HATS

Highly Adaptable and Trustworthy Software using Formal Models

Project N^{o}: **FP7-231620**

Project Acronym: **HATS**

Project Title: **Highly Adaptable and Trustworthy Software using Formal Methods**

Instrument: **Integrated Project**

Scheme: **Information & Communication Technologies**

Future and Emerging Technologies

Deliverable D1.1A
Report on the Core ABS Language and Methodology: Part A

Due date of deliverable: (T12)

Actual submission date: 1st March 2010

Revision date: 1st April 2010

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

Organisation name of lead contractor for this deliverable: UIO

Revised version

Integrated Project supported by the 7th Framework Programme of the EC		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Executive Summary:
Report on the Core ABS Language and Methodology: Part A

This document summarizes deliverable D1.1A 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`.

The report contains the definition of the *core ABS* language, an abstract executable modeling language intended to form the basis for extensions later that will support evolvability, features, and further concepts relevant for the modeling and analysis of software families. The report contains conceptually three parts concerning the language:

Formal language definition: We specify the language in form of abstract syntax (including the type system operational semantics). In particular we describe the data type language and the concurrency model, based in *concurrent objects groups* of asynchronously communicating objects.

Example: The concrete syntax of the language is defined by the parsers. We illustrate the language in this document by way of an example, which also is meant to convey the pragmatics of the modeling language.

Tools: We describe the intended tool chain dealing with the (currently core) ABS language, and the current state of the implementation.

List of Authors

Reiner Hähnle (CTH)
Einar Broch Johnsen (UIO)
Bjarte M. Østvold (NR)
Jan Schäfer (UKL)
Martin Steffen (UIO)
Arild B. Torjusen (NR)

Contents

Chapter 1

Introduction

The HATS Deliverable D1.1 is presented in two parts.

- **D1.1A** *Report on the the Core ABS Language and Methodology: Part A* reports on the core ABS language, and
- **D1.1B** *Report on the the Core ABS Language and Methodology: Part B* reports on the HATS methodology.

This is Deliverable D1.1A which presents a core subset of the ABS modeling language. The HATS Description of Work (DoW) describes our intentions for the core ABS: it is intended to be a subset of the full ABS language which does not in itself address software product lines, but rather be a basis for extensions which will capture software product line artifacts such as features and feature integration. Crucial for this basis is that it should inherently support concurrency and distribution, and that components form units of computation which are encapsulated by interfaces and which allow a flexible communication model. Furthermore, as a foundation for a formal method, the core ABS must have a formally defined semantics.

For this core ABS language we propose a *syntax* and, as a formal foundation for the language, a *type system* based on interface types and an *operational semantics*. We have adopted a Java-like syntax for the ABS language. The operational semantics of the core ABS language forms the basis for a prototype language interpreter, defined in Maude [27]. This interpreter supports the simulation of models in the core ABS, as defined by the operational semantics. The tool chain transforming models expressed in the syntax of the (core) ABS language into the run-time syntax of the interpreter is currently being developed. This development will form the basis for the HATS tool platform (the extension of this tool chain with further analysis techniques will be realized in Task 1.5, 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 [52, 31], 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 *abstract 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 [74, 75]. 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 core of the ABS language with a proposed Java-like syntax, a type system, an operational semantics, and a prototype tool chain from source syntax to the simulation environment. We do not present formal properties of the language such as, e.g., a subject reduction proof. We intend to show such properties for the proposed language in a planned conference paper. 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 D5.1 [70]. Chapter 3 introduces the ABS language and proposes an abstract syntax and a basic assertion language. The core ABS language consists of two parts, a functional part and an imperative part. The functional part is presented in Chapter 4 and the imperative part in Chapter 5. Chapter 6 presents an example of a peer-to-peer network node as a model in the core ABS. Chapter 7 shows the current and planned stages of development of the tool chain for the ABS language. Chapter 8 discusses related work. Finally, Chapter 9 summarizes the work presented in this report.

Conventions in this document

The document is intended as a reference (for the further development) of the Core 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. Another purpose is to illustrate the manner in which *concrete* programs are/will be 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. Notational conventions are set out at the beginning of Chapter 3.

We intend the (formal) definitions 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".

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 D5.1 [70].

2.1 Motivation for an Abstract Behavioral Modeling Language

The main rationale of the abstract behavioral modeling is to fill the gap between highly abstract, generic, and often only structural modeling languages such as UML [65], B [3], and ASM [18], and highly specific ones such as JML [22] or Spec# [14]. 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 [30]. 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 deterministic whereas non-determinism is an essential abstraction mechanism in design of adaptable systems.

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 intend to 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*. This deliverable presents the core ABS language. Below, in Section 2.2, we elaborate on design choices guiding the design of the core language and then, in Section 2.3, we discuss how these choices relate to the high-level requirements of Deliverable D5.1 [70]. In the continuation of this work, we intend to

- extend the Core ABS with structuring mechanisms to support feature integration in ABS models, and
- exploit the abstractions of the Core ABS to adapt models to particular deployment scenarios for the deployment of a product from a Software Product Line.

2.2 Design Choices for the Core ABS

This section discusses some of the main design choices underlying the design of the Core ABS 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, which will be incorporated into the language in the later stages, will *not* necessarily be based on standard class *inheritance*.

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* [74, 75]. COGs generalize the concurrency model of Creol [52, 31], 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 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* [43]. This is widely considered to be a simple and adequate technique for dealing with partiality in specifications [44].

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 [75].

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 on, 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 [7] 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 the universe types [78].

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 2.2. This is necessitated by the fact that feature modeling capabilities and other constructs are added after the core language design is established.

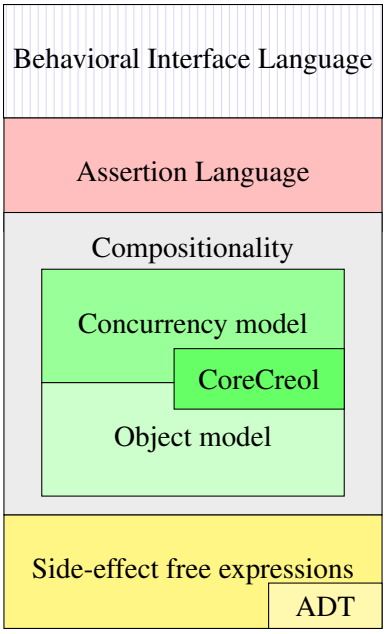

Figure 2.1: HATS core ABS language layers.

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 D5.1 [70].

As this document only provides the description of the *Core* ABS language, several requirements are not yet addressed and will be addressed by other work tasks. These include aspects of the integration of ABS with product line engineering (MR3–MR6), resource guarantees (MR16), protocol analysis (MR17), and extensibility (MR20). In addition, several requirements concerning the HATS methodology are addressed in deliverable D1.1B (MR1, MR2, MR14). Concerning the Core ABS language, requirements MR11, MR12, MR13, and MR18 are relevant.

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 [60]. Functional language concepts have lately also been integrated into object-oriented languages like Scala[1], 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

[1] `http://www.scala-lang.org`

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 difficult, difficult to understand, and requires expert programmers [66]. In particular the thread-model is very error-prone with respect 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.

MR12 (Usability), MR13 (Reducing manual effort), and MR18 (Integrated Environment Support). Realizing usability is not a central objective of D1.1A, 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 (see Section 7.4). Tool support will constantly be improved during the project and integrated into the Eclipse IDE (addressing MR18). All tools for ABS will 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.

Chapter 3

Syntax

In this chapter we present the syntax of the core ABS language.

The language is specified in abstract syntax. In writing down the syntax, we will make use of a number of syntactic conventions. We use mathematical *italics* for non-terminals in the grammars. Lists of syntactic entities are denoted by an overbar, as in $\overline{e}$ for a list of expressions e, and non-empty ones by a raised plus symbol, as in I^+. Optional parts are written as $[\ldots]$. The language fixes some design choices in one way or the other. We discuss the reasons behind these choice or alternatives in the text.

3.1 A Language for Abstract Behavioral Specification

3.1.1 Abstract Syntax

We first discuss the abstract syntax of the core ABS language, which is given in Figure 3.1. 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 55.

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*[1]) 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 $Co(\overline{T})$. *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.

[1] If 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.

P	::=	$\overline{Dd}\ \overline{F}\ \overline{In}\ \overline{Cl}\ [B]$	program
In	::=	interface I [extends I^+] $\{\overline{M_s}\}$	interface declaration
Cl	::=	class $C(\overline{Tf})$ [implements I^+] { $\overline{Tf}$ $[B]$ $\overline{M}$ }	class definition
M	::=	$M_s\ B$	method definition
M_s	::=	$T\ m(\overline{T\,x})$	method signature
B	::=	{ $\overline{T\,x}\ s$ }	blocks
T	::=	$I \mid D \mid \mathsf{Fut}(T) \mid \mathsf{Void} \mid \mathsf{Bool} \mid \mathsf{Guard}$	types
v	::=	$x \mid \mathsf{this}.f$	state variables
e	::=	$e_p \mid e_e$	expressions
e_p	::=	$v \mid e_f \mid \mathsf{null} \mid e_p = e_p$	pure expressions
e_e	::=	new [cog] $C(\overline{e_p}) \mid e_p!m(\overline{e_p}) \mid e_p.m(\overline{e_p}) \mid e_p.\mathtt{get}$	expressions with side effects
s	::=	$v := e \mid \mathsf{await}\ g \mid \mathsf{skip} \mid \mathsf{suspend} \mid e$	statements
		$\mid \mathsf{if}\ (e_p)\ s\ \mathsf{else}\ s \mid \mathsf{while}\ (e_p)\ s \mid s;s$	
g	::=	$v? \mid g \wedge g \mid e_f$	guards

Dd	::=	data $D\ \{\overline{Co(\overline{T})}\}$	data type declaration
F	::=	def $T\ fn(\overline{T\,z}) = e_f$	function declaration
t	::=		term
		z	logical variables
	$\mid$	$Co(\overline{e_p})$	constructor expressions
	$\mid$	(e_p, e_p)	pair constructor
p	::=	$z \mid Co(\overline{p}) \mid (p,p)$	pattern
e_f	::=	t	functional expressions
	$\mid$	$\mathsf{let}\,z{:}T = e_p\ \mathsf{in}\ e_f$	local value definition
	$\mid$	$fn(\overline{e_p})$	function application
	$\mid$	$\mathsf{case}\ \ e_p\ \mathsf{of}\ \overline{b}$	case expression
b	::=	$p\ \triangleright\ \ e_f$	branch

Figure 3.1: ABS abstract syntax

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 theoretical) similarities. To be explicit, we call the constructor used to create new instances of the class as a constructor method or a class constructor, while the constructors for data types are called data type constructors. The class constructor takes arguments, which are given as a number of formal parameters $\overline{f}$ of types $\overline{T}$ to the class. These parameters correspond to a subset of the fields of the class, which are set to the corresponding actual parameters upon instantiation. This already corresponds to a restricted use of a class constructor. A class can implement a number of interfaces, whose declared methods must then be supported by the class. The body of a class definition consists of *field definitions* $\overline{T\ f}$ and *method definitions* $\overline{M}$. We assume that the field definitions mentioned in the body of the class definition are disjoint from the fields initialized by the class parameters. (This can be enforced by the typing rules.) In the class declaration, there is an (optional) explicit initialization block that is used to instantiate the fields of a class instance. During this initialization, further objects may be instantiated in turn, but the new object(s) cannot start interacting yet or pass their new identity on to other activities. Fields that are not initialized by the constructor method nor by the initializing block are left undefined.

Method definitions M have a return type T, a name m, formal parameters $\overline{T\ x}$, and a method body which consists of locally declared variables $\overline{T\ x}$ and a sequence of statements $\overline{s}$.

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_e. 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 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* $Co(\overline{e_p})$ 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 let-construct. The expression $fn(\overline{e_p})$ 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 skip, conditionals, while-loops, grouping $\{\overline{s}\}$ of statements, assignments $v := e$ where v is a variable and e an expression. In addition there are statements for scheduling control; 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, await g releases the lock if the guard g evaluates to 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 [31].

3.1.2 Concrete Syntax

The concrete user syntax will be defined by the ABS parser, which is currently at the development stage. The concrete syntax is close to what is listed in Fig. 3.1, however, a few details have been left out; for example:

- The list notation $\overline{X}$ is assumed to indicate a possibly empty sequence of Xs. A separator is introduced where necessary. Consequently $\overline{In}$ stands for $In_1 \ldots In_n$ and similarly for $\overline{Cl}$, $\overline{M_s}$, and $\overline{b}$. $\overline{I}$ stands for $I_1, \ldots, I_n$, $\overline{T\,x}$ stands for $T_1\ x_1, \ldots, T_n\ x_n$, and similarly for $\overline{T\,f}$, $\overline{e}$, and $\overline{Co(\overline{D})}$. $\overline{s}$ stands for $s_1; \ldots; s_n$.
- Semicolon is used as statement terminator. We also introduce an empty statement which consists of a semicolon only.
- Semicolon separates a nonempty list: $\overline{T\,x}$ or $\overline{T\,f}$, and the following list of statements or methods.
- For a class without parameters, C abbreviates $C()$.
- In the abstract syntax there is no return statement, instead the last statement of a body "returns" the value. To be compatible with *Java*, the concrete syntax has a return statement.

Examples We illustrate declarations in the functional sub-language.

- Function declaration (F):

```
def Int inc(Int x) = plus(x,1) ;

def Int plus(Nat x, Nat y) =
  case y {
```

```
    0 => x ;
    S(z) => plus(S(x),z) ;
  }
```

- Data type declaration (*Dd*):

```
data IntList { IntNil , Cons(Int , IntList) }
```

Further examples are given in Section 3.1.3 and Chapter 6.

3.1.3 Assertion Language

There is a built-in data-type for Booleans in the core ABS. We give the basic definitions of the data type Bool below, which also illustrates the concrete syntax of the core ABS.

```
data Bool { True , False }

def Bool and(Bool x , Bool y) =
  case x {
    True => y ;
    False => False ;
  }

def Bool or(Bool x , Bool y) =
  case x {
    True => True ;
    False => y ;
  }

def Bool not(Bool x) =
  case x {
    True => False ;
    False => True ;
  }
```

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 Bool.

The basic assertions over local state are given in terms of possibly quantified Boolean expressions. We define the state predicates with the following BNF (where z is a logical variable as before):

$$pr ::= e_p \mid \forall z \cdot pr \mid \exists z \cdot pr$$

Chapter 4

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. Left out are polymorphism, subtyping, and genericity, which are planned for future extensions.

4.1 Syntax

The abstract syntax is given in Table 4.1. 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 *delare* an identifier (a variable name, a name for a data type etc). In the state of the overall language now (cf. Figure 3.1), 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 3.1 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 σ and the heap, i.e., the values of the fields. The operational semantics dealing with thec imperative part is covered in Figures 5.5 and 5.6. 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 $\lambda\vec{x}{:}\vec{T}$ are not included into the category of values. Implicitly, λ-abstractions are values in the sense that there is no evaluation under a λ-abstraction. In our syntax, however, λ-abstractions are no stand-alone expressions; they can be used in function declarations only.

v	::=	$x \mid t$	values
t	::=		term
		$Co(\overline{t})$	constructor term
	$\mid$	(t,t)	pair constructor
p	::=	$x \mid Co(\overline{p}) \mid (p,p)$	pattern
e	::=		expressions
		v	values
	$\mid$	$\mathsf{let}\, x{:}T = v \,\mathsf{in}\, e$	value definition
	$\mid$	$\mathsf{let}\, x{:}T = \lambda x{:}T.e_f \,\mathsf{in}\, e$	function definition
	$\mid$	$\mathsf{let}\ \mathtt{data}\ D = Co(T) \ldots Co(T) \,\mathsf{in}\, e$	data type definition
	$\mid$	$\mathsf{case}\ \ v\ \mathsf{of}\ \overline{b}$	case
	$\mid$	$e\ e$	function application
b	::=	$p \rhd e$	branch

Table 4.1: Abstract syntax of the expression language

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

Remark 4.1.1 (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.* □

4.2 Typing

The type system[2] 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 $*$ is the kind for proper types, $K_1 \to K_2$ 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 $*$ 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 $*$ are given in Table 4.2.

K	::=	$*$	kinds
T	::=	$U \mid \underline{U \to U}$	types
U	::=	$n \mid T \times T \mid \mathsf{Unit}$	
Γ	::=	$\bullet \mid \Gamma, x{:}T \mid \Gamma, D{:}K \mid \Gamma, Co{:}T$	contexts

Table 4.2: 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 bother to include the rest of the types mentioned in the abstract syntax of Figure 3.1 resp. of Figure 5.1 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.

At 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 4.2.1). 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 4.4 introduces a number of *predefined* types.

Remark 4.2.1 (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 *type contexts* (see Table 4.2). The empty context is written as $\bullet$. Otherwise, Γ 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 Γ 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 Γ. 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 Γ, each constructor mentioned in the context belongs to exactly one data type. We write $\Gamma \vdash \sum \vec{Co}{:}\vec{T} \to D$ to assert that Co_i are all the constructors for the data type n and their respective input type is T_i. That statement $\Gamma \vdash \sum \vec{Co}{:}\vec{T} \to D$ 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 n. 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 Γ 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 $dom(\Gamma)$ for the domain of those mappings.

The type system uses *judgments* as shown in Table 4.3. We discuss and formalize them in turn in the following.

Γ	$\vdash$	ok	well-formed context
Γ	$\vdash$	$T : K$	kinding
Γ	$\vdash$	$e : T$	typing
Γ	$\vdash_m$	$p : T :: \Gamma$	typing for guarded expressions

Table 4.3: Judgments

4.2.1 Well-formedness

Well-formedness basically assures that a context Γ does not contain a binding to a name twice and that the variables or names in the domain of Γ are bound only to type of base kind $*$. The judgments are of the form $\Gamma \vdash ok$ (see Table 4.3) and the corresponding rules are given in Table 4.4. 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 $*$. Also a data type named D must be of kind $*$. For constructor names, we require that the input type U is of kind $*$. Note that for checking the kind of type U, it is assured that n is already defined in Γ as implied by the first premise of the rule. In this way, U can contain n recursively.

$$\frac{}{\bullet \vdash ok}\ \text{C-EMPTY} \qquad \frac{\Gamma \vdash ok \qquad x \notin dom(\Gamma) \qquad \Gamma \vdash T : *}{\Gamma, x{:}T \vdash ok}\ \text{C-VAR}$$

$$\frac{\Gamma \vdash ok \qquad D \notin dom(\Gamma)}{\Gamma, D{:}* \vdash ok}\ \text{C-DNAME}$$

$$\frac{\Gamma \vdash ok \qquad Co \notin dom(\Gamma) \qquad \Gamma \vdash D : K \qquad \Gamma \vdash U : *}{\Gamma, Co{:}U \to D \vdash ok}\ \text{C-CONAME}$$

Table 4.4: Well-formed contexts

Remark 4.2.2 (Contexts & well-formedness)**.** *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* Γ *correspond to the* syntax table *and might be implemented by a hash table or a similar data structure.* □

4.2.2 Kinding

As said, kinding is captured in judgments of the form $\Gamma \vdash T : K$. The corresponding rules are given in Table 4.5. In rule K-NAME, the kind for a data type name D is looked up in the context. Note that, at the current state, the well-formedness restriction on the contexts assures, that K equals $*$. Arrow types are of kind $*$, provided all mentioned constituent types are, as well (see rule K-ARROW).

$$\frac{\Gamma(D) = K \qquad \Gamma \vdash ok}{\Gamma \vdash D : K}\ \text{K-NAME} \qquad \frac{\Gamma \vdash U_1 : * \qquad \Gamma \vdash U_2 : *}{\Gamma \vdash U_1 \times U_2 : *}\ \text{K-PAIR}$$

$$\frac{\Gamma \vdash U_1 : * \qquad \Gamma \vdash U_2 : *}{\Gamma \vdash U_1 \to U_2 : *}\ \text{K-ARROW} \qquad \frac{\Gamma \vdash ok}{\Gamma \vdash \mathsf{Unit} : *}\ \text{K-UNIT}$$

Table 4.5: Kinding

4.2.3 Type system

The type system is shown in Table 4.6 (and Table 4.7). The type for variables is looked up from the type context (see rule 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 T-VDEF checks the body e in a type context extended by the binding $x{:}T_1$ for the the variable, and furthermore, the value v is checked to be of the expected type T_1. Function definitions are treated by rule T-FDEF, which works similarly: The function used in the definition and bound to f (represented as λ-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 T-DDEC. 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.[3] 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 λ-expressions at the user syntax (avoiding higher-order functions this way). We sometimes write $f(e)$ and $Co(e)$ for applications $f\ e$ and $Co\ e$.

Remark 4.2.3 (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 $\overline{b}$. Thus, e must be typed by the name n 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.

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

$$\Gamma \vdash_m p{:}T :: \Gamma' \,. \tag{4.1}$$

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

Definition 4.2.4 (Matching)**.** Matching *of a pattern with a type is given inductively by the rules of Table 4.7.*

The pattern p is a constructor term possibly containing variables (see the abstract syntax of Table 4.1). 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 routing" 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 Γ. Matching a term whose top-level construct is a constructor name c_i is captured in TM-CONSTR. The first premise consults Γ 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 Γ' 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

[3] That differentiates the constructor types from *variant types*, which are technically related and which we do not include in the language.

$$\frac{\Gamma(x) = T \qquad \Gamma \vdash T : *}{\Gamma \vdash x : T}\ \text{T-VAR} \qquad \frac{\Gamma \vdash v : T_1 \qquad \Gamma, x{:}T_1 \vdash e : T_2}{\Gamma \vdash \mathsf{let}\, x{:}T_1 = v \,\mathsf{in}\, e : T_2}\ \text{T-VDEF}$$

$$\frac{\Gamma, f{:}T_1 \to T_2, x{:}T_1 \vdash e' : T_2 \qquad \Gamma, f{:}T_1 \to T_2 \vdash e : T_3}{\Gamma \vdash \mathsf{let}\, f{:}T_1 \to T_2 = \lambda x{:}T_1.e' \,\mathsf{in}\, e : T_3}\ \text{T-FDEF}$$

$$\frac{D, Co_i \text{ fresh} \qquad \Gamma, D{:}*, \ldots, Co_i : T_i \to D, \ldots \vdash e : T}{\Gamma \vdash \ \mathsf{let\ data}\, D = Co_1(T_1) \ldots Co_n(T_n) \,\mathsf{in}\, e : T}\ \text{T-DDEC}$$

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash (e_1, e_2) : T_1 \times T_2}\ \text{T-PAIR} \qquad \frac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}\ \text{T-APP}$$

$$\frac{\Gamma \vdash Co : T_1 \to T_2 \qquad \Gamma \vdash e : T_1}{\Gamma \vdash Co(e) : T_2}\ \text{T-CONSTR}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : D \qquad \Gamma \vdash \sum \overline{c}{:}\overline{T} \to D \qquad \overline{p} \text{ cover the constructors exhaustively} \\ b_i = p_i \rhd e_i \qquad \Gamma \vdash_m p_i : D :: \Gamma' \qquad \Gamma' \vdash e_i : T\end{array}}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \overline{b} : T}\ \text{T-CASE}$$

Table 4.6: Type system

$$\frac{x \notin dom(\Gamma)}{\Gamma \vdash_m x : T :: (\Gamma, x{:}T)}\ \text{TM-VAR} \qquad \frac{\Gamma \vdash Co_i{:}T_i \to T \qquad \Gamma \vdash_m p : T_i :: \Gamma'}{\Gamma \vdash_m Co_i(p) : T :: \Gamma'}\ \text{TM-CONSTR}$$

$$\frac{\Gamma \vdash_m e_1 : T_1 :: \Gamma' \qquad \Gamma' \vdash_m e_2 : T_2 :: \Gamma''}{\Gamma \vdash_m (e_1, e_2) : T_1 \times T_2 :: \Gamma''}\ \text{TM-PAIR}$$

Table 4.7: Type system (matching)

independently. Conceptually, the left sub-pattern is treated first, and afterwards the right. The *effect* of successfully matching the p_1 is changing the context Γ to Γ', which is taken as the "pre-context" for checking p_2, which may affect a further change to Γ'', which also is the resulting context overall in the conclusion of the rule. Note that by requiring that bindings occur unique in context Γ, 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 4.2.5 (Type checking)**.** *Let's assume a (monomorphic) data type named* Pair *with one constructor* pair *of type* $\mathsf{Int} \times \mathsf{Bool} \to \mathsf{Pair}$. *We assume that the type context* Γ *contains the relevant bindings, and the expression to type check is the one-armed case expression:*

$$\mathsf{case}\ \mathtt{pair}(\mathtt{1}, \mathtt{true})\ \mathsf{of}\ (\mathtt{pair}(x, y))\ . \tag{4.2}$$

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

$$\dfrac{\ldots \qquad \dfrac{\dfrac{\Gamma \vdash_m x{:}\mathsf{Int} :: \Gamma' \quad \Gamma' \vdash_m y{:}\mathsf{Bool} :: \Gamma''}{\Gamma \vdash_m (x,y){:}\mathsf{Int} \times \mathsf{Bool} :: \Gamma''}\ \text{TM-PAIR}}{\Gamma \vdash_m \mathsf{pair}(x,y){:}\mathsf{Pair} :: \Gamma''}\ \text{T-CONSTR} \qquad \Gamma'' \vdash e' : T}{\Gamma \vdash \mathsf{case}\ \mathsf{pair}(1,\mathsf{true})\ \mathsf{of}\ (\mathsf{pair}(x,y) \rhd e') : T}\ \text{T-CASE}$$

[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.

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* $\mathsf{pair}(1,\mathsf{true})$ *with the data type named* Pair*, whose definition is looked up in* T-CONSTR*, i.e.,* $\Gamma \vdash \mathsf{pair} : \mathsf{Int} \times \mathsf{Bool} \to \mathsf{Pair}$ *(this premise of* T-CONSTR *is not shown in the derivation). This then is matched with the pair* (x,y)*, which is then split by* TM-PAIR *into one sub-goal for each variable, where the derivation ends. Matching x against the type of integers adds the binding* x:Int *to* Γ*, i.e.,* $\Gamma' = \Gamma, x{:}\mathsf{Int}$*, which is the "starting context" used to match y, which then yields* $\Gamma'' = \Gamma, x{:}\mathsf{Int}, y{:}\mathsf{Int}$*.* □

The type system is split into 3 levels, i.e., 3 recursive procedures, one depending on the other. Type checking (including matching) from Tables 4.6 and 4.7 uses kinding derivation from Table 4.5 as sub-derivation, which in turn is relying on well-formedness check for context from Table 4.4. The following lemmas are standard "sanity" checks for the type system, that the different parts fit together appropriately.

Lemma 4.2.6 (Well-formedness)**.** $\Gamma \vdash T : K$ *implies* $\Gamma \vdash ok$*.*

Proof. By straightforward induction on derivations over the rules from Table 4.5.

Case: K-NAME and K-UNIT
Immediate, by the premises of the corresponding rule.

Case: K-PAIR and K-ARROW
By straightforward induction.

□

Lemma 4.2.7 (Well-kindedness)**.** *If* $\Gamma \vdash e : T$*, then* $\Gamma \vdash T : *$*.*

Proof. By straightforward induction on the length of the derivation. Without arrow-kinds, i.e., without type operators at the current stage, the well-kindedness boils down to a context-free syntax check.

Case: T-VAR
Directly by the premise of the rule.

Case: T-VDEF, T-FDEF, and T-DDEC
By straightforward induction.

Case: T-PAIR
By induction, $\Gamma \vdash T_1 : *$ and $\Gamma \vdash T_2 : *$, and the result follows by rule K-PAIR.

Case: T-APP and T-CONSTR
By induction, $\Gamma \vdash T_1 \to T_2 : *$, and inverting rule K-ARROW yields $\Gamma \vdash K_1 : *$, as required. The case for T-CONSTR works analogously.

Case: T-CASE
By induction.

□

Lemma 4.2.8 (Matching preserves well-formedness)**.** *If* $\Gamma \vdash ok$ *and* $\Gamma \vdash_m p : T :: \Gamma'$*, then* $\Gamma' \vdash ok$*.*

Proof. By induction on the derivation.

Case: TM-VAR
Straightforward, by the premise that $x \notin dom(\Gamma)$.

Case: TM-CONSTR
By straightforward induction.

Case: TM-PAIR
By straightforward induction.

□

Lemma 4.2.9 (Monomorphism)**.** *If* $\Gamma \vdash e : T_1$ *and* $\Gamma \vdash e : T_2$*, then* $T_1 = T_2$*.*

Proof. By straightforward induction on the length of derivation. The case for rule T-CASE depends also on the (easy) observation that matching is deterministic, as well: If $\Gamma \vdash p : T :: \Gamma_1$ and $\Gamma \vdash p : T :: \Gamma_2$, then $\Gamma_1 = \Gamma_2$. □

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 4.2.10 (Termination)**.** *Type checking is decidable.*

Proof. The type checking is grouped into 4 different formal systems/procedures: well-formedness check, kinding, type-checking, and matching. The 4 systems are not strictly layered: the rules for well-formedness and the ones for kinding are *mutually* recursive (and so must be considered jointly in termination). Typing depends on kinding and matching, but not vice versa.[5] Now: Well-formedness and kinding from Tables 4.4 and 4.5, using as easy termination measure the "size" of the judgment (in terms of "symbols"). The type system and the system for matching clearly terminate, as they recursively deconstruct the term. □

4.3 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, also the semantics poses no real challenges. The semantics is given as reduction steps over configurations given as:

$$\Gamma \vdash e \tag{4.3}$$

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 4.8.

The right-hand side of each rule has the let construct as top-level construct, and the form of the let-expression determines the choice of rule.

The first rule 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 ;-construct, the rule expresses that $(e_1';e_2');e$ can be restructured to $e_1';(e_2';e)$). Rule 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 Γ 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.[6]

The two 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 4.3.2). 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 σ the matching substitution). In that case the branch is selected and evaluation continues with the "body" of the branch, with the matching substitution σ applied. Otherwise, with no match, the branch is discarded and the rest of the branch list is tried for a match (see rule 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 deadlocks.

Remark 4.3.1 (Recursion)**.** *The semantics is defined as a (small-step) operational semantics of configurations as given in equation (4.3). 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* Γ*-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.* □

[5] To be precise, matching, technically, depends on typing by the corresponding premise of TM-CONSTR, but this is a leaf of the derivation, so no mutual recursion necessary here for the termination argument.

[6] 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.

$$\Gamma \vdash \mathsf{let}\, x{:}T = (\mathsf{let}\, x' : T' = e'_1 \,\mathsf{in}\, e'_2)\,\mathsf{in}\, e \to \Gamma \vdash \mathsf{let}\, x' : T' = e'_1 \,\mathsf{in}\, (\mathsf{let}\, x : T = e'_2 \,\mathsf{in}\, e) \quad \text{R-SEQ}$$

$$\Gamma \vdash \mathsf{let}\, x{:}T = v \,\mathsf{in}\, e \to \Gamma \vdash e[v/x] \quad \text{R-LET}$$

$$\Gamma \vdash \mathsf{let}\, f{:}T' = \lambda(x{:}T).e_1 \,\mathsf{in}\, e_2 \to \Gamma, f : T' = \lambda x{:}T.e_1 \vdash e_2 \quad \text{R-FDEC}$$

$$\frac{\Gamma' = \Gamma, D{:}*, \ldots Co_i{:}T_i \to n, \ldots}{\Gamma \vdash \mathsf{let}\, \mathtt{data}\, D = Co_1(T_1) \ldots Co_n(T_n)\,\mathsf{in}\, e \to \Gamma' \vdash e}\ \text{R-DDEC}$$

$$\frac{}{\Gamma \vdash \mathsf{let}\ x{:}T = (\mathsf{if}\, v = v \,\mathsf{in}\, e_1\, \mathsf{else}\, e_2)\,\mathsf{in}\, e \to \Gamma \vdash \mathsf{let}\ x{:}T = e_1 \,\mathsf{in}\, e}\ \text{R-COND}_1$$

$$\frac{v_1 \neq v_2}{\Gamma \vdash \mathsf{let}\ x{:}T = (\mathsf{if}\, v_1 = v_2 \,\mathsf{in}\, e_1\, \mathsf{else}\, e_2)\,\mathsf{in}\, e \to \Gamma \vdash \mathsf{let}\ x{:}T = e_1 \,\mathsf{in}\, e}\ \text{R-COND}_2$$

$$\frac{\Gamma = \Gamma_1, f : T = \lambda(y{:}T).e_1, \Gamma_2}{\Gamma \vdash \mathsf{let}\, x{:}T = f(e_2)\,\mathsf{in}\, e \to \Gamma \vdash \mathsf{let}\, x{:}T = e_1[y/e_2]\,\mathsf{in}\, e}\ \text{R-APP}$$

Table 4.8: Semantics

Remark 4.3.2 (First-match)**.** *The two* R-CASE*-rules of Table 4.8 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 4.3.3 (Subject reduction)**.** *If* $\Gamma \vdash e : T$ *and* $\Gamma \vdash e \longrightarrow \Gamma' \vdash e'$*, then* $\Gamma' \vdash e' : T$*.*

Proof. By inspection of the rules from Table 4.8.

Case: R-SEQ
Straightforward by inverting the rule T-LET two times and induction.

Case: R-LET
By preservation of typing under substitution.

Case: R-DDEC
We are given $\Gamma \vdash \mathsf{let}\, \mathtt{data}\, D = Co_1(T_1) \ldots Co_n(T_n)\,\mathsf{in}\, e : T$. Inverting the corresponding type rule T-DDEC directly gives the required well-typedness of the post-configuration.

Case: R-FDEC
We are given $\Gamma \vdash \mathsf{let}\, f{:}T = \lambda(x{:}T_1).e_1\,\mathsf{in}\, e_2 : T'$. By inverting the type rule for function declaration we know that T is of the form $T_1 \to T_2$ for some type T_2, and furthermore that $\Gamma, f{:}T_1 \to T_2 \vdash e_2 : T'$, as required.

Case: R-CASE$_1$
We are given $\Gamma \vdash \mathsf{let}\ x{:}T = \mathsf{case}\ v\ \mathsf{of}\ b\ \overline{b}\,\mathsf{in}\, e : T_2$. By the premise of the typing rule T-CASE we have $\Gamma' \vdash e_i : T$ for the bodies e_i of all the branches of the case construct, including the first one that matches in case T-CASE$_1$. The result follows by preservation of typing under substitution, using the appropriate rule from T-VDEF, T-FDEF, or T-DDEC, depending on the form of the body of the branch.

Case: R-CASE$_2$
Straightforward, as the branch list is only shortened in the step.

Case: R-APP
We are given $\Gamma \vdash \mathsf{let}\, x = f\ e_2\,\mathsf{in}\, e : T$ and furthermore $\Gamma(f) = \lambda x{:}T'.e'$. The result follows with the help of preservation of typing under substitution.

□

The following lemma states formally that the evaluation order of ABS expressions does not matter:

Lemma 4.3.4 (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* $\Gamma_2 \vdash e_2$*.*

Proof. By inspection of the rules. □

4.4 Predefined types

The data type language allows to introduce user-defined data types. It is convenient to have a collection of standard ones plus their operations predefined. The examples we show in Chapter 6 and at various different places contain a number of those (partly in concrete syntax). They include Boolean, natural numbers, lists, etc.

Chapter 5

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.

5.1 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 5.1 repeats the available types T.

$$T \quad ::= \quad I \mid D \mid \mathsf{Fut}(T) \mid \mathsf{Void} \mid \mathsf{Bool} \mid \mathsf{Guard}$$

Figure 5.1: Types

The type system (including the part for subtyping) uses *typing contexts* Γ. They serve as a finite mapping from identifiers/names to (mainly) their types. Considering Γ as a finite mapping, we use $\Gamma(x) = T$ for looking up the type binding for x; i.e., if Γ contains the pair $x{:}T$. Similarly for other bindings in Γ.

5.1.1 Subtyping

The type system, described in Section 5.1.2, uses a *subtype relation* as subsidiary statement. We write

$$\vdash T \preceq T' \tag{5.1}$$

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 5.2. 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 5.1.1 (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.* □

$$\frac{}{\vdash T \preceq T}\ (\textsc{Sub-Refl}) \qquad \frac{\vdash T \preceq T' \quad \vdash T' \preceq T''}{\vdash T \preceq T''}\ (\textsc{Sub-Trans}) \qquad \frac{\vdash T \preceq T'}{\vdash \mathsf{Fut}(T) \preceq \mathsf{Fut}(T')}\ (\textsc{Sub-Fut})$$

$$\frac{\mathsf{interface}\ I\ \ \mathsf{extends}\ I^+ \ldots \quad I' \in I^+}{\vdash I \preceq I'}\ (\textsc{Sub-IntfDecl}) \qquad \frac{\mathsf{class}\ C(\overline{T f})\ \ \mathsf{implements}\ I^+ \ldots \quad I' \in I^+}{\vdash I_C \preceq I'}\ (\textsc{Sub-ClassDecl})$$

Figure 5.2: Subtype relation

5.1.2 Typing

The type system for the imperative part is given in Figure 5.3.

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). 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 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 Γ (see T-VAR). A field lookup is typed by first determining the type of this and then using this type to find the type of the field declaration, using the *ftype* function (see T-FIELD). The 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 $\mathsf{new}[\ \mathsf{cog}\ ]\ C(\overline{e})$. Instantiation gives a new object, which is typed by I_C. The judgment $\vdash C : \overline{T} \to I_C$ 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 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 *mtype* (see T-SYNCCALL). The get operation can only be applied to a future, which has to be of some type $\mathsf{Fut}(T)$. 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. Being a statement, the type of the assignment itself is Void. The await statement expects a guard typed to Guard as argument and is then typed to Void (see T-AWAIT). The skip and suspend statements have no premises and are simply typed to Void (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' my be different than Void. 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 $T \succeq T'$ (see rule T-SUB).

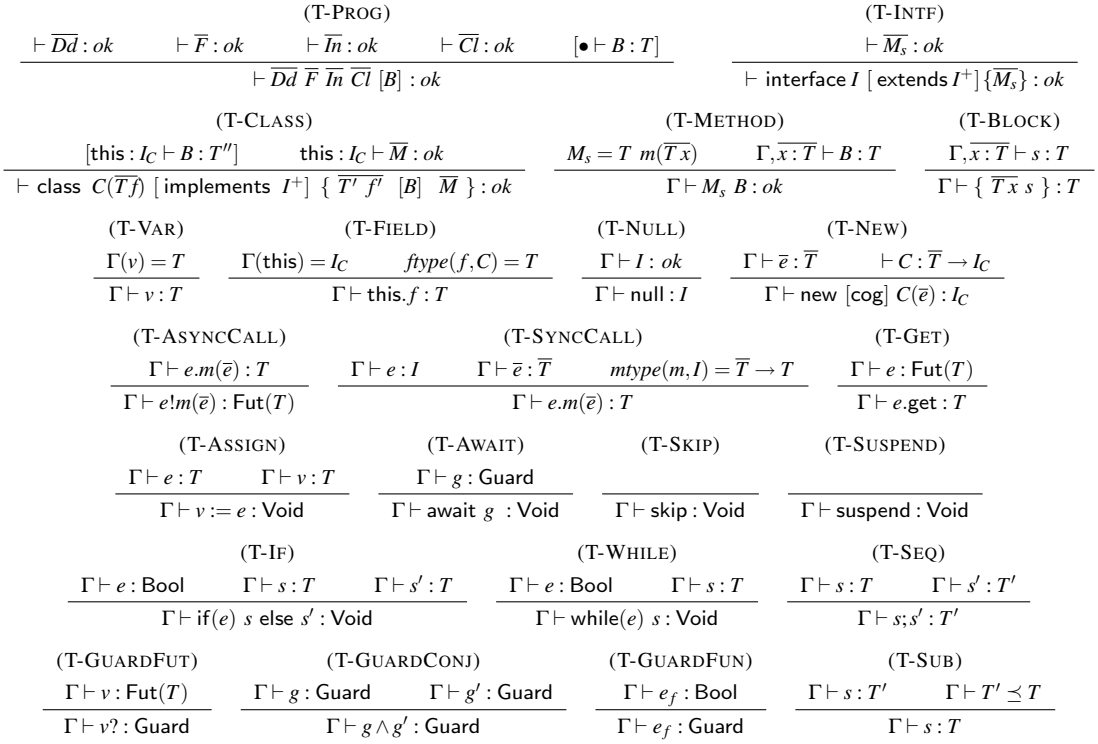

Figure 5.3: The type system

5.2 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 *associative* and *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:

$$\begin{array}{rcll} P & ::= & o[b, C, \sigma] & \text{object } o \\ & | & n\langle b, o, s, \sigma\rangle & \text{task } n \\ & | & b[l] & \text{lock } b \text{ for a concurrent object group} \\ & | & P \parallel P & \text{composition .} \end{array} \tag{5.2}$$

An object

$$o[b, C, \sigma] \tag{5.3}$$

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

$$n\langle b, o, \sigma, s\rangle \tag{5.4}$$

are the active parts in a configuration/program; i.e., they represent statements 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 *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).

Remark 5.2.1 (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 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* 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 (5.2), 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* $n\langle b, o, \sigma, s\rangle$ *is a future reference.* □

The last basic entity of (the changeable part of) the configuration represents a concurrent object group. This is done by a *lock* of the form

$$b[l] \tag{5.5}$$

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 *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 5.2.2 (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 new object, the concurrency model corresponds to that of Creol. When using only one global object group, i.e., using only* new*C 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 3.1.

$$s \quad ::= \quad \ldots \mid \mathsf{grab}\ (b) \mid \mathsf{release}(b) \mid \mathsf{let}\, y{:}T = s \,\mathsf{in}\, s$$

Figure 5.4: 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 4: 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 the randomness of a non-deterministic rewriting strategy (for purely functional expression 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 "expressions" 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* [36]. 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 $e_1.m(f(e_1))$ (where f is a function name) can be expanded to

$$\mathsf{let}\, x_1 = e_1 \,\mathsf{in}(\mathsf{let}\, x_2 = e_2 \,\mathsf{in}(\mathsf{let}\, x_3 = f(x_1) \,\mathsf{in}\, x_1.m(x_3))) \,. \tag{5.6}$$

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 $\mathsf{this}.f_1 + \mathsf{this}.f_2$ (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 place: 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 [7]; 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 5.5 (dealing with object-local statements and expressions) and Figure 5.6 (for statements and expressions dealing with cooperation, message passing, etc.). The first rules of Figures 5.5 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-COND$_1$ and R-COND$_2$). Similarly, whether the body of a while-loop is entered or skipped depends on the loop condition (see the R-WHILE-rules).

$$n\langle b,o,\sigma,\mathsf{let}\,z{:}T = v\,\mathsf{in}\,s\rangle \rightsquigarrow n\langle b,o,\sigma,s[v/z]\rangle \quad \text{R-RED}$$

$$n\langle b,o,\sigma,\mathsf{let}\,z_2{:}T_2 = (\mathsf{let}\,z_1{:}T_1 = s_1\,\mathsf{in}\,s)\,\mathsf{in}\,s'\rangle \rightsquigarrow n\langle b,o,\sigma,\mathsf{let}\,z_1{:}T_1 = s_1\,\mathsf{in}\,(\mathsf{let}\,z_2{:}T_2 = s\,\mathsf{in}\,s')\rangle \quad \text{R-SEQ}$$

$$n\langle b,o,\sigma,\mathsf{skip};s\rangle \rightsquigarrow n\langle b,o,\sigma,s\rangle \quad \text{R-SKIP}$$

$$n\langle b,o,\sigma,\mathsf{let}\,z{:}T = x\,\mathsf{in}\,s\rangle \rightsquigarrow n\langle b,o,\sigma,\mathsf{let}\,z{:}T = \sigma(x)\,\mathsf{in}\,s\rangle \quad \text{R-PURE}$$

$$n\langle b,o,\sigma,\mathsf{let}\,z{:}T = (\mathsf{if\ true\ then}\,s_1\,\mathsf{else}\,s_2)\,\mathsf{in}\,s\rangle \rightsquigarrow n\langle b,o,\sigma,\mathsf{let}\,z{:}T = s_1\,\mathsf{in}\,s\rangle \quad \text{R-COND}_1$$

$$n\langle b,o,\sigma,\mathsf{let}\,z{:}T = (\mathsf{if\ false\ then}\,s_1\,\mathsf{else}\,s_2)\,\mathsf{in}\,s\rangle \rightsquigarrow n\langle b,o,\sigma,\mathsf{let}\,z{:}T = s_2\,\mathsf{in}\,s\rangle \quad \text{R-COND}_2$$

$$n\langle b,o,\sigma,\mathsf{let}\,z{:}T = (\mathsf{if}\,v = v\,\mathsf{then}\,s_1\,\mathsf{else}\,s_2)\,\mathsf{in}\,s\rangle \rightsquigarrow n\langle b,o,\sigma,\mathsf{let}\,z{:}T = s_1\,\mathsf{in}\,s\rangle \quad \text{R-COND}_3$$

$$\frac{v_1 \neq v_2}{n\langle b,o,\sigma,\mathsf{let}\,z{:}T = (\mathsf{if}\,v_1 = v_2\ \mathsf{then}\,s_1\,\mathsf{else}\,s_2)\,\mathsf{in}\,s\rangle \rightsquigarrow n\langle b,o,\sigma,\mathsf{let}\,z{:}T = s_2\,\mathsf{in}\,s\rangle}\ \text{R-COND}_4$$

$$n\langle b,o,\sigma,\mathsf{let}\,z{:}T = (\mathsf{while\ true\ do}\,s_1)\,\mathsf{in}\,s_2\rangle \rightsquigarrow n\langle b,o,\sigma,\mathsf{let}\,z'{:}T = s_1\,\mathsf{in}(\mathsf{let}\,z{:}T = (\mathsf{while\ true\ do}\,s_1)\,\mathsf{in}\,s_2)\rangle \quad \text{R-WHILE}_1$$

$$n\langle b,o,\sigma,\mathsf{let}\,z{:}T = (\mathsf{while\ false\ do}\,s_1)\,\mathsf{in}\,s_2\rangle \rightsquigarrow n\langle b,o,\sigma,s_2\rangle \quad \text{R-WHILE}_2$$

$$n\langle b,o,\sigma,x := v;s\rangle \rightsquigarrow n\langle b,o,\sigma[x\mapsto v],s\rangle \quad \text{R-ASSIGN}$$

$$o[b,C,\sigma] \parallel n\langle b,o,\sigma',\mathsf{let}\,z : T = o.f\,\mathsf{in}\,s\rangle \rightsquigarrow o[b,C,\sigma] \parallel n\langle b,o,\sigma',\mathsf{let}\,z : T = \sigma(f)\,\mathsf{in}\,s\rangle \quad \text{R-FLOOKUP}$$

$$o[b,C,\sigma] \parallel n\langle b,o,\sigma',o.f := v;s\rangle \rightsquigarrow o[b,C,\sigma[f\mapsto v]] \parallel n\langle b,o,\sigma',s\rangle \quad \text{R-FUPDATE}$$

Figure 5.5: Reduction rules (1)

An assignment $x := v$ of a value v to a local variable x simply updates the local state σ 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 3.1), 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 5.6). This means, at runtime in the rules R-FLOOKUP 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-FLOOKUP. Field update works inversely, copying a value from the task-local state space to the heap, where $\sigma[f \mapsto v]$ denotes this update (see rule R-FUPDATE).

Now to the rules of Figure 5.6, dealing with message passing and task handling. The first two rules deal in this table 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 5.2.1). 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 this (which is a reserved local variable for each method) is replaced by the identity o' of the callee, and furthermore where the formal parameters x are replaced by the actual ones. On the caller side, the task "blocks" in that it immediately tries to use the fresh identity n' as handle to get the value back, using n'.get. In that way the "control" is passed through the new task n'. As far as the group affiliation is concerned: the new task clearly has to belong to the same group, as synchronous calls can never cross the border between object groups.

Asynchronous calls in rule R-ACALL are treated similarly, and 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 $l \in \{\bot, \top\}$. 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 5.4). See also Remark 5.2.4.

Remark 5.2.3 (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 5.2.1). 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 new$C$, 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 initializor block. In the premise, the instance state σ_{init} and the local state σ'_{init} have the respective variables/fields initialized appropriately by default values or by corresponding expressions. Note that the return value of the initializor

$$\frac{n' \textit{ fresh} \qquad \textit{body}(m,C) = s(\bar{x}) \qquad s_{task} = (\mathsf{let}\, z' {:} T = s[o'/\mathsf{this}][\bar{v}/\bar{x}]\,\mathsf{in}\, z')}{o'[b,C,\sigma] \parallel n\langle b,o,\sigma,\mathsf{let}\, z{:}T = o'.m(v)\,\mathsf{in}\, s_2\rangle \rightsquigarrow o'[b,C,\sigma] \parallel n\langle b,o,\sigma,\mathsf{let}\, z{:}T = n'.\mathtt{get}\,\mathsf{in}\, s_2\rangle \parallel n'\langle b,o',\sigma_{init},s_{task}\rangle}\ \text{R-SCALL}$$

$$\frac{n' \textit{ fresh} \qquad \textit{body}(m,C) = s(\bar{x}) \qquad s_{task} = (\mathsf{let}\, z{:}T = \textit{grabs}(b); s[o'/\mathsf{this}][\bar{v}/\bar{x}]\,\mathsf{in}\,\mathsf{release}(b); z)}{o'[b',C,\sigma] \parallel n\langle b,o,\sigma,\mathsf{let}\, z{:}T = o'!m(v)\,\mathsf{in}\, s_2\rangle \rightsquigarrow o'[b',C,\sigma] \parallel n\langle b,o,\sigma,\mathsf{let}\, z{:}T = n'\,\mathsf{in}\, s_2\rangle \parallel n'\langle b,o',\sigma,s_{task}\rangle}\ \text{R-ACALL}$$

$$\frac{o' \text{ and } n' \text{ fresh} \qquad B = \{\overline{Tf}\ s'\} \text{ is initializer block of } C \qquad s_{task} = (s'[o'/\mathsf{this}]; o')}{n\langle b,o,\sigma,\mathsf{let}\, z{:}T = \mathsf{new}\ C(\bar{v})\,\mathsf{in}\, s\rangle \rightsquigarrow n'\langle b,o',\sigma'_{init},s_{task}\rangle \parallel o'[b,C,\sigma_{init}[\overline{f'} \mapsto \bar{v}]] \parallel n\langle b,o,\sigma,\mathsf{let}\, z{:}T = n'.\mathtt{get}\,\mathsf{in}\, s\rangle}\ \text{R-NEWO}$$

$$\frac{b', o', \text{and } n' \text{ fresh} \qquad B = \{\overline{Tf}\ s'\} \text{ is initializer block of } C \qquad s_{task} = (s'[o'/\mathsf{this}]; \mathsf{release}(b'))}{n\langle b,o,\sigma,\mathsf{let}\, z{:}T = \mathsf{new}\ C(\bar{v})\,\mathsf{in}\, s\rangle \rightsquigarrow b'[\top] \parallel n'\langle b',o',\sigma'_{init},s_{task}\rangle \parallel o'[b',C,\sigma_{init}] \parallel n\langle b,o,\sigma,\mathsf{let}\, z{:}T = o'\,\mathsf{in}\, s\rangle}\ \text{R-NEWOG}$$

$$b[\bot] \parallel n\langle b,o,\sigma,\mathsf{grab}\ (b); s\rangle \rightarrow b[\top] \parallel n\langle b,o,\sigma,s\rangle \qquad \text{R-GRAB}$$

$$b[\top] \parallel n\langle b,o,\sigma,\mathsf{release}(b); s\rangle \rightarrow b[\bot] \parallel n\langle b,o,\sigma,s\rangle \qquad \text{R-RELEASE}$$

$$n\langle b,o,\sigma,\mathsf{suspend}; s\rangle \rightsquigarrow n\langle b,o,\sigma,\mathsf{release}(b); \mathsf{grab}\ (b); s\rangle \qquad \text{R-SUSPEND}$$

$$n_1\langle b',o',\sigma',\mathsf{let}\, x : T = n_1.\mathtt{get}\,\mathsf{in}\, t\rangle \parallel n_2\langle b,o,\sigma,v\rangle \rightsquigarrow n_1\langle b',o',\sigma',\mathsf{let}\, x : T = v\,\mathsf{in}\, t\rangle \parallel n_2\langle b,o,\sigma,v\rangle \qquad \text{R-GET}$$

$$\frac{\textit{futnames}(g) = n_1 \ldots n_k \text{ with } k \geq 1}{n_1\langle b_1,o_1,\sigma_1,v_1\rangle \parallel \ldots \parallel n_k\langle b_k,o_k,\sigma_k,v_k\rangle \parallel n\langle b,o,\sigma,\mathsf{await}\ (g); s\rangle \rightsquigarrow n\langle b,o,\sigma,\mathsf{release}(b); \mathsf{await}\ (g)[\overline{\mathsf{true}}/\overline{n?}]; s\rangle}\ \text{R-AWAIT}_1^?$$

$$\frac{n' \in \textit{futnames}(g) \qquad s' \neq v}{n'\langle b',o',\sigma',s'\rangle \parallel n\langle b,o,\sigma,\mathsf{await}\ (g); s\rangle \rightsquigarrow n\langle b,o,\sigma,\mathsf{release}(b)\ \mathsf{await}\ (g); s\rangle}\ \text{R-AWAIT}_2^?$$

$$\frac{\textit{futnames}(g) = \emptyset \qquad [\![g]\!]_\sigma = \textit{true}}{n\langle b,o,\sigma,\mathsf{await}\ (g); s\rangle \rightsquigarrow n\langle b,o,\sigma,s\rangle}\ \text{R-AWAIT}_1 \qquad \frac{\textit{futnames}(g) = \emptyset \qquad [\![g]\!]_\sigma = \textit{false}}{n\langle b,o,\sigma,\mathsf{await}\ (g); s\rangle \rightsquigarrow n\langle b,o,\sigma,\mathsf{release}(b); \mathsf{await}\ (g); s\rangle}\ \text{R-AWAIT}_2$$

Figure 5.6: Reduction rules (2)

block is the new object identity o (at the end of s_{init}). This value is handed back (after termination of the initializor) to the creating thread via $n'.\mathtt{get}$.

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: in order 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 initializor code of the class does *not* need to apply for the lock as other methods, it holds it initially. Of course, 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 5.2.4). They work very simple. Grabbing a lock requires that the lock is free (i.e., it is in state $\bot$), and it changes the state to $\top$. 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 $\bot$.

Remark 5.2.4 (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 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 $n_2\langle\_,\_,\_,v\rangle$ (as required by R-GET) or the statement is empty, when no value is returned and the method is of type Void, 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 5.2.3); 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 objects/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 become 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 3.1). Apart from the (unproblematic) fact that guards can contain functional Boolean expressions, they are constructed with $\wedge$. 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 $\wedge$ —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 5.7(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 5.7(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 *monotonously,* this cannot happen and the order described is safe. Furthermore it avoids looping back and re-applying for lock and can this be more efficiently implemented than the first solution. Guards are monotone, if the do not refer to instance fields. Note

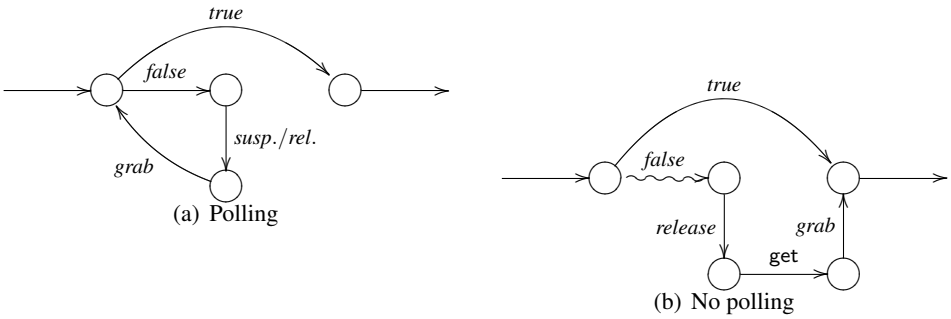

Figure 5.7: Await statement

in particular, the the polling expression $n?$ trying to dereference a future are *monotonic*.

These considerations are reflected in the rules in the following way. In the R-AWAIT-rules in Figure 5.6 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, $\mathit{futnames}(g)$ extracts all the names for futures mentioned as $n?$ expressions in g. the second stage (after all the $n?$-guards have been replaced by `true`, the guard is evaluated again; the $[\![g]\!]_\sigma$ gives back the Boolean value of the guard, relative to the instance state σ.

Chapter 6

Example: A Peer-To-Peer Node in ABS

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.

6.1 The Functional Specification

Data type definitions.

```
data Void // built−in
data String // built−in
data Int // built−in
data SetOfString {EmptyStringSet, InsertString(String, SetOfString)}
data ListOfPacket {NilPacket, ConsPacket(Packet, ListOfPacket)}
data ListOfServer {NilServer, ConsServer(Server, ListOfServer)}
data PairOfServerNFilenames { PairOfServerNFilenames(Server, Filenames) }
data ListOfPairOfServerNFilenames {
  NilPairOfServerNFilenames,
  ConsPairOfServerNFilenames(PairOfServerNFilenames, ListOfPairOfServerNFilenames)
}
data PairOfFilenameNFile { PairOfFilenameNFile(Filename, File) }
data MapOfFilenameToFile { EmptyMap, InsertAssoc(PairOfFilenameNFile, MapOfFilenameToFile) }
```

(Note that these definitions can be generalized using parameterized data structures, which are not contained in the core ABS language but planned as a language extension.)

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).

```
type Filename = String
type Filenames = SetOfString
type Server = String
type Packet = String
type File = ListOfPacket
type Catalog = ListOfPairOfServerNFilenames
```

Function definitions. We define the following functions on lists. Below, we just give them for File (i.e., ListOfPacket). The built-in data type Int provides infix functions $+$, $-$, and $>$, which are used in the sequel.

```
def Packet head(File file) = case file { ConsPacket(p,l) => p } // the head of a non−empty file
def File tail(File file) = case file { ConsPacket(p,l) => l } // the tail of a non−empty file
def Bool isEmpty(File file) = file == NilPacket // test for empty file

def Int length(File file) = // the length of a list
  case file {
    NilPacket => 0
    ConsPacket(p, list) => 1 + length(list)
  }

def Packet nth(File file, Int n) =
  case n {
    0 => head(file)
    _ => nth(tail(file), n−1)
  }

def File concatenate(File file1, File file2) =
  case file1 {
     NilPacket => file2
     ConsPacket(head, tail) => ConsPacket(head, concatenate(tail,file2))
  }

def File appendright(File file, Packet p) = concatenate(file, ConsPacket(p,NilPacket))
```

We define the following functions on sets. Below, we give the definitions for SetOfString.

```
def Bool element(String string, SetOfString set) =
  case set {
    EmptyStringSet => False
    InsertString(string2, set2) => string == string2 or element(string, set2)
  }
```

We define the following functions on maps. Below, we just give the definitions for MapOfFilenameToFile.

```
def File getFromMap(Filename fId, MapOfFilenameToFile map) = // retrieve a file from the map
  case map {
    InsertAssoc(PairOfFilenameToFile(fId',file), tail) =>
      if (fId==fId') {file} else {getFromMap(fId,tail)}
  }

def MapOfFilenameToFile insert(PairOfFilenameNFile assoc, MapOfFilenameToFile map) = InsertAssoc(assoc, map)

def Filenames keys(MapOfFilenameToFile map) =
  case map {
    EmptyMap => EmptyStringSet
    InsertAssoc(PairOfFilenameNFile(filename, file), tail) => InsertString(filename, keys(tail))
  }
```

The functions fst and snd return the first and second element of a pair, respectively. Thus, $p = (\mathsf{fst}(p), \mathsf{snd}(p))$ if p is a pair. We only give these functions for pairs of strings below:

```
def String fst(Pair p) = case p { (l,r) => l }
def String snd(Pair p) = case p { (l,r) => r }
```

6.2 The Imperative Model

6.2.1 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 6.1 above). The database functionality is given by an interface DB, defined as follows:

```
interface DB {
  File getFile(Filename fId)
  Int getLength(Filename fId)
  Void 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 defined by the type MapOfFilenameToFile in Section 6.1 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, 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):

```
class DataBase(MapOfFilenameToFile db) implements DB {
  File getFile(Filename fId) { return getFromMap(db, fId); }
  Int getLength(Filename fId){ return length(getFromMap(db, fId)); }
  Void storeFile(Filename fId, File file) { db = insert(PairOfFilenameNFile(fId,file), db); }
  Filenames listFiles() { return keys(db); }
}
```

6.2.2 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:

```
interface Client {
  ListOfPairOfServerNFilenames availFiles(ListOfServer sList)
  Void reqFile(Server sId, Filename fId)
}

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

interface Peer extends Client, Server {
  ListOfServer 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

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 6.1.

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.

```
class Node(DB db, Peer admin, Filename file) implements Peer {
  Catalog catalog ; ListOfServer myNeighbors ;

  ListOfServer getNeighbors() { return myNeighbors ; }

  Server findServer(Filename fId, Catalog catalog) {
    if (isEmpty(catalog)) {return null ;
    } else if (element(fId,snd(head(catalog)))) { return fst(head(catalog)) ;
    } else { return findServer(fId, tail(catalog)) ;
    }
  }

  Void run() {
    Fut(Catalog) c ; Fut(ListOfServer) f; ListOfServer newNeighbors; Server server ;
    neighbors = ConsServer(admin, NilServer);
    f = admin!getNeighbors(); // Asynchronous call to admin
    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; // Build the catalog
    server = findServer(file,catalog); // Find the server for the requested file
    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(File) = ff ;
    ff = db!getFile(fId) ; await ff? ; f = ff.get;
    return nth(f, pNbr);
  }

  Catalog availFiles (ListOfServer sList) {
    Catalog cat ; Fut(Filenames) fNames ; Fut(Catalog) catList ;
    if (sList = NilServer) { cat = NilPairOfServerNFilenames ;
    } 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.get, (head(sList), fNames.get));
    }
    return cat;
  }

  Void 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 = ConsPacket(pack,file) ;
      lth = lth − 1 ;
    }
    db!storeFile(fId, file) ;
  }
}
```

Figure 6.1: The Node class in core ABS

Chapter 7

Tool Support and Integration

This chapter presents the design of the tools for the core ABS language, a compiler and a virtual machine, and it also gives an overview of the implementation of the virtual machine for testing and executing models in the language.

7.1 The HATS Framework Vision

This report and accompanying implementation is the first stepping stone in 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 families rigorously using the ABS language.

We therefore take a look at the overall picture into which this deliverable fits, emphasizing the tool perspective. Figure 7.1 shows our vision of the HATS framework. In the figure, the *blue parts* are the tools designed in this deliverable and the *orange parts* are examples of tools to be designed later. The *green parts* are tool inputs or outputs. The *gray parts* are tools, which already exist.

The abstract syntax tree (AST) is the cornerstone of tool integration: it is the representation for ABS models, and tools will typically reason about one or more such models or produce them. In addition, we will define a ABS model file format for representing ABS models independently of the concrete syntax of the ABS language. Currently planned is to base this format on XML, but other technologies are also possible. In contrast to the ABS source language, the ABS model files will not be edited by programmers, but will be used as an interchange format between tools. The model format of ABS will be fully type annotated and will not require any name analysis. Creating the AST from the model format will thus be straightforward. All tools will be able to read and write ABS models in the model format using a common ABS model reader and writer. Some tools will also be able to read the ABS source format. These tools will then use a common compiler frontend to read ABS models. All tools will work on a common representation of the AST, which 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.

Examples given in the figure are the ABS model miner (Deliverable D3.2), a verifier for behavioral properties (Deliverable D2.5), and a Java code generator. 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. Fig. 7.1 is not exhaustive; for example, we have not integrated tools for resource analysis (Deliverable D4.2) or debugging (Deliverable D2.3). A detailed presentation of the final HATS tool chain will be part of Deliverable D1.5.

7.2 The ABS Compiler

The role of the ABS compiler front-end is to translate textual ABS models into an internal representation and check the models for syntax errors and semantic errors. The role of the compiler back-end is to generate code for the models

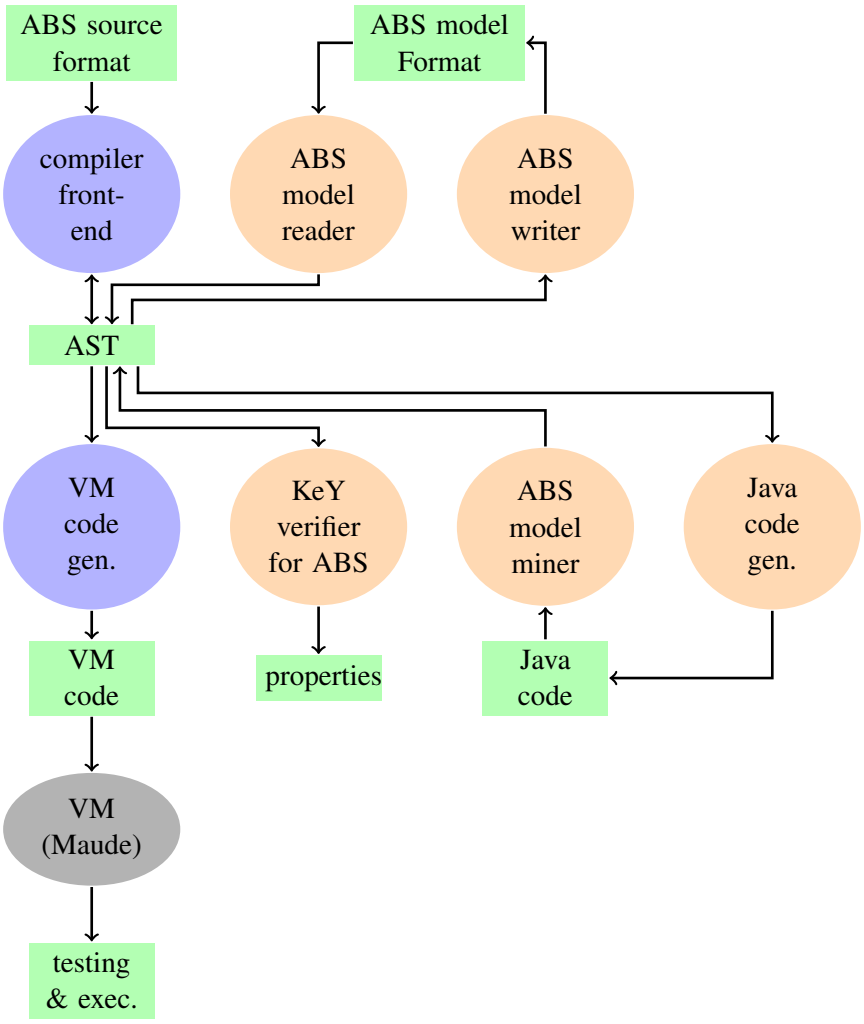

Figure 7.1: Tools in the HATS framework

targeting some suitable execution or simulation environment. Figure 7.2 shows a generic compiler architecture for some programming language and target platform code. Note that the compiler back-end has been simplified into just a code generator since the optimization is outside the scope of this deliverable.

The core ABS compiler is characterized by this diagram after replacing *program* with *ABS model* and replacing *code* with *ABS virtual machine code*.

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 will be 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 the AST to be augmented during semantic analysis, in order to ease both the analysis itself and the code generation. The reason for choosing the Beaver parser generator is its straightforward integration with JastAdd. (We may later decide to replace JFlex and Beaver with the industry standard parser generator ANTLR[3]).

[1] JFLex: `http://www.jflex.de/`, Beaver: `http://beaver.sourceforge.net/`
[2] JastAdd: `http://www.jastadd.org/`
[3] ANTLR: `http://www.antlr.org/`

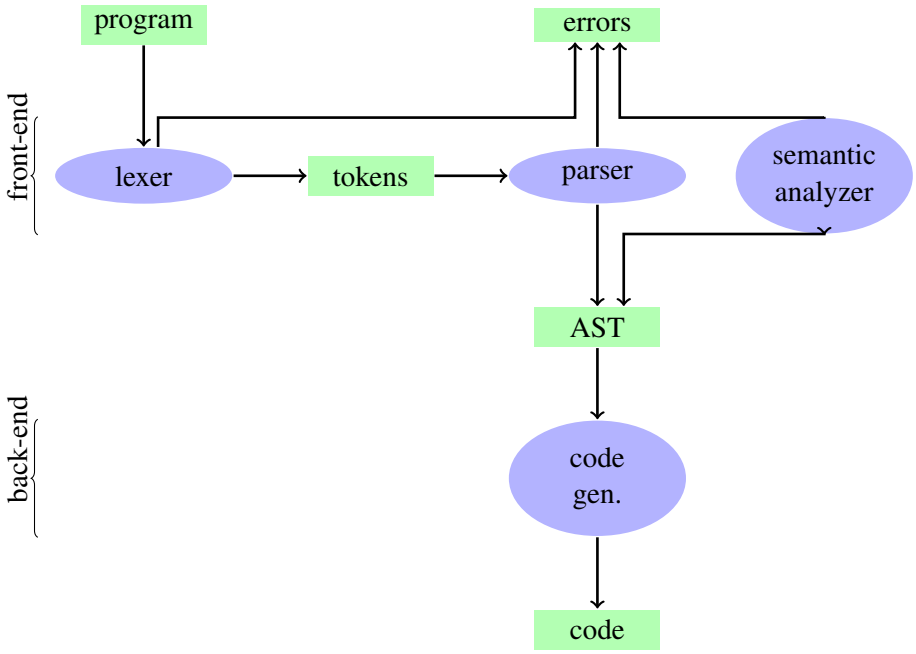

Figure 7.2: ABS Compiler

7.3 Virtual Machine for Testing and Execution

In order to have a concrete simulator for ABS models, the operational semantics described in Section 5.2 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, will be converted into Maude terms describing the model in terms and expressions that can be used for simulation by the interpreter. 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 (Deliverable D2.5).

7.4 Editor Support

Implementation efforts for tools to support the editing of ABS models are underway, specifically support in the Eclipse IDE and Emacs editor. The usual features expected by users of these environments will be available, including compilation and execution support, semantic highlighting and cross-referencing of ABS code. Figure 7.3 gives a snapshot of the current prototype of the Eclipse IDE editor with the ABS model of the peer-to-peer system given in Chapter 6.

[4]Maude: `http://maude.cs.uiuc.edu/`

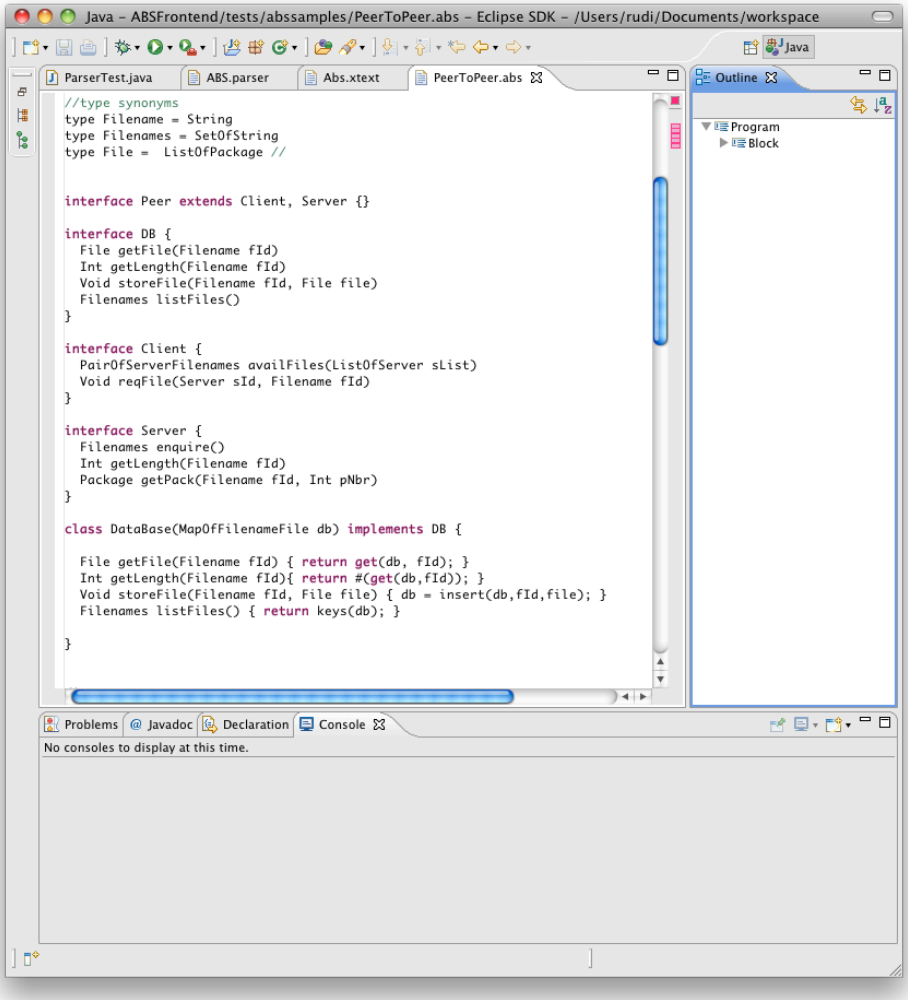

Figure 7.3: Screenshot of the Eclipse ABS editor

Chapter 8

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 (`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 [45]. Erlang [11] 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 [59, 49]. 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 [50], 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 [68, 81, 73]. 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 [41] 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 [23], a programming language based on similar primitives which targets distributed concurrent objects. The concurrent object calculus of [32] 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*[52] and may be compared to monitors [48] 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 [33, 10], 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 [29]. 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 [34].

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 [76, 37]. In this vein of work, TCOZ [56] 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 [57, 27] 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 [65]: 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 [82], 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 [30] 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 [3] or Abstract State Machines (ASMs) [19] 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 [67, 21]. A common approach has been to keep activity (threads) and objects distinct, as done in Hybrid [64] and Java [42]. 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 [20], 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 [10] and POOL-T [8].

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 [28]. 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 [6, 5, 47, 54] 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 [85], Argus [55], ConcurrentSmalltalk [84], Eiffel// [26], CJava [28], and in the Join calculus [39] based languages Polyphonic C$^\sharp$ [16] and Join Java [51]. 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 [86], POOL2 [9], Eiffel// [24], and Creol [52]. Hybrid [63] introduced the concept of *domains*, which group active objects. However, communication in Hybrid is via synchronous method calls. ASP [25] 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 an 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 [12].

The E programming language [58] 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 [79]. 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 [52]. In addition, combining cooperative multitasking with futures, was also pioneered by Creol [31]. Creol, however, has single objects as the unit of concurrency, which does not allow for multiple service objects. The Creol model corresponds to concurrent object groups with only single objects in ABS. In Symbian OS [61] 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 [77] allows to schedule tasks cooperatively if the assigned scheduler is configured to be single-threaded. Rodriguez and Rossetto [71] combine cooperative multitasking with asynchronous RMI in the Lua language.

Thorn [17] 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.

Futures. The notion of futures used in ABS stems from Creol [31]. 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 [13], and later rediscovered by Liskov and Shrira as Promises [55] and by Halstead in the context of MultiLisp [46]. Futures appear in languages like Alice [72], Oz-Mozart [80], Concurrent ML [69], C++ [53] and Java [83], 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 [62]. 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 $\lambda(\mathtt{fut})$ [62] 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 [38] 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 [25] 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 [4] 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 [31]: concurrent objects with futures basically allow local reasoning similar to reasoning for sequential programs. Proof systems for actor languages exist [35], but these require explicit structures in the proof rules for reasoning about message queues, which our proof theory avoids. Previous work by UIO [34] 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 [40, 48]. 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 [15] has been presented in [7].

Chapter 9

Summary

In this report, we have proposed a Core ABS language. The 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 abstract data types (ADTs) to abstractly model the internal data structures inside concurrent objects and a side-effect free functional expression language, including case-constructs, to manipulate ADTs.

We have provided a formalization of the Core ABS language in terms of an EBNF for the abstract syntax, a basic type system using ADTs and interface types, and an operational semantics. The language has been demonstrated by means of an example 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.

We have investigated the design of the HATS framework and developed a prototype parser for the surface syntax of the Core ABS language and a prototype interpreter which can perform concrete simulations of Core ABS models. The completion of the tool chain from the surface syntax of the Core ABS to the syntax of the interpreter remains to be done; in particular, the type checker and the translator from the abstract syntax tree into the syntax of the interpreter have not yet been completed. Front-ends to the tool chain are currently being developed for Eclipse and Emacs.

In the proposed design of the Core ABS language, we have tried to keep the language fairly simple, yet incorporate flexible mechanisms for concurrency control and composition, as promised in the DoW. An obvious extension to the core ABS that we intend to introduce, are polymorphic data types. The need for this feature was clearly demonstrated by the example in Section 6. Polymorphic data types will allow much more succinct presentations of the functional part of the models. We believe further extensions to the ABS language should be driven by the needs of software product families, in order to keep the development of the modeling language reasonably focused. In addition, the inclusion of language constructs in the final ABS language should also depend on their properties with respect to the analysis of ABS models.

In this report, we have not attempted to introduce possible structuring mechanisms for software product families. In particular, we have not included class inheritance in the Core ABS language. This is not because it causes any particular difficulties at the level of the type system or operational semantics, which are the focus of this report, but rather to avoid introducing class inheritance for the verification system unless we find that this is a good way to structure code for software product families.

Bibliography

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, New York, NY, 1996.

[2] Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. An assertion-based proof system for multithreaded Java. *Theoretical Computer Science*, 331, 2005.

[3] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[4] Gul Agha and Carl Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In *Research Directions in Object-Oriented Programming*, pages 49–74. MIT Press, 1987.

[5] Gul A. Agha. Abstracting interaction patterns: A programming paradigm for open distributed systems. In Elie Najm and Jean-Bernard Stefani, editors, *Proc. 1st IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'96)*, pages 135–153, Paris, 1996. Chapman & Hall.

[6] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, January 1997.

[7] Wolfgang Ahrendt and Maximilian Dylla. A verification system for distributed objects with asynchronous method calls. In Karin Breitman and Ana Cavalcanti, editors, *Formal Methods and Software Engineering, International Conference on Formal Engineering Methods (ICFEM'09)*, volume 5885 of *Lecture Notes in Computer Science*, pages 387–406. Springer-Verlag, 2009.

[8] Pierre America. POOL-T: A parallel object-oriented language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. The MIT Press, Cambridge, Mass., 1987.

[9] Pierre America. Issues in the design of a parallel object-oriented language. *Form. Asp. Comput.*, 1(4):366–411, 1989.

[10] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[11] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007.

[12] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. Programming, deploying, composing, for the Grid. In José C. Cunha and Omer F. Rana, editors, *Grid Computing: Software Environments and Tools*. Springer, January 2006.

[13] Henry G. Baker and Carl E. Hewitt. The incremental garbage collection of processes. In *Proceeding of the Symposium on Artificial Intelligence Programming Languages*, number 12 in SIGPLAN Notices, page 11, August 1977.

[14] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69, New York, NY, 2005. Springer-Verlag.

[15] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2007.

[16] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C♯. *ACM Transactions on Programming Languages and Systems*, 26(5):769–804, September 2004.

[17] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn: robust, concurrent, extensible scripting on the JVM. *SIGPLAN Not.*, 44(10):117–136, 2009.

[18] Egon Börger. Abstract state machines and high-level system design and analysis. *Theoretical Computer Science*, 336(2–3):205–207, May 2005.

[19] Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, Berlin, 2003.

[20] Per Brinch Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34(4):38–45, April 1999.

[21] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and distribution in object-oriented programming. *ACM Comput. Surv.*, 30(3):291–329, 1998.

[22] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, 2005.

[23] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.

[24] Denis Caromel. Towards a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[25] Denis Caromel, Ludovic Henrio, and Bernard Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL'04)*, pages 123–134. ACM Press, 2004.

[26] Denis Caromel and Yves Roudier. Reactive programming in Eiffel//. In J.-P. Briot, J. M. Geib, and A. Yonezawa, editors, *Proceedings of the Conference on Object-Based Parallel and Distributed Computation*, volume 1107 of *Lecture Notes in Computer Science*, pages 125–147. Springer-Verlag, Berlin, 1996.

[27] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, August 2002.

[28] Gianpaolo Cugola and Carlo Ghezzi. CJava: Introducing concurrent objects in Java. In M. E. Orlowska and R. Zicari, editors, *4th International Conference on Object Oriented Information Systems (OOIS'97)*, pages 504–514, London, 1997. Springer-Verlag.

[29] Ole-Johan Dahl. Monitors revisited. In A. W. Roscoe, editor, *A Classical Mind, Essays in Honour of C.A.R. Hoare*, pages 93–103. Prentice Hall, 1994.

[30] Werner Damm, Bernhard Josko, Amir Pnueli, and Angelika Votintseva. Understanding UML: A formal semantics of concurrency and communication in Real-Time UML. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *First International Symposium on Formal Methods for Components and Objects (FMCO 2002), Revised Lectures*, volume 2852 of *Lecture Notes in Computer Science*, pages 71–98. Springer-Verlag, 2003.

[31] Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In Rocco de Nicola, editor, *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer-Verlag, March 2007.

[32] Paolo Di Blasio and Kathleen Fischer. A calculus for concurrent objects. In Ugo Montanari and Vladimiro Sassone, editors, *7th International Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 655–670. Springer-Verlag, August 1996.

[33] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[34] Johan Dovland, Einar Broch Johnsen, and Olaf Owe. Observable behavior of dynamic systems: Component reasoning for concurrent objects. In Dina Goldin and Farhad Arbab, editors, *Proc. Workshop on the Foundations of Interactive Computation (FInCo'07)*, volume 203 of *Electronic Notes in Theoretical Computer Science*, pages 19–34. Elsevier, May 2008.

[35] Carlos H. C. Duarte. Proof-theoretic foundations for the design of actor systems. *Mathematical Structures in Computer Science*, 9(3):227–252, 1999.

[36] Martin Felleisen and Daniel P. Friedmann. The revised report in the syntactic theories of sequential control and state. *Theoretical Computer Science*, 2(4):235–271, 1992. An earlier version as Technical Report 100, Rice University June 1989.

[37] Clemens Fischer. CSP-OZ: a combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Proc. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 423–438. Chapman and Hall, London, 1997.

[38] Cormac Flanagan and Matthias Felleisen. The semantics of future and an application. *J. Funct. Program.*, 9(1):1–31, 1999.

[39] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 372–385, 1996.

[40] Rob Gerth and Willem P. de Roever. A proof system for concurrent ADA programs. *Sci. Comput. Program.*, 4(2):159–204, 1984.

[41] Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. In Uwe Nestmann and Benjamin C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages (Nice, France, September 12, 1998)*, volume 16(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998.

[42] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *The Java language specification*. Java series. Addison-Wesley, Reading, Mass., 2nd edition, 2000.

[43] David Gries and Fred B. Schneider. Avoiding the undefined by underspecification. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 366–373. Springer-Verlag, New York, NY, 1995.

[44] Reiner Hähnle. Many-valued logic, partiality, and abstraction in formal specification languages. *Logic Journal of the IPGL*, 13(4):415–433, July 2005.

[45] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009.

[46] Robert H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[47] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI*, pages 235–245. William Kaufmann, August 1973.

[48] C. A. R. Hoare. Monitors: an operating systems structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[49] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ., 1985.

[50] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In Pierre America, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, 1991.

[51] G. Stewart Itzstein and Mark Jasiunas. On implementing high level concurrency in Java. In A. Omondi and S. Sedukhin, editors, *Proc. 8th Asia-Pacific Computer Systems Architecture Conference (ACSAC 2003)*, volume 2823 of *Lecture Notes in Computer Science*, pages 151–165. Springer-Verlag, 2003.

[52] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software and System Modeling*, 6(1):35–58, March 2007.

[53] R. Greg Lavender and Douglas C. Schmidt. Active object: an object behavioral pattern for concurrent programming. *Proc. Pattern Languages of Programs*, 1995.

[54] Henry Lieberman. Concurrent object-oriented programming in Act 1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 9–36. MIT Press, 1987.

[55] Barbara H. Liskov and Liuba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In David S. Wise, editor, *Proceedings of the SIGPLAN Conference on Programming Lanugage Design and Implementation (PLDI'88)*, pages 260–267, Atlanta, GE, USA, June 1988. ACM Press.

[56] Brendan P. Mahony and Jin Song Dong. Sensors and actuators in TCOZ. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *World Congress on Formal Methods (FM'99), Proceedings, Volume II*, volume 1709 of *Lecture Notes in Computer Science*, pages 1166–1185. Springer-Verlag, 1999.

[57] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[58] Mark S. Miller, Eric Dean Tribble, and Jonathan S. Shapiro. Concurrency among strangers. In Rocco De Nicola and Davide Sangiorgi, editors, *TGC*, volume 3705 of *Lecture Notes in Computer Science*, pages 195–229. Springer, 2005.

[59] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, May 1999.

[60] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML - Revised*. The MIT Press, 1997.

[61] Ben Morris. Cactive and friends. *Symbian Developer Network*, June 2008.

[62] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364:338–356, 2006.

[63] Oscar Nierstrasz. Active objects in Hybrid. In *OOPSLA*, pages 243–253. ACM Press, 1987.

[64] Oscar Nierstrasz. A tour of Hybrid – A language for programming with active objects. In D. Mandrioli and B. Meyer, editors, *Advances in Object-Oriented Software Engineering*, pages 167–182. Prentice Hall, 1992.

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

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

[67] Michael Philippsen. A survey on concurrent object-oriented languages. *Concurrency: Practice and Experience*, 12(10):917–980, August 2000.

[68] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. The MIT Press, 1998.

[69] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[70] Requirement elicitation, August 2009. Deliverable 5.1 of project FP7-231620 (HATS), available at `http://www.cse.chalmers.se/research/hats/sites/default/files/Deliverable5.1.pdf`.

[71] Noemi Rodriguez and Silvana Rossetto. Integrating remote invocations with asynchronism and cooperative multitasking. *Parallel Processing Letters*, 18(1):71–85, 2008.

[72] Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. *Alice Through the Looking Glass*, volume 5 of *Trends in Functional Programming*, pages 79–96. Intellect Books, Bristol, UK, ISBN 1-84150144-1, Munich, Germany, February 2006.

[73] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[74] Jan Schäfer and Arnd Poetzsch-Heffter. CoBoxes: Unifying Active Objects and Structured Heaps. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS 2008)*, pages 201–219, 2008.

[75] Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *European Conference on Object-Oriented Programming (ECOOP 2010)*, Lecture Notes in Computer Science. Springer-Verlag, June 2010. To appear. Preprint available at `softech.cs.uni-kl.de/pub?id=137`.

[76] Graeme Smith and John Derrick. Specification, refinement and verification of concurrent systems - an integration of Object-Z and CSP. *Formal Methods in System Design*, 18(3):249–284, 2001.

[77] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In Jan Vitek, editor, *ECOOP*, volume 5142 of *Lecture Notes in Computer Science*, pages 104–128. Springer, July 2008.

[78] Alexander J. Summers, Sophia Drossopoulou, and Peter Müller. Universe-type-based verification techniques for mutable static fields and methods. *Journal of Object Technology*, 8(4):85–125, 2009.

[79] Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In *SCCC*, pages 3–12. IEEE Computer Society, 2007.

[80] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, March 2004.

[81] Vasco Thudichum Vasconcelos. Typed concurrent objects. In Mario Tokoro and Remo Pareschi, editors, *Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP'94)*, volume 821 of *Lecture Notes in Computer Science*, pages 100–117. Springer-Verlag, 1994.

[82] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Object Technology Series. Addison-Wesley, 1999.

[83] Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for Java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA'05)*, pages 439–453, New York, NY, USA, 2005. ACM Press.

[84] Yasuhiko Yokote and Mario Tokoro. Concurrent programming in ConcurrentSmalltalk. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 129–158. The MIT Press, Cambridge, Mass., 1987.

[85] Akinori Yonezawa. *ABCL: An Object-Oriented Concurrent System*. Series in Computer Systems. The MIT Press, 1990.

[86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming ABCL/1. In Norman Meyrowitz, editor, *OOPSLA*, pages 258–268. ACM Press, 1986.

Glossary

Terms and Abbreviations

ABS Abstract Behavioral Specification (Language)

ASM Abstract State Machine

AST Abstract Syntax Tree

COG Concurrent Object Group

CSP Communicating Sequential Processes

CCS Calculus of Communicating Systems

Future A place holder for the result of an asynchronous method call.

IDE Integrated Development Environment

OOL Object-Oriented (Programming) Language

RMI Remote Method Invocation

UML Unified Modeling Language

Meta-Variables

b	branches
B	blocks
C	class names
Cl	class definition
Co	constructor terms
D	data type names
Dd	data type declaration
e	expressions
e_e	expressions with side effects
e_f	functional expressions
e_p	pure expressions
f	field names
fn	function names
F	function declarations
g	guards
Γ	type contexts
I	interface names

In	interface declarations
K	kinds
m	method names
M	method definition
M_s	method signatures
p	patterns
P	programs
s	statements
t	terms
T	types
U	types
v	state variables / functional values
x	local variables
z	logical variables