet => 0;
    Insert(s, ss) => 1 + size(ss);
  };

112
def Set\(\langle A \rangle\) union\(\langle A \rangle\)\((\text{Set}\langle A \rangle\) set1, \text{Set}\langle A \rangle\) set2\) =
  case set1 {
    EmptySet => set2;
    Insert(a, s) => union(s,insertElement(set2,a));
  };

/**
 * Returns a set with all elements of set 'xs' plus element 'e'.
 * Returns 'xs' if 'xs' already contains 'e'.
 */
def Set\(\langle A \rangle\) insertElement\(\langle A \rangle\)\((\text{Set}\langle A \rangle\) xs, A e\) =
  case contains(xs, e) {
    True => xs;
    False => Insert(e, xs);
  };

/**
 * Returns a set with all elements of set 'xs' except element 'e'.
 */
def Set\(\langle A \rangle\) remove\(\langle A \rangle\)\((\text{Set}\langle A \rangle\) xs, A e\) =
  case xs {
    EmptySet => EmptySet ;
    Insert(e, ss) => ss;
    Insert(s, ss) => Insert(s,remove(ss,e));
  };

// checks whether the input set has more elements to be iterated.
def Bool hasNext\(\langle A \rangle\)\((\text{Set}\langle A \rangle\) s\) = ~ emptySet(s);

// Partial function to iterate over a set.
def Pair\(\langle\text{Set}\langle A \rangle,A \rangle\)\(\text{next}\langle\text{Set}\langle A \rangle,A \rangle\)\((\text{Set}\langle A \rangle\) s\) =
  case s {
    Insert(e, set2) => Pair(set2,e);
  };

// Lists
data List\(\langle A \rangle\) = Nil | Cons(A, List\(\langle A \rangle\));

def List\(\langle A \rangle\) list\(\langle A \rangle\)\((\text{List}\langle A \rangle\) l\) = l; // list constructor helper

/**
 * Returns the length of list 'list'.
 */
def Int length\(\langle A \rangle\)\((\text{List}\langle A \rangle\) list\) =
  case list {
    Nil => 0 ;
    Cons(p, l) => 1 + length(l) ;
  };

/**
* Returns True if list 'list' is empty, False otherwise.

```scala
def Bool isEmpty<A>(List<A> list) = list == Nil;
```

* Returns the first element of list 'list'.

```scala
def A head<A>(List<A> list) =
    case list { Cons(p,l) => p ; };
```

* Returns a (possibly empty) list containing all elements of 'list' except the first one.

```scala
def List<A> tail<A>(List<A> list) =
    case list { Cons(p,l) => l ; };
```

* Returns element 'n' of list 'list'.

```scala
def A nth<A>(List<A> list, Int n) =
    case n {
        0 => head(list) ;
        _ => nth(tail(list), n-1);
    };
```

* Returns a list where all occurrences of a have been removed.

```scala
def List<A> without<A>(List<A> list, A a) =
    case list {
        Nil => Nil;
        Cons(a, tail) => without(tail,a);
        Cons(x, tail) => Cons(x, without(tail,a));
    };
```

* Returns a list containing all elements of list 'list1' followed by all elements of list 'list2'.

```scala
def List<A> concatenate<A>(List<A> list1, List<A> list2) =
    case list1 {
        Nil => list2 ;
        Cons(head, tail) => Cons(head, concatenate(tail, list2));
    };
```

* Returns a list containing all elements of list 'list' followed by 'p'.

```scala
def List<A> appendright<A>(List<A> list, A p) =
    concatenate(list, Cons(p, Nil));
```
>Returns a list containing all elements of 'list' in reverse order.

```scala
def List[A] reverse[A](List[A] list) =
  case list {
    Cons(hd, tl) => appendright(reverse(tl), hd);
    Nil => Nil;
  };
```

>Returns a list of length 'n' containing 'p' n times.

```scala
def List[A] copy[A](A p, Int n) =
  case n { 0 => Nil; m => Cons(p, copy(p, m-1)); };
```

// Maps

data Map[A, B] = EmptyMap | InsertAssoc(Pair<A, B>, Map<A, B>); // map constructor helper (does not preserve injectivity)

def Map[A, B] map[A, B](List<Pair<A, B>> l) =
  case l {
    Nil => EmptyMap;
    Cons(hd, tl) => InsertAssoc(hd, map(tl));
  };

def Map[A, B] removeKey[A, B](Map<A, B> map, A key) = // remove from the map
  case map {
    InsertAssoc(Pair(key, _), map) => map;
    InsertAssoc(pair, tail) => InsertAssoc(pair, removeKey(tail, key));
  };

def List<B> values<A, B>(Map<A, B> map) =
  case map {
    EmptyMap => Nil ;
    InsertAssoc(Pair(_, elem), tail) => Cons(elem, values(tail)) ;
  };

//**
* Returns a set containing all keys of map 'map'.
/**

def Set<A> keys<A, B>(Map<A, B> map) =
  case map {
    EmptyMap => EmptySet ;
    InsertAssoc(Pair(a, _), tail) => Insert(a, keys(tail));
  };

115
/∗ ∗
* Returns the value associated with key ’k’ in map ’ms’.
*/
def B lookup<A, B>(Map<A, B> ms, A k) = // retrieve from the map
    case ms {
        InsertAssoc(Pair(k, y), _) => y;
        InsertAssoc(_, tm) => lookup(tm, k);
    };

/* ∗
* Returns the value associated with key ’k’ in map ’ms’, or the value ’d’
* if ’k’ has no entry in ’ms’.
*/
def B lookupDefault<A, B>(Map<A, B> ms, A k, B d) = // retrieve from the map
    case ms {
        InsertAssoc(Pair(k, y), _) => y;
        InsertAssoc(_, tm) => lookupDefault(tm, k, d);
        EmptyMap => d;
    };

/* ∗
* Returns a map with all entries of ’map’ plus an entry ’p’,
* which might override but not remove another entry with the same key.
*/
def Map<A, B> insert<A, B>(Map<A, B> map, Pair<A, B> p) = InsertAssoc(p, map);

/* ∗
* Returns a map with all entries of ’ms’ plus an entry mapping ’k’ to ’v’,
* minus the first entry already mapping ’k’ to a value.
*/
def Map<A, B> put<A, B>(Map<A, B> ms, A k, B v) =
    case ms {
        EmptyMap => InsertAssoc(Pair(k, v),EmptyMap);
        InsertAssoc(Pair(k, _), ts) => InsertAssoc(Pair(k, v), ts);
        InsertAssoc(p, ts) => InsertAssoc(p, put(ts, k, v));
    };

/* ∗
* Returns a string with the base–10 textual representation of ’n’.
*/
def String intToString(Int n) =
    case n < 0 {
        True => "-" + intToStringPos(-n);
        False => intToStringPos(n);
    };

def String intToStringPos(Int n) =
    let (Int div) = (n / 10) in
    let (Int res) = (n % 10) in
    case n {
        0 => "0"; 1 => "1"; 2 => "2"; 3 => "3"; 4 => "4";
5 => "5"; 6 => "6"; 7 => "7"; 8 => "8"; 9 => "9";
_ => intToStringPos(div) + intToStringPos(res);

/**
 * Returns a substring of string str of the given length starting from start (inclusive)
 * Where the first character has index 0
 *
 * Example:
 * substr("abcde",1,3) => "bcd"
 *
 */
def String substr(String str, Int start, Int length) = builtin;

/**
 * Returns the length of the given string
 */
def Int strlen(String str) = builtin;

// A Time datatype.
data Time = Time(Int);
def Int currentms() = builtin;
def Time now() = Time(currentms());
def Int timeval(Time t) =
   case t { Time(v) => v; };
// use this like so:
// Time t = now(); await timeDifference(now(), t) > 5;
def Int timeDifference(Time t1, Time t2) =
   abs(timeval(t2) - timeval(t1));

/**
 * Annotation data type to define the type of annotations
 * currently only TypeAnnotation exists
 */
data Annotation = TypeAnnotation;

[TypeAnnotation]
data LocationType = Far | Near | Somewhere | Infer;

/**
 * Can be used to annotated classes and to ensure that
 * classes are always instantiated in the right way.
 * I.e. classes annotated with [COG] must be created by using
 * new cog, class annotated with [Plain] must be created by using
 * just new, without cog.
 */
data ClassKindAnnotation = COG | Plain;

/**
 * Declare local variables to be final
 */
data FinalAnnotation = Final;
/**
 * Declare methods to be atomic, i.e., such methods must not
 * contain scheduling code and also no .get
 */
data AtomicityAnnotation = Atomic;
Appendix B

Syntax of Core ABS

In this chapter we present the syntax of the core ABS language. The language is specified in abstract syntax.

B.1 A Language for Abstract Behavioral Specification

B.1.1 Abstract Syntax

We first discuss the abstract syntax of the core ABS language, which is given in Figure B.1. We will use some conventions for metavariables representing certain syntactic categories throughout this report, for example, $T$ for types, etc. These are collected for ease of references in the Glossary on page 109.

The syntax is essentially divided into an “object-oriented” part dealing with classes, objects, etc. and a “functional” part, dealing with inductive data types, function definitions, etc. We start by explaining the object-oriented part.

Classes, objects, interfaces

A program $P$ consists of a number of declarations (or definitions) and an optional method body which constitutes the initial activity. This initial activity would correspond to the body of the designated main-method in Java. Alternatively we could also allow an arbitrary number of activities, which would be more compositional. Note that the model is one of “concurrent objects”. If we want to have “active objects”, we would need to have a designated method in a class which is invoked by default as part of the object creation. However, we chose in the abstract syntax not to designate such an activity that is automatically started after initialization. This would typically be solved at the level of the surface syntax using a run method or qualifier.

The syntax contains different categories of names (or identifiers). We distinguish names for classes $C$, interfaces $I$, abstract data types $D$, constructors $Co$, methods $m$, and functions $fn$. As a convention, names for classes, interfaces, data types, and constructors start with uppercase letters. Names for methods and functions start with lowercase letters.

At the top-level of programs, there are four different declarations: data types $Dd$, functions $F$, interfaces $In$, and classes $Cl$. Data type declarations $Dd$ define the data type named $D$, over a list of data type constructors $Co(T)$. Function declarations $F$ define a function with the name $fn$. Abstract data types and functions have global scope. In contrast, methods defined in classes have local scope. Interface declarations $In$ describe the publicly available methods of objects, i.e., objects, variables referring to objects, etc., are typed by interfaces, not by class names. Even if the language does not feature inheritance (nor considers class names as types at the user level), interfaces are hierarchically arranged; i.e., an interface $I$ can be a sub-interface of one or more other interfaces. The subtype relation for interfaces is given by the extends-relationship between interfaces.

Class declarations $Cl$ introduce the name $C$, which also serves as constructor method. The constructor method of a class should not be confused with the constructors in the context of a data type, even if there are (practical and

---

1If one likes to draw a distinction between declarations and definitions, a declaration is about introducing a “name” and fixing the “interface/type” for an entity, and a definition is providing (additionally) a “value/implementation”. In that sense, introducing a class declares the class name etc., and defines the implementation. For interfaces, the word “declaration” would be more appropriate, etc. We are not too strict about this terminology here.
fields of other objects, the only fields accessible are the ones which are local to the object. Pure expressions include
as usual, to the object in which the corresponding method is executing. Note that we do not allow direct access to the
local variables

Computation is expressed in terms of expressions and statements. We distinguish between pure expressions $e_p$ and
expressions with side effect $e_f$. The language is constructed such that side effects occur at the outermost level of
the statements. Therefore, only pure expressions can be nested. Pure expressions $e_p$ include variables $v$ (which may be
local variables $x$, or fields $f$ in the imperative part of the syntax), or (functional) terms $e_f$. The "constant" this refers,
as usual, to the object in which the corresponding method is executing. Note that we do not allow direct access to the
fields of other objects, the only fields accessible are the ones which are local to the object. Pure expressions include

Expressions and statements

Computation is expressed in terms of expressions and statements. We distinguish between pure expressions $e_p$ and
expressions with side effect $e_f$. The language is constructed such that side effects occur at the outermost level of
the statements. Therefore, only pure expressions can be nested. Pure expressions $e_p$ include variables $v$ (which may be
local variables $x$, or fields $f$ in the imperative part of the syntax), or (functional) terms $e_f$. The "constant" this refers,
as usual, to the object in which the corresponding method is executing. Note that we do not allow direct access to the
fields of other objects, the only fields accessible are the ones which are local to the object. Pure expressions include

Figure B.1: ABS abstract syntax.
expressions $e_f$, defined in the functional sub-language. Pure expressions can be compared using $e_1 = e_2$. Expressions with side effects $e_e$ are expressions for object creation, object group creation, and method calls. These can occur at the right hand side of an assignment, but not as an actual parameter to a method call, etc.

*Functional expressions* $e_f$ contain terms, local value definitions, function application, and a case construct. A *term* $t$ may be a logical variable $z$, a constructor term, or a pair of expressions. A constructor term $Co(e_p)$ applies a constructor name $Co$ to the appropriate sub-expressions. By constructor, we mean the data-type constructors of the functional data-type language, not constructor (methods) for instantiating new objects. Note that the arguments $e_p$ to the constructor may be fields or local variables. This links the result from evaluating a functional expression to the current state of execution. We use the special constructor $(e_p,e_p)$ for pairs. Local values are defined using a \texttt{let}-construct. The expression $fn(e_p)$ represents function application. A case construct collects a number of branches $b$, each single one of the form $p \triangleright e_f$, where the pattern $p$ guards the body $e_f$ of the branch. The pattern may be a variable, a (constructor) term, or a pair. In contrast to a term, the sub-expressions of a pattern must also be patterns.

*Statements* $s$ include standard statements \texttt{skip}, conditionals, while-loops, grouping \{\texttt{s}\} of statements, assignments $v := e$ where $v$ is a variable and $e$ an expression. In addition there are statements for scheduling control; \texttt{suspend} (known also as yield) introduces a scheduling point where the current task temporarily stops executing. The task frees the lock, giving other tasks the opportunity to be scheduled instead. Finally, \texttt{await} $g$ releases the lock if the guard $g$ evaluates to \texttt{true} and otherwise continues execution. A guard can be a Boolean expression $e_f$, the polling $v?$ of a reply to a method call, or a conjunction of guards (but not disjunction). Note especially that negated polling is not allowed, as this would necessitate a non-interference conditions in the proof system \cite{64}.

All types $T$ in core ABS have a built-in equality predicate, which recognizes terms. For data types, two terms are equal if they are syntactically equal when reduced to constructor terms. For objects, two object references of the same type are equal if they point to the same object identifier. We denote the equality predicate by the infix operator $==$; thus, for any type $T$ and well-typed expressions $e_1$ and $e_2$ of type $T$, $e_1 == e_2$ is a well-typed expression of type $\text{Bool}$.

Basic assertions rely on the Boolean Data Type. For more detail, refer to Chapter \cite{4} devoted to the Behavioral Interface Specification Framework.
Appendix C

Data type and functional sub-language

In this chapter we describe the “functional” part of the modeling language, basically allowing inductive data types as function definitions over them, using pattern matching. The basic design principle for the language is simplicity. We start with a simple data-type language, and add more advanced features later. The core of the features is just to have inductive data types and the corresponding pattern matching. Omitted are polymorphism, subtyping, and genericity, which are planned for future extensions.

C.1 Syntax

The abstract syntax is given in Table C.1. The syntax is slightly more general than is needed for the current stage of the object-oriented part. The most obvious generalization we use here is that more free form of declaration (using the let-construct). It is used to declare an identifier (a variable name, a name for a data type etc). In the state of the overall language now (cf. Figure B.1), the declarations are largely “flat”: The code for the initial activity is preceded by all declarations of data types, function definitions etc, which makes the definitions/declarations global. In this section, we use, as mentioned, the let-construct to define/declare data-types and values in the functional language, which introduces a (static) scope for the definition. That makes the type checking presentation more “standard” without actually complicating the presentation, and furthermore, in an implementation, being block-structure neither incurs much complication; one needs a stack-oriented syntax table as opposed to a “flat” one that requires all identifiers to be globally unique. This way, we will later straightforwardly be able to extend the language by, for instance, allowing data type definitions local to a method or a COG or similar additional expressivity, without changing the data type part. So when writing data $D\{\ldots\}$ in Figure B.1 at the beginning of a program $P$, we mean more formally here let data $D \ldots$ in “rest of the program”.

We assume a set of variables (represented by $x, x_1, x', \ldots$); in case the variable represents a function, we also use $f$. Symbols $d$ represent data types, and $c$ constructors. Values $v$ include variables and (ground) terms $t$. Apart from that, in the context of the object-oriented language, there might be more values, especially object-references. Patterns $p$ are constructor terms which can contain variables.$^1$

In this section, we work with functional expressions denoted by $e$ and do not distinguish pure expressions $e_p$ or consider expressions with side effects. In order to formulate the functional language independently of the imperative part, the expressions do not contain state variables as in pure expressions. Instead, we import the needed state variables from the instance state into the functional expressions $e$, deal with them in the local state space by means of let-bindings, and the results are stored back to the imperative part by means of field updates or destructive assignments. This way, the reduction semantics of the functional part is independent from both the task-local mutable state $\sigma$ and the heap, i.e., the values of the fields. The operational semantics dealing with the imperative part is covered in Figures D.5 and D.6. Concentrating on the non-imperative part, functional expressions $e$ contain values, three forms of (local) definitions (using a let-construct), a case-construct, and application. The case construct collects a number of branches.

$^1$Note that function abstractions $\lambda x. T$ are not included into the category of values. Implicitly, $\lambda$-abstractions are values in the sense that there is no evaluation under a $\lambda$-abstraction. In our syntax, however, $\lambda$-abstractions are no stand-alone expressions; they can be used in function declarations only.
Table C.1: Abstract syntax of the expression language

\[
\begin{align*}
    v & ::= x \mid t & \text{values} \\
    t & ::= \text{Co}(t^*) & \text{constructor term} \\
         & \mid (t, t) & \text{pair constructor} \\
    p & ::= x \mid \text{Co}(p^*) \mid (p, p) & \text{pattern} \\
    e & ::= v \mid \text{let } x:T = v \text{ in } e & \text{value definition} \\
        & \mid \text{let } x:T = \lambda x:T. e \text{ in } e & \text{function definition} \\
        & \mid \text{let data } D = \text{Co}(T) \ldots \text{Co}(T) \text{ in } e & \text{data type definition} \\
        & \mid \text{case } v \text{ of } b^* & \text{case} \\
        & \mid e \ e & \text{function application} \\
    b & ::= p \triangleright e & \text{branch}
\end{align*}
\]

\(b\), each single one of the form \(p \triangleright e\), where the pattern \(p\) guards the body \(e\) of the branch. As for local definitions, the syntax supports representing values by variable, function definition, and data type definitions/declarations.

**Remark C.1.1** (Let-expression). We based the semantics on using let-constructs for two reasons. First it allows a more simple representation of the operational semantics, for instance, doing without evaluation contexts to fix the evaluation order. This also will allow a simpler representation in rewriting logics, the underlying theory of the Maude tool.

### C.2 Typing

The type system contains types \(T\) and (in a future extension) type operators, i.e., types that take other types as arguments to yield types. The proper usage of types and type operators needs to be regulated, as well; for instance, it is an error to apply a type to another one. To impose an appropriate discipline, the types are themselves equipped with a (simple) type system, where the “types” of the types are known as kinds. The kind \(\ast\) is the kind for proper types, \(K_1 \rightarrow K_2\) represents type operators with argument types of kind \(K_1\) and result types of kind \(K_2\). Currently, type operators are not yet relevant, which means, we only support kind \(\ast\) which makes the system checking well-kinded rather trivial, it only becomes more interesting when introducing type operators and polymorphism at a later stage. Types and their (only) kind \(\ast\) are given in Table C.2

\[
\begin{align*}
    K & ::= \ast & \text{kinds} \\
    T & ::= U \mid U \rightarrow U & \text{types} \\
    U & ::= n \mid T \times T \mid \text{Unit} \\
    \Gamma & ::= \bullet \mid \Gamma, x:T \mid \Gamma, D:K \mid \Gamma, \text{Co}:T & \text{contexts}
\end{align*}
\]

Table C.2: Kinds, types, and contexts.

---

\(^2\)As mentioned, we concentrate on the data type language, we are not discussing classes and interfaces in this chapter. In particular, we do not include the rest of the types mentioned in the abstract syntax of Figure B.1 resp. of Figure D.1 into this table here. Their treatment is orthogonal to the data type part, which we concentrate in this chapter, and they play an orthogonal role of non-descript “basic types” as far as the technical development of this chapter is concerned.
In the current state, we do not introduce type operators in this document. However, they will be the next step for the data type language together with polymorphism, so the kinding system is already prepared to deal with them.

Pairs will be typed by \( T_1 \times T_2 \). The unit type is written Unit. The \( \times \) type constructor is assumed associative with Unit as unit. Functional or arrow types of the form \( U_1 \to U_2 \) will be used to type check constructors and also methods. As the tuple types, they are not part of the user syntax. Later, sequences of types will be introduced as syntactic sugar using \( \times \) and analogously for sequences at the term level (see also Remark C.2.1). The type language does not include general arrow types at the user level, as we do not feature higher-order functions. Concentrating on the data type language, we do not cover types for classes/interfaces here, which makes the type language rather small. Section C.3 introduces a number of predefined types.

Remark C.2.1 (Tuples and sequences). In the data type system, we intend to have constructors of fixed but arbitrary arity. Likewise, methods are in general of type \( T_1 \times \ldots \times T_n \to T \) even if we do not concentrate on the object-oriented part here. To simplify the technical account of type checking, and working with abstract syntax, the type system supports binary product \( T_1 \times T_2 \) and the unit type Unit. With associativity of \( \times \) and Unit as unit, this allows to represent finite sequences, as well. Later, when treating pattern matching, the constructor for products can be treated in the same way as constructors for the user-defined user types. A difference between the pair constructor and the data-type constructors will be that the pair type is polymorphic while (currently) the user-defined constructor types are not. A further difference is that we write the constructor of the tuple type \( T_1 \times T_2 \) by the special syntax \( (e_1, e_2) \), but that is of course only a syntactical particularity and theoretically irrelevant (and we are dealing with abstract syntax here anyhow, not with the user syntax).

The type system is given, as usual, as a derivation system, making use of typing contexts (see Table C.2). The empty context is written as \( \bullet \). Otherwise, \( \Gamma \) contains bindings for variables, constructor names, and the names for the data types. The system will assure that the bindings are unique, so (well-formed) contexts \( \Gamma \) will act as a finite mapping from the respective entities to their binding, and we write \( \Gamma(x), \Gamma(D) \), and \( \Gamma(Co) \) to refer respectively to the type of the variable \( x \), the kind of the name \( n \), and the type of the constructor \( Co \), as defined in \( \Gamma \). Concerning the constructor names \( Co \): each is typed by an arrow type \( S \to T \), where the type system makes sure that \( T \) is the name of a data type; the data type, that the constructor \( Co \) is contributing to construct, of course. Since the rules will assure that constructors are unique in \( \Gamma \), each constructor mentioned in the context belongs to exactly one data type. We write \( \Gamma \vdash \sum Co \colon T \to D \) to assert that \( Co \) are all the constructors for the data type \( D \) and their respective input type is \( T_1 \). That statement \( \Gamma \vdash \sum Co \colon T \to D \) is not a formal judgment of the derivation system in its own right (in the sense that there are no derivation rules to justify that statement), it is a notation to express the constructors and their types for a given data type \( D \). The type system will assure “global” uniqueness of constructors; i.e., each constructor can be part of at most one data type (within one scope). We can consider \( \Gamma \) as a finite mapping from constructor names to their types, as well, and we refer by \( \Gamma(Co) \) to the type of \( Co \). Furthermore, we write \( \text{dom}(\Gamma) \) for the domain of those mappings.

The type system uses judgments as shown in Table C.3. We discuss and formalize them in turn in the following.

| \( \Gamma \vdash \) ok | well-formed context |
| \( \Gamma \vdash T : K \) | kinding |
| \( \Gamma \vdash e : T \) | typing |
| \( \Gamma \vdash_m p : T :: \Gamma \) | typing for guarded expressions |

Table C.3: Judgments.

C.2.1 Well-formed Contexts

Well-formedness of contexts basically assures that a context \( \Gamma \) does not contain a binding to a name twice and that the variables or names in the domain of \( \Gamma \) are bound only to type of base kind \( * \). The judgments are of the form \( \Gamma \vdash \) ok (see Table C.3) and the corresponding rules are given in Table C.4. The empty context \( \bullet \) is well-formed (see rule C-EMPTY). A context can be extended by a type binding for a variable or the binding for a data type name (see rules C-VAR and C-DNAME). In both cases, the name must be fresh, i.e., not occur in the context to be extended. The type
of the variable must be a proper type, i.e., of kind \( \ast \). Also a data type named \( D \) must be of kind \( \ast \). For constructor names, we require that the input type \( U \) is of kind \( \ast \). Note that for checking the kind of type \( U \), it is assured that \( n \) is already defined in \( \Gamma \) as implied by the first premise of the rule. In this way, \( U \) can contain \( n \) recursively.

\[
\begin{align*}
\text{(C-EMPTY)} & \\
\Gamma \vdash \text{ok} & \\
\hline
\text{(C-VAR)} & \\
\text{\( x \notin \text{dom}(\Gamma) \)} \quad \Gamma \vdash T : \ast & \\
\Gamma, x : T \vdash \text{ok} & \\
\hline
\text{(C-DNAME)} & \\
\Gamma \vdash \text{ok} \quad D \notin \text{dom}(\Gamma) & \\
\Gamma, D : \ast \vdash \text{ok} & \\
\hline
\text{(C-CoNAME)} & \\
\text{\( \Gamma \vdash \text{ok} \)} \quad \text{Co} \notin \text{dom}(\Gamma) \quad \Gamma \vdash D : K & \\
\Gamma, \text{Co} : U \rightarrow D \vdash \text{ok} & \\
\end{align*}
\]

Table C.4: Well-formed contexts.

**Remark C.2.2 (Well-formed Contexts).** The well-formed judgment specifies in a theoretical way what it means for a context to be well-formed. The rules can be understood as a recursive procedure to assure these conditions, but in a concrete implementation, one would check these conditions in a more efficient way than given literally by the rules. The contexts \( \Gamma \) correspond to the syntax table and might be implemented by a hash table or a similar data structure.

### C.2.2 Kinding

As said, kinding is captured in judgments of the form \( \Gamma \vdash T : K \). The corresponding rules are given in Table C.5. In rule \( \text{K-NAME} \), the kind for a data type name \( D \) is looked up in the context. Note that, in the current state, the well-formedness restriction on the contexts assures, that \( K \) equals \( \ast \). Arrow types are of kind \( \ast \), provided all mentioned constituent types are, as well (see rule \( \text{K-ARROW} \)).

\[
\begin{align*}
\text{(K-NAME)} & \\
\Gamma(D) = K & \\
\Gamma \vdash \text{ok} & \\
\Gamma \vdash D : K & \\
\hline
\text{(K-PAIR)} & \\
\Gamma \vdash U_1 : \ast & \\
\Gamma \vdash U_2 : \ast & \\
\Gamma \vdash U_1 \times U_2 : \ast & \\
\hline
\text{(K-ARROW)} & \\
\Gamma \vdash U_1 : \ast & \\
\Gamma \vdash U_2 : \ast & \\
\Gamma \vdash U_1 \rightarrow U_2 : \ast & \\
\hline
\text{(K-UNIT)} & \\
\Gamma \vdash \text{ok} & \\
\Gamma \vdash \text{Unit} : \ast & \\
\end{align*}
\]

Table C.5: Kinding.

### C.2.3 Type system

The type system is shown in Table C.6 (and Table C.7). The type for variables is looked up from the type context (see rule \( \text{T-VAR} \)). The next three rules deal with defining values (proper values of function) as well as with introducing a new data type. In each case, the scope of the newly introduced identifier is the body of the let construct. For variables, the premise of rule \( \text{T-VDef} \) checks the body \( e \) in a type context extended by the binding \( x : T_1 \) for the variable, and furthermore, the value \( v \) is checked to be of the expected type \( T_1 \). Function definitions are treated by rule \( \text{T-FDef} \), which works similarly: The function used in the definition and bound to \( f \) (represented as \( \lambda \)-expression) is checked in the first premise to be of the expected type, by checking the function body \( e' \). Note that this type checking premise uses the context extended not only by the function parameter, but also by the function name \( f \). The function name can thus be used in the function body, allowing to type check recursive function definitions.

Defining/declaring a data type is shown in rule \( \text{T-DDecl} \). The rule formalizes type checking without *mutual* recursion and deals just with one name. It is straightforward to generalize the definition for mutual recursion (see later). A data type declaration consists of a name for the data type on the left-hand side of the defining equation, \( n \) in
the rule. The right-hand side specifies the constituent constructors and their respective “input” types \( T_i \). It is required for constructors mentioned in those types that their names are globally unique. This is assured by maintaining that the contexts are all well-formed, i.e., no binding occurs twice. The data type name \( D \) is of kind \(*\) (in current absence of type operators), and we consider the type constructors to be functional types with the name of the data type as range type.

Rules T-APP for applications and rule T-PAIR for pairs are standard. In our language, in applications of the form \( e_1 e_2 \), the expression \( e_1 \) will always be the name of a constructor (introduced by rule T-DDEC or the name of a function (introduced by rule T-FDEF), since we do not support general \( \lambda \)-expressions at the user syntax (avoiding higher-order functions this way). We sometimes write \( f(e) \) and \( \text{Co}(e) \) for applications \( f e \) and \( \text{Co} e \).

Remark C.2.3 (Pairs). Concerning pairs, constructed by \( (_{-},_{-}) \): Pairs are used only in connection with constructor/function arguments but not part of the user syntax expression. Therefore, the type system only has an “introduction rule” for pairs (rule T-PAIR that is) but no corresponding elimination rules, i.e., rules for projections. Deconstructing pairs (used in a constructor) is done via pattern matching. The reason why (at the current stage) we do not support free-form pairing at the user level is that type checking those would require polymorphism, whose treatment we postponed.

Rule T-CASE deals with the case construct and pattern matching. The expression \( e \) is the constructor term used to select one of the branch expressions of \( \vec{b} \). Thus, \( e \) must be typed by the name \( D \) of a data type, which is used in the second premise to reference the constructors and their respective types. Checking that such an expression is appropriately typed can be split into “global” and “local” conditions. The global conditions make sure that there is no overlap between the cases, and that all cases are covered. The local ones make sure that each individual branch is well-typed. Each branch \( b \) is of the form \( p_i \triangleright e_i \) and consists of two parts, a guarding pattern \( p_i \) and the body of the branch \( e_i \). The patterns \( p_i \) are used to select between the different branches. Consequently they must be typed by a constructor pattern, whose selected constructor must be a constructor corresponding to the type of the matching expression \( e \).

---

<table>
<thead>
<tr>
<th>Rule</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T-VAR)</td>
<td>( \Gamma(x) = T ) ( \Gamma \vdash x : T )</td>
</tr>
<tr>
<td>(T-DEF)</td>
<td>( \Gamma \vdash \lambda x : T_1 \Gamma, x : T_1 \vdash e : T_2 )</td>
</tr>
<tr>
<td>(T-FDEF)</td>
<td>( \Gamma, f : T_1 \rightarrow T_2, x : T_1 \vdash e' : T_2 )</td>
</tr>
<tr>
<td>(T-DDEC)</td>
<td>( \Gamma \vdash \text{let } x : T_1 = \text{in } e : T_2 )</td>
</tr>
<tr>
<td>(T-PAIR)</td>
<td>( \Gamma \vdash e_1 : T_1 \Gamma \vdash e_2 : T_2 )</td>
</tr>
<tr>
<td>(T-APP)</td>
<td>( \Gamma \vdash e_1 : T_1 \rightarrow T_2 \Gamma \vdash e_2 : T_1 \rightarrow T_2 \rightarrow T_2 )</td>
</tr>
<tr>
<td>(T-CONSTR)</td>
<td>( \Gamma \vdash \text{Co}(e) : T_2 )</td>
</tr>
<tr>
<td>(T-CASE)</td>
<td>( \Gamma \vdash e : T ) ( \sum_{i} \Delta_i \rightarrow D ) ( { p_i \triangleright e_i } \vdash \sum_{i} \Delta_i \rightarrow D )</td>
</tr>
</tbody>
</table>

Table C.6: Type system.

To be able to check that connection between the branch and the particular data type, we need to match the pattern with the expected type, and in the premise of T-CASE, match (for each branch) \( p_i \) with the \( n \) of the data type. This is done in the premise of the form

---

3That differentiates the constructor types from variant types, which are technically related and which we do not include in the language.
\[ \Gamma \vdash_m p : T :: \Gamma' . \]  \hspace{1cm} (C.1)

The judgment thus corresponds to a matching problem, where \( \Gamma' \) corresponds to the matching substitution, and the judgment can be read as “given the type context \( \Gamma \), the pattern \( p \) matches the type \( T \), where the context \( \Gamma’ \) extends \( \Gamma \) by the corresponding variable-type bindings that make \( p \) match with \( T’ \).

**Definition C.2.4** (Matching). Matching of a pattern with a type is given inductively by the rules of Table C.7.

<table>
<thead>
<tr>
<th>Rule Type</th>
<th>Premises</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>(TM-VAR)</td>
<td>( x \notin \text{dom}(\Gamma) )</td>
<td>( \Gamma \vdash_m x : T :: (\Gamma, x : T) )</td>
</tr>
<tr>
<td>(TM-CONSTR)</td>
<td>( \Gamma \vdash \text{Co}_i ; T_i \rightarrow T )</td>
<td>( \Gamma \vdash_m p_i : T_i :: \Gamma' )</td>
</tr>
<tr>
<td>(TM-PAIR)</td>
<td>( \Gamma \vdash_m e_1 : T_1 :: \Gamma' )</td>
<td>( \Gamma \vdash_m e_2 : T_2 :: \Gamma'' )</td>
</tr>
<tr>
<td></td>
<td>( \Gamma \vdash_m (e_1, e_2) : T_1 \times T_2 :: \Gamma'' )</td>
<td></td>
</tr>
</tbody>
</table>

**Table C.7**: Type system (matching).

The pattern \( p \) is a constructor term possibly containing variables (see the abstract syntax of Table C.1). Technically, matching is a procedure taking two “terms” from the same domain. Here we match a pattern term against a type term, but that difference is not crucial. In the type system, the pattern matching is used, as mentioned, to treat the case-construct. Considering the rules in a goal-directed manner, the “matching routine” is called in the premise of rule T-CASE. Rule TM-VAR matches the variable \( x \) with type \( T \), and the match immediately succeeds, adding the binding \( x : T \) to the context \( \Gamma' \). Matching a term whose top-level construct is a constructor name \( c_i \) is captured in TM-CONSTR. The first premise consults \( \Gamma \) to look up the type of that constructor, and where that result type must correspond to the type \( T \) the pattern is matched against. The procedure then continues with a recursive call on the sub-pattern \( p \) and the binding context \( \Gamma' \) given back from that sub-derivation is also the result of the pattern match of TM-CONSTR. Rule TM-PAIR finally deals with pairs, treating both sub-patterns recursively. Note the sub-goals are not treated independently. Conceptually, the left sub-pattern is treated first, and afterwards the right. The effect of successfully matching \( p_1 \) is changing the context \( \Gamma \) to \( \Gamma' \), which is taken as the “pre-context” for checking \( p_2 \), which may affect a further change to \( \Gamma'' \), which also is the resulting context overall in the conclusion of the rule. Note that by requiring that bindings occur unique in context \( \Gamma \), we assure that in a pattern, no variable occurs more than once.\(^4\)

The following example illustrates type checking for the case construct and pattern matching expression.

**Example C.2.5** (Type checking). Let’s assume a (monomorphic) data type named Pair with one constructor pair of type \( \text{Int} \times \text{Bool} \rightarrow \text{Pair} \). We assume that the type context \( \Gamma \) contains the relevant bindings, and the expression to type check is the one-branched case expression:

\[
\text{case pair}(1, \text{true}) \text{ of } (\text{pair}(x,y)) .
\]  \hspace{1cm} (C.2)

The following sketches the derivation, concentrating on the core premises in the derivation:

\[
\Gamma \vdash_m x : \text{Int} \rightarrow \Gamma' \quad \Gamma \vdash_m y : \text{Bool} :: \Gamma''
\]
\[
\Gamma \vdash_m (x, y) : \text{Int} \times \text{Bool} :: \Gamma''
\]  \hspace{1cm} TM-PAIR

\[
\ldots \quad \Gamma \vdash_m \text{pair}(x,y) : \text{Pair} :: \Gamma''
\]  \hspace{1cm} T-CONSTR

\[
\Gamma' \vdash e' : T
\]  \hspace{1cm} T-CASE

Arguing in a goal-directed manner; i.e., going from the conclusions to the premises, as a recursive procedure does, the derivation starts with T-CASE. Its first premise matches the pattern pair(1,true) with the data type named Pair, whose definition is looked up in T-CONSTR, i.e., \( \Gamma \vdash \text{pair} : \text{Int} \times \text{Bool} \rightarrow \text{Pair} \) (this premise of T-CONSTR is not shown

\(^4\)This linearity assumption for variables in patterns is common and corresponds to the fact that all (different) variables are matched independently. The use of further constraints on variables in patterns is left for future extensions.
in the derivation). This then is matched with the pair \((x, y)\), which is then split by \(\text{TM-PAIR}\) into one sub-goal for each variable, where the derivation ends. Matching \(x\) against the type of integers adds the binding \(x:\text{Int}\) to \(\Gamma\), i.e., \(\Gamma = \Gamma, x:\text{Int}\), which is the “starting context” used to match \(y\), which then yields \(\Gamma' = \Gamma, x:\text{Int}, y:\text{Int}\).

The type system is split into 3 levels, i.e., 3 recursive deduction systems, one depending on the other. Type checking (including matching) from Tables C.6 and C.7 uses kinding derivation from Table C.5 as sub-derivation, which in turn is relying on well-formedness check for context from Table C.4. The following lemmas are standard “sanity” checks for the type system, that the different parts fit together appropriately. (Proofs have been omitted, but these can be found in Deliverable 1.1A).

**Lemma C.2.6** (Well-formedness). \(\Gamma \vdash T : K\) implies \(\Gamma \vdash \text{ok}\).

**Lemma C.2.7** (Well-kindness). \(\text{If } \Gamma \vdash e : T, \text{ then } \Gamma \vdash T : *, \text{.}\)

**Lemma C.2.8** (Matching preserves well-formedness). \(\text{If } \Gamma \vdash \text{ok and } \Gamma \vdash_{\text{m}} p : T :: \Gamma', \text{ then } \Gamma' \vdash \text{ok}\).

**Lemma C.2.9** (Monomorphism). \(\text{If } \Gamma \vdash e : T_1 \text{ and } \Gamma \vdash e : T_2, \text{ then } T_1 = T_2\).

The next result shows, that the rules, interpreted as recursive procedures, actually terminate. Without complications such as subtyping, polymorphism, etc., termination is rather straightforward.

**Lemma C.2.10** (Termination). Type checking is decidable.

### C.3 Semantics

Next we specify the semantics of the data type language. The data type language is simple, functional (i.e., side-effect free) and there is no real interaction with the object-oriented part of the language. This means that the semantics also poses no real challenges. The semantics is given as reduction steps over configurations given as:

\[
\Gamma \vdash e \tag{C.3}
\]

An expression \(e\) evaluates to a value (= an evaluated expression), if the computation exists. The evaluation will be deterministic. The steps are as specified by the rules given in Table C.8.

The right-hand side of each rule has the form of the let-expression determines the choice of rule.

The first rule \(\text{R-SEQ}\) simply restructures a nested occurrence of let and corresponds thus to associativity of sequential composition/the let-construct (in terms of the simpler \(\vdash\)-construct, the rule expresses that \((e_1; e_2); e\) can be restructured to \(e_1'; (e_2'; e))\). Rule \(\text{R-LET}\) deals with an evaluated expression where the value \(x\) is substituted by the value \(v\) in the rest of the expression. The next two rules deal with declarations, the first one with the function declarations and the second one with data type declarations. In both cases, the context \(\Gamma\) is extended appropriately in the step. In the first case by the name of the function, in the second case by the name name of the data type and its kind \(*\) plus type information for the constructors. By extending the context, we assume that the additional bindings are new, which assured that constructor names are globally unique\(^5\).

The two \(\text{R-CASE}\)-rules deal with the case-construct. The case expression is evaluated against the guards from a list of branches. The branches are evaluated according to a first-match strategy, which makes the evaluation deterministic (see also Remark C.3.2). So if there is more than one branch, there are two cases possible, based on the first branch. Either, the pattern match-expression \(e\) matches with the pattern \(p\) guarding the branch (written \(e \leq \sigma p\), meaning \(e = p\sigma\) with \(\sigma\) the matching substitution). In that case the branch is selected and evaluation continues with the “body” of the branch, with the matching substitution \(\sigma\) applied. Otherwise, with no match, the branch is discarded and the rest of the branch list is tried for a match (see rule \(\text{R-CASE}_2\)). In case the branch list is empty (for instance after exhausting unsuccessfully all branches without a match), no rule applies and the evaluation deadlock.

\(^5\)Note that constructor names are assumed to be “constant” identifiers. That is different from local variables and function names, which are also identifiers, but they are interpreted up-to alphabetic renaming. Thus, global uniqueness of local variables and function names is not enforced. As a further aside: at the current stage of the surface language, we do not support nested local scopes of function definitions, all data is declared globally. The type system can later be easily used when the surface language is extended.
\[
\Gamma \vdash \text{let } x : T = (\text{let } x' : T' = e'_1 \text{ in } e_2) \text{ in } e \rightarrow \Gamma \vdash \text{let } x' : T' = e'_1 \text{ in } (\text{let } x : T = e_2 \text{ in } e)
\]

(R-LET)
\[
\Gamma \vdash \text{let } x : T = v \text{ in } e \rightarrow \Gamma \vdash e[v/\lambda x]
\]

(R-FDEC)
\[
\Gamma \vdash \text{let } f : T' = \lambda (x : T). e_1 \text{ in } e_2 \rightarrow \Gamma, f : T' \vdash e_2
\]

(R-DDEC)
\[
\frac{\Gamma = \Gamma, D : \alpha, \ldots ; \text{Co}_1 : T_1 \rightarrow \ldots \rightarrow \text{Co}_n : T_n \text{ in } e \rightarrow \Gamma' \vdash e}{\Gamma \vdash \text{let data } D = \text{Co}_1(T_1) \ldots \text{Co}_n(T_n) \text{ in } e \rightarrow \Gamma' \vdash e}
\]

(R-COND)
\[
\frac{\Gamma \vdash \text{let } x : T = (\text{if } v = v_1 \text{ else } e_2) \text{ in } e \rightarrow \Gamma \vdash \text{let } x : T = e_1 \text{ in } e}{\text{Cond1}}
\]
\[
\frac{v_1 \neq v_2}{\Gamma \vdash \text{let } x : T = (\text{if } v_1 = v_2 \text{ else } e_1 \text{ else } e_2) \text{ in } e \rightarrow \Gamma \vdash \text{let } x : T = e_1 \text{ in } e}{\text{Cond2}}
\]

(R-APP)
\[
\frac{\Gamma = \Gamma_1, f : T = \lambda (y : T). e_1, \Gamma_2}{\Gamma \vdash \text{let } x : T = f(e_2) \text{ in } e \rightarrow \Gamma \vdash \text{let } x : T = e_1[y/e_2] \text{ in } e}
\]

| Table C.8: Semantics. |

**Remark C.3.1** (Recursion). The semantics is defined as a (small-step) operational semantics of configurations as given in equation (C.3). This gives a semantics in terms of (top-most) rewriting rules on the level of expression. It is not presented in terms of a pure substitution-based semantics, i.e., formulated without the extra \( \Gamma \)-part of the configuration, but just rewriting expressions. The reason for that is, that the representation allows an easy representation of recursive functions. The alternative is possible too and would require availability of some “Y-combinator” to allow recursion.

**Remark C.3.2** (First-match). The two R-CASE-rules of Table C.8 implement a first-match strategy. That was chosen for sake of simplicity, as it renders the evaluation deterministic. An alternative may be a non-deterministic strategy.

**Lemma C.3.3** (Subject reduction). If \( \Gamma \vdash e : T \) and \( \Gamma \vdash e \rightarrow \Gamma' \vdash e' \), then \( \Gamma' \vdash e' : T \).

The following lemma states formally that the evaluation order of ABS expressions does not matter:

**Lemma C.3.4** (Determinism). If \( \Gamma \vdash e \rightarrow \Gamma_1 \vdash e_1 \) and \( \Gamma \vdash e \rightarrow \Gamma_2 \vdash e_2 \), then \( \Gamma_1 \vdash e_1 \) is identical to \( \Gamma_2 \vdash e_2 \).
C.4 Predefined types

The data type language allows user-defined data types to be introduced. It is convenient to have a collection of standard ones plus their operations predefined. The examples we show in Appendix and at various different places contain a number of those (partly in concrete syntax). They include Boolean, natural numbers, lists, etc.
Appendix D

The Concurrent Object-Oriented Language

In this part, we describe the object-oriented, concurrent part of the language. We start with the type system before we come to the reduction semantics.

D.1 The Type System

The ABS language uses a nominal type system, where object references are typed by (names of) class interfaces (but not by class names, as for instance in Java). Note that interfaces provide a hiding mechanism for ABS. Interfaces only export methods, but not fields. A class may implement many interfaces, exporting different subsets of its methods. There is no other hiding mechanism in ABS (i.e., no qualifiers like public or private). Furthermore, the language supports nominal subtyping or subtype polymorphism on interfaces. Figure D.1 repeats the available types $T$.

$$ T ::= I \mid D \mid \text{Fut}(T) \mid \text{Unit} \mid \text{Bool} \mid \text{Guard} $$

Figure D.1: Types.

The type system (including the part for subtyping) uses typing contexts $\Gamma$. They serve as a finite mapping from identifiers/names to (mainly) their types. Considering $\Gamma$ as a finite mapping, we use $\Gamma(x) = T$ for looking up the type binding for $x$; i.e., if $\Gamma$ contains the pair $x:T$. Similarly for other bindings in $\Gamma$.

D.1.1 Subtyping

The type system, described in Section D.1.2, uses a subtype relation as subsidiary statement. We write

$$ \vdash T \preceq T' $$

for the subtyping relation, “$T$ is a subtype of $T'$”, resp. “$T'$ is a supertype of $T$”. Subtyping is a partial order on types; i.e., it is reflexive, transitive, and anti-symmetric. Its definition is shown in Figure D.2. Observe that future types are covariant in their type parameter. Otherwise the subtyping relation is generated by the extends- and implements-declarations on classes and interfaces (which we assume to be acyclic). We write $T \prec T'$ if $T \preceq T'$ and $T \neq T'$ (“$T$ is a proper subtype of $T'$”).

Remark D.1.1 (Subtyping relation). As mentioned, the $\preceq$-relation is a partial order on types. Interfaces are the types for objects, so object references (plus variables containing those references etc.) are typed by interfaces. For technical reasons it is advantageous when dealing with subtyping that, if a program entity is typed at all, there is a minimal type. In the absence of intersection types (following a nominal (sub)-typing discipline), we assume that each class $C$ has an interface $I_C$ capturing the exact method signature of $C$. Thus, $I_C$ is the full type of $C$, making all methods of $C$ available to the environment.
D.1.2 Typing

The type system for the imperative part is given in Figure D.3.

Declarations. An ABS program is typed by typing all its components; i.e., the data declarations, class declarations, interface declarations, and the optional main-block (see T-PROG). Judgments of the form \( \vdash \text{De} : \text{ok} \), \( \vdash \text{F} : \text{ok} \), etc. stipulate well-formedness of data declarations, function definitions, etc. As they are straightforward and concentrating on the imperative part here, we do not formalize them in this section but refer to the part about the data types (Chapter C). Interfaces are typed by typing their method signatures (see T-INTF). Classes are typed by typing the optional init-block and the method declarations under the type environment that types \( \text{this} \) to the interface type of the class (see T-CLASS). Methods are typed by typing their body-block under a type environment that maps the formal parameters to their declared types. In addition, the type of the body-block has to be the type of the declared return type of the method (see T-METHOD). The type of a block is the type of its statement, which is typed under the type environment that maps the local variables to their declared types (see T-BLOCK).

Expressions. Variables are typed as expected by looking up the corresponding binding in \( \Gamma \) (see T-VAR). A field lookup is typed by first determining the type of \( \text{this} \) and then using this type to find the type of the field declaration, using the \texttt{ftype} function (see T-FIELD). The \texttt{null} expression can be typed to any interface type (see T-NULL). Rule T-New deals with instantiation, i.e., with expressions of the form \texttt{new}\[\text{cog \ C(\overline{\sigma})}\]. Instantiation gives a new object, which is typed by \( \text{I}_C \). The judgment \( \vdash C : \overline{\text{F}} \rightarrow \text{I}_C \) determines the “type” of a class, which are the types of its constructor parameters and the class interface. Giving back a reference to the new object, the new-expression is typed by the classes minimal interface \( \text{I}_C \) of the class.

Asynchronous calls are typed by typing the corresponding synchronous call and returning the future type of the corresponding return type (see T-ASYNCCALL). Synchronous calls are typed by typing the receiver and the argument expressions. These types must then match the types of the corresponding method declaration, looked up by function \texttt{mtype} (see T-SYNCALL). The \texttt{get} operation can only be applied to a future, which has to be of some type \( \text{Fut}(T) \). The result is the value of the future, thus type \( T \) (see T-GET).

Statements. The assignment statement is handled in rule T-ASSIGN in an obvious manner: both the variable assigned to as well as the expression on the right-hand side need to be well-typed with the same type. The immediate type of the variable must be used, for soundness. Being a statement, the type of the assignment itself is Unit. The await statement expects a guard typed to Guard as argument and is then typed to Unit (see T-AWAIT). The skip and suspend statements have no premises and are simply typed to Unit (see rules T-SKIP and T-SUSPEND). The if, while, and sequence statements are typed in the obvious manner. Conditions have to be of type Bool; sub-statements must be typeable to some type. The sequence statement \( s; s' \) is typed to the type of \( s' \) (see T-SEQ). Note that expressions can also be statements. Thus the type of \( s' \) may be different than Unit. This is used to obtain the type of a block; i.e., it is the type of the last statement (see T-BLOCK).
Guards. A guard v? is only typed if v is a future type (see T-GUARDFUT). A conjunction of guards requires that the operands are guards (see T-GUARDCONJ). A functional expression can be used as a guard if it is of type Bool (see T-GUARDFUN).

Subsumption. Subsumption is a standard property of type systems with subtyping, and connects the typing judgment with the subtyping judgment: a statement of T' is also of a larger type T, i.e., where T ⊇ T' (see rule T-SUB).

\[
\begin{align*}
\Gamma, x : T &\vdash s : T' & \text{(T-Block)} \\
\Gamma, x : T &\vdash s : T' & \text{(T-Var)} \\
\Gamma \vdash \{ T \bar{x} s \} : T & & \text{(T-Class)} \\
\Gamma \vdash \text{class } C(\ell) &\text{ implements } \bar{T} \{ T' \bar{f} \bar{B} \bar{M} \} : ok & \text{(T-Method)} \\
\Gamma, x : T &\vdash f : T & \text{(T-Field)} \\
\Gamma \vdash e : I & & \text{(T-SyncCall)} \\
\Gamma \vdash e : \text{Fut}(T) & & \text{(T-AsyncCall)} \\
\Gamma \vdash \text{new } \text{proxy } C(\bar{c}) : \ell_C & & \text{(T-New)} \\
\Gamma \vdash e : \text{Fut}(T) & & \text{(T-Get)} \\
\Gamma \vdash e : T & & \text{(T-Assign)} \\
\Gamma \vdash v : T & & \text{(T-Assign)} \\
\Gamma \vdash e : \text{Unit} & & \text{(T-Assign)} \\
\Gamma \vdash e : \text{Unit} & & \text{(T-Assign)} \\
\Gamma \vdash s : T & & \text{(T-Assign)} \\
\Gamma \vdash s : T & & \text{(T-Assign)} \\
\Gamma \vdash s' : T & & \text{(T-Assign)} \\
\Gamma \vdash s' : T & & \text{(T-Assign)} \\
\Gamma \vdash v : \text{Fut}(T) & & \text{(T-Assign)} \\
\Gamma \vdash v : \text{Guard} & & \text{(T-Assign)} \\
\Gamma \vdash g : \text{Guard} & & \text{(T-Assign)} \\
\Gamma \vdash g' : \text{Guard} & & \text{(T-Assign)} \\
\Gamma \vdash g \land g' : \text{Guard} & & \text{(T-Assign)} \\
\end{align*}
\]

Figure D.3: The type system.

D.2 The Operational Semantics

The operational semantics is defined by reduction rules on configurations (see below). The semantics will be defined in structural operational manner. The configurations contain the code being executed, the heap with the instantiated objects, and a representation of the concurrent object groups (or groups for short in the sequel). We represent the
configuration by the “parallel composition” of those mentioned entities. The binary operation for parallel composition is \textit{associative} and \textit{commutative}. In the terminology of rewriting theory as implemented in Maude, the reduction relation is interpreted modulo AC (or associative and commutative equations).

At runtime, a program looks as follows:

\[
P \ ::= \ o[b,C,\sigma] \quad \text{object } o
\mid n(b,o,s,\sigma) \quad \text{task } n
\mid b[l] \quad \text{lock } l \text{ for a concurrent object group } b
\mid P \parallel P \quad \text{composition}.
\]

An object

\[
o[b,C,\sigma]
\]

has an identity \(o\), contains information about its group \(b\), its class \(C\) (to look up the methods), and an instance state, which gives values to the object fields. Tasks of the form

\[
n(b,o,\sigma,s)
\]

are the active parts in a configuration/program; i.e., they represent statements \(s\) under execution. A task named \(n\) has information about its group \(b\) and about the object \(o\) in which it is currently executing. The information about the group is needed for lock-manipulation and also to decide whether a synchronous method call is allowed (as reentrance is only allowed \textit{inside} a group). Finally, the task contains a local state which keeps the values of the local variables. One task is one method body under execution, and when the task terminates, the method has terminated and ready to “make available” its return value (if any).

\textbf{Remark D.2.1 (Tasks, methods, and futures).} The representation “one method activation, one task” is done uniformly for methods which are called synchronously and for those called asynchronously. The difference between the two cases is whether the calling task can proceed independently after the call —the asynchronous case— or blocks waiting for the result —the synchronous case. See also the rules \textit{R-ACall vs. R-SCall} in the operational semantics below.

The blocking in the synchronous case means that the callee’s activation must finish and return the result before the calling task/method can continue; in other words, the tasks running “in parallel” in the configuration actually form the call stack in this case. So even if all the method activation records have different task identities, they conceptually all are part of the same single thread of control. In the representation of equation (D.2), such a thread containing the call stack of synchronous method calls (inside a group) has no identity of its own (only the identities of the single tasks exist). Since the model does not support reentrance at the level of components/object groups, this identity is not needed; it would be needed to determine whether or not an activity applying for a lock already owns it and thus would be allowed to reenter the group.

In the case of an asynchronous call, the caller task continues executing (and blocks only if/when it claims the result later). To be able to do so one needs a mechanism by which the caller can refer to that result, a place holder for that eventual return value. This place holder is known as a future and the reference or identity referring to the future is the future reference. It is natural in our representation, to use the identity of the task that is responsible to calculate the result as the handle to the result. In other words: the task identifier \(n\) in a task \(n(b,o,\sigma,s)\) is a future reference.

\textbf{Remark D.2.1.}\n
The last basic entity of (the changeable part of) the configuration represents a concurrent object group. This is done by a \textit{lock} of the form

\[
b[l]
\]

has an identity \(b\). The group is the unit of concurrency: at each point in time, there is only one task active in the group and the groups shares a common scheduling strategy with cooperative (and non-preemptive) scheduling. At the level of the operational semantics, we leave the scheduling unspecified; i.e., the scheduling is non-deterministic. This mutual exclusion among the activities in the group is specified here by a simple lock, which all “members” of the group share. So the object group realizes a \textit{monitor}. Unlike the monitors in the multi-threading concurrency model of Java, we do not support re-entrant monitor calls: communication between monitors is done via asynchronous message passing. Without reentrance, locks \(l\) are binary and can take 2 states, \(\bot\) when it is free and \(\top\) when not. The lock is used to assure mutual exclusion at the level of a concurrent object group. Upon instantiation, the lock is free.
Remark D.2.2 (Concurrency model). The concurrency model with concurrent object groups generalizes both the multi-threading concurrency model of Java and the concurrent object model of Creol. The concurrent object groups are sets of objects which are closely collaborating and which share a common scheduler/message queue. Inside such a group, synchronous method calls are supported, i.e., basically the multi-threaded concurrency model of Java of concurrent threads which share access to the instances. Each thread corresponds to the call stack of its (synchronous) method invocations. Inside an object group, however, there is no “locking” as is done in Java using synchronized methods. In ABS, all methods are synchronized by default and the area of protection is the object group. Inside the group there is only one activity at a time, with cooperative scheduling (using the await and suspend statements). Inside the group, with synchronous method calls, (mutual) recursion, i.e., call-backs are allowed. However, inter-group communication is based on asynchronous calls, and no monitor reentrance (synchronous call-backs) is possible. More precisely: a task in one group calling an object in another group causes a task in that other group. Of course that activity may “call back” to objects in the original group. However, being asynchronous, the message back does not belong to the same thread, and so, even if the original task holds the lock of the group, this gives the call-back no privilege to re-enter the monitor. Indeed, the locks associated with each object groups are binary locks and these support no reentrance. Note further that the identity of a task here and the identity of a thread in Java play rather different roles, even if both identify an “activity”. In Java, the thread identity can be used for thread communication; i.e., one thread can notify another, kill another thread etc, and using the currentThread-keyword, a thread can inspect its own identity. In our model and lacking a corresponding keyword, a task cannot find its own identity and cannot communicate in the described way with other tasks. In the case of synchronous method calls, the identity of the task exists at the runtime level, only, and is not reflected at the programming level. For asynchronous method calls, the identity of the new task does play an important role for the caller, in that it is the handle to the eventual result (the future reference). In a way, an asynchronous method call between object groups here and a spawning of a new thread in Java (by instantiating a Runnable object and starting the activity) are similar, but the way of interacting with the new activity is different.

Object groups generalize the concurrency model of Creol in that groups of collaborating objects are protected by a common cooperative scheduling scheme. In Creol, the protected domain is confined to single objects.

In summary: when attaching a concurrent object group to each object; i.e., when using only new cog to instantiate a new object, the concurrency model corresponds to that of Creol. When using only one global object group, i.e., using only newC to instantiate new objects, the concurrency model roughly corresponds to Java (plus the possibility to support asynchronous method calls without reentrant monitor calls, but without synchronized modifier for synchronous calls).

In order to formulate the semantics we will make use of a few pieces of auxiliary syntax or runtime syntax. They are introduced to specify the reduction rules, but are not to be used at program level. See also Figure D.4.

\[
s \ ::= \ldots | \text{grab}(b) | \text{release}(b) | \text{let } T = s \text{ in } s
\]

Figure D.4: Runtime syntax

The first two statements grab and release are used for lock handling, in particular to define the behavior of the suspend and the await statements. As argument, they take the group identity/the shared lock. The let construct is introduced mainly for technical reasons as a means to specify the semantics. It is the same construct we used in the data type part of Chapter C: it introduces local variables that can be used to temporarily store values without interfering with other tasks and can be specified in a simple manner (which is substitution-based and purely functional). Secondly, it specifies sequentiality. In the functional part without any side effects, that property is not so important, as, due to confluence, different orders of reduction do not play an important role. With side effects and statements, as here, the order (within one task or thread of execution) can, of course, not be left to a non-deterministic rewriting strategy (for purely functional expressions it could).

Clearly, the reduction relation must assure that in a statement \(s_1; s_2\), the statement \(s_1\) is reduced before \(s_2\). That in
general is simple when defining an SOS, the rules and redexes are just defined in a way to apply to the “left-most”
piece of syntax. Problematic are nested constructs, as with arbitrarily nested syntax, one cannot directly specify in a
plain reduction rule which the next redex should be. That typically is the case when having some form of “expression”
and it becomes problematic if the expressions may involve side-effects. For instance in a method call $e_1.m(e_2,e_3)$,
one should specify the order of reduction of $e_1$, $e_2$, and $e_3$ (and recursively nested deeper inside the $e_i$’s), especially if
these expressions involve side-effects. There are different ways to deal with this. One way of specifying such order of
evaluations is to use of so-called evaluation or reduction contexts $\sigma$. Evaluation contexts are expressions/statements,
etc., “with a hole”. By defining the structure of such a context, one can make the choice of redex more specific, in
particular, rendering it deterministic (per thread); in the above expression, for instance, fixing a left-to-right reduction.

A different approach is to avoid nesting and make the evaluation order explicit. This avoids reduction contexts at the
expense of making the (abstract) syntax less compact. This is the way we specify the reduction rules here, and we
use the let construct to do that. For instance, a nested expression such as $e_1.m(f(e_2))$ (where $f$ is a function name)
can be expanded to

$$\text{let } x_1 = e_1 \text{ in (let } x_2 = e_2 \text{ in (let } x_3 = f(x_2) \text{ in } x_1, m(x_3)) \text{).}$$

(D.6)

In this way we have specified, that $e_1$ is evaluated before $e_2$, for instance. This transformation is standard and not
made more explicit in this document. It is related to the well-known transformation into continuation passing style
(CPS). A further advantage of using this representation is that it allows us to specify the functional, data-type part
largely independent from the object-oriented, imperative part. For instance, when writing a function application\this.f + this.f (here a “+”) which refers to the local instance state by mentioning the fields $f_1$ and $f_2$, then the
functional reductions would need to be defined relative to the instance state of an object. By using the local let variable
and forcing that the mutable state, for example, the instance state is copied into the local “functional” state space, we
can better decouple the functional and the imperative part and define them independently. A further remark may be in
order: avoiding reduction contexts at the price of a more explicit syntax is advantageous for realizing the semantics
in an executable manner in the rewriting engine Maude or in symbolic execution [8], i.e., the implementation does
not use reduction contexts. However, as other aspects of the language definition here, the representation using let
constructs is meant as the specification of the semantics. In the concrete rewriting implementation, we chose more
compact ways for instance of representing the state (avoiding to represent it as part of the “syntax”).

Now to the reduction rules. They are shown in Figure D.5 (dealing with object-local statements and expressions)
and Figure D.6 (for statements and expressions dealing with cooperation, message passing, etc.). The first rules of
Figures D.5 deal with sequential composition (represented here by the let construct). If a statement/expression has
terminated, i.e., evaluated to a value, the reduction step in rule R-RED substitutes the variable by the value in the rest
of the statement. Note that the step has no side effect on the local state, nor does it change the instance state of an
object, nor is the lock involved. The same applies to the step of rule R-SEQ which deals with sequential composition
(represented here by the let construct). Executing a skip statement or a pure expression statement has no effect, the
control is just transferred to the remainder of the statement (see rules R-Skip, R-Pure). The rules for conditionals
work in the expected way: depending of the Boolean condition in the if construct, either the left or the right branch is
taken (see rules R-Cond1 and R-Cond2). Similarly, whether the body of a while-loop is entered or skipped depends
on the loop condition (see the R-While rules).

An assignment $x := v$ of a value $v$ to a local variable $x$ simply updates the local state $\sigma$ of the task (see rule R-Assign).
An assignment has a side effect. The next two rules deal with reading and writing to instance variables/fields of an object.
Remember that we disallow that one object directly changes the fields of another object; i.e., each method can access the fields only of its own instance. This is achieved in that in the syntax, fields can be addressed only qualified as this.$f$ (see Figure B.1), where this is a reserved local variable for each method. The syntax this.$f$ is used for the definition of methods in classes, only, i.e., for the static code. Upon method call, when the method body is activated as a task, the local variable this is substituted by the object identity of the caller (see the call rules in Figure D.6). This means, at runtime in the rules R-lookup and R-FUpdate, the expression this.$f$ becomes o.$f$. Looking up a field means simply to copy the respective value $\sigma(f)$ from the object into the local state space of the task, using the let variable $y$ in rule R-lookup. Field update works inversely, copying a value from the task-local state space to the heap, where $\sigma(f \rightarrow v)$ denotes this update (see rule R-FUpdate).

Now to the rules of Figure D.6 dealing with message passing and task handling. The first two rules in this table
n(b, o, σ, let z: T = v) \leadsto n(b, o, σ, s[v/z])

(R-SEQ)

n(b, o, σ, let z2: T2 = (let z1: T1 = s1 ins) ins') \leadsto n(b, o, σ, let z1: T1 = s1 in (let z2: T2 = s ins'))

(R-SKIP)

n(b, o, σ, skip; s) \leadsto n(b, o, σ, s)

(R-PURE)

n(b, o, σ, let z: T = x ins) \leadsto n(b, o, σ, let z: T = σ(x) ins)

(R-COND1)

n(b, o, σ, let z: T = (if true then s1 else s2) ins) \leadsto n(b, o, σ, let z: T = s1 ins)

(R-COND2)

n(b, o, σ, let z: T = (if false then s1 else s2) ins) \leadsto n(b, o, σ, let z: T = s2 ins)

(R-COND3)

n(b, o, σ, let z: T = (if v = v then s1 else s2) ins) \leadsto n(b, o, σ, let z: T = s1 ins)

(R-COND4)

v1 \neq v2

n(b, o, σ, let z: T = (if v1 = v2 then s1 else s2) ins) \leadsto n(b, o, σ, let z: T = s2 ins)

(R-WHILE1)

n(b, o, σ, let z: T = (while true do s1) ins) \leadsto n(b, o, σ, let z': T = s1 in (let z: T = (while true do s1) ins))

(R-WHILE2)

n(b, o, σ, let z: T = (while false do s1) ins) \leadsto n(b, o, σ, s)

(R-ASSIGN)

n(b, o, σ, x := v; s) \leadsto n(b, o, σ[x \mapsto v], s)

(R-LOOKUP)

o[b, C, σ] \parallel n(b, o, σ', let z: T = o.f ins) \leadsto o[b, C, σ] \parallel n(b, o, σ', let z: T = σ(f) ins)

(R-FUPDATE)

o[b, C, σ] \parallel n(b, o, σ', o.f := v; s) \leadsto o[b, C, σ[f \mapsto v]] \parallel n(b, o, σ', s)

Figure D.5: Reduction rules (1).
deal with method calls, synchronous and asynchronous. In both cases, a new task is created, and the main difference
is whether the calling task can continue executing or not (see Remark D.2.1). Rule R-SCALL deals with synchronous
method calls. Remember that synchronous method calls are only allowed inside a group, not between concurrent
object groups. This means that the callee object and the task that issues the call (in the rule n) must belong to the same
group (i.e., they share the same lock, in this representation); this is specified in the rules in that both the callee o and
the task n refer to the same lock b in their configuration. The call spawns a new task n’ in the rule with a fresh identity.
The task is dedicated to execute the body of method m, more precisely the body of the method where the \( \text{thia} \) (which
is a reserved local variable for each method) is replaced by the identity o’. The calling activity does not block as before.

Asynchronous calls in rule R-ACALL are treated similarly; the differences are as follows. First of all, there are
no restrictions on the group affiliation of the callee object o’: asynchronous calls can address any object whether it
belongs to the group of the calling task/object or not. Furthermore, the calling activity does not block as before.
This means, in the reduction rule, the call is simply replaced by the future reference n’ (without an immediate get).

Another crucial difference concerns the new task named n’: As before (and always), the task is responsible to execute
the corresponding method body (after the appropriate argument passing, as before). The old task may continue
independently; however, the new task cannot just start executing. As discussed earlier, an object group does not only
assemble a number of objects that can closely collaborate, it also provides a domain of concurrency control in that at
most one thread is active per group (or that there is a common scheduler per such group). The access to such groups
is regulated by the lock entities b[l], where \( l \in \{\perp, \top\} \). There are two statements that manipulate the lock: grab tries
to acquire a lock, and release releases a held lock. Neither operation is part of the user syntax, they are injected by the
rewriting rules at runtime (see Figure D.4). See also Remark D.2.4.

Remark D.2.3 (Future references and synchronous method calls). We use a uniform representation for tasks executing
methods that have been called synchronously and for those called asynchronously, and we use the task identity as
future reference (see Remark D.2.1). The latter is relevant, in some sense, for asynchronous calls only, as only there
one can speak of a reference to a value which is computed only later in the future. In the synchronous case, with
the caller blocking there is no actual need for such a future reference. Note that the semantics, especially R-SCALL,
assures that even if the result-passing is done using the task identity, the calling and blocked task cannot use that
identity for other purposes than getting back the result. Especially and unlike asynchronous calls, it cannot store the
future reference and pass it on to others. For asynchronous calls, the task identity has a user-level function (known as
future reference). In contrast, in the synchronous case the user cannot (and should not) make use of the identity of the
stack frame.

The next two rules R-NEWO and R-NEWOG deal with object creation. There are two ways to instantiate an
object, both by using the new command on a class name/constructor. The difference is how the newly instantiated
object relates to the structure of object groups. In the standard object instantiation using newC, the newly created
object belongs to the object group of its creator; in R-NEWO, the newly created object o’ “inherits” the group identifier
b from the instantiating task n. This is different when creating a new object with new cog in rule R-NEWOG: a new
object is created as in NEWO and at the same time a new object group, represented by b’[\top]. There is another crucial
difference between the two ways of instantiating an object: in the first case, the instantiation is synchronous, i.e., the
instantiator blocks until the initialization has terminated. In rule R-NEWO, a new task n’ is created which executes
the initializer block. In the premise, the instance state \( \sigma_{\text{init}} \) and the local state \( \sigma_{\text{init}}' \) have the respective variables/fields
initialized appropriately by default values or by corresponding expressions. Note that the return value of the initializer
block is the new object identity o’ (at the end of s_task). This value is handed back (after termination of the initializer)
to the creating thread via n’.get.

In the second case of object creation in rule R-NEWOG, the instantiation is asynchronous in that the creator
need not wait until the initialization of the new object is done. There is a caveat to that: to avoid interaction with a
partially initialized object, the lock of the newly created object group is initially taken and not free, and furthermore
the initializer code of the class does not need to apply for the lock as other methods, it holds it initially. Of course,
\[
\begin{align*}
(n' \text{ fresh}) & \quad \text{body}(m,C) = s(x) \quad s_{task} = (\text{let } z':T = s'[0'/this][\overline{v}/\overline{x}] \text{ in } z') \\
& \quad o'[b,C,\sigma'] \parallel n(b,o,\sigma,\text{let } z:T = o'm(\overline{v}) \text{ in } s_2) \rightsquigarrow o'[b,C,\sigma'] \parallel n(b,o,\sigma,\text{let } z:T = n' \text{ get } s_2) \parallel n'(b,o',\sigma_{init},s_{task}) \\
(n' \text{ fresh}) & \quad \text{body}(m,C) = s(x) \quad s_{task} = (\text{let } z':T = \text{grab } (b);s[0'/this][\overline{v}/\overline{x}] \text{ in } \text{release}(b);z') \\
& \quad o'[b',C,\sigma'] \parallel n(b,o,\sigma,\text{let } z:T = o'm(\overline{v}) \text{ in } s_2) \rightsquigarrow o'[b',C,\sigma'] \parallel n(b,o,\sigma,\text{let } z:T = n' \text{ get } s_2) \parallel n'(b,o',\sigma_{init},s_{task}) \\
& \quad o' \text{ and } n' \text{ fresh} \quad B = \{T \overrightarrow{f} s'\} \text{ is initializer block of } C \quad s_{task} = (s'[0'/this];o') \\
& \quad n(b,o,\sigma,\text{let } z:T = \text{new } C(\overline{v}) \text{ in } s) \rightsquigarrow n'(b,o',\sigma'_{init},s_{task}) \parallel o'[b',C,\sigma'_{init}] \parallel n(b,o,\sigma,\text{let } z:T = o' \text{ in } s) \\
& \quad b',o',\text{ and } n' \text{ fresh} \quad B = \{T \overrightarrow{f} s'\} \text{ is initializer block of } C \quad s_{task} = (s'[0'/this];\text{release}(b')) \\
& \quad n(b,o,\sigma,\text{let } z:T = \text{new } C(\overline{v}) \text{ in } s) \rightsquigarrow b'[\overrightarrow{f}] \parallel n'(b,o',\sigma'_{init},s_{task}) \parallel o'[b',C,\sigma_{init}] \parallel n(b,o,\sigma,\text{let } z:T = o' \text{ in } s) \\
(R-\text{GRAB}) & \quad b[\bot] \parallel n(b,o,\sigma,\text{grab } (b);s) \rightarrow b[\top] \parallel n(b,o,\sigma,s) \\
(R-\text{RELEASE}) & \quad b[\top] \parallel n(b,o,\sigma,\text{release } (b);s) \rightarrow b[\bot] \parallel n(b,o,\sigma,s) \\
(R-\text{SUSPEND}) & \quad n(b,o,\sigma,\text{suspend };s) \rightsquigarrow n(b,o,\sigma,\text{release } (b);\text{grab } (b);s) \\
&(R-\text{GET}) \quad n_1(b',o',\sigma',\text{let } x:T = n_2.\text{get } \text{int}) \parallel n_2(b,o,\sigma,v) \rightarrow n_1(b',o',\sigma',\text{let } x:T = \text{vin} x) \parallel n_2(b,o,\sigma,v) \\
(R-\text{AWAIT}_1) & \quad \text{fnames}(g) = n_1 \ldots n_k \text{ with } k \geq 1 \\
& \quad n_1(b_1,o_1,\sigma_1,v_1) \parallel \ldots \parallel n_k(b_k,o_k,\sigma_k,v_k) \parallel n(b,o,\sigma,\text{await } (g);s) \rightsquigarrow n_1(b_1,o_1,\sigma_1,v_1) \parallel \ldots \parallel n_k(b_k,o_k,\sigma_k,v_k) \parallel n(b,o,\sigma,\text{release } (b);\text{await } (g)[\text{true}/\overline{?}];s) \\
(R-\text{AWAIT}_2) & \quad n' \in \text{fnames}(g) \quad s' \neq v \\
& \quad n'(b',o',\sigma',s') \parallel n(b,o,\sigma,\text{await } (g);s) \rightsquigarrow n'(b',o',\sigma',s') \parallel n(b,o,\sigma,\text{release } (b);\text{await } (g);s) \\
(R-\text{AWAIT}_1) & \quad \text{fnames}(g) = \emptyset \quad [g]_\sigma = \text{true} \\
& \quad n(b,o,\sigma,\text{await } (g);s) \rightsquigarrow n(b,o,\sigma,s) \\
(R-\text{AWAIT}_2) & \quad \text{fnames}(g) = \emptyset \quad [g]_\sigma = \text{false} \\
& \quad n(b,o,\sigma,\text{await } (g);s) \rightsquigarrow n(b,o,\sigma,\text{release } (b);\text{await } (g);s)
\end{align*}
\]
at the end it must release the lock before termination. The distinction between a synchronous instantiation inside an object group and an asynchronous instantiation for a new object group corresponds to the distinction for ordinary method calls: Inside an object group component, method calls are synchronous and between groups, information is exchanged by asynchronous message passing (using the future mechanism).

The two rules R-GRAB and R-RELEASE do the lock handling; i.e., they are responsible for assuring mutex (see also Remark D.2.4). They work very simple. Grabbing a lock requires that the lock is free (i.e., it is in state ⊥), and it changes the state to ⊤. Trying to grab a lock which is not free blocks the contender until the lock becomes free. Releasing the lock works dually in setting the lock back to ⊥.

Remark D.2.4 (Lock discipline and mutex). The cogs b[l] represent the object groups and regulate the access. The operations which manipulate these entities are the dual operations grab and release. To maintain mutual exclusion as invariant, it is crucial that the use of those operations follow a strict discipline. Since they are not user-operations, at least the user cannot misuse them to destroy non-interference, for instance by releasing the lock (and let thus someone else grab it and start executing) while continue executing. In the rule for asynchronous method calls, the whole body is enclosed in a matching pair of grab and release. Furthermore, a suspend, which is a user operation, is a release immediately followed by a grab. Under this discipline one can prove non-interference.

Another invariant which is structurally assured is that release is wait-free, it never blocks: even if the rule R-RELEASE requires that the lock is taken in order to release it. It is an invariant that a release is never attempted on a free lock at runtime.

By using the suspend command —known also as yield— the user can introduce scheduling points, i.e., where the current task temporarily stops executing. In the semantics, it means that the task makes the lock free, giving other tasks the opportunity to be scheduled instead. To be precise, exactly one task, including the one just suspended, gets the chance to re-enter the monitor. So a suspend is nothing else than releasing the lock and immediately re-apply for it using grab (see rule R-SUSPEND).

Rule R-GET treats the get statement, where the requesting task (\(n_1\) in the rule) attempts to read the result from future reference \(n_2\). This reference \(n_2\) is at the same time the identity of the task that calculates the result (and the task corresponds to one method activation). If the result is not yet ready resp. the task not yet terminated, the requester needs to block and wait until (if ever) the result is available. In that case, the task \(n_2\) is not of the form \(n_2(\ldots,v)\) (as required by R-GET) or the statement is continue empty, when no value is returned and the method is of type Unit, in which case no rule applies and \(n_1\) is blocked. Otherwise, \(n_1\) copies back the result \(v\) into its local space. Note that in the case of a synchronous call, the entity \(n_2\) could be garbage collected (see Remark D.2.3): if \(n_2\) is the future reference to an asynchronous call, the value might not be garbage collected, as it could be read more than once (and by different tasks, when supporting first-class futures).

The await construct await \(g\) is central for synchronization. The \(g\) is a guard which, value-wise, corresponds to a Boolean. It is, however, more than just a Boolean in that guards can be used to introduce (conditional) scheduling points using await, namely conditional on the value of the guard; suspend, as mentioned, is an unconditional scheduling point.

A statement awaiting a condition stops executing until (if ever) the condition, i.e., the guard becomes true. When executing the await when the condition is false causes the executing task to suspend itself, to allow other potentially suspended tasks to acquire the lock and to progress. The task having executed the await and having suspended itself will need to recheck the guard in order to eventually proceed. To be able to do so, the task needs to acquire the lock to have safe access to the state. To appreciate the working of the await statement, we need to have a closer look at the form of the guards (see the corresponding line of the abstract syntax from Figure B.1). Apart from the (unproblematic) fact that guards can contain functional Boolean expressions, they are constructed with \(\land\). The expression part is unproblematic as the value of an expression does not depend on the state. The parts that do depend on the state are the basic polling guards of the form \(n^?\), which check whether a future has already been evaluated, and field access. If so, the guard is considered true, otherwise false. The important point here is that a guard \(n^?\) is monotonic in the following sense: once true, it remains true as a future value remains available once it becomes available. As the only constructor is \(\land\) —connectors like negation or implication are not allowed— composite guards are monotonic, as well.
This fact simplifies the way the *await* statement can be implemented. Remember that executing an *await* on a guard corresponding to false causes the executing task to suspend itself, i.e., to release the lock, to try the guard later again. If the guard happens to be true, the task may continue *after* having re-acquired the lock. Ideally, the check of the guard and, if positive, the taking of the lock is done *atomically*. The semantics does the guard-checking and the lock-grabbing, however, in two separate steps. The order in which the two steps are done is important. At first sight, the safe way to proceed is: 1) first take the lock and, if successful, 2) check the guard in the second step. If that check fails, suspend and try again. This behavior is sketched in Figure D.7(a) in the state where the activity is suspended, *first* the lock is re-taken and then afterwards checked (again) whether the guard evaluates to true, if not, the activity suspends again.

An alternative order of the two steps is shown in Figure D.7(b): trying to acquire the lock *after* checking whether the guard evaluates to true. This alternative may lead to an error: it may cause the task to continue with the guard actually being false (which is an error) if in the point between evaluating the guard and the attempt to acquire the lock the value of the guard may change from true to false. If, however, the guards behave *monotonically*, this cannot happen and the order described is safe. Furthermore it avoids looping back and re-applying for lock and this can be more efficiently implemented than the first solution. Guards are monotonic, if they do not refer to instance fields. Note in particular, that the polling expression *n?* trying to dereference a future is monotonic.

These considerations are reflected in the rules in the following way. In the R-AWAIT-rules in Figure D.6 we split the evaluation of a guard into two steps, corresponding to the monotone part dealing with the *n?*-parts of the guard, and afterwards, with the rest of the guard, in particular, the part that checks instance fields. The first part corresponds to the two alternatives R-AWAIT$^1$ and R-AWAIT$^2$: if in the first case all futures have terminated, then the corresponding parts of the guard are replaced by true, if not the task suspends itself. In the rules, *futnames*(g) extracts all the names for futures mentioned as *n?* expressions in g. In the second stage (after all the *n?*-guards have been replaced by true, the guard is evaluated again; the $\llbracket g \rrbracket_\sigma$ gives back the Boolean value of the guard, relative to the instance state $\sigma$. 

![Figure D.7: Await statement](image-url)
Appendix E

Example: A Peer-To-Peer Node in ABS

In this appendix we present the full details of the Peer-to-Peer system used as running example throughout this deliverable. We consider a peer-to-peer file sharing system which consists of nodes distributed across a network. Peers are equal: each node plays both the role of a server and of a client. The changes between these two roles illustrate the use of suspension points in the methods of the core ABS model. In a typical peer-to-peer network, nodes may appear and disappear dynamically. As a client, a node requests a file from a server in the network, and downloads it as a series of packet transmissions until the file download is complete. In the dynamic network, the connection to the server may be broken, in which case the download will automatically resume if the connection is reestablished. A client may run several downloads concurrently, at different speeds. A peer node in our model can thus be busy with a number of downloads and a number of uploads at the same time.

We assume that every node in the network has an associated database in which it stores its shared files. Downloaded files are stored in this database, which is modeled here in a rudimentary manner. However, the database model illustrates the use of the functional sub-language of ABS to deal with internal data structures.

E.1 The Imperative Model

E.1.1 Type synonyms

To clarify the presentation, we introduce the following type synonyms (type synonyms are not contained in the core ABS language since they can be implemented in the compiler).

```plaintext
type Filename = String;
type Filenames = Set<Filename>;
type Servername = String;
type Packet = String;
type File = List<Packet>;
type Catalog = List<Pair<Servername,Filenames>>;
```

E.1.2 The Database

We consider a very simple database system, in which a database object provides the following functionality:

- `getFile` returns a file from the database
- `getLength` returns the number of transmission packets of a given file
- `storeFile` saves a given file in the database
- `listFiles` returns a list of file names for files available from the database
We use the data type filename for file names and the data type file for files (see Section E.1.1 above). The database functionality is given by an interface DB, defined as follows:

```java
interface DB {
    File getFile(Filename fId);
    Int getLength(Filename fId);
    Unit storeFile(Filename fId, File file);
    Filenames listFiles();
}
```

In the implementation of the database, we use a map between file names and files to store the files. This map is of the type `Map<Filename,File>`, where `Map` is included in the ABS standard library in Appendix A, and `Filename` and `File` are defined in Section E.1.1 above. For simplicity, we let the class `DataBase` be parametric in its stored files (i.e., we can instantiate a database with several files directly). Database objects should support the DB interface, so the `DataBase` class will implement this interface. The methods in the `DataBase` class simply use functions defined over the map. The `DataBase` class is defined as follows (recall that the concrete ABS syntax has a `return` statement):

```java
class DataBase(Map<Filename,File> db) implements DB {
    File getFile(Filename fId) { return lookup(db, fId); } 
    Int getLength(Filename fId){ return length(lookup(db, fId)); } 
    Unit storeFile(Filename fId, File file) { db = insert(Pair(fId,file), db); } 
    Filenames listFiles() { return keys(db); } 
}
```

### E.1.3 The Peer Node

In the peer-to-peer network, a network client allows a user to request a specific file from a server and to get a catalog of all files available in the network. A network server, on the other hand, allows a client to inquire about available files from that particular server, to get the length of a specific file (i.e., the number of packets comprising the file), and to get a specific packet during the download of a file. A peer in the network provides both client and server functionality. In addition, a peer will share its neighbor servers. The `Client`, `Server`, and `Peer` interfaces can be specified as follows:

```java
interface Client {
    List<Pair<Server,Filenames>> availFiles(List<Server> sList);
    Unit reqFile(Server sId, Filename fId);
}
interface Server {
    Filenames inquire();
    Int getLength(Filename fId);
    Packet getPack(Filename fId, Int pNbr);
}
interface Peer extends Client, Server {
    List<Server> getNeighbors();
}
```

A node in the peer-to-peer network is an object which implements the `Peer` interface. The node takes a database of interface `DB` as a formal parameter, so a database may be private to a node or shared with other nodes. We assume that the node knows one other node in the network at creation time, from which it will get a list of neighbor servers. For simplicity, we capture the user’s interest by providing a file name as an explicit parameter; the node will try to download this file to its database. Observe that this is the active behavior of the node and is given by a method `run`. In the concrete syntax, `run` is a reserved method name used to designate active behavior. In the abstract syntax, this
**class** Node(DB db, Peer admin, Filename file) **implements** Peer {
  Catalog catalog;
  List<Server> myNeighbors;

  List<Server> getNeighbors() { return myNeighbors; }
  Server findServer(Filename fId, Catalog catalog) {
    if (isEmpty(catalog)) {
      return null;
    } else if (lookup(fId, snd(head(catalog)))) {
      return fst(head(catalog));
    } else {
      return findServer(fId, tail(catalog));
    }
  }

  Unit run() {
    Fut<Catalog> c; Fut<List<Server>> f; List<Server> newNeighbors; Server server;
    neighbors = Cons(admin, NilServer); f = admin!getNeighbors(); await f?; newNeighbors = f.get;
    neighbors = concatenate(neighbors, newNeighbors);
    c = this!availFiles(neighbors); // Asynchronous call
    await c?; // Allow other peers to call in the meantime
    catalog = c.get;
    server = findServer(file, catalog); // Find the server for the requested file
    if (server != null) {
      reqFile(server, file); // Download file
    }
  }

  Filenames inquire() {
    Fut<Filenames> f;
    f = db!listfiles(); await f?; return f.get;
  }

  Int getLength(Filename fId) {
    Fut<Int> length;
    length = db!getLength(fId); await length?; return length.get;
  }

  Packet getPack(Filename fId, Int pNbr) {
    File f; Fut<Packet> ff;
    ff = db!getFile(fId); await ff?;
    f = ff.get; return nth(f, pNbr);
  }

  Catalog availFiles (List<Server> sList) {
    Catalog cat; Fut<Filenames> fNames; Fut<Catalog> catList;
    if (sList == Nil) {
      cat = Nil;
    } else {
      fNames = head(sList)!inquire(); // Asynchronous call to the first server
      catList = this!availFiles(tail(sList)); // Asynchronous self-call with the tail of the list
      await fNames? & catList?; // Wait for both replies
      cat = appendright(catList, Pair(head(sList), fNames));
    }
    return cat;
  }

  Unit reqFile(Server sId, Filename fId) {
    File file; Packet pack; Int lth;
    Fut<Int> l1; Fut<Packet> l2;
    l1 = sId!getLength(fId); await l1?; lth = l1.get;
    while (lth > 0) {
      l2 = sId!getPack(fId, lth); await l2?;
      pack = l2.get; file = Cons(pack, file); lth = lth - 1;
    }
    db!storeFile(fId, file);
  }
}

**Figure E.1:** The Node class in core ABS
is transformed into a method call to the run method immediately after object initialization for instances of the class. The Node class is given in Figure E.1.

For file transfer between objects of the Node class, files are transferred as a series of packet transmissions. The model has some nice properties with respect to the loosely connected nodes of peer-to-peer networks:

- An object can do both file uploads and downloads;
- Many file transfers may occur at the same time;
- File transfers may have different speeds, depending on the network;
- A delay in one file transfer does not influence the others;
- The model offers automatic resumption of temporarily disabled network connections: if a server becomes unavailable, the file transfer from that server simply resumes later;
- Packet overtaking in the network is tolerated;
- The implementation does not use synchronous calls to exchange files, there is no active waiting and no deadlock in Peer objects;
- The implementation uses concurrent method calls: in availFiles, all inquire() calls to the servers in sList are initiated concurrently possibly before the replies are picked up. This results in increased concurrency and efficiency, compared to more conventional sequential solutions without asynchronous method calls.

### E.2 Variability in a Peer-To-Peer Node

A node in our peer-to-peer system searches for a given file in the network and requests the file from the first server found to offer the file, as described in the last two lines of the run() method in Figure E.1. In this section we propose two simple transformations using the techniques described in Chapter 5. First we introduce the possibility of filtering undesired files, according to a new method isValid. Second we introduce a basic parental control that filters searches for file names that contain the string “xxx”. We define these two delta modules below in the DML.

```plaintext
delta DFileFilter() {
modifies class Node {
    modifies Server findServer(Filename fId, Catalog catalog) {
        if (isValid(fId)) {
            original(fId, catalog);
        } else {
            return null;
        }
    }
    adds Bool isValid(Filename fId) {
        return True;
    }
}
}
delta DParentalControl() {
modifies class Node {
    modifies Bool isValid(Filename fId) {
        return (~isSubstring("xxx",fId));
    }
}
}
```
The function `isSubstring(String substr, String str)` used by DParentalControl searches for the `substr` inside `str`.

The delta DFileFilter introduces a new method `isValid` that is used to allow the search of only valid file names. In turn, the delta DParentalControl overrides this method by a new version that checks if the string "xxx" is contained in the file name. Hence, the latter delta can only be applied if the former is applied before. The choice of which delta modules should be applied is dealt with at the level of the feature model. We define the following feature model to describe the product line.

```plaintext
root Peer2Peer {
  group oneof {
    Basic,
    Extended {
      group [0..2] {
        FileFilter,
        ParentalControl { require: FileFilter; }
      }
    }
  }
}
```

The `Basic` feature represents the core module, while the `Extended` feature represents the program extended with our delta modules. The `FileFilter` and `ParentalControl` features are associated to the delta modules with the same name, and the constraints guarantee the dependency of the `ParentalControl` feature on the `FileFilter` feature. The constraints in the feature model do not impose any restrictions regarding the order of application of the two delta modules. This is done in the product line configuration. The delta modules and features in our example are connected via the following product line configuration.

```plaintext
productline Peer2Peer {
  features Basic, FileFilter, ParentalControl;
  delta DFileFilter when FileFilter;
  delta DParentalControl after DFileFilter when ParentalControl;
}
```

Consider the following product selections.

```plaintext
product P1 (Basic) {}
product P2 (FileFilter) {}
product P3 (ParentalControl) {}
product P4 (FileFilter, ParentalControl) {}
```

Each of these product selections produce a different peer-to-peer application, with the exception of `P3`—it will fail the satisfiability check with respect to the feature model. The product `P1` yields the core module, the product `P2` applies the delta module `DFileFilter` to the core module, and product `P4` applies the delta modules `DFileFilter` followed by `DParentalControl` to the core module.
Appendix F

JCoBox: Generalizing Active Objects to Concurrent Components

The paper “JCoBox: Generalizing Active Objects to Concurrent Components” [160] follows.
JCoBox: Generalizing Active Objects to Concurrent Components

Jan Schäfer and Arnd Poetzsch-Heffter
University of Kaiserslautern
{jschaefer,poetzsch}@cs.uni-kl.de

Abstract. Concurrency in object-oriented languages is still waiting for a satisfactory solution. For many application areas, standard mechanisms like threads and locks are too low level and have shown to be error-prone and not modular enough. Lately the actor paradigm has regained attention as a possible solution to concurrency in OOLs.

We propose JCoBox: a Java extension with an actor-like concurrency model based on the notion of concurrently running object groups, so-called coboxes. Communication is based on asynchronous method calls with standard objects as targets. Cooperative multi-tasking within coboxes allows for combining active and reactive behavior in a simple and safe way. Futures and promises lead to a data-driven synchronization of tasks.

This paper describes the concurrency model, the formal semantics, and the implementation of JCoBox, and shows that the performance of the implementation is comparable to state-of-the-art actor-based language implementations for the JVM.

1 Introduction

The Internet and the broad availability of multi-processors radically influence the way software has to be written [56, 20]. On the one hand standard desktop programs have to deal with distribution aspects like network transmission delay and failure, on the other hand they must utilize multiple cores to scale in the future.

The object-oriented programming paradigm is widely used in practice and supported by industry-strength languages like Java and C#. These languages have built-in mechanisms for multi-threaded and distributed programming, but support for structuring and encapsulation of the state space, higher-level communication mechanisms and a common model for local and distributed concurrency is missing. Concurrency in these languages is introduced by dividing the control flow over a number of concurrently running threads working on a shared state. Threads are scheduled preemptively, allowing any thread to be suspended or activated at any time. To prevent threads from unwanted interleavings, low-level, basic synchronization concepts, like locks, have to be used. Experience shows that software written in such a way is prone to errors, difficult to debug, hard to maintain and to extend.

⋆ This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Methods.
In addition, threads can freely cross component boundaries, making it difficult to modularly describe the behavior of a component and to maintain its invariants.

To motivate our approach and to relate it to other work (see Sect. 6 more details), we shortly discuss solutions that have been developed to overcome the mentioned problems.

As a core concurrency concept, the actor model [33, 3, 42] has regained attention as, in contrast to thread-based concurrency, it encapsulates control flow and data. Prominent examples are Erlang [6] based on a functional language background and Scala actors [29] integrated into a modern OOL. Combining actors with object-orientation comes with some challenging problems. One problem is that actors communicate by message passing instead of invoking methods. Thus, resulting systems are written with two incompatible communications mechanisms. In Scala actors, it is for example unsafe to simply call a method on an actor, as method calls are not protected by the actor. In addition, messages are often only dynamically typed, or require additional mailboxes to become statically type safe, further complicating the communication mechanism. Instead of using messages, asynchronous method calls provide a type safe communication mechanism, compatible with standard method calls, which is adopted by many active object approaches [13, 14, 11, 10].

Object-oriented components are often realized by groups of interacting objects. Similar to many other component models, an OSGi component [59], for example, provides a number of so-called services, where each service can be realized by a different object. In contrast, typical active object approaches have single objects as the unit of concurrency, which makes it difficult to implement such components, as each service object must be an active object again, which cannot share state with the main component object. In some actor approaches, e.g. ASP [14], the state of an actor can be represented by multiple objects. These objects, however, cannot be referenced by other actors. This issue is solved in the E programming language [44] and AmbientTalk [60], where multiple objects can be referenced.

Most actor and active object approaches have in common that a single thread is responsible for executing the code inside an actor. This makes it difficult to have multiple independent control flows within an actor, which is important for two reasons: first, it is not easy to combine active with reactive behavior, and second, waiting for certain messages requires the actor to completely block the actor for other activities. Creol proposes a solution, namely to have multiple cooperatively scheduled tasks within a single active object [11].

**Contributions.** We developed a concurrency model and language that unifies existing solutions for the design problems with the following features:

- A concurrency model suitable for local and distributed concurrency.
- An appropriate integration of synchronous and asynchronous communication.
- A scaling support for components, in particular for components with several services provided by distinct objects.
- A partitioning of the object space into so-called coboxes such that each cobox can control local concurrency and maintain its invariants.
– A formal semantics that extends and simplifies that of an earlier version [54].
– An implementation of the language as an extension to a Java that can compete with existing actor implementations.

Overview. The paper continues with the description of the concurrency model and its realization as an extension of the sequential subset of Java (Sect. 2.1). Section 3 presents the formal semantics of the core fragment of JCoBox. Section 4 briefly explains the implementation of JCoBox. Section 5 evaluates the performance of JCoBox and summarizes practical experiences. Section 6 discusses related work, limitations, and future work. Finally, Section 7 concludes.

2 The Core Concepts

This section describes and motivates the core concepts of the coxbox model, shows how they are realized as a Java extension, the so-called JCoBox language, and sketches additional language constructs provided by JCoBox.

2.1 The CoBox Model: Informal Description

The central concept of the coxbox model is the coxbox. A coxbox can be considered as a container for objects. CoBoxes are concurrently running, isolated, object-oriented components. Figure 1 presents a schematic view of the coxbox model. A coxbox encapsulates both state and behavior and is in this sense similar to active objects [61, 4, 13]. The state of a coxbox is constituted of a heap of objects. The coxbox owns these objects for their entire lifetime. The behavior of a coxbox is constituted of a set of cooperative tasks, which, again, are owned by the coxbox for their entire lifetime.

CoBox-Local Computations. Inside a coxbox, computations are similar to sequential object-oriented programming. All objects of a coxbox can be directly accessed by accessing their fields or by invoking methods. Such direct method calls are immediately executed by the calling task in the standard stack-based way. To realize concurrency, a coxbox supports multiple, possibly interleaved control flows, called tasks. Tasks are created when methods are asynchronously called on objects of a coxbox and are responsible for executing the called method.

Tasks are scheduled cooperatively. This means that at most one task can be active in a coxbox at a time, and that the active task has to give up its control explicitly to allow other tasks to become active. While a task is active, it has exclusive access to the coxbox-local state. This allows the active task to reestablished invariants ranging over several objects of the coxbox before it gives up control. If the active task never gives up control until it terminates, sequential execution of the task is enforced.

Tasks never leave a coxbox; they stay in a coxbox until they terminate. This is an important aspect, which differs from typical thread-based approaches, where threads are not bound to a component, but are allowed to leave and reenter components, which makes the externally visible behavior of such components very complex as it depends on concrete thread identifiers [2].
If a task is not active, it is either ready or suspended. Initially a task is in the ready state. Ready tasks are organized in a FIFO queue (the ready queue). If the active task gives up control the next task from the ready queue becomes active. A task can give up control in three ways: it can terminate, it can yield, which immediately adds it to the end of the ready queue, or it can suspend, waiting for some condition to be satisfied. When the condition of a suspended task becomes satisfied, the task is awaked and added to the ready queue, for eventual execution.

Inter-CoBox Communication. As a cobox is only represented by its objects, inter-cobox communication means that objects of one cobox communicate with objects of another cobox. To do so, an object in one cobox needs to reference objects of other coboxes. Such references are called far references (following the naming of E [44]), in contrast to local references, which refer to objects of the same cobox. Dually, we speak of far and local objects. Far references are the analogue of remote references used in RMI.

Far references can only be used as targets for asynchronous method calls. It is not possible to directly invoke methods or access fields on far references. Asynchronous method calls return futures [8, 30, 61, 42, 46, 1]. Futures are place holders for the results of asynchronous method calls. Tasks can synchronize in the cobox model by waiting on futures. Using futures to wait for the results of asynchronous method calls leads to a data-driven synchronization, where tasks can only wait for data. Futures are first-class values and can be passed to other coboxes.

Futures cannot only be resolved implicitly by asynchronous method calls, but also explicitly using promises [43, 44, 46, 1]. A promise is a special object that can be explicitly resolved once and can have multiple associated futures. Promises can be shared between coboxes. They are a safe, flexible communication and coordination

Fig. 1. Schematic view of the cobox model. Legend: standard objects, transfer objects, immutable objects, active task, ready tasks, suspended tasks, local reference, far reference, task scheduling.

\footnote{Note that the terms future and promise are not used consistently in the literature. This paper uses definitions similar to Ábrahám et al. [1] and Niehren et al. [46].}
interface Client {
  void onChatMsg(Msg m); }
interface Server {
  Session connect(Client c); }
interface Session {
  void publish(Msg m);
  void keepAlive();
  void close(); }

Fig. 2. The interfaces of the different components of a simple chat application.

Data Transfer and Object Sharing. To combine encapsulation, data transfer between coboxes, and sharing of objects among coboxes, the cobox model distinguishes three kinds of objects. Standard objects in the cobox model are passed by reference between coboxes. CoBoxes can only interact with standard objects of other coboxes by using asynchronous method calls via far references. To transfer data between coboxes without exposing objects of a cobox by passing it via a far reference, transfer objects can be used. Transfer objects are always local to a cobox and can never be referenced by other coboxes using far references. They are deep-copied when passed to another cobox. The target cobox then gets a local reference to the object copy instead of a far reference to the original object. Transfer objects are like passive objects in ASP [14], and isolates in AmbientTalk [60]. They can also be compared to non-remote, serializable objects in Java [58].

As transferring data using transfer objects can be inefficient due to the copying overhead, it is possible to directly share state between coboxes using immutable objects [28, 18]. Immutable objects never change their state, and it is thus safe to access this state concurrently. Like standard objects, immutable objects are passed by reference between coboxes. Conceptually, immutable objects are not owned by any cobox. Nevertheless references to immutable objects are treated as local references. It is thus be possible to directly call methods and to directly access the fields of immutable objects. Immutable objects can only reference standard objects or other immutable objects. Transfer objects cannot be referenced by immutable objects.

2.2 Motivating Example: A Chat Application

To illustrate and motivate central design decisions of our model, we use a simple IRC-like chat application as an example. The chat scenario consists of multiple clients communicating with a single server. For simplicity, there is only one public chat room which all clients use. Whenever a client sends a message to the server it is broadcast to all other clients. All clients should see the messages in the same order as they appeared at the server. A user should interact with its client instance via a graphical user interface. The implementation should work in a distributed environment, and thus must deal with network latencies. For simplicity, we assume no network failures, but allow clients to silently disconnect from the server.

In the typical usage scenario, a client sends a connect message to the server, together with a callback object, which is used by the server later to send published
messages to the client. The server answers with a reference to a session object, which is a typical service object. The session object can then be used by the client to send messages to the server. Using a separate session object for each client simplifies the server logic for tracking client-specific state. Whenever the server gets a message from a client it broadcasts the message to all connected clients. The server also assumes that a client constantly sends alive messages to the server. Figure 2 shows how possible interfaces of these components may look in Java.

Implementing such an application in an object-oriented language has some challenges. On the client side, the GUI and the network-related parts should be decoupled and run concurrently, to prevent network latencies from affecting GUI responsiveness. Frameworks for graphical user interfaces are in general not thread-safe, further complicating concurrency handling in desktop applications. Futhermore, the client has to deal with asynchronous messages from the server and has to constantly send alive messages to the server.

The server component has to deal with concurrently accessing clients. In addition, these clients interact with different objects of the server. Standard OO monitors cannot deal with such multi-object interfaces. In addition, the server has to separate the communications of the different clients from each other, to prevent slow clients from affecting the communication with other clients. In addition, the ordering of messages must be guaranteed by the server.

Using the cobox model, the chat application could be structured as shown in Fig. 3. The client is split into two coboxes, one for running the GUI-related code and one for the client logic. Both parts thus run concurrently and communicate asynchronously. The server is realized by a single cobox, which owns the main server object as well as the session objects for each client. Thus the state of the server belongs to a single cobox and programming within the server is done using cooperative multitasking. The server runs concurrently to all clients and communicates with the clients asynchronously. Messages would be realized as either transfer or immutable objects.

2.3 The Core JCoBox Language

A programming model alone cannot be used to write programs. JCoBox is an extension of sequential Java that implements the cobox model to realize concurrency. As JCoBox extends Java, it is a class-based object-oriented language. The extension is conservative, which means that a well-formed sequential Java program is a well-
formed JCoBox program. There is only one exception, which is that static fields in JCoBox have some restrictions (see Sect. 2.4). The syntax of Java is only minimally extended by an additional operator for asynchronous method calls and an extended new expression. All other extensions are done by standard Java annotations and using special purpose classes and interfaces. We assume that the reader is familiar with sequential Java and thus we only explain the concepts introduced by the JCoBox extension. This subsection explains the core constructs of JCoBox in some detail; in particular it describes how

- coboxes are created and objects are assigned to coboxes;
- the different object kinds (standard, transfer, immutable) are distinguished;
- asynchronous method calls and futures are addressed;
- the mechanisms for task cooperation is realized.

Creating CoBoxes. Coboxes are not first-class citizens of the language; a cobox is only represented by the objects it contains. A cobox is implicitly created when a cobox class is instantiated. Instances of cobox classes are standard objects in the cobox model, but with the guarantee that they are always created in a new cobox. They are thus the first object of a cobox and can then be used to interact with the cobox. CoBox classes are declared by the @CoBox annotation. An implementation of the Server interface from the chat application could thus be done by writing a cobox class:

```java
@CoBox class AServer implements Server {
    List<Session> sessions = new ArrayList<Session>();
    Session connect(Client c) { return new ASession(c); }
    void broadcast(Msg m) { ... }
    class ASession implements Session { ... }
}
```

Assigning Objects to CoBoxes. Objects of cobox classes are always created in a new cobox. In contrast, objects of non-cobox classes are created by default in the cobox of the creating task. The ASession class, for example, could be implemented as follows:

```java
class ASession implements Session {
    Client client; Date lastActivity;
    ASession(Client c) { client = c;
        sessions.add(this); keepAlive(); this!checkAliveness(); }
    void publish(Msg m) { broadcast(m); keepAlive(); }
    void keepAlive() { lastActivity = new Date(); }
    void close() { sessions.remove(this); }
    void checkAliveness() { ... }
}
```

The class has no annotation and is thus called a plain class. As it is a non-cobox class, its objects are created in the cobox of the creating task. In the example, it is always created by the server cobox and thus can access fields of server objects and can

---

2 For brevity we ignore access modifiers in example code.
directly call methods on the server object. There is no mechanism in the language that statically specifies the cobox of an object. Such a guarantee could, however, be given by extending the language with an ownership type system [17, 18].

It is possible to create non-cobox objects in other coboxes than the current cobox by using an extended \texttt{new} expression that allows for specifying a target cobox. For example, \texttt{new A() in (b)} creates an object of \texttt{A} in the cobox that owns object \texttt{b}.

\textit{Transfer and Immutable Objects.} Standard objects in \textit{JCoBox} are instances of cobox classes or plain classes. Transfer and immutable objects are instances of \textit{transfer} and \textit{immutable} classes, respectively, declared by @\texttt{Transfer} and @\texttt{Immutable} annotations. For example, the \texttt{Msg} class of the chat application could be implemented as:

\begin{verbatim}
@Transfer class Msg { String user; String content; }
\end{verbatim}

To guarantee immutability of immutable objects, \textit{JCoBox} currently adopts a simple mechanism, namely to make all fields of immutable classes implicitly \texttt{final}. In addition, fields may not refer to transfer classes. This is a rather restrictive definition, but is, nevertheless, sufficient in many cases.

\textit{Asynchronous Method Calls.} Asynchronous method calls are expressed by using the \texttt{!} operator instead of a dot. For example, \texttt{a ! m()} asynchronously invokes the method \texttt{m()} on object \texttt{a}. Any method of plain or cobox classes can be invoked asynchronously in \textit{JCoBox}. Methods of immutable and transfer classes can only be called directly.

Asynchronous method calls are partially ordered. Partially means that two subsequent calls, from the same cobox, targeting objects of the same cobox, are executed in the given order. For example, the methods sequence \texttt{x ! m(); y ! n()} are executed in the given order if \texttt{x} and \texttt{y} refer to objects owned by the same cobox. As in a distributed setting, guaranteeing ordering of messages can impose additional costs, \textit{JCoBox} also supports an unordered variant of an asynchronous method call indicated by the \texttt{!!} operator.

\textit{Futures.} Futures in \textit{JCoBox} are instances of the special interface \texttt{Fut<\textit{V}>}, where \textit{V} is the type of the future value. The following code invokes an asynchronous method and stores the result in a future variable:

\begin{verbatim}
Fut<Session> fut = server ! connect(this);
\end{verbatim}

In order to get the value of a future, it has to be explicitly \textit{claimed} \cite{1}. Similar to the Creol approach \cite{11}, claiming a future in \textit{JCoBox} can be done in two different ways: a blocking and a cooperative one. Claiming a future blockingly is done by using the \texttt{get()} method. Blockingly means that the active task waits for the future without giving up control, thus preventing other tasks of the same cobox from being activated. This guarantees that no other task can modify the state of the cobox while waiting for the future value.

A task can also wait cooperatively for a future using the \texttt{await()} method. Cooperatively means that, when the future is not ready yet, the waiting task gives up control and is added to the suspend set of the cobox, allowing other tasks to be executed. When the future is ready, the waiting task is added to the ready queue again. The following code cooperatively waits for a future using \texttt{await()}:
Session session = fut.await();

Whether a future should be claimed by `get()` or by `await()` depends on the given scenario. As a general rule, `get()` should only be used if the invoking task cannot establish the invariant of its cobox without knowing the value of the claimed future. In addition, a `get()` can only be used if the waited future can be resolved by the called method without a callback to the waiting cobox. As otherwise a (deterministic) deadlock would happen. When using `await()`, the claiming task must establish the invariant of the cobox before the call of `await()`, because other tasks could be activated in between.

Synchronous communication can be simulated by asynchronous method calls with an immediate `get()` or `await()`. In fact, a synchronous call `x.m()` is treated as `x!m().get()`, when `x` refers to a far object.

**Task Cooperation.** Beside using `await()` to give up control to another task, it is also possible to directly yield control to the next ready task by using the `JCoBox.yield()` method. Optional time parameters can be used to specify a period of time to wait before the task is added to the ready queue again. For example, the `ensureAliveness` method of the client could constantly send `keepAlive` messages to the server session:

```java
void ensureAliveness() {
    while (!stopped) {
        session!keepAlive();
        yield(1, TimeUnit.SECONDS); }
}
```

**Start Up and Termination of JCoBox Programs.** JCoBox programs are started by executing the standard `main` method. The main method is executed in a special `main` cobox, which is created at start up. A JCoBox program terminates when all tasks of all coboxes have terminated.

### 2.4 Further Features

JCoBox has several additional features, which are important for writing practical applications. This section briefly describes some of them.

**Futures.** Beside explicitly claiming a future, it is possible to asynchronously invoke methods on futures, which are invoked on the value of the future, when it becomes ready. It is also possible to register event handlers at futures, which are executed when the future is ready. This is similar to when expressions in E [44].

**Exceptions, Arrays, Static Fields, and Static Methods.** Uncatched exceptions, which are thrown during the execution of asynchronous method calls, are rethrown when the future of the call is claimed. Arrays are treated like transfer objects and are copied when passed to another cobox. JCoBox restricts the usage of static fields similar to Kilim [55]. Static fields are implicitly `final` and may not refer to transfer classes. As global state is thus immutable, static methods can be executed by any cobox and, in particular, in parallel.
Java Interoperability. For pragmatical reasons, objects created from standard Java classes can be arbitrarily used in JCoBox. The programmer has to ensure that only thread-safe objects are shared between coboxes, otherwise objects can be wrapped by a dynamic proxy object to be shared by coboxes. JCoBox has special support for Swing applications. CoBox classes can be annotated with the @Swing annotation, in which case all tasks of that cobox are executed by the Swing event handling thread.

Distributed Programming. JCoBox has prototypical support for writing distributed applications using RMI, where the unit of distribution is a cobox. This allows for asynchronous communication via RMI with futures and promises.

3 Formal Semantics

This section presents a formal calculus for a core of JCoBox, called JCoBox\(^c\). The calculus serves as a precise definition of the semantics of JCoBox. The calculus focuses on the key features of the language, namely: cobox classes, asynchronous method calls, cooperative multitasking, futures, promises, and transfer classes. Immutable objects are not included as they can be regarded as transfer objects optimized for performance reasons. The semantics of JCoBox\(^c\) is implemented in the Maude rewriting framework [19], which allows JCoBox\(^c\) programs to be executed, and can be obtained from [35]. We only present a dynamic semantics for JCoBox. The type system for JCoBox is a straightforward extension of a standard Java type system, which mainly adds the typing of asynchronous method calls, futures, and promises. The sequential part of the dynamic semantics is based on Featherweight Java [34] and ClassicJava [24]. The combination of futures and cooperative multitasking is similar to that of Creol [11]. The treatment of transfer objects is similar to that of passive objects in ASP [14]. The combination of promises and asynchronous method calls is similar to Ábrahám et al. [1]. The formalization is a modification and simplification of previous work [54], which treats hierarchical coboxes and which does neither feature transfer classes nor promises.

3.1 Syntax

The abstract syntax of JCoBox\(^c\) is shown in Fig. 4. Terms enclosed by brackets are optional; capital letters denote sets; an overbar indicates a sequences. • denotes the
empty sequence, • adds an element to a sequence, and \( \circ \) concatenates sequences. In addition, we often implicitly treat single elements as sequences or sets of size one when needed. Most of the syntax should be clear from the description in Sect. 2. A program \( p \) is a pair \( D, e \) consisting of a set of class declarations \( D \) and a main expression \( e \). A promise for a value of type \( \tau \) is created by the promise \( \tau \) expression. A future can be retrieved from a promise with \( e.fut \). The expression \( e.resolve(\epsilon') \) resolves a promise \( e \) to the evaluated value of \( \epsilon' \). A type \( \tau \), can either be a class name \( c \), a future type \( r(\tau) \), or a promise type \( p(\tau) \).

### 3.2 Dynamic Semantics

The dynamic semantics is formulated as a small step operational semantics. The semantic entities used by the semantics are shown in Fig. 5. The state of a program is represented by a set \( K \) of configurations \( k \), which can either be coboxes \( b \) or promises \( p \). A cobox \( B(\mathit{tb}, O, T, T) \) consists of a cobox identifier \( \mathit{tb} \), a set of objects \( O \), a set of suspended tasks \( T \) and a sequence of tasks \( T \). The head of \( T \) represents the active task of the cobox, the tail represents the ready queue. Promises are "degenerated" coboxes, that do not have any tasks. Instead, a promise \( \epsilon \) consists of a single expression \( e \), which is in general of the form \( t_p.resolve(\epsilon') \), where \( t_p \) is the promise that is resolved by the task. Only the initial task of a program has no associated promise. Values are either references \( r \) or null, where a reference can either be an object reference \( t_o.t_o \) or a promise identifier \( t_p \). Note that object references in general are of the form \( \mathit{tb},t_o \), \( t_p,t_o \) references can only appear in objects of promises.
oids(O) def = \{ι_o | O(ι_o, _, _) ∈ O ∨ F(ι_o, _, _) ∈ O \}

init(c) def = null, where the length is equal to the number of fields of c

coboxl(c) def = c is defined as a cobox class

transferl(c) def = c is defined as a transfer class

mexpr(c, m, r, \overline{v}) def = [r | this, \overline{v}/\overline{x}]e , where \overline{x}, e = body(c, m)

body(c, m) def = \overline{x}, e , where τ m(\overline{v}/\overline{x})[e] is the declaration of m

reach(0, \overline{v}) def = \emptyset

reach(O, c) def = \emptyset

reach(O ⊕ o, \overline{v} · t_o, t_o) def = reach(O, \overline{v} o \overline{v}) ∪ \{o\} if o = O(ι_o, c, \overline{v}) \land transferl(c)

reach(O ⊕ o, \overline{v} · t_o, t_o) def = reach(O, \overline{v}) ∪ \{F(ι_o, t_o, e)\} if o = F(ι_o, t_o, \overline{v})

reach(O, \overline{v} · v) def = reach(O, \overline{v}) else

m;copy(t_o, O, \overline{v}, t_o', O') def = (σO', σ\overline{v})

where O' = reach(O, \overline{v})

and σ = \{ t_o', t_o \mapsto t_o', t_o \mapsto t_o' | t_o ∈ oids(O') \land t_o' fresh \}

Fig. 6. Auxiliary functions and predicates

Evaluation Contexts. To abstract from the context of an expression and to define the evaluation order of expressions we use evaluation contexts [22]. An evaluation context is an expression with a “hole” □ at a certain position. By writing e[□] that hole is replaced by expression e.

\[ e[□] := □ | e_1.f | e_2.f = e | v.f = e_3 | \text{new } c \text{ in } e_4 \]
\[ | \text{let } x = e_3 \text{ in } e | e_1.n(\overline{v}) | v.n(\overline{v}, e_3, \overline{x}) | e_2.ln(\overline{v}) | v!n(\overline{v}, e_3, \overline{x}) \]
\[ | e_5.get | e_5.await | e_5.fut | e_5.resolve(e) | v.resolve(e) \]

Auxiliary Functions. Figure 6 gives a definition of auxiliary functions and predicates used by the rules. As many of these functions are technically simple, we only provide an informal description for them. For a precise definition we refer to the Maude formalization [35].

Data Transfer. Transfer objects and futures are copied between coboxes/promises by the copy function. The copy function uses the reach(O, \overline{v}) function to extract all transfer objects of O reachable from \overline{v}. In addition, futures are extracted, but their value is reset to e and not regarded for the reachability of objects. This point is important for a deterministic transfer of futures, independent of their actual resolving status. Whenever a future is transferred to another cobox it has to be resolved again by the associated promise. copy(t_o, O, \overline{v}, t_o', O') creates a copy of all transfer objects from cobox/promise t_o, reachable by \overline{v}, with fresh identifiers for cobox/promise t_o'.

Evaluation Rules. The operational semantics is defined in terms of a relation on sets of configurations, K \rightarrow K'. It is implicitly parametrized by a fixed underlying
(LET)
\( B(t_0, O, T, \overline{T} \cdot T (e_0[\text{let } x = v \text{ in } e])) \rightarrow_\eta B(t_0, O, T, \overline{T} \cdot T (e_0[[v/x]e])) \)

(NEW-LOCAL-OBJECT)
\[ e = \text{new } c \quad v = e \quad \text{new } c' \quad i_\eta, i_\xi \notin \text{oids}(O) \quad o = O(t_0, c, \text{init}(c)) \]
\[ B(t_0, O, T, \overline{T} \cdot T (e_0[\text{yield}])) \]
\[ B(t_0, O, T, \overline{T} \cdot T (e_0[[ yields ]])) \]
\[ \overline{T} \text{NEW-LOCAL -OBJECT } \]
\[ e = \text{new } c ... create a new promise for holding the \]
result of the call and add a new task, which executes the body expression of the

(FIELD-READ)
\[ O(t_0, c, \overline{T}) \in O \]
\[ B(t_0, O, T, \overline{T} \cdot T (e_0[t_0,t_2,f_i])) \]
\[ B(t_0, O, T, \overline{T} \cdot T (e_0[t_0,t_2])) \]

(FIELD-UPDATE)
\[ O = O' \cup O(t_0, c, \overline{T}) \]
\[ B(t_0, O, T, \overline{T} \cdot T (e_0[t_0,t_2,f_i] = v))) \]
\[ B(t_0, O, T, \overline{T} \cdot T (e_0[t_0,t_2]))) \]

(RESUME-TASK)
\[ t = \overline{T} (e_0[t_0,t_2,f_i] \text{yield}])) \]
\[ t \in O \]
\[ B(t_0, O, T, T \cdot T (e_0[t_0,t_2])) \]
\[ \rightarrow_\eta B(t_0, O, T, \overline{T} \cdot T (e_0[t_0,t_2]))) \]
\[ \rightarrow_\eta B(t_0, O, T, \overline{T} \cdot T (e_0[t_0,t_2]))) \]

(FUTURE-GET)
\[ \overline{T} \text{FUTURE-GET } \]
\[ B(t_0, O, T, \overline{T} \cdot T (f(t_0,t_2,f_i))) \]
\[ B(t_0, O, T, \overline{T} \cdot T (e_0[t_0,t_2,f_i] = v))) \]
\[ B(t_0, O, T, \overline{T} \cdot T (e_0[t_0,t_2]))) \]

(FUTURE-AWAIT)
\[ t = \overline{T} (e_0[t_0,t_2,f_i] \text{await}))) \]
\[ t \in O \]
\[ B(t_0, O, T, T \cdot T (e_0[t_0,t_2])) \]
\[ \rightarrow_\eta B(t_0, O, T, \overline{T} \cdot T (e_0[t_0,t_2]))) \]

Fig. 7. CoBox-local rules of JCoBox'.

program, which we omit for conciseness. The rules defining the relation are split into
two parts: cobox-local rules and global rules. Splitting up the rules in this way makes
it explicit which steps can be executed in isolation and which require interaction
between coboxes and/or promises.

CoBox-Local Rules. The cobox-local rules are shown in Fig. 7. The relation is denoted by
\( \rightarrow_\eta \) and is defined on coboxes. These rules essentially model programming
inside a cobox. The sequential programming rules are more or less standard. The
important aspect is that the target of field reads, updates, and direct method calls
must be objects of the same cobox. In addition, cooperative task scheduling and
future claiming is covered by the cobox-local rules.

Global Rules. The global rules are shown in Fig. 8. The (NEW-FAR-OBJECT) rule is
equivalent to (NEW-LOCAL-OBJECT), but creates the new object in a different
cobox and does not allow to create transfer classes. (NEW-CoBox) creates a new
cobox with an initial object, whose reference is the result of the new-expression. A
new cobox has no tasks. Asynchronous method calls are distinguished into local
(LOCAL-ASYNC-CALL) and far calls (FAR-ASYNC-CALL). The local one addresses the current
cobox and does not copy the method parameters. The far one addresses a different
cobox and copies the parameters. Both calls create a new promise for holding the
result of the call and add a new task, which executes the body expression of the
Fig. 8. Global reduction rules of JCoBox
Finally, rule (CONGRUENCE) integrates the cobox-local relation into the global relation:

\[
\begin{align*}
\text{(CONGRUENCE)} & \quad b \rightarrow_b b' \\
K \cup b & \rightarrow K \cup b'
\end{align*}
\]

Initial Configuration. A JCoBox program is started by executing \( e \) in the initial main cobox: \( b \langle \text{main}, \emptyset, \emptyset, \tau(e) \rangle \).

Properties. When proposing a new concurrency model, the two most interesting properties are properties concerning data-races and deadlocks. JCoBox is by design data-race free, which is directly reflected in the operational semantics by distributing the object heap over the coboxes. As the active task of a cobox can only access fields of its own cobox, it is immediately clear that data-races cannot occur. Deadlocks are possible in JCoBox by blockingly waiting for futures. In many cases, deadlocks result from direct data-dependencies between coboxes, which, in general, lead to deterministic deadlocks, which can easily be found by standard testing. Deadlocks can be avoided by imposing a partial order (e.g. a hierarchy) on coboxes and only blockingly wait for futures of smaller coboxes, according to that order.

4 Implementation

We implemented JCoBox on top Java\(^3\). The JCoBox implementation consists of two parts: the compiler and the runtime system. The JCoBox compiler (JCoBoxC) is implemented as an extension of Polyglot 1.3.5 \([51, 48]\). Support for Java 5 features, e.g. generics, is realized by the JL5 extension \([36]\). JCoBoxC takes JCoBox source files and Java class files as input and generates Java source files, which are then compiled to JVM bytecode by a standard Java compiler.

CoBoxes. Every cobox at runtime is represented by a cobox object, which is created when an instance of a cobox class is created. To realize the relationship between objects and coboxes, every object gets an additional field pointing to its owning cobox, which is set when the object is created. This additional field is added by inheriting all classes without an explicit super class, implicitly from CoObjectClass, which has such a field. Transfer, immutable and classes inheriting from standard Java classes do not inherit from CoObjectClass and thus do not have this field. Whenever a method is invoked on an object of CoObjectClass, it is checked whether the invoking cobox is equal to the cobox of the object. If this is not the case, the call is passed to the scheduler of the cobox for eventual execution. The invoking cobox is determined by the invoking thread. Every thread has a thread-local field referring to the cobox of the task which it currently executes. The check is cheap as no synchronization is required, so that cobox local calls only have a small overhead. In addition, the compiler can optimize cases, where it is statically clear that the cobox is not left, e.g. for calls on this.

\(^3\) The implementation can be obtained from the JCoBox website \([35]\)
Asynchronous Method Calls. Asynchronous calls are realized by task objects invoking the corresponding method synchronously. For each asynchronous call a separate class is generated that contains code for copying of parameters, as well as the invocation of the corresponding method. Each task object is assigned a promise object for holding the result of the call, which is resolved when the call has finished. Asynchronous method calls return a future obtained from the assigned promise. If it is statically clear that the future is not needed, the compiler uses an optimized task object, without promises and futures.

Synchronous Method Calls. For efficiency reasons, synchronous method calls targeting far objects are not realized by asynchronous method calls and an immediate get(), but by standard synchronous calls. To guarantee that they are semantically equivalent, the JCoBox compiler generates wrapper methods for every method of a class that gives that guarantee.

Task Scheduling. The tasks of a cobox are managed by a scheduler. The scheduler manages the ready queue of the cobox. The ready queue contains either asynchronous task objects, or Thread objects, which are used to represent synchronous calls. The scheduler executes the tasks in FIFO order. Asynchronous tasks are executed by passing them to a global thread pool for execution. Synchronous threads are blocked until they reach the head of the task queue. This all happens without the need of an additional scheduler thread. So that a cobox without any running task does not require a separate thread.

Asynchronous tasks are executed by a thread pool. The size of the thread pool is dynamically adapted to ensure that there are always threads that make progress. The thread pool is increased, for example, when a task waits for an unresolved future. In addition, liveness is ensured by monitoring progress and adding additional threads when needed.

Transfer Classes. For each transfer class a copy method is generated, which recursively copies all fields of the class. This is similar to the default implementation of serializable classes, but is much faster, because no intermediate byte stream is used. In the distributed setting, the standard serialization mechanism of RMI is used.

5 Evaluation

5.1 Performance

A programming model is only useful in practice if it can be efficiently implemented. To evaluate the performance of the JCoBox implementation, we compare JCoBox with two industry-strength languages that run on the JVM and have some kind of actor abstraction: Scala and Clojure. We also planned to include Kilim in our performance evaluation, but until the end, Kilim suffered from a data-race, which made it impossible to get reliable measurements.
Scala features an actor library, which belongs to the fastest actor implementations on the JVM [38]. Clojure is a Lisp-dialect targeting the JVM and has the concept of agents to allow for the asynchronous execution of functions. We used Scala v2.7.7-final and Clojure-1.0.0 in all of our benchmarks. We also included Scala v2.7.5-final, because it uses a different actor backend.

As each language has different names for similar things, we use in the following the term actor for coboxes in JCoBox, actors in Scala, and agents in Clojure; and we speak of message sending, when talking about asynchronous invocation of methods in JCoBox, message sending in Scala, and asynchronous invocation of functions in Clojure.

**Benchmark Programs.** As there is currently no standard benchmark for measuring actor-like languages, we measure performance by three micro-benchmarks, The first two are taken from the Computer Language Benchmark Game [57], the third one is an example, which is used in Srinivasan and Mycroft [55].

*Thread Ring* creates a ring of $n$ actors. A single message is then send around the ring $m$ times, resulting in a total of $n \times m$ sent messages. We consider two different configurations. The first configuration (ringmsgs) sets $n$ to 1 000 and increases $m$ from 1 000 up to 10 000. This configuration measures mainly message sending and receiving performance and is similar to the configuration used in [57]. The second configuration (ringnodes) sets $m$ to 10 and increases $n$ from 100 000 up to 1 000 000. This configuration mainly measures the creation and handling of a large amount of actors. Both configurations have no concurrency, no contention, and there always exists only one unreceived message at a time.

The *Chameneos* example [57], creates $n$ (first 3 then 10) chameneos, which all try to meet another chameneos at a single mall to complement their colors. The number of meets $m$ is increased from 200 000 up to 2 000 000. This benchmark measures performance of a high frequency of messages under high contention, but the message load is relatively low and the number of actors is also very low. There is a small potential for parallel execution.

The *BigPingPong* (pingpong) example [55] creates $n$ actors, which sends each other actor a single message, so that $n^2$ messages are sent and received. This benchmark measures performance under a high load of simultaneous messages, under low contention, with a medium number of actors and with a high potential of parallel execution.

**Setup.** We ran all benchmarks on five different hardware platforms: An Intel Atom N270 1.6GHz CPU and 1GB RAM (Atom). An Intel Core 2 Duo T7400 2.16GHz CPU and 2GB RAM (Core2Duo). An AMD Athlon dual-core 4850e CPU with 2.5Ghz and 4GB RAM (Athlon). An AMD machine with two dual-core AMD Opteron 270 2GHz CPUs and 4GB RAM (Opteron). An Intel Xeon X3220 quad-core CPU with 4GB RAM (Xeon).

The Atom, Core2Duo and Athlon platforms run a 32-bit Linux 2.6.31, the Opteron platform run in a XEN virtual machine with 64-bit Linux 2.6.16-xen, and the Xeon platform run a 64-bit Linux 2.6.25. All benchmarks were executed on the Sun JDK...
version 1.6.16. On the Core2Duo and the Athlon platform the 32-bit Server VM, and on the the Opteron and Xeon platform the 64-bit Server VM was used. On the Atom platform we used the 32-bit Client (Atom Client), which is the default on this platform, as well as the Server VM (Atom Server). All benchmarks were executed with `java -Xmx1024M`, thus with a maximal memory of 1GB.

We followed the advice of Georges et al. [26] and executed each benchmark \( k \) times within \( n \) JVM invocations. For each JVM invocation we took the arithmetic mean, \( \bar{x}_i \), of the last 10 of \( k \) benchmark runs, where \( k \) was at most \( 10 + 5 \), but could be less if the JVM reached a steady state earlier. A steady state was assumed if the CoV\(^5\) of the last 10 runs was less or equal to 0.02. The time of a benchmark run was measured by using the Java `System.nanoTime()` method. We then calculated the mean and the 95% confidence interval of the \( n \) means, \( \bar{x}_i \), where \( n \) was either 10 or less if the size of the 95% confidence interval fell below 3% earlier.

**Results.** The chart in Fig. 9 gives an overview over all benchmark runs with maximal input parameters. Each point represents the execution time of a single language-program-platform combination relatively to the corresponding JCoBox execution time. The y-axis can also be read as the speedup of JCoBox compared to the other languages. This chart allows for comparing the different languages as well as the

---

\(^5\) CoV is defined as the standard deviation \( s \) divided by the mean \( \bar{x} \), see [26] for details.
Fig. 10. The different benchmark executions on the Xeon-Platform

different platforms. Figure 10 exemplarily shows the different benchmarks runs on the Xeon platform with increasing input parameters.

Discussion. In all our benchmarks JCoBox outperforms Scala as-well-as Clojure. The largest speedups (between 2 and 9) are in the pingpong example, which shows that JCoBox can deal with a high load of messages. It also shows the effect of additional cores. Scala v2.7.5 seems to profit, whereas Clojure and Scala v2.7.7 are slowed down, compared to JCoBox. The lowest speedups (∼1–4) are in the chameneos example, which focuses on high-contention, which shows that a JVM limit might be reached by all languages. The ringmsg benchmark shows that message sending and receiving is fast in JCoBox (speedups between 1.5 and 7). The ringnodes benchmark shows that coboxes are very cheap and scale up to millions of coboxes. Even though it was not the focus of the performance tests, it can be noticed that Clojure is surprisingly fast, even though it is a dynamically typed language. There is also a significant difference between Scala v2.7.5 and v2.7.7, where the latter has been significantly slower in 3 of 4 of our benchmarks, and the former ran out of memory or timed out in the ringnodes benchmark on some platforms, a known bug, which has been fixed in v2.7.7.

It is always critical to use micro-benchmarks to compare the performance of different languages. It cannot be concluded from these benchmarks that in practice there will be a significant difference in speed between the different languages, as the
dominant factor of an application might not lie in the actor framework. However, we believe that in practice, JCoBox will be at least as fast as the compared languages.

5.2 Example Applications

Among several small examples and applications, we implemented four mid-size desktop applications in JCoBox. A connect four game, which also supports a computer player, which utilizes multiple cores and can run on a different machine. The CoCoME example [52], which implements a distributed trading system, which was implemented by a master student. A disk usage visualizer, which incrementally visualizes the contents of a file system and allows for navigating through the view, while it is built. Finally, we implemented a distributed chat application as described in Sect. 2, but with multiple chat rooms. Except the CoCoME example, all applications have been written by the authors.

Our experience with JCoBox is very promising. The cobox model matches very well with the typical object-oriented programming style and does not require a radical new way of thinking. CoBoxes naturally appear in typical OO applications. Asynchronous method calls are also a natural communication mechanism and lead to loosely coupled systems. The cooperative multi-tasking inside coboxes proved to be very useful for combining active and reactive behavior and for simulating synchronous communication with flexible reentrancy control. JCoBox prevents data-races by design and makes it difficult to create non-deterministic deadlocks, which greatly simplifies the development of concurrent OO applications.

6 Related and Future Work

JCoBox does not introduce radically new ideas or features. It rather unifies existing ideas into a single formalized programming model and provides a practical implementation on top of Java.

6.1 Related Work

There exists a vast amount of work on how to combine concurrency with object-orientation (see, for example, [50, 12] for surveys). This subsection necessarily concentrates on the closest work.

The principle idea of a data-centric concurrency model, i.e. a model where concurrency is structured by the data, stems from the actor model [33, 3, 42]. The pure actor model is defined in a functional setting, without mutable state, with Erlang [6] being the most prominent implementation of the recent past. The combination of stateful objects and actors as active objects was first done in ABCL/1 [61], POOL2 [4], and Eiffel// [13]. These approaches have single objects as the unit of concurrency. Hybrid [47] generalizes this to groups of objects called domains, which, however, communicate via synchronous method calls. ASP [14] groups objects into so-called activities. Activities, however, have only one distinguished object, the active object, which can be referenced by other activities. Multiple service objects for one activity
are thus not possible. ASP also introduced the notion of **passive objects**, which can only be referenced inside a single activity and are deep-copied when passed to other activities. Passive objects correspond to transfer objects in JCoBox. The ASP concurrency model is implemented in ProActive [7].

The E programming language [44] introduces a concurrency model called **communicating event loops**. The unit of concurrency in E is a **vat**, which hosts a group of objects. All objects of a vat can be referenced by other vats, allowing multiple service objects, equally to coboxes. Computations inside a vat, however, are only executed by a single thread of control. This leads to an event-based programming model, where the control flows has to be spread over event handlers. The E programming model can be simulated in JCoBox by never suspending or yielding a task. Ordering of messages in E is only with respect to single objects, where in JCoBox it is with respect to coboxes, which we believe is what a programmer in general expects. Like JCoBox, E can execute a vat by the event handling thread of Swing, to allow for a seamless GUI interaction. The E programming model has been lately adopted by AmbientTalk [60]. AmbientTalk also integrates a notion of transfer objects called **isolates**.

The idea of using cooperative multitasking for concurrency inside of active objects stems from Creol [11, 37]. In addition, combining cooperative multitasking with futures, was also pioneered by Creol. Creol, however, has single objects as the unit of concurrency, which does not allow for multiple service objects. The Creol model can be simulated by the cobox model by putting every object in a unique cobox. In Symbian OS [45] active objects are scheduled cooperatively within the same active scheduler, which are thread-local. Each active object only has a single thread of control. Symbian OS also shows how to combine active objects with GUI programming. Kilim [55] can schedule tasks cooperatively if the assigned scheduler is configured to be single-threaded. Rodriguez and Rossetto [53] combine cooperative multitasking with asynchronous RMI in the Lua language.

Thorn [10] is an actor approach, which combines message sending and asynchronous method calls. Unlike JCoBox, Thorn does not support multiple tasks within a process. However, Thorn has a special `splitSync` construct, which can be used to solve the problems of the single-threading of Thorn components in some cases. Only methods declared to be asynchronous can be invoked in an asynchronous way, where in JCoBox, any method can be invoked asynchronously.

### 6.2 Current Limitations and Future Work

Although JCoBox showed to be usable in practice, there are some limitations and improvable aspects.

**Data-Parallel Programming.** JCoBox can be used for many parallelization problems, namely ones, where the problem state can be partitioned into separate parts and be worked on in isolation. However, there are some parallelization problems, which cannot be efficiently addressed in the cobox model, namely data-parallel algorithms, where parallel threads work on shared mutable state. For example, a parallel in-place sorting of an array will not be possible to implement in the cobox model. We
argue that JCoBox should be used as a high-level programming model, where it is possible in certain cases to “escape” from the high-level model and use special purpose libraries that address these issues [41, 40]. In addition, parallel access to shared state is possible in JCoBox if the state is immutable.

Receiver Coordination. Currently, a cobox serves all method calls it gets in the same order as they appear. There is no direct mechanism to define a certain communication protocol. Instead, a programmer has to use promises or monitor-like condition variables. There exist several different solutions for receiver-side coordination. One are guards at method declarations [32] or at suspension points [11]. Another are join patterns [25, 9]. An actor like mechanism, where a body explicitly specifies a communication protocol [4, 5, 14, 39, 10] could be integrated into JCoBox. Finally, one may specify the scheduling of messages in a separate object as done in SOM [16] and POM [15]. This is already partly supported by JCoBox as the scheduler object is independent from the cobox object, a feature which is used to implement the Swing support.

Data Transfer. Currently, data in JCoBox is transferred between coboxes either by copying or by using a simple form of class immutability. The current form of immutability is very strict. We are currently working on an implementation of a more flexible immutability notion based on [27, 18, 28]. Sometimes neither copying nor immutability is an option. A network connection object, for example, can neither be immutable nor can be copied. It is thus desirable to be able to safely transfer mutable objects by reference. Several solutions to this problem exist [18, 55, 21], which could be added to JCoBox.

Cooperative Multitasking. The strongest limitation of the current JCoBox implementation is the fact that suspending a task can lead to a suspension of the underlying thread and may require to increase the thread pool to prevent starvation. This can be an expensive operation especially if the number of simultaneously suspended tasks is large. This problem is effectively solved by continuation frameworks for Java [55, 23], which we plan to integrate in our implementation.

Distributed Programming. The cobox model is well-suited for distributed programming, supported by the fact that languages explicitly designed for distribution have a similar programming model [44, 7, 60]. The current JCoBox implementation only features a proof-of-concept implementation based on RMI. It can already be used to write distributed programs in an asynchronous style, where the underlying programming model is, despite network failures, identical in the local and distributed case.

7 Conclusions

Concurrency in OOP is an ongoing research topic. This paper presents JCoBox: a language that unifies several concepts for OOP-concurrency. The concurrency model
of JCoBox is based on coboxes, concurrently running, isolated object-oriented components. The language integrates asynchronous method calls for a loosely coupled communication, suitable for distributed systems. Futures and promises allow for a data-driven synchronization between tasks. The behavior of a cobox is constituted by a set of cooperatively scheduled tasks, enabling the easy combination of active and reactive behavior. Data can be transferred either by copy or by reference using immutable objects.

The semantics of JCoBox is precisely described by a formal core calculus. JCoBox is implemented on top of sequential Java and was successfully used to write several concurrent and distributed desktop applications with graphical user interfaces. The JCoBox implementation is a research prototype, nevertheless, its performance is comparable to other state-of-the-art actor implementations for the JVM.

Acknowledgments. The authors thank Martin Steffen for a discussion on the programming model, Ilham Kurnia and Yannick Welsch for proof-reading, and the anonymous ECOOP reviewers for their helpful comments.

References

34. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. 23(3), 396–450 (May 2001)
45. Morris, B.: Captive and friends. Symbian Developer Network (Jun 2008)
49. Ousterhout, J.: Why threads are a bad idea (for most purposes) (1996), invited talk at the 1996 USENIX Conference
56. Sutter, H.: The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb's Journal 30(3) (Mar 2005)
Appendix G

Prototyping a tool environment for run-time assertion checking in JML with Communication Histories

The paper “Prototyping a tool environment for run-time assertion checking in JML with Communication Histories” [65] follows.
Prototyping a tool environment for run-time assertion checking in JML with Communication Histories

Frank S. de Boer  
CWI  
Science Park 123  
Amsterdam, Netherlands  
F.S.de.Boer@cwi.nl

Stijn de Gouw  
CWI  
Science Park 123  
Amsterdam, Netherlands  
C.P.T.de.Gouw@cwi.nl

Jurgen Vinju  
CWI  
Science Park 123  
Amsterdam, Netherlands  
Jurgen.Vinju@cwi.nl

ABSTRACT
In this paper we present prototype tool-support for the run-time assertion checking of the Java Modeling Language (JML) extended with communication histories specified by attribute grammars. Our tool suite integrates Rascal, a meta programming language and ANTLR, a popular parser generator. Rascal instantiates a generic model of history updates for a given Java program annotated with history specifications. ANTLR is used for the actual evaluation of history assertions.

Categories and Subject Descriptors
D2.1 [SOFTWARE ENGINEERING]: Requirements/Specifications—Tools; D2.2 [SOFTWARE ENGINEERING]: Design Tools and Techniques; D2.4 [SOFTWARE ENGINEERING]: Software/Program Verification; D2.5 [SOFTWARE ENGINEERING]: Testing and Debugging—Tracing; F3.1 [LOGICS AND MEANINGS OF PROGRAMS]: Specifying and Verifying and Reasoning about Programs; F4.2 [MATHEMATICAL LOGIC AND FORMAL LANGUAGES]: Grammars and Other Rewriting Systems

General Terms
Verification

Keywords
Interface Specification, Run-Time Assertion Checking, Attribute Grammars

1. INTRODUCTION
We present in this paper prototype tool-support for the run-time assertion checking of the Java Modeling Language (JML) [3] extended with communication histories specified by attribute grammars (as originated from [12]).

The main problem addressed in our paper is how to specify properties of communication sequences which form the very semantic foundations of object-oriented programming. In [10], a fully abstract trace semantics for a core Java-like language is given, where traces (or communication histories) are (finite) sequences of messages. A fully abstract semantics in general captures the externally observable behavior abstracting from irrelevant implementation details.

Communication sequences have also been used in proof systems for concurrent object-oriented languages: Dowland et al. [6] derived a complete proof system for concurrent objects from a sequential proof system using non-deterministic assignments over the local communication history. This work formed the basis for an extension of the KeY verification system for concurrent objects [1].

Using naively data structures like lists to represent such sequences however gives rise to very complicated assertions. In our paper, we represent these sequences as words of a language generated by a grammar. Grammars as such allow in a declarative and highly convenient manner to describe the protocol structure of the communication events.

However, the question now rises how to represent such grammars in JML assertions, and how to describe the data flow of the sequence. We argue that the data flow can be conveniently described by extending the grammar with attributes, providing a powerful separation of concerns between high-level protocol-oriented properties, which focus on the kind of messages sent and received, and data-oriented properties, which focus on the actual data communicated.

The main technical contribution of the paper is to show how the parser generated from the attribute grammar can be seamlessly integrated with JML assertions.

Our tool suite integrates Rascal, a meta programming language and ANTLR, a popular parser generator. Rascal instantiates a generic model of history updates for a given Java program annotated with history specifications. ANTLR is used for the actual evaluation of history assertions.

2. MODELING FRAMEWORK
Abstracting from the implementation details, an execution of an object can be represented by its communication history, i.e., the sequence of messages corresponding to the invocation and completion of its methods (as declared by its interface).

In this section we describe our modeling framework in Java for the behavioral description of the interface of an object, expressed in terms of its communication histories. Our framework integrates into JML the use of attribute grammars for specifying properties of user-defined abstractions of communication histories. We explain the basic modeling
concepts in terms of the interface of a class implementing a stack as given in Figure 1:

interface Stack {
    void push(Object item);
    Object pop();
}

Figure 1: Stack Interface

Consider an instance of a class implementing this interface. The messages of the communication history of this object are modeled in our framework as instances of the following classes: 'call-push' and 'call-pop' which represent invocations of the methods 'push' and 'pop', and 'return-push' and 'return-pop' which represent the completion of the methods 'push' and 'pop'. Both classes 'call-push' and 'return-push' contain as an attribute the formal method parameter 'item' which stores the value of the actual parameter of the corresponding method invocation. The class 'return-pop' contains an attribute 'result' which stores the object returned. Our example demonstrates that it indeed is convenient to include the formal parameters as attributes also in messages which represent the completion of a method (see below).

A communication view is a general mechanism to introduce user-defined abstractions over the communication history in terms of the terminals of the (attribute) grammar used to specify its high-level (protocol) structure. For example, the communication view in Figure 2 introduces an abstraction of the communication history in terms of its projection onto the messages which correspond to the completion of the methods 'push' and 'pop'. We thus abstract in this particular case from the invocations of these methods and rename the selected messages 'PUSH' and 'POP', respectively.

The abstract behavior of this view can be defined in terms of sequences of the terminals 'PUSH' and 'POP' generated by the context-free grammar given in Figure 3.

This grammar describes the prefix closure of the standard grammar for balanced sequences of the terminals 'PUSH' and 'POP'. Note that in general the specification of the ongoing behavior of an object requires prefix closed grammars.

In order to specify the relation between the actual parameters of calls to the 'push' method and the result values of returns from the 'pop' method we introduce an attribute 'stack' of type EList, which extends the class List in Java with side-effect free implementations of the (overloaded) methods 'EList append(Object element)' and 'EList append(EList list)'. In Figure 4 we extend the grammar rules of Figure 3 with updates to the 'stack' attribute. Note that only the production rules for the non-terminal 's' are extended with updates to the attribute 'stack', since balanced stacks modeled by the non-terminal 'b' are empty. According to the formal semantics of attribute grammars, the value of the attribute 'stack' of a node in the parse tree labeled by the non-terminal 'S' is thus synthesized from the attributes of its descendant nodes. The built-in attributes of the leaf nodes of the parse tree are given by the message types of the corresponding terminals, as defined by the given communication view. As an example, 'item' is (in this case, the only) built-in attribute of the 'PUSH-terminal. The attribute 'stack' of a sequence of the terminals 'PUSH' and

view StackHistory {
    return-push PUSH;
    return-pop POP;
}

Figure 2: Communication View of a Stack

's ::= PUSH s
     | s s
b ::= PUSH b POP
     | ε

Figure 3: Abstract Stack Behavior

'POP' generated by the above grammar thus denotes the list of items pushed but not yet popped (i.e., the current content of the stack).

's ::= PUSH S1 stack = S1.stack.append(PUSH.item)
     | s1 s2 stack = s1.stack.append(s2.Stack)
     | b stack = stack.clear()

Figure 4: Attribute Grammar Stack Behavior

In order to use the attribute 'stack' of this grammar in assertions for specifying the contract of the 'push' and 'pop' methods of the 'Stack' interface (Figure 1) in terms of communication histories, the modeling framework provides a class 'StackHistory' which corresponds to the communication view of Figure 2. This class contains a 'getter' method 'EList stack()' corresponding to the attribute 'stack' defined by the grammar in Figure 4. In assertions specifying properties of the interface 'Stack' we refer to the communication history of the underlying implementation by a model field which denotes an instance of class 'StackHistory', as illustrated by the JML specification of the interface 'Stack' in Figure 5.

Here we assume that the class EList additionally provides side-effect free methods for returning the head and tail of a list, and comparing the contents of two lists. As described above, the identifier 'history' is declared in this interface specification as a model field. The JML keyword 'instance' indicates that each implementation of the interface 'Stack' will contain an instance variable 'history'. The above 'ensures' and 'requires' clauses describe the methods 'push' and 'pop' in terms of corresponding 'PUSH' and 'POP' operations on the stack abstraction of the communication history, specified by the above attribute grammar.

Figure 6 summarizes the main concepts of our framework for modeling communication histories by means of attribute grammars. The concept of a communication view introduces a user-defined abstraction of communication histories in terms of the declaration of the terminals of the attribute grammar. The given interface provides a name-space of the message types of these terminals. The rules of the grammar define invariant properties of the high-level protocol structure of the corresponding abstraction. The attributes of the grammar are introduced in JML as attributes of a model field in the interface which represents the projection of the communication history of any implementation unto to the messages declared by the corresponding view. How this model field is updated in an implementation, is described in
interface Stack {
//@ public model instance StackHistory history;
//@ ensures history.stack().equals(
 \old(history.stack()).append(item));
void push(Object item);
//@ requires history.stack().size != 0;
//@ ensures history.stack().equals(
 \old(history.stack()).tail());
//@ ensures \result == \old(history.stack()).head();
Object pop();
}

Figure 5: JML Specification Stack Interface

3. RUN-TIME ENVIRONMENT
We describe in this section the run-time environment of a single-threaded Java program annotated with JML-assertions over the communication history. The thread of control on a method invocation is depicted in the UML sequence diagram in Figure 7 and of a method completion in Figure 8.

The actors in both diagrams are:

- 'Program': contains an interface we wish to check, an implementation of this interface (StackImpl) and a client test class instantiating and using the implementation.
- 'History (instance)': an instance of the history class. In the stack example this is an object of type 'StackHistory' storing the local history of a 'StackImpl' object.
- 'Parser': an instance of a parser for the given attribute grammar.
- 'JML API': provides facilities to check assertions at run-time.
- 'stdout': the standard output stream of the system. Error messages (such as an assertion failure) can be send to this stream.

Suppose a program invokes 'push' on a 'StackImpl' object \texttt{s}. We can distinguish between two communication events related to the 'push' method: an incoming method call to push, corresponding to the terminal 'PUSH' and a completion of the method call to push, which corresponds to the terminal 'POP'. We first describe the actions taken on the return of push and next explain how the situation differs for incoming method calls.

When 'push' returns (see Figure 8), a call to update the communication history of \texttt{s} is made since we capture returns of 'push' (according to the communication view in Figure 2). Based on the observation that the communication history can be considered to be a token stream (where the tokens are communication events), the object storing the local history of \texttt{s} triggers the parser to create the parse tree for this history, now viewed as a token stream. The parser also computes the values of the attributes of the non-terminals in the parse tree. Since parse errors correspond to violations of the protocol structure of the communication events imposed by the grammar, an assertion failure is generated on parse errors.

Once the parser returns it communicates to the history the values of the attributes of the root in the parse tree. Conceptually, these are the values of the history attributes describing properties of the data (actual parameters, return value) sent by the calls and returns that were recorded in the history. Finally, the method 'push' returns and a call to the JML API is done by the program to check the postcondition of 'push'. The JML API retrieves from \texttt{\{History instance\}} the values of the attributes of the communication history (in our case: 'history.stack()') used in the postcondition and checks the postcondition. If the postcondition is false, a message describing the Assertion Failure is send to 'stdout'.

Figure 6: The Modeling Framework

Figure 7: Sequence Diagram of method invocation

Figure 8: Sequence Diagram of postcondition assertion failure
4. GENERATIVE FRAMEWORK

In this section we describe how the classes that implement our verification architecture are obtained from the given annotated source code and the model (shaped as an attribute grammar). We used Rascal [11] for general meta programming and ANTLR [14] for parser generation. See Figure 9 for an overview. We will describe the details of our experience with these tools in Section 4.3.

4.1 Manipulating models and source code

Rascal [11] is a domain specific language for meta programming. We use its parsing, source code analysis, source-to-source transformation and source code generation features. A ± 250 line Rascal program\(^1\) takes care of:

- generating the token classes for each call and return event for the methods in the Java interfaces.
- extracting the declared top-level attributes from the ANTLR grammar.
- generating the History class, which specifically accepts new events from the provided methods in the interface, allows retrieval of the specified attributes by the JML assertion checker, and acts as a token stream for the generated parser.

Note that we require general meta programming features for several input languages, not just aspect weaving for Java. This application of Rascal has three languages as input (ANTLR grammars, View declarations and Java), and one output language (Java). Rascal runs on a JVM, such that it integrates into any Java environment.

4.2 Generating parsers

The choice of a particular parser generator has consequences for the implementation of the History class, as well as for the kind of attribute grammars that can be used and the efficiency of the run-time checker. We have therefore identified several requirements guiding this choice.

- We need to be able to code attribute evaluation with a grammar.
- Since we interact with the annotated parse tree (in order to retrieve the attribute values of the root node), the target language for the parser should be Java.
- The terminals of the grammar are communication events, not string tokens. Thus the parser should be flexible enough to allow handling of any kind of stream of objects, or at least allow a correspondence between token types and other semantic values (the communication events).
- The type of a 'return-push'-token differs from a 'return-pop'-token. Therefore there must be support for heterogeneous token types: we must be able to specify a different token type for each abstract terminal in the grammar (or a different type for the semantic value of a token).
- Since we parse the communication history each time it is updated, parse trees for all prefixes are constructed. To improve efficiency, the parser should probably reuse parse trees of prefixes as much as possible, and avoid re-computation of attributes.
- If the user defines an attribute grammar for which no parser could be generated, no run-time checking can be done. Ideally the parser generator should be able to create a parser for any context-free grammar.

ANTLR facilitates many of these requirements. It generates fast recursive descent parsers for Java, allows encoding of attribute evaluation using grammar actions, and most importantly allows a custom stream of token classes.

As a result, our current prototype has two scalability issues. Firstly, the parsing of the history is done from scratch after each update of the trace. This means that while running a program, checking each assert

\(^1\) Excluding the grammar for Java.

Figure 9: Generating the modeling framework
is in $O(h)$, where $h$ is the size of the history. Secondly, the histories themselves will not easily fit into memory for industrially sized applications.

We postulate that our use of the attribute grammar formalism will help us approach these two scalability issues. Results on incremental parsing and attribute grammar evaluation, e.g. Hedlin [7] and Hudson [8], should be applicable to allow us to evaluate asserts more independently of the size of the history. Instead of having to parse the entire history, an incremental parsing and attribute evaluation mechanism computes the changes that have to be done to an attributed syntax tree after an edit action has been applied to the input string. Such an algorithm might therefore make our assertion checking more independent of the size of the history and solve our first scalability issue.

As an aside, since our attribute grammars are small and simple they produce simple abstract trees. It can be expected that many sub-trees are similar. Maximal sub-term sharing [16] (hash-couling) and dynamic programming techniques should help to exploit such redundancy. If a similar sub-tree is produced again and again, we may reuse the values of its attributes to skip redundant computations.

Related work on the use of context-free grammars for runtime verification also suggests that a scalable implementation is quite feasible [13]. This work extends and optimizes an LR parsing algorithm for checking context-free patterns in execution traces. The authors claim their prototype scales well on the DaCapo benchmark. Further study is needed to see if their approach can be extended to fit attribute evaluation.

### 4.3 Experience Report

Here we describe in more detail our use of ANTLR [14] and Rascal [11].

As explained above we need to pass custom token classes to the ANTLR generated parsers, which represent events in the communication history. This can be accomplished by subclassing the ANTLR Token class. So, using Rascal we generate such sub-classes for every kind of event.

Values for the attributes of a non-terminal can be defined in ANTLR grammars using semantic actions: normal Java code written inside brackets `{` and `}` which is attached to grammar rule. Inherited attributes can be passed as parameters of a non-terminal, synthesized attributes can be represented as return values of a non-terminal. There is no direct support for specifying heterogeneous token types, but one can cast a token in the grammar to the appropriate type when needed.

An important drawback of ANTLR is the lack of support for parsing arbitrary context-free grammars. ANTLR can only handle LL(*) grammars\(^2\), which means that our stack grammar (in Figure 4) must be manually rewritten (factored) to accommodate the parser generator. For this purpose, additional attributes are used to count the number of surplus pushes (`$size`) and we add the constraint that the number of pops may not exceed the number of preceding pushes at any point (facilitated by the syntactic predicate `$($s.ok)`, specifying `$s.ok` must be true).

The ANTLR adaptation of the Stack grammar is given in Figure 10. As ANTLR can not handle left-recursive grammars, the grammar is right-recursive. With synthesized attributes alone, the definition of the 'stack' attribute in the production `s ::= POP s` is problematic, suggesting the use of an inherited attribute 'Stack' to build up the stack, and a synthesized attribute 'CompleteStack' to propagate the full stack up the parse tree. The structural information over the 'PUSH', 'POP' events explicitly present in the original stack grammar is encoded in additional attributes (`$Size` and `$Ok`) and is therefore highly obfuscated, also violating the separation of concerns that dictates the use of attributes to describe data-oriented (instead of protocol) properties. The grammar can be rewritten to an equivalent grammar without inherited attributes, but this further impairs readability.

This example clearly shows the importance of support for general CFG grammars. As ANTLR does not satisfy this requirement we are exploring other parsing generators now. There exist several general context-free parsing algorithms that could be applied. For example the SDF system [17] uses a General LR algorithm. There also exist incremental versions of GLR which may be applicable [18].

\(^2\)A strict subset of the context-free grammars. Left-recursive grammars are not LL(*). A precise definition can be found in [14].

---

**Figure 10: Stack grammar in ANTLR**

---

```
grammar Stack;

start returns [EList<Object> stack]
@init {
    $stack = new EList<Object>();
}

s := POP s | PUSH s |
    returns [ELit<String> completeStack, boolean ok]

PUSH:
    s1 = s[$size+1, $stack.appendElement(
        (return_push)$PUSH).item{}])
    $completeStack = $s1.completeStack;
    $ok = ($s1.ok && $size>=0);
}

POP:
    s1 = s[$size-1, $stack.tail()]
    $completeStack = $s1.completeStack;
    $ok = ($s1.ok && $size>=0);
}

public void update(return_<id> e) {
    <if (r in tokens){>
        e.setType(<grammarName>Lexer.<tokens[r]>);
        addAndParse(e);
    }>
    };
```

---
The return statement contains three levels. The Rascal language level (in boldface) provides the return statement, the string, and embedded in the string expressions marked by `<...>` angular brackets. The string that is generated represents an (unparsed) Java fragment. The fragments embedded in backticks (`') represent parsed Java fragments from the input interface. Inside those fragments Rascal expressions occur again between angular brackets.

The string template language of Rascal allows us to instantiate a number of methods called update using a for loop and an if statement. The data that is used in the for loop is extracted directly from the parse trees of the methods in a Java interface file. The concrete Java source pattern between the backticks (`') matches the declaration of a method in the interface, extracting the name of the method (`<id>`). Note that this snippet uses variables declared earlier, such as tokens which is a map from method names to token names taken from the view declaration in the interface and grammarName which was also extracted from the view earlier.

Albeit complex code due to the many levels required for this task, the code is short and easy to adapt to other kinds of analysis and generation patterns.

5. CONCLUSION

We developed in this paper a modeling framework in Java for the integration of attribute grammars in JML and discussed the corresponding tool-support for run-time assertion checking. An open-source prototype can be downloaded from the author’s website http://www.cwi.nl/~cdegouw.

Related Work.

Work on integrating assertions over communication history in Java programs has been the topic of Cheon and Perumandla in [5]. They present an extension of JML with call sequence assertions: regular expressions over method names, defining the set of allowed traces. A run-time checker is implemented to verify such assertions over the local history of an object. Data communicated by calls and returns is not considered. Hurlin [9] has elegantly extended this approach by using data (in particular, return values of calls) to parametrize the regular expression protocol specifications. Though no run-time checker is implemented (and no integration with JML), a static checker with proof rules for specifications in which the precondition has a specific form is provided.

Bartetzko et al. [2] introduce Jass, an alternative for JML with trace assertions. Trace assertions are given in a CSP-like notation and a precompiler translating trace assertions into Java 5 Annotations is provided. Treneltman and Huisman [15] describe a new formalism on top of JML for assertions using Temporal Logic formulae. A translation for a subset of the Temporal Logic formulae back to standard JML is described, and as future work they intend to integrate their extension into the standard JML-grammar and develop a run-time assertion generator to accommodate run-time checking. In Chen et al. [4] also context-free grammars are used to recognize patterns in event histories. However a distinguishing feature of our approach is the use of attribute grammars to specify data-oriented properties of histories, and their full and seamless integration into JML. The work of Meredith et al. [13] is especially interesting from the optimization point of view (see Section 4).

6. REFERENCES


Appendix H

Unifying Trace- and State-based Specifications for Object-based Components

The revised version of [118] entitled “Unifying Trace- and State-based Specifications for Object-based Components” follows.
Unifying Trace- and State-based Specifications for Object-based Components

Ilham W. Kurnia, Arnd Poetzsch-Heffter, Yannick Welsch

University of Kaiserslautern, Germany
ilham,poetzsch,welsch@cs.uni-kl.de

Abstract. The literature distinguishes between trace-based and state-based specification techniques for object-oriented components. Trace-based techniques describe behavior in terms of the message histories of components. State-based specifications explain component behavior by defining how the state is changed by method calls and what the returned results are. The state space can either be abstract or concrete. Abstract states are used to model the behavior without referring to the implementation. Concrete states are expressed by the underlying implementation. State-based specifications are usually described in terms of pre- and postconditions of methods.

In this paper, we investigate the relationship between trace-based specifications and specifications based on abstract states for sequential, object-based components. We develop a technique for specifying interaction patterns of components and show that the technique allows to formulate both trace-based and state-based specifications. In particular, we illustrate how callbacks can be handled. We define the semantics of specifications in terms of transition systems and give precise definitions of transition system reduction, refinement and extension. Reducibility allows to compare two behavioral equivalent components with respect to their state representation. Refinement allows to weed out non-determinism in components. Extension allows to evolve components in a compatible way.

1 Introduction

Objects or, more generally, object-based components communicate with their environment via messages. Usually, the reaction to an incoming message depends on the state of the object or component. To avoid uncontrolled access and to achieve implementation-independency, it is an accepted principle to encapsulate the concrete state of the implementation. In particular, the concrete state should not be exposed in specifications of classes and components [1, §1.3]. Two different kinds of implementation-independent specification techniques have been investigated in the literature. In so-called model- or abstract state-based specifications, behavior is explained based on a model or space of abstract states [2]. The specification expresses the result values and changes of the abstract state in reaction

* This work is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Methods
to an incoming message. In trace-based specifications, behavior is explained with respect to the history of messages an object or a component has seen.

State-based specifications are closer to the implementation, can directly be complemented with invariants – which is important for verifying implementations –, and are supported by a set of well-developed specification constructs going far beyond the basic pre- and postconditions [3,4]. On the other hand, trace-based specifications have a natural link to semantics [5], avoid the design of an abstract state space, and can more easily deal with callbacks and concurrency [6, §5]. Furthermore, invariants may also be specified to restrict, e.g., the structure of the allowed traces.

The goal of our research is to work out the relationship between state- and trace-based specifications to combine the best of the two techniques. In this paper we investigate, as an initial step, specifications of sequential components with possible callback behavior. We consider specifications of object-based components where a component description consists of one or more classes. A (run-time) component is created by instantiating a class. Internally, the component can create further objects and expose these objects to the environment. To keep the presentation focused on the comparison between trace- and state-based techniques, we do not discuss concurrency and inheritance. However, we believe that our results can be generalized to such settings.

Contributions and overview. We develop a new technique that allows specifying the interactions of components with its environment (Sect. 2). As we only consider sequential components, an interaction consists of a message \( m \) sent to a component and a reaction to \( m \) by the component. In JML-like pre- and postconditions of method calls [4], the message sent corresponds to a method call, the reaction to the corresponding return. Our technique is more general. It supports, e.g., reacting to an incoming call with a call to the environment. Analogously, one can specify how the component behaves if a call to the environment returns. This especially allows the handling of callback scenarios. Furthermore, the technique supports flexible ways to express non-deterministic behavior.

To specify the interactions of a component, the designer can use any suitable model. In particular, the model can be based on the history of interactions a component has experienced. We call this the canonical model, because this model directly reflects the trace semantics. We illustrate that different models can specify the same behavior, but can nevertheless have different properties. To illustrate the power of the approach, we present a non-deterministic specification of a Subject component following the Subject-Observer Pattern [7].

Section 3 defines the semantics of interaction-based specifications in terms of transition systems. We study three different formal relationships between these interaction-based transition systems in Sect. 4. The first relationship (Sect. 4.1) allows us to compare behavioral equivalent specifications with respect to the size of their state spaces. In particular, we show that the state spaces resulting from canonical models are often unnecessarily large. Sections 4.2 and 4.3 investigate refinement and extension of components.
spec IntSet {
    Set<int> mySet = {}; // abstract state

    boolean contains(int j)
      ensures result = (j ∈ mySet) ∧ mySet = old(mySet)

    void add (int j)
      ensures mySet = old(mySet) ∪ {j}

    int some()
      requires mySet ≠ {} 
      ensures result ∈ old(mySet) ∧ mySet = old(mySet) \ {result}
}

Fig. 1. State-based specification of component IntSet using a set datatype

2 Specification Approach

This section explains our approach to unify state- and trace-based specifications. We illustrate both kinds of specifications by simple examples, discuss the relationship between them, and present a combined specification technique.

State-based specifications. In state-based specifications, behavior is explained based on a model or space of abstract states. A specification expresses the result values and changes of the abstract state in reaction to an incoming message. Figure 1 presents a specification of a component IntSet implementing integer sets. An existing set datatype Set<int> is used to model the component behavior. The abstract variable mySet captures the component’s current state. The method contains checks whether an element is in the set and returns the result, while add inserts an element. The keyword old can be used in a postcondition to refer to the value of an expression in the prestate. The method some deletes and returns an arbitrary element from the set; a call to some requires that the set is nonempty.

Trace-based specifications. Trace-based specifications explain behavior with respect to the history of messages a component has seen. In sequential settings, a history is a sequence of pairs (inMsg, outMsg) where inMsg represents a message from the environment to the component and outMsg represents a message from the component to the environment. In general, incoming messages are calls on the component or returns from method executions in the environment. Outgoing messages are calls to the environment or returns from method execution in the component. In the IntSet-example, the incoming messages are call messages of the form call contains(j), call add(j) or call some(). The outgoing messages are returns of the form rtrn res where res denotes the result.

A trace-based specification provides a function next that maps a history and a newly incoming message to potential outgoing messages. To handle non-
next :: List<InMsg × OutMsg> → InMsg → OutMsg → boolean

next h (call contains(j)) (rtrn res) =
    res = (∃m. m < length(h) ∧ h!m = (call add(j), rtrn ()))
    ∧ ∃n. n < length(h) ∧ n > m ∧ h!n = (call some(), rtrn j)
next h (call add(j)) (rtrn ()) = true
next h (call some()) (rtrn res) = next h (call contains(res)) (rtrn true)

Fig. 2. Trace-based specification of component IntSet

deterministic behavior, next is modeled as a predicate which takes a history h, an incoming message im, and an outgoing message om, and checks whether om is a possible response to im after having seen h. Figure 2 defines the next-predicate for the IntSet-example. The definition uses pattern matching from functional programming and can be read as follows: With h being the current history, the result of a call to contains(j) is true if and only if there exists a call of add(j) which appears after any calls of some() with a return value j. This shows that return messages are relevant for the behavior. The result of a call to add(j) is a dummy value denoted by (). The result of a call to some() is a value that is contained in the component, which is specified through a recursive application of next.

For this paper purposes, the specification details of next are not so important. However, specifying the next-predicate can, in general, become complicated.

Comparing state- and trace-based specifications. The two specifications in Figs. 1 and 2 look different at first. To relate the two specifications, we convert the trace-based specification to a state-based one using histories as abstract state. That is, instead of a set datatype, we use the histories together with the next-predicate to capture the state of the component. Figure 3 illustrates this approach for the IntSet-example. An abstract state variable h captures the history. For each method, variable h is explicitly updated with the new call-/rtrn-pair where # is used as the operator to append an element to a list. The result value of the methods is specified using the next-predicate from Fig. 2. Similarly, the requires clause of some uses the next-predicate to check for nonemptiness.

Embedding trace-based specifications into the form of state-based specifications gives us a clear ground to compare them. As the first observation we note that both specifications express the same component behavior where behavior is defined in terms of the input/output traces (a detailed semantics is given in Sect. 3). It should also be clear that there are many other behavioral equivalent specifications (e.g., we could use a list datatype or a richer specification technique for histories not using explicit indices).

From a software engineering perspective, it is interesting to compare behavioral equivalent specifications. For example, enhancements and modifications might come at different costs. Consider, e.g., the addition of a delete method to

\(^1\) Of course, in a more elaborate specification language this could be left implicit.
spec IntSet {
  List<InMsg × OutMsg> h = []; // history as abstract state

  boolean contains(int j)
    ensures (next h (call contains(j)) (rtn result))
    ∧ h = old(h)#(call contains(j), rtn result)

  void add (int j)
    ensures h = old(h)#(call add(j), rtn ())

  int some()
    requires ∃ j. (next h (call contains(j)) (rtn true))
    ensures (next h (call some()) (rtn result)) ∧ h = old(h)#(call some(), rtn result)
}

Fig. 3. Specification of component IntSet using history as abstract state

the IntSet component. In the first specification, deletion can simply be expressed using the set difference operation of the datatype Set<int>. In the trace-based specification, the modification not only adds a new case to the definition of next, but also causes a modification of the case for contains. That is why trace-based specifications, in particular larger ones, are sometimes quite brittle.

As we will analyze in Sect. 4.1, the specifications are also different with respect to the transition systems they induce. For example, the transition system of the state-based specification of IntSet (Fig. 1) is simpler than the transition system of the history-based specification (Fig. 3) in that it has “less” states, e.g., the state representing the empty set in the former system corresponds to infinitely many different history states in the latter one. For verification purposes, it can be very helpful to work with reduced state spaces.

On the other hand, trace-based specifications of components are more powerful than classical pre- and postcondition-style specifications. For example, they can express that the call of a component’s method can lead to a call on the environment. They can also handle callbacks and concurrency aspects. Our goal is to combine the advantages of both specification techniques.

Interaction-based specifications. To combine the expressive power of trace-based specifications with the flexibility of state-based specifications, we specify the interaction of a component in terms of the incoming and outgoing messages, but assume an abstract component state and allow the use of appropriate datatypes/models to express the behavior. A specification consists of

- a list of interfaces specifying the message names and parameters of the object references that can cross the component boundary,
- a state specification having the form of variable list, and
- a list of incoming and outgoing message pairs annotated by pre- and postconditions and a modifies clause (empty modifies clauses can be omitted).
interface Observer {
    void notify();
}

interface Subject {
    SubjState get();
    void register(Observer o);
    void update(SubjState s);
}

spec Subject {
    Set<Observer> obs = {};
    SubjState sbst = null;
    boolean notifying = false;
    Set<Observer> toNotify = {};

    call this.get() rtn result
    ensures result = sbst

    call this.register(o) rtn ()
    requires ¬notifying
    modifies obs
    ensures obs = old(obs) ∪ {o}

1 call this.update(s) rtn ()
   requires obs = {} 
   modifies sbst
   ensures sbst = s

2 call this.update(s) call o.notify()
   requires ¬notifying ∧ obs ≠ {} 
   modifies sbst, notifying, toNotify
   ensures sbst = s ∧ notifying ∧ 
   o ∈ obs ∧ toNotify = obs \ {o}

3 rtn o.notify(): () call o1.notify()
   requires toNotify ≠ {} 
   modifies toNotify
   ensures o1 ∈ old(toNotify) ∧ 
   toNotify = old(toNotify) \ {o1}

4 rtn o.notify(): () rtn ()
   requires toNotify = {} 
   modifies notifying
   ensures ¬notifying

Fig. 4. An interaction-based specification of the Subject component

Messages are denoted similarly to the trace-based specifications above. In addition, return messages can be optionally extended by the method name and the parameters of the corresponding call, i.e., we support the forms \texttt{rtn \hspace{0.1cm} o.m()} : \texttt{v} and \texttt{rtn v}, where \texttt{v} denotes the return value. The latter shorthand can be used only if it acts as a direct response to an incoming call message.

Figure 4 illustrates this specification style for a subject component that nondeterministically notifies its observers in an unspecified order. The observers are outside the component. The abstract variable \texttt{obs} contains the set of registered observers, \texttt{sbst} stores the state of the subject. The variable \texttt{notifying} captures the control flow information that the component is currently notifying its observers and is used to prohibit updates and register operations within this process. The observers remaining to be notified in a notification process are managed by the variable \texttt{toNotify}. The full power of interaction-based specifications appears in the specification of the \texttt{update} method. If \texttt{update} is called and the set of registered observers \texttt{obs} is empty, the component simply returns (1). Otherwise, the component notifies one of the observers (2); note that \texttt{o} is the receiver of the outgoing \texttt{notify} call message. If the component returns from notifying an observer, it checks whether there are still other observers to notify. If so, the component picks another observer to notify (3). Otherwise, it returns (4).
3 Semantics of Interaction-based Specifications

The semantics of interaction-based component specifications is defined by a special form of labeled transition system, so-called interaction-based transition systems. This section introduces such transition systems and uses them to define the semantics of specifications. The message formalization of [8,9] is adopted.

Let \( \text{Obj} \) be a set of objects, \( \text{Mtd} \) a set of methods with unique names, and \( \text{Dir} \) a two-element set \{call, rtrn\} representing method call (invocation) and return (completion), respectively. Furthermore, let \( \text{Value} \) be a set of values which encompasses all possible parameter values in actual method calls and return values. Then, with \( \overline{v} \) being a shorthand for a possibly empty list of values \( v_1, v_2, \ldots \), the set of messages can be defined as follows.

**Definition 1 (Message).** The set of messages \( \text{Msg} \) (whose instances are denoted by \( \mu \)) is a subset of \( \text{Obj} \times \text{Mtd} \times \text{List}(\text{Value}) \times \text{Dir} \). A tuple \( \mu = (o, m, \overline{v}, d) \) is a message if the callee \( o \) supports the method \( m \).

Instead of representing messages \( \mu \) in the tuple format, we depict them textually: 

\[
\text{call } o.m(\overline{v}) \text{ or } \text{rtrn } o.m() : v.
\]

We define a component as a collection of objects \( O \subseteq \text{Obj} \) (called component objects). All other objects (called environment objects) are part of the environment \( O_{env} = \text{Obj} \setminus O \). Based on this partitioning, we can categorize the messages into incoming and outgoing messages from the perspective of the component. An incoming message is either an invocation message whose callee is a component object, or a completion message whose callee is an environment object. An outgoing message is either an invocation message whose callee is an environment object, or a completion message whose callee is a component object.

**Example (Ingredients for the Subject component).** The set of objects \( \text{Obj} \) is a superset of \{\( \text{sbj}, \text{obs}_1, \text{obs}_2, \ldots, \text{st}_1, \text{st}_2, \ldots \}\) where \( \text{sbj} \) is an instance of the subject component, \( \text{obs}_1, \text{obs}_2, \ldots \) are instances of observers and \( \text{st}_1, \text{st}_2, \ldots \) capture the state information. The set \( \text{Obj} \) might also contain other objects (part of the component or the environment) which do not appear in messages at the boundary of the component. The Subject component is defined as \( O = \{\text{sbj}, \ldots\} \). The set of methods \( \text{Mtd} \) can be seen from the Observer and Subject interfaces. The messages in the example are composed from methods and objects according to the message patterns of the specification. For example, \( \text{call } \text{sbj}.\text{get}() \), \( \text{rtrn } \text{sbj}.\text{get}() : \text{st}_1 \), are two possible messages involving the get method. Note that this in the specification is now replaced by \( \text{sbj} \) in \( O \).

Using the ingredients we now define interaction-based transition systems that represent a component’s possible behavior. A state in such a transition system corresponds to an abstract internal state of a component and a transition captures an interaction (challenge/response) of the component with its environment.

**Definition 2 (Interaction-based Transition System).** Given a set of messages \( \text{Msg} \) and a component \( O \), an interaction-based transition system \( \mathcal{M}(\text{Msg}, O) \),
ITS for short, is a triple $(S, \Theta, s_0)$ where $S$ is a set of states, $s_0 \in S$ is the initial state and $\Theta \subseteq S \times Msg \times Msg \times S$ is a transition relation, where the first message represents an incoming message and the second one an outgoing message.

We drop the set of messages $Msg$ and objects of the component $O$ from the argument of $M$ when they are clear from the context. A transition between two states $s, s'$ is graphically depicted as $s \xrightarrow{\mu_a, \mu_b} s'$.

Given an interaction-based specification, a corresponding ITS can be derived in several steps. The first step is to eliminate the modifies clauses. A clause of the form $\text{modifies } x_1, ..., x_i$ is translated into the additional conjunct $y_1 = \text{old}(y_1) \land ... \land y_j = \text{old}(y_j)$ for the corresponding postcondition where $y_1, ..., y_j$ is the set of abstract variables not appearing in the clause. To reflect the call stack property of our sequential setting\(^2\), states are pairs where the first element is from the Cartesian product of the ranges of the abstract variables in the specification and the second element is the current call stack. The call stack contains for each call the message header which consists of the callee and the method name. The initial state is given by the initial values of the variables and an empty call stack. If an interaction pattern

\text{inMsg outMsg requires pre ensures post}

exists within a specification, then there is a transition $s \xrightarrow{\mu_a, \mu_b} s' \in \Theta$ when

- $s$ satisfies pre, $s'$ satisfies post, the pattern inMsg matches the message $\mu_a$ and the pattern outMsg matches the message $\mu_b$, and
- $cs$ is the call stack of $s$ and $cs'$ of $s'$ and one of the following holds:
  1. $\mu_a$ and $\mu_b$ are calls and $cs' = cs \# \text{header}(\mu_a) \# \text{header}(\mu_b)$
  2. $\mu_a$ and $\mu_b$ are returns and $cs = cs' \# \text{header}(\mu_a) \# \text{header}(\mu_b)$
  3. $\mu_a$ is a call, $\mu_b$ is a return, $\text{header}(\mu_a) = \text{header}(\mu_b)$, and $cs = cs'$
  4. $\mu_a$ is a return, $\mu_b$ is a call, $cs = cs'' \# \text{header}(\mu_a)$, and $cs' = cs'' \# \text{header}(\mu_b)$

What is left is to discard all non-reachable states from the ITS.

In the following, we will refer to the specification and the corresponding ITS interchangeably. The ITS semantics implicitly reflects the possible behavior of the environment, i.e., the clients of the component. As long as the call stack property and the conditions are satisfied, the clients can call any method of the component. Furthermore, clients can stop to communicate at any moment.

Example (ITS of the IntSet component). The ITS $M_{\text{IntSet}} = (S, \Theta, s_0)$ is a transition system of the IntSet component specified in Fig. 1, where $S$ is a subset of $\text{Set<stdint>} \times \text{Stack<Msg>}$, $s_0 = (\{\}, \epsilon)$ and the transition relation $\Theta$ is derived from its interaction patterns.

\(^2\) Other sequential properties (e.g., a component cannot send a message to an object from the environment before that object has been captured by the component) are not considered as they complicate the definitions without adding further insight.
A (finite) component trace \( t = (\mu_1, \mu_2), \ldots \) is induced by a state \( s \) of \( M \) if there exists a sequence of states \( s, s_1, \ldots \) such that \( s_i \xrightarrow{\mu_1, \mu_2} s_{i+1} \) for all \( i > 0 \). We define \( T \) to be the set of all traces induced by the initial state \( s_0 \) of \( M \). This set of traces is prefix-closed, since the components we are considering here are passive (they only respond to incoming messages) and the environment may decide to stop at any moment. Two ITSs \( M_1 \) and \( M_2 \) are behavioral equivalent if they induce the same trace sets.

The trace set induced by an ITS \( M \) can be used to construct another ITS \( M' \), called the canonical ITS of \( M \), where the states are the possible traces. The initial state is the empty trace. A transition \( t' = (\mu_i, \mu_{i+1}) \) exists if \( t = (\mu_i, \mu_{i+1}) \).

The ITS of the specification in Fig. 2 is isomorphic to its canonical ITS.

4 Relating Interaction-based Specifications

In this section we look into three different ways of relating interaction-based specifications via their semantics which is their ITSs. The first relation is based on the number of states two behavioral equivalent ITSs have. This gives the possibility to check whether the specification is encoded efficiently enough compared to its “straightforward” canonical model. The other two relations, namely refinement and extension, describe two possibilities of developing the specification further. Refining a specification means eliminating non-determinism, while extension expands the range of service the specified component offers.

4.1 Reducibility

Two behavioral equivalent ITSs can have very different sets of states. In particular, an ITS may use “less” states than another one to encode the same behavior.

**Definition 3 (Transition System Reduction).** Given two interaction-based transition systems \( M_i = (S_i, \Theta_i, s_{0,i}), i = 1, 2 \), \( M_2 \) is smaller than \( M_1 \) if there is a total onto function \( \alpha : S_1 \to S_2 \) which is a bisimulation ([10]).

For example, the ITS of the state-based specification of \texttt{IntSet} in Fig. 1 is smaller than the ITS of the history-based specification in Fig. 3, although the specifications describe the exact same behavior. The state abstraction function \( \alpha \) maps for example the (history) state \( h = \{(\text{call add}(5), \text{rtn })\} \) and the state \( h' = \{(\text{call add}(5), \text{rtn } ()), (\text{contains}(3), \text{rtn false})\} \) from the second transition system to the same state \( \text{mySet} = \{5\} \) in the first transition system. More generally, \( \alpha \) maps all histories representing the same set into one state.

This demonstrates in particular that well-chosen state-based specifications usually have an ITS that is smaller than their canonical ITS. Smaller transition systems usually simplify verification. Thus, a good specification should lead to an ITS that has no redundant states where a state is called redundant if it is bisimilar to another state, i.e., both states induce the same traces. For example, the ITS of the \texttt{IntSet} specification in Fig. 1 has no redundant states.
4.2 Refinement

Component specifications are usually quite loose and allow room for developers to implement them. This looseness can be characterized by the presence of non-determinism in the transition system. Such non-determinism is weeded out by refining the choices of responses a component can make after receiving an incoming message from the environment. In other words, refinement makes the transition system more precise. In this section, we define the notion of a non-deterministic transition system and refinement in our setting and present a refinement of the IntSet example.

The concept of non-determinism in our case is more specific than in usual transition systems, as our transitions consist of two messages. Non-determinism can arise if two different responses can be given for the same incoming message.

**Definition 4 (Non-deterministic Interaction-based Transition System).**
Given an interaction-based transition system $M = \langle S, \Theta, s_0 \rangle$, $M$ is a non-deterministic transition system if there exist two distinct transitions $s_1 \xrightarrow{\mu_a, \mu_b} s_2$ and $s_1 \xrightarrow{\mu_c, \mu_d} s_3$ such that $\mu_b \neq \mu_c \lor s_2 \neq s_3$.

This definition subsumes the usual notion of non-determinism, where the same labeled transition can lead to two different states ($s_2 \neq s_3$).

An example for refinement of ITSs can be seen in Fig. 5. The solid edges represent incoming messages, while the dashed edges represent outgoing messages and the round black nodes are the actual states of the transition system, while the intermediary gray rectangular nodes are used only for illustration purposes.

The refinement relation is illustrated by the first two transition systems. First, the refined version cannot respond to new incoming messages that are not present in the original transition system (e.g., serving an incoming message $\mu_x$) and cannot produce new responses to an incoming message (e.g., responding $\mu_a$ with $\mu_f$). In other words, the trace sets of the two transition systems are related by a subset relation.

Second, a refined component should be able to respond to an incoming message whenever the original component can. In the illustration, this is shown by...
the preservation of at least one outgoing message for each response (e.g., \( \mu_d \) as a response to \( \mu_a \) while both responses to \( \mu_b \) are preserved). The formalization of the refinement characterization can be seen in the following definition.

**Definition 5 (Trace-based Refinement).** Let \( M_1 \) and \( M_2 \) be two interaction-based transition systems, and \( T_1 \) and \( T_2 \) their trace sets. \( M_2 \) refines \( M_1 \) if

1. \( T_2 \subseteq T_1 \), and
2. \( t, (\mu_a, \mu_c) \in T_1 \land t \in T_2 \implies t, (\mu_a, \mu_d) \in T_2 \), for some \( \mu_d \in \text{Msg} \).

The second requirement states that the refined transition system can respond to a message whenever the original transition system can do. Because the trace sets are prefix-closed, the property is guaranteed for all trace prefixes \( t \) in \( T_2 \).

For verification purposes, it is often simpler to use simulation [11] to express and handle refinement properties (e.g., in [12]):

**Definition 6 (Simulation [11]).** Let \( M_i = \langle S_i, \Theta_i, s_{0,i} \rangle, \ i = 1, 2 \) be interaction-based transition systems. A simulation for \( (M_1, M_2) \) is a binary relation \( \mathcal{R} \subseteq S_2 \times S_1 \) such that

1. \( (s_{0,1}, s_{0,2}) \in \mathcal{R} \), and
2. if \( (s_1, s_2) \in \mathcal{R} \) and \( s_1 \xrightarrow{\mu_a, \mu_c} s'_1 \), then \( s_2 \xrightarrow{\mu_a, \mu_b} s'_2 \) and \( (s'_1, s'_2) \in \mathcal{R} \), for some \( s'_2 \in S_2 \).

However, this is not enough to describe the notion of refinement in the ITS since it does not protect an incoming message present in the original system to have a response in the refined system. Simulation only covers the requirement that the refined transition system does not deviate from the original transition system. Therefore, we have to extend the definition of refinement for ITS.

**Definition 7 (Interaction-based Refinement).** Given two interaction-based transition systems \( M_i = \langle S_i, \Theta_i, s_{0,i} \rangle, \ i = 1, 2 \), \( M_2 \) refines \( M_1 \) if there exists a binary relation \( \mathcal{R} \subseteq S_2 \times S_1 \) such that

1. \( \mathcal{R} \) is a simulation,
2. if \( (s_2, s_1) \in \mathcal{R} \), \( s_2 \xrightarrow{\mu_a, \mu_c} s'_2 \), and \( s_1 \xrightarrow{\mu_a, \mu_c} s'_1 \), then \( (s'_1, s'_2) \in \mathcal{R} \), and
3. if \( (s_2, s_1) \in \mathcal{R} \) and \( s_1 \xrightarrow{\mu_a, \mu_c} s'_1 \), then \( s_2 \xrightarrow{\mu_a, \mu_b, \mu_d} s'_2 \), for some \( \mu_d \in \text{Msg} \) and \( s'_2 \in S_2 \).

To capture the second requirement, we first extends the simulation relation by relating all pair of states that are connected by the same interaction, in addition to insisting on the existence of such pairs. This is needed to ensure that all parts of the refined transition system is considered especially in face of non-determinism. Then for each pair \( (s_2, s_1) \) in the relation and for each transition originating from \( s_1 \) that replies to an incoming message \( \mu_a \) we want that the refined transition system is able to response to that incoming message from \( s_2 \).
Example (Refinement of \texttt{IntSet}). The \texttt{IntSet} ITS can be refined by strengthening the postcondition of the \texttt{some} method specification, requiring that the returned element is a minimal element of the set, i.e., \texttt{result} = \texttt{min(old(mySet))}. There is a simulation relation between the refined and the original transition system as the refined transition system allows less post-states. However, whenever there is a transition where the incoming message involves the \texttt{some} method in the old transition system, then there is also a transition in the refined transition system, namely the one which has as response the minimal element of the set. It is clear that no new responses are added to the refined transition system.

The following theorem shows that we have a sufficient interaction-based definition of refinement. The proof can be found in the appendix.

**Theorem 1.** Interaction-based refinement implies trace-based refinement.

For the necessary condition, we need the ITSs to be deterministic in a traditional way, i.e., ITSs whose transition relation $\Theta$ is a function $S \times \text{Msg} \times \text{Msg} \rightarrow S$.

### 4.3 Extension

Another operation that complements refinement is extension, which allows extension of functionality. In this section, we investigate the notion of extension for our framework.

Extending a component means that existing parts of the component must remain valid, but additional functionality can be specified. This does not only mean that the extended component may have more methods to handle, but also that the extended component may handle new requests of existing methods that were not supported by the original component. The only restriction we have is that the extended component has to retain the same set of answers to an incoming message as the original component. This is illustrated in Fig. 5. The figure on the right represents the extended transition system, which is now able to respond to a new incoming message $\mu_x$. However, the extension can neither add a new response (e.g., $\mu_e$) nor remove any possible responses given $\mu_a$ as the incoming message. Similar as for refinement, we start by giving a trace-based characterization of extension.

**Definition 8 (Trace-based Extension).** Let $M_1$ and $M_2$ be two interaction-based transition systems, and $T_1$ and $T_2$ their trace sets. $M_2$ extends $M_1$ if

1. $T_1 \subseteq T_2$, and
2. $t, (\mu_a, \mu_c) \in T_1 \land t, (\mu_a, \mu_d) \in T_2 \implies t, (\mu_a, \mu_d) \in T_1$.

This definition tells us that if the environment can interact with the original component in a sequence of requests, then the environment must expect exactly the same breadth of responses from the extended transition system. In other words, if the environment assumes interactions with the original component, the extended component can substitute the original without any observable difference. The following definition offers the interaction-based view of extension.
Definition 9 (Interaction-based Transition System Extension). Given two interaction-based transition systems $M_i(M_{i1}, O_i) = \langle S_i, \Theta_i, s_{0,i} \rangle$, $i = 1, 2$, $M_2$ extends $M_1$ if there exists a binary relation $R \subseteq S_1 \times S_2$ where

1. $R$ is a simulation,
2. if $(s_1, s_2) \in R$, $s_1 \overset{\mu_1, \mu_2}{\rightarrow} s'_1$, and $s_2 \overset{\mu_1, \mu_2}{\rightarrow} s'_2$ then $(s'_1, s'_2) \in R$, and
3. if $(s_1, s_2) \in R$, $s_1 \overset{\mu_1, \mu_2}{\rightarrow} s'_1$, and $s_2 \overset{\mu_1, \mu_2}{\rightarrow} s'_2$ then $s_1 \overset{\mu_1, \mu_2}{\rightarrow} s''_1$, for some $s''_1 \in S_1$.

Similar to the refinement definition, we let the relation be an instance of simulation and force this relation to capture pairs of states that are linked together by non-deterministic transitions. Now if we find a pair of states $(s_1, s_2)$ in the relation such that both original transition systems can make a response to the same incoming message $\mu_a$ at those states, then the original transition system must be able to respond in the same manner as its extension.

Example (Extension of IntSet). In this example, we extend the IntSet transition system in two ways. First we add a new method, empty, whose specification is $\text{boolean empty()} \text{ ensures result} = (\text{mySet} = \{\})$. This method checks whether the integer set is empty. Second we change the method specification of some such that it returns a fixed integer, say 0, when the set is empty: $\text{int some()} \text{ ensures (old(mySet) = \{\})} \Rightarrow \text{result} = 0 \land \text{mySet = old(mySet)} \land (\text{old(mySet) \neq \{\})} \Rightarrow \text{result} \in \text{old(mySet)} \land \text{mySet = old(mySet) \{result\}}$

By adding a new method empty, the extension definition is fulfilled since this new method does not add new responses to existing requests. The change of some's specification is also an extension, since in the original transition system some is not called by the environment when the set is empty.

The following theorem shows the relation between the interaction-based extension and the trace-based extension. The proof can be found in the appendix.

Theorem 2. Interaction-based extension implies trace-based extension.

The reverse holds when we relate ITSs which are deterministic in a traditional way, i.e., ITSs whose transition relation $\Theta$ is a function $S \times Msg \times Msg \rightarrow S$.

5 Related Work

State-based models are usually, in the object-oriented setting, specified using contracts as introduced in Eiffel [3]. JML [4] and Spec# [13] adopt this approach to allow modular specifications in Java and C#, respectively. Contracts mainly consist of method pre-/postconditions and class invariants, making it non-trivial to deal directly with callbacks as in the subject/observer example.

Trace-based specifications are well-known in the literature of processes and modules [14,15]. In an object-oriented setting, trace-based specifications have
been used in relation to proof systems [8,16]. A component is described exactly as the set of traces it can admit, where a trace of a component is seen as a projection from the global trace and checked against message invariants which capture communication patterns.

Jass [17] gives specifications in the form of trace assertions, whose semantics is based on CSP. This technique allows to specify our subject example in a similar fashion. However, a clear component model is missing. Similarly, Cheon and Perumandla [18] extend JML to introduce assertions based on call sequences in form of regular expressions, by abstracting from argument values and callee.

A generalized state-based model similar to the one given here is present in the Z++ methodology [19], where object-oriented real-time systems are specified using real-time logic. While similar notions of message predicates are used within the state-based specifications, their purpose is for timing measurement.

In [20], a specification technique for component interaction based on attribute grammars is presented and supported by runtime checks. Akin to our approach, it aims to specify the interaction patterns of components in a clear relation to a trace semantics. Whereas in our approach the state abstraction from the traces and the call stack is implicit, the grammar approach makes these aspects explicit.

Similar to [21], but unlike e.g. [22], we distinguish refinement and extension. A refinement step narrows the options of non-determinism without reducing functionality. Extension allows to add functionality, i.e., its converse cannot be refinement. In this paper, we do not consider data refinement [23], e.g. a refinement of the data passed as parameters.

6 Conclusion and Future Work

In this paper, we presented a specification technique for sequential object-based components that allows us to use state- and trace-based approaches in a unified framework. We gave a formal semantics for the specifications in form of interaction-based transition systems. We explored three different kinds of comparison of the component specifications, namely reduction, refinement, and extension, for the purpose of relating models and evolving them. We illustrated the approach by a non-trivial example.

Our next goal is to extend the approach to concurrent, fully object-oriented components. This includes specifying the flexibility of active components (in the sense that they can initiate communication) and needs more capabilities to specify the exceptions of the component with respect to the environment.

Another important application area of our approach that we want to explore is the relation to verification methods. Three problem areas are of interest: (1) Implementation correctness means that a component implementation satisfies the component specification; (2) refinement and extension proofs use specifications to show that one component is a refinement/extension of another component; (3) compositional reasoning allows to verify properties of systems constructed from components.

Acknowledgment. The authors would like to thank the anonymous referees for their constructive comments and suggestions.
References

Appendix: Proofs for Theorems 1 and 2

For convenience, the related definitions and the two theorems are reiterated.

**Definition 5 (Trace-based Refinement).** Let $M_1$ and $M_2$ be two interaction-based transition systems, and $T_1$ and $T_2$ their trace sets. $M_2$ refines $M_1$ if

1. $T_2 \subseteq T_1$
2. $t, (\mu_a, \mu_c) \in T_1 \land t \in T_2 \implies t, (\mu_a, \mu_d) \in T_2$, for some $\mu_d \in \text{Msg}$.

**Definition 7 (Interaction-based Refinement).** Given two interaction-based transition systems $M_i = (S_i, \Theta_i, s_{0,i}), i = 1, 2, M_2$ refines $M_1$ iff there exists a binary relation $R \subseteq S_2 \times S_1$ such that

1. $R$ is a simulation,
2. if $(s_2, s_1) \in R$, $s_2 \xrightarrow{\mu_a, \mu_c} s_2'$, and $s_1 \xrightarrow{\mu_a, \mu_c} s_1'$ then $(s_2', s_1') \in R$, and
3. if $(s_2, s_1) \in R$ and $s_1 \xrightarrow{\mu_a, \mu_c} s_1'$, then $s_2 \xrightarrow{\mu_a, \mu_d} s_2'$, for some $\mu_d \in \text{Msg}$ and $s_2' \in S_2$

**Theorem 1.** If $M_1$ and $M_2$ fulfills the interaction-based refinement (Def. 7), then their trace sets $T_1$ and $T_2$ fulfills the trace-based refinement (Def. 5). If $M_1$ and $M_2$ are traditionally deterministic and their trace sets $T_1$ and $T_2$ fulfills the trace-based refinement (Def. 5), then $M_1$ and $M_2$ fulfills the interaction-based refinement (Def. 7).

**Proof.** Interaction-based refinement $\implies$ trace-based refinement

The trace inclusion $T_2 \subseteq T_1$ (Def. 5.1) is fulfilled due to the simulation and prefix-closedness property. For Def. 5.3, let $t$ be a trace as shown on the first row of Fig. 6. By induction and Defs. 7.1, 7.2, we can lift this trace to form the same trace in $M_1$, as illustrated by the lower half of Fig. 6 such that all pairs of states are in $R$. 

![Fig. 6. Trace lifting illustration](image-url)
– If $s_{i,1}$ has no outgoing transition, Def. 5.2 is vacuously true.
– If $s_{i,1} \xrightarrow{\mu_a, \mu_c} s_{i+1,1}$, then by Def. 7.3, we get $s_{i,2} \xrightarrow{\mu_a, \mu_d} s_{i+1,2}$ and Def. 5.2 is fulfilled.

Trace-based refinement and traditional determinism $\Rightarrow$ interaction-based refinement

Deterministic $M_1$ and $M_2$ mean that in both transition systems, there is exactly one path from the initial state to some other state $s_1$ (or $s_2$) that induces the trace $t$. Adding trace inclusion and prefix-closedness of the trace sets, we get that there exists a relation $R$ which is a simulation where $(s_2, s_1) \in R$. Definition 7.2 is fulfilled immediately due to the deterministic nature of the transition systems. Furthermore, since there is only one path to $t$, if Def. 5.2 holds, then Def. 7.3 also holds at the point of $(s_2, s_1)$. □

Definition 8 (Trace-based Extension). Let $M_1$ and $M_2$ be two interaction-based transition systems, and $T_1$ and $T_2$ their trace sets. $M_2$ extends $M_1$ if

1. $T_1 \subseteq T_2$
2. $t, (\mu_a, \mu_c) \in T_1 \land t, (\mu_a, \mu_d) \in T_2 \Rightarrow t, (\mu_a, \mu_d) \in T_1$.

Definition 9 (Interaction-based Extension). Given two interaction-based transition systems $M_i = \langle S_i, \Theta_i, s_{0,i} \rangle$, $i = 1, 2$, $M_2$ extends $M_1$ iff there exists a binary relation $R \subseteq S_1 \times S_2$ such that

1. $R$ is a simulation,
2. if $(s_1, s_2) \in R$, $s_1 \xrightarrow{\mu_a, \mu_c} s_1'$, and $s_2 \xrightarrow{\mu_a, \mu_d} s_2'$ then $(s_1', s_2') \in R$, and
3. if $(s_1, s_2) \in R$, $s_1 \xrightarrow{\mu_a, \mu_c} s_1'$, and $s_2 \xrightarrow{\mu_a, \mu_d} s_2'$ then $s_1 \xrightarrow{\mu_a, \mu_d} s_1''$, for some $s_1'' \in S_1$.

Theorem 2. If $M_1$ and $M_2$ fulfills the interaction-based extension (Def. 9), then their trace sets $T_1$ and $T_2$ fulfills the trace-based extension (Def. 8). If $M_1$ and $M_2$ are traditionally deterministic and their trace sets $T_1$ and $T_2$ fulfills the trace-based extension (Def. 8), then $M_1$ and $M_2$ fulfills the interaction-based extension (Def. 9).

Proof. The proof goes in a similar way to the proof for the Theorem 1.

Interaction-based extension $\Rightarrow$ trace-based extension

The trace inclusion $T_1 \subseteq T_2$ (Def. 8.1) is fulfilled due to the simulation and prefix-closedness property. For Def. 8.2, let $t$ be a trace as shown on the first row of Fig. 6 (after exchanging the $M_1$ trace with the $M_2$ trace). By induction and Defs. 9.1, 9.2, we can lift this trace to form the same trace in $M_1$, as illustrated by the lower half of Fig. 6 such that the end state pair is in $R$. If both states $s_{1,1}$ and $s_{1,2}$ can respond to an incoming message $\mu_a$, then by Def. 9.3, $M_1$ can imitate the reply of $M_2$. Therefore, Def 8.2 holds.
Trace-based extension and traditional determinism $\implies$ interaction-based extension

Deterministic $\mathcal{M}_1$ and $\mathcal{M}_2$ mean that in both transition systems, there is exactly one path from the initial state to some other state $s_1$ (or $s_2$) that induces the trace $t$. Adding trace inclusion and prefix-closedness of the trace sets, we get that there exists a relation $\mathcal{R}$ which is a simulation where $(s_2, s_1) \in \mathcal{R}$. Definition 9.2 is fulfilled immediately due to the deterministic nature of the transition systems. Furthermore, since there is only one path to $t$, if Def. 8.2 holds, then Def. 9.3 also holds at the point of $(s_2, s_1)$. $\square$