CHALMERS

A Verification System for the Distributed Object-Oriented Language Creol

Master of Science Thesis

MAXIMILIAN DYLLA

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, June 2009

A Verification System for the Distributed Object-Oriented Language Creol

MAXIMILIAN G. DYLLA

Examiner: Kristian Lindgren
Supervisor: Wolfgang Ahrendt

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

The travels within this theses were supported by the EU COST Action IC0701: ”Formal Verification of Object-Oriented Software” and the EU-project FP7-ICT-2007-3 HATS: ”Highly Adaptable and Trustworthy Software using Formal Methods”.

Department of Computer Science and Engineering
Göteborg, Sweden June 2009

Abstract

Open distributed systems are composed of geographically distributed components which may be changed during run-time. The importance of such systems is growing as they are often part of safety-critical infrastructure. Creol is an experimental object-oriented modeling language for such systems. Its objects execute concurrently and contain their own virtual processor with cooperative scheduling. Inter-object communication takes place asynchronously via interfaces.

This work presents a first-order dynamic logic and calculus for formal verification of Creol together with a prototypical implementation in the interactive theorem prover KeY. The proof system is compositional in the sense that all methods of all classes are verifiable independently. Object internal concurrency is addressed by class invariants serving as a contract between all threads of an object. For external communication, a history in form of a ghost variable local to the object provides independence of modular proofs. Later on, composition of local histories leads to system wide correctness.

Keywords: program verification, dynamic logic, concurrency, communication history, distributed systems

Preface

This project has been carried out as a 60 credit-points Master of Science Thesis within the Complex Adaptive Systems Master-program in the Software Engineering using Formal Methods Group of the Department Computer Science and Engineering at Chalmers University of Technology. My task was to develop a verification system for Creol.

I would like to thank my supervisor Wolfgang Ahrendt for investing a lot of time in discussing my thesis with me even when I showed up several times per week. Despite his obligations to his family, he spent some days with me at the University of Oslo to proceed with my thesis.

I owe Richard Bubel much as he seems to know everything regarding the KeY system no matter whether it is a question about logic or about the source code.

During his time at University of Oslo, Marcel Kyas helped me in understanding many details of the Creol language. It was a pleasure for me to work together with Martin Steffen, Olaf Owe, and Einar Broch Johnsen who were always open to discussions and even left me a copy of the Festschrift in Memory of Ole-Johan Dahl.

Also I am very grateful to Wolfgang Ahrendt, Justin Schneiderman and Max Jair Ortiz Catalan for proofreading the final version of this report.

I want to thank my friends Max Jair Ortiz Catalan, Clemens Buss, and Justin Schneiderman for helping me to sometimes stay away from my thesis. Finally I thank my family and especially my girlfriend Anna Drescher for encouraging me throughout the completion of my thesis.

Table of Contents

List of Figures

Listings

Chapter 1

Introduction

This thesis builds the link between two long term research projects namely KeY[1] and Creol[2].

KeY [ABB+05, BHS07] is a research project started in 1998 at University of Karlsruhe, Germany. Nowadays, it is a joint project of the University of Karlsruhe, University of Koblenz-Landau, and Chalmers University of Technology, Gothenburg. Its purpose is the development of a formal software development tool that integrates design, implementation, formal specification, and formal verification of object-oriented software as seamlessly as possible. The core of the system is a first-order dynamic logic theorem prover supplied with a user friendly graphical interface. KeY was originally developed for verification of Java[3] and recently adapted for the verification of C and other purposes.

Creol (Concurrent reflective object-oriented language) is an ongoing research project at the Precise Modeling and Analysis group at University of Oslo where object-orientation was originally invented. Its goal is the development of a formal framework and tool for reasoning about dynamic and reflective modifications in object-oriented open distributed systems. The main part of the work is the modeling language Creol which is strongly-typed and object-oriented. It supports classes, interfaces, multiple inheritance, polymorphism and two different levels of concurrency. Between objects, executing in parallel with their own virtual processors, asynchronous method calls via the interfaces are the only form of communication. Inside an object, cooperation between threads is yielded by explicit processor release-points. The operational semantics of Creol are implemented in the rewriting logic of Maude [CDE+08] which provides an executable prototype environment to run Creol.

[1] http://www.key-project.org
[2] http://heim.ifi.uio.no/creol/
[3] http://www.java.com

1.1 Problem statement

The objective of this thesis is to establish the theoretical and technological foundations for an efficient and user-friendly verification environment for distributed systems modelled with Creol. Experience and knowledge from both Creol and the KeY system must therefore be incorporated. The major milestones completed in this work are:

1. The design of a program logic for (a significant subset of) Creol
2. The design of a proof calculus for this logic
3. The implementation of the calculus, in terms of executable rules
4. The implementation of a rudimentary proof strategy

The major challenge of the set goals is mastering the uncertainty inherent in compositional reasoning about distributed systems.

1.2 Related work

In the 1940s Goldstine, von Neumann [GvN47], and Turing [Tur49] first attempted to verify computer programs. About 20 years later Floyd [Flo67] formalized this principle for flowchart programs and Hoare [Hoa69] for imperative programs.

Nowadays, many program-verification systems are structured in a modular way consisting, at minimum, of a verification condition generator and a theorem prover like CVC [SBD02], PVS [COR+95] or Isabelle/HOL [NPW02]. The most well-known systems, besides KeY, are Boogie [ByECD+06] for C# using Z3 [dMB08] in its back-end and Why/Krakatoa/Caduceus [FM07] for Java and C, which hands its generated verification conditions over to a number of different theorem provers.

It seems that, in the future, these techniques might be integrated in a fully automated "verifying compiler" [Hoa03].

However, the verification of concurrent programs has proven itself to be an tedious task. There were efforts by Ábrahám et al. [ÁdBdRS03] and Beckert and Klebanov [BK07] to handle concurrent Java, but they are still at an early stage of development. The theoretical foundations of the verification of concurrent programs are discussed in [AO97, dRdBH+01].

Calculi for the verification of Creol were elaborated in [DJO08a, DJO06] and in [Bla08], a corresponding verification condition generator was established. The notion of the history or trace they are using was originally introduced by Dahl [Dah77] and Hoare [Hoa83].

A similar adaption of KeY for the programming language C is described in [Mür08].

1.3 Thesis outline

The main focus of this thesis is program verification. However, it should be accessible to computer scientists feeling comfortable with topics from discrete mathematics, such as first-order logic and graph theory. These concepts will be introduced briefly before their use. Another prerequisite is the understanding of principles and terminology from object-oriented programming. An elaborated introduction can be found in [Bal04].

The remaining parts of this thesis are structured as follows:

- Chapter 2 introduces mathematical foundations and notations necessary to understand the following parts of the thesis.
- Chapter 3 presents the modeling language Creol that is the subject of our analysis.
- Chapter 4 documents a parsing software for Creol used by KeYCreol.
- Chapter 5 provides an overview of the KeY tool.
- Chapter 6 explains typed dynamic logic and sequent calculi.
- Chapter 7 uses the theory of the previous chapter to extend it for reasoning about Creol.
- Chapter 8 reviews the KeYCreol software that implements the calculus from the preceding chapter.
- Chapter 9 contains two illustrative examples that are provable with KeYCreol.
- Chapter 10 summarizes the results and condenses the discussions of further work.
- Appendix A specifies a Creol Grammar.
- Appendix B comprises a glossary of mathematical symbols.

1.4 Summary contributions

The contributions of this thesis are of both theoretical and practical nature.

On the theoretical side is the creation of a dynamic logic and a corresponding calculus for a major subset of Creol. It reuses the concepts of the KeY system known from the verification of Java and combines them with ideas for the verification of Creol to handle the uncertainty occurring in distributed systems.

The main practical achievement is the establishment of a prototype of the KeYCreol software forming a verification system for Creol based on the

KeY system. An alternative design concept for parts of the KeY system is used reducing the lines of code by more than 80% in comparison to previous adaptions of KeY.

A minor practical accomplishment is the realization of a LL(1) grammar for Creol that significantly broadens the range of applicable parser generators for Creol in contrast to the prior existing LR(1) grammar.

Chapter 2

Preliminaries

This chapter introduces two mathematical concepts, namely sets and graphs, used throughout the thesis. These topics are usually taught in undergraduate computer science curricula, therefore this chapter focuses on terminology and notations. Readers feeling comfortable with these subjects are encouraged to move on to the next chapter.

2.1 Sets

Definition 2.1.1. *A* set *is a collection of pairwise different objects*

Example 2.1.1. A simple example is the set $A = \{1, 2, 3\}$ inclosing the numbers 1 to 3.

The following definition establishes operations on sets.

Definition 2.1.2. *For given sets A, B and a element e the following relations exist:*

- *The* membership *$e \in A$ denotes that e is contained in A.*
- *The* subset *relation $A \subseteq B$ expresses that all members of A are also members of B.*
- *The* relative complement *$A \backslash B$ includes all elements inclosed in A but not in B.*
- *The* intersection *$A \cap B$ holds all elements contained in both A and B.*
- *The* union *$A \cup B$ consists of all elements which are in A or B (or both).*
- *The* cardinality *$|A|$ denotes the number of elements in a set.*
- *For the* empty set *we write $\emptyset$.*

Example 2.1.2. Let $B = \{1, 2\}$ and $A = \{1, 2, 3\}$ be given. Then $B \subseteq A$ holds because 1 and 2 are contained in A. The union two sets B and $\{3\}$ is $B \cup \{3\} = A$. 1 is a member of the set A: $1 \in A$. The relative complement of A and B is $A \backslash B = \{3\}$. The cardinality of set describes number of objects in a set: $|A| = 3$. The empty set contains no elements: $|\emptyset| = 0$.

The members of a set can be ordered by a given relation R.

Definition 2.1.3. *A relation R is called partial order for a set A iff for all $a, b \in A$:*

- *aRa (reflexivity)*
- *if aRb and bRa then $a = b$ (antisymmetry)*
- *if aRb and bRc then aRc (transitivity)*

Example 2.1.3. The less or equal relation $\leq$ is a partial order on the integers $\mathbb{Z}$.

Given a partial order we can define the minimum.

Definition 2.1.4. *For a given set A and a partial order R the minimum is an element $a \in A$ such that for all $b \in A$: aRb.*

The minimum does not necessarily exist.

Example 2.1.4. The integers $\mathbb{Z}$ do not have a minimum. The natural numbers $\mathbb{N}_0$ have the minimum 0.

Definition 2.1.5. *The Cartesian product of two sets A and B is the set of ordered pairs where the first element belongs to A and the second to B:* $A \times B = \{(a, b) | a \in A, b \in B\}$

Example 2.1.5. For two sets $A = \{1, 2\}$ and $B = \{3, 4\}$ the Cartesian product is: $A \times B = \{(1, 3), (1, 4), (2, 3), (2, 4)\}$.

Definition 2.1.6. *The Cartesian product of n sets is a n-tuple:*
$A_1 \times \cdots \times A_n = \{(a_1, \ldots, a_n) | a_1 \in A_1, \ldots, a_n \in A_n\}$

We will write A^n for $\underbrace{A \times \cdots \times A}_{\text{n times}}$ and A^* for $\bigcup_{i=0}^{\infty} A^i$. Tuples abbreviated by an variable will be indicated by an bar: $\bar{w} \in A^*$.

Example 2.1.6. For a set $A = \{1, 2, 3\}$ the following tuples are contained in A^*: (), $(2, 3, 3, 1)$, and (2).

2.2 Graphs

Definition 2.2.1. *A* directed graph *$G = (V, E)$ consists of a set of vertices V such that $0 < |V| < \infty$ and a set of directed edges connecting vertices: $E = \{(v_1, v_2) | v_1, v_2 \in V\}$*

We will use the word graph meaning only directed graphs.

Example 2.2.1. In figure 2.1 is a graph with four nodes and five edges between them.

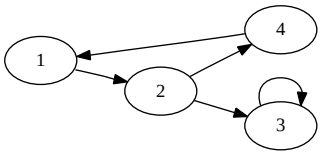

Figure 2.1: A directed graph

Definition 2.2.2. *On a graph $G = (V, E)$ a sequence $P = v_1, \ldots, v_n$ $(2 \leq n)$ with $v_i \in V$ satisfying $\forall i \in \{1, \ldots, n-1\} : (v_i, v_{i+1}) \in E$ is called a* path *from v_1 to v_n.*

Example 2.2.2. There is a path $P = 1, 2, 3, 3$ in figure 2.1.

Definition 2.2.3. *A* directed acyclic graph *is a graph $G = (V, E)$ where the property holds: For all paths $P = v_1, \ldots, v_n$ in G: $v_1 \neq v_n$.*

Example 2.2.3. The graph in figure 2.2 is an acyclic graph whereas figure 2.1 is cyclic.

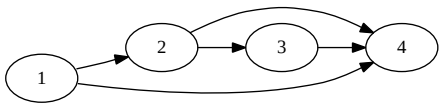

Figure 2.2: An acyclic directed graph

Definition 2.2.4. *Given a graph $G = (V, E)$ and a node $v_0 \in V$ named* root*, the graph is called* tree *iff $\forall v \in V : v \neq v_0$ there is exactly one path $P = v_0, \ldots, v$*

Example 2.2.4. Neither the graph in figure 2.2 nor in figure 2.1 is a tree, but the graph in figure 2.3 is.

Definition 2.2.5. *For a given tree $G = (V, E)$ a* leaf *is a node $v \in V$ such that $\forall v' \in V : (v, v') \notin E$*

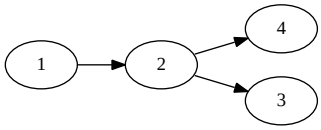

Figure 2.3: A tree

Example 2.2.5. The vertices numbered by 3 and 4 are leafs of the tree in figure 2.3.

Definition 2.2.6. *Given a graph $G = (V_G, E_G)$ with a node v_0 a* directed spanning tree *is $T = (V_T, E_T)$ with $V_T = V_G$ and $E_T \subset E_G$ such that T is a tree with root v_0.*

Example 2.2.6. Figure 2.3 contains one of the many spanning trees of figure 2.2.

Chapter 3

Overview of Creol

This chapter introduces Creol by example. The focus lies on the features significant to the verification of Creol. There are many aspects of Creol which are not covered by this work, like multiple inheritance [DJOS08, DJO04] and class upgrades [YJO06]. In the literature, there are different Creol dialects depending on their respective purpose. This thesis uses a new dialect which is largely consistent with the standard of the Creol compiler [Kya08]. The specific changes result from insights gained while building the verification system and have been discussed with the Creol team at University of Oslo. As Creol is still an experimental programming language, information in this chapter might have been subject to changes since this chapter has been written.

In the following paragraphs different facets of Creol will be discussed and the chapter finishes with an example in section 3.1.

Levels of parallelism

Each Creol object is assumed to be executed on its own (maybe virtual) processor. Hence, one has to think of a Creol program as each object is executed in parallel. The objects communicate among each other via message passing where a message can be a method call or a method completion.

Inside an object there can be several different threads. A new thread is created as soon as a method is invoked. Threads are scheduled cooperatively[1], meaning there is no timer which enforces a thread switch but a thread switch can only occur if the execution reaches a releasing statement. However, there are no assumptions about the scheduling behavior so one cannot assume that a particular thread is chosen at such a release-point. Threads of one object can communicate via shared variables, the class attributes. A thread can create new threads by calling a method of the same object.

[1] This comes with the drawback of blocked objects in case of an infinite loop.

Object viewpoints

References to objects are typed by interfaces. An interface contains methods which have to be provided by the object implementing the interface. Casting an object reference to another interface will enable other methods to be invoked or in other words changes the view on the object. Consider the following example of an interface declaration:

```
1 interface I
2 begin
3   with I2
4     op meth1(in a : Int)
5     op meth2(out b : Bool)
6 end
```

Listing 3.1: Interface declaration

The interface *I* provides two different methods, namely *meth1* and *meth2*. The input parameter of *meth1* is *a* type *Int*. The method *meth1* has one output parameter *b* of type *Bool*. An object storing a reference to another object typed by *I* can call the methods *meth1* and *meth2* if it implements itself the (co-)interface *I2*.

Inter object communication

In contrast to most other programming languages all methods calls are asynchronous meaning that the invocation is separated from the retrieving and assignment of the result. For an example consider the following listing:

```
1 var l : Label[Bool];
2 l!obj.meth(a,b);
3 ...;
4 l?(y)
```

Listing 3.2: Asynchronous communication

The method *meth* of object *obj* is invoked with input parameters *a* and *b*. As the retrieving of the results is separated from the invocation the method call must be remembered. Therefore labels exist in which a reference to a method call can be stored. In our case the label is *l*. The last statement l?(y) assigns the result of the method call corresponding to the given label to the variable *y*. If there is no answer available yet, the object will block (busy waiting) until the answer arrives. *y* must be of type *Bool* because the label is declared like this.

In general there are no assumptions on the underlying network which transmits the invocation or completion messages. So message overtaking is possible and must be considered by the programmer. This implies that the

sequential call of two methods on a given object does not ensure that the messages arrive at the object in the same order.

Classes

The only form, executable code can occur in, are classes. All objects occurring during run time are instances of a class. Classes have attributes where each object has its own copy of them. Attributes of an object can only be modified by the methods of that object. There is no remote access to attributes. Therefore objects can alter attributes of other objects only indirectly.

There are two kinds of methods. The first are typed by a co-interface and implemented because the class implements an interface. Such methods can be called by other objects. The second kind of methods are local methods not typed by an co-interface. They can only be called from other methods inside the class.

Let us consider an example:

```
1  interface I
2  begin
3    with Any
4      op meth1(in a : Bool)
5  end
6
7  class A(x : Int) implements I
8  begin
9    var attr1 : Int;
10   var attr2 : Bool;
11
12   op init == attr1 := x
13   op run == skip
14
15   with Any
16     op meth1(in a : Bool) == attr2 := a
17 end
```

Listing 3.3: Class

The class *A* implements the interface *I* which provides the method *meth1*. This method has to be implemented in the class. The class attributes *attr1* and *attr2* are accessible to all methods of the class. The two local methods, *init* and *run*, are special and exist in every class. *init* is called on object creation where the class parameters (in the example x) are its parameters. After *init* has terminated *run* is called which usually invokes actions in the object. In our case it does nothing stated by *skip*.

Releasing statements

A releasing statement allows the scheduler of an object to interrupt the current thread, and to choose another thread to run. The simplest statement is the *release* statement:

```
1 release
```

Listing 3.4: Release

This simply makes it possible for other threads to be scheduled.

The mostly used statement is the *await* statement which is bound to a condition usually called guard. If the guard evaluates to true, the execution will continue without releasing. If the guard is evaluated to false, the thread will release. It can only be rescheduled when the guard is true.

```
1 await g
```

Listing 3.5: Await

In the above listing g is a placeholder for a guard. Guards can either be a Boolean expression like $a > 0$ or *l?* or *wait*, but not a combination of them. The *l?* notion is true if an answer to the given label has arrived and false otherwise. *await wait* is a long form of expressing *release*.

Indeterministic choice statement

In contrast to popular programming languages Creol contains an indeterministic choice statement that has two branches of statements where only one is executed. The decision depends on the first statements of the branches:

```
1 await l?; l(x) [] a:=b+c
```

Listing 3.6: Indeterministic choice

In the above listing the left branch can only be executed if the answer to the label l has arrived. In the latter case one of the branches will be chosen randomly. Otherwise the right branch will be executed. When both first statements are not ready for execution the statement will block the object until one is.

Type system

The Creol type hierarchy used throughout the thesis is a subset of the actual one available in [Kya08]. The top of the type hierarchy of Creol is *Data*. All other data types are subtypes of it. This sets the Creol type hierarchy apart from popular programming languages like Java because *Data* is a common top element for primitive and reference types. All interfaces inherit the *Any* type which is also implemented by all classes.

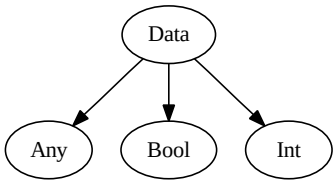

Figure 3.1: Types in Creol

Exceptions

In the Creol literature there is no exception handling as this was postponed for the sake of more innovative features. The Maude machine interpreting Creol machine code initializes all declared variables with *null* and simply stops if an exception occurs. If we implemented the same semantics in a calculus we would have to inspect every variable whether is it *null* or not (a:=**null**+2 leads to an exception) leading to a remarkable overhead.

Therefore in this work all declared variables are implicitly initialized with a standard value. For integers it is 0, for Boolean variables it is $false$ and for labels and object references we use *null*. Even under this precautions an exception emerges if the execution reaches a point where a method of *null* is called or if a division by zero is attempted. We deal with such exceptions by simply blocking the object. This means the current thread does a infinite loop which could be described by:

```
1 while true do
2   skip
3 end
```

Listing 3.7: While

We the abbreviate the above behavior by a single statement:

```
1 block
```

Listing 3.8: Block

3.1 Example: Unbounded buffer

We conclude this chapter with an example which implements an unbounded First In First Out (FIFO) buffer. In the flavor of Creol we will not use a queue as a predefined data structure but create it as linked list of objects. Every object will store exactly one element of the buffer and a reference to the next object in the buffer.

Let us start with the declaration of the corresponding interface:

```
1  interface BufferSpec
2  begin
3    with Any
4      op put(in x : Any, seq : Int)
5      op get(in seq : Int ; out y : Any)
6  end
```

Listing 3.9: Buffer Interface

There are two methods namely *put* which receives a reference to an object and a sequence number and *get* which just has a sequence number as a parameter and returns a reference to an object. The *put* method will intuitively add an object to the buffer whereas *get* retrieves an object. The sequence numbers are necessary because of message overtaking.

```
1  class Buffer implements BufferSpec
2  begin
3    var cell:Any;
4    var next:Buffer;
5    var ins:Int;
6    var outs:Int;
7
8    with Any
9      op put(in x:Any, seq:Int) == await ins=seq;
10        if ins-outs = 0
11        then cell:=x
12        else if next=null then next:=new Buffer end;
13          !next.put(x,seq)
14        end; ins:=ins+1
15     op get(in seq:Int; out y:Any) == var l:Label[Any];
16        await outs=seq; await ins-outs>0;
17        if cell=null
18        then l!next.get(seq,x); l?(y)
19        else y:=cell end; outs:=outs+1
20 end
```

Listing 3.10: Buffer Class

The class has four attributes which are the stored object, the reference to the next element of the buffer, and the last sequence numbers for *put* and *get*. The *put* method assigns its own cell with the object to store if the following parts of the buffer are empty (ensured by *ins-outs=0*). Otherwise the element will be forwarded in the buffer. This maintains the property that older elements are first in the buffer. Hence the *get* method simply returns the first stored object to be found in the buffer. If a *put* or *get* call arrives out of order its execution is delayed by the initial await statement.

Readers still feeling uncomfortable with Creol can have look at chapter 9 where there are two more examples or consult the Creol literature [DJO05, DJO08a, Bla08, Kya08, Bla07] which contains an extensive amount of examples.

Chapter 4

jCreol: A parsing library

jCreol is a library which supports parsing and simple reference resolving of Creol code. It is designed for the linkage with other programs. jCreol is protected by the GNU general public license and can be obtained upon request from http://www.key-project.org. To simplify the linking to the KeY system it is written in Java.

We will open this chapter by a documentation of the architecture of jCreol. Thereafter a brief guidance into the use and the deployment of jCreol in other projects is presented in section 4.2. The chapter ends in section 4.3 with discussions on immanent constraints and suggestions on prospective work.

4.1 Architecture

The architecture of jCreol is centered around the *Main* class which invokes the other parts of the system. A UML diagram[1] providing an overview can be found in figure 4.1.

A pass of jCreol consists of these steps:

- First, the ANTLR lexer converts the sequence of characters given by the Creol input file to a sequence of tokens each representing categorized text.
- Second, the parser determines the grammatical structure of the sequence of tokens.
- Third, the resulting parse tree is translated into the graph data structure.

[1] Unified Modeling Language created by the Object Management Group http://www.omg.org

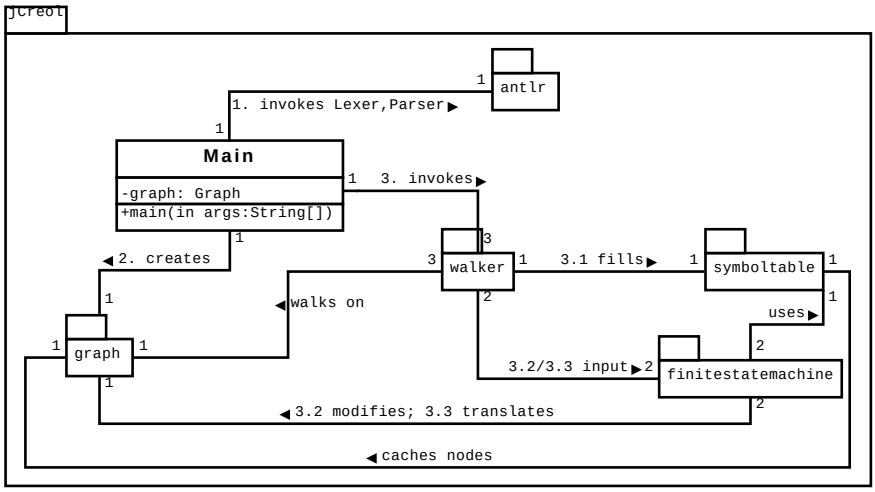

Figure 4.1: jCreol architecture UML diagram. The *antlr*, *graph*, *walker*, *symboltable* and *finitestatemachine* packages are depainted in detail in the figures 4.2, 4.4, 4.6, 4.5 and 4.7, respectively.

- Next, a walker traversing the graph fills a symbol table which maintains a dictionary relating identifiers of classes, interfaces, data structures, functions, and variables with their corresponding nodes in the graph.
- In the following step another walker moves on the graph which provides the names of the nodes of the graph as an input for a finite state machine which recognizes identifiers and modifies the graph to resolve these.
- Finally, a third walker is called which wanders on the modified graph and again controls a finite state machine. The automaton is supplied by an external program which in our case is KeYCreol.

ANTLR

For creating the lexer and the parser the ANTLR parser generator[2] was used. A good introduction to ANTLR can be found in [Par07]. A comprehensive description of lexers and parsers is available in [ASU86] chapters 3 and 4, respectively.

In figure 4.2 there is an overview of the *antlr* package in jCreol, visualizing that the tokens produced by the lexer are handed over to the parser. ANTLR does not strictly separate between lexer and parser. Hence both are in the same source code file called "Creol.g" which can be found in the *antlr* package of jCreol.

[2]ANother Tool for Language Recognition. http://www.antlr.org

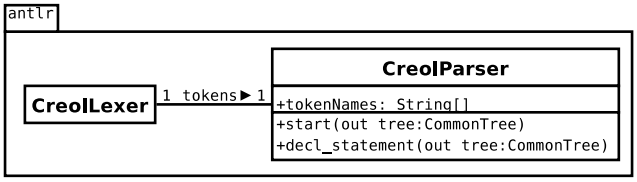

Figure 4.2: UML diagram of the architecture of the antlr package in jCreol. Methods and attributes are simplified. The displayed methods of *CreolParser* are used to launch the parser at the corresponding rule.

Token names consist of capital letters in ANTLR so we will keep this notation in the examples.

Lexer The task of a lexer is to determine the category of each word in the Creol source code and indicate its category by a token. For example, a token can represent an integer, one or more keywords of the Creol language or the plus operator. Let us consider an example:

```
1 class Example
2 begin
3   var i:Bool;
4   op foo == i:=true
5 end
```

Listing 4.1: Creol code example

The sample class given above would be converted into the token stream:

```
1 CLASS CLASS_IDENTIFIER BEGIN VAR IDENTIFIER COLON
2 CLASS_IDENTIFIER SEMICOLON OP IDENTIFIER DOUBLE_EQUAL
3 IDENTIFIER ASSIGN TRUE END
```

Listing 4.2: Token stream resulting from listing 4.1

We note that the meaning of the token *IDENTIFIER* is not unique for instance as it might betoken a variable name or a function name. This is where the parser comes into play.

Parser The parser determines the syntactical structure of a token stream using a grammar. For example the grammar rule given in listing 4.3 is able to accept all arithmetic expressions just consisting of plus, minus and integers.

```
1 expr -> ( expr ( ( PLUS | MINUS ) expr)* )
2         | INTEGER
```

Listing 4.3: Grammar rule example

The words in capital letters are the tokens produced by the lexer whereas the words in lower case are non-terminal symbols which should be replaced by the parser until there are just tokens left. The vertical symbolizes a decision meaning that just one of the alternatives is chosen. The asterisk, usually called Kleene star, stands for zero or arbitrary many occurrences. A parser examining an expression would start just with *expr* and replace it by appropriate instances of the right side of the rule until there are no appearances of *expr* left.

Continuing the previous examples, the resulting parse tree for the code given in listing 4.1 and its tokens in listing 4.2 is pictured in figure 4.3.

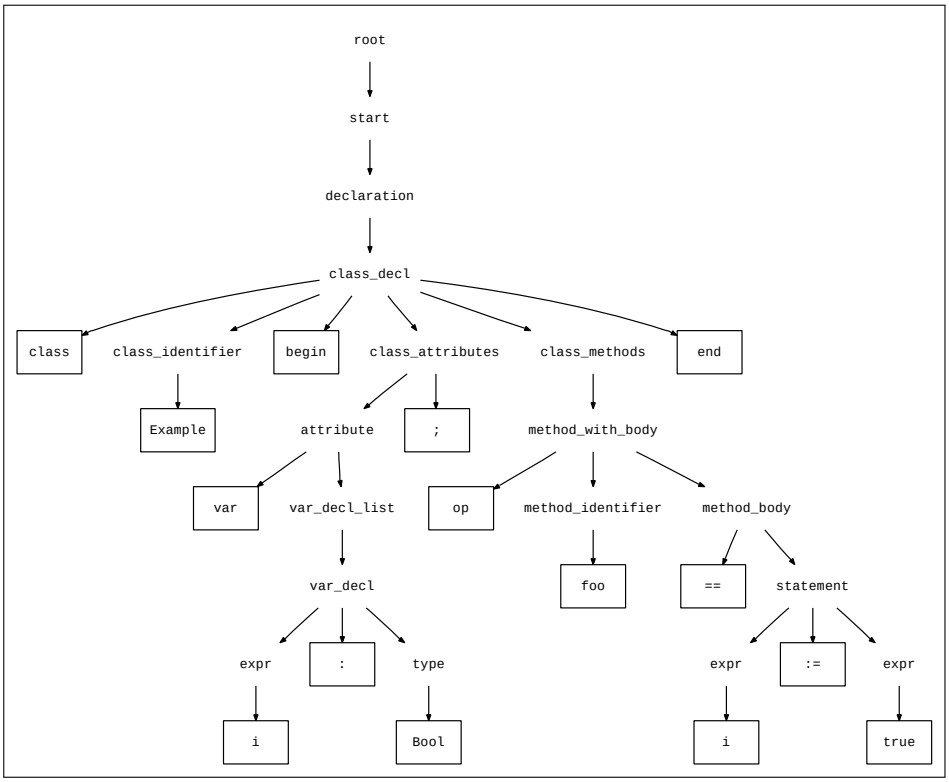

Figure 4.3: Parse tree of listing 4.1. Each node without an frame represents one application of a rule of the grammar. The (non-)terminals occurring in the right hand side of the rule are displayed as children of the left hand side. The positions of the code snippets (internally represented by tokens) in the tree are illustrated by the nodes with a frame. Some non-terminals were skipped.

There are different strategies for parsers. Two of them will be mentioned here.

LR(1) parsers use a bottom-up strategy. They read from left to right emblematized by L, try to expand the right most instance of a non-terminal

typified by R and just look at one token to decide which grammar rule to apply expressed by the (1). They are known to be complicated to design. A LR(1) parser is used in the Creol compiler. The corresponding grammar can be found in Appendix A of [Kya08]).

LL(1) parsers differ from LR(1) parsers in their strategy, which is top-down, and in the way of expansion. They use the left most non-terminal for rule application instead of the right most. Such a parser is used by ANTLR and therefore a LL(1) grammar for Creol was created within this thesis (see Appendix A for the grammar). The languages recognized by LL(1) are a proper subset of the LR(1) languages, so a LL(1) grammar does not necessarily exist for a given LR(1) grammar.

More detailed information about LL and LR parsers can be looked up in [ASU86].

On a parse tree some cosmetic transformations like cutting unnecessary chains of nodes and creating of new nodes with meaningful names are performed by rewriting rules. The successive tree is the abstract syntax tree (AST) which is processed by other parts of the program.

Graph

The graph data structure is a directed, acyclic graph (see section 2.2). An overview of its program structure be found in figure 4.4.

In the program for vertices the corresponding class *GraphNode* and for edges the class *GraphEdge* exist. As the abstract syntax tree is translated to the graph the notion of parents and children are kept assuming that the directed edges always point from a parent to its children. Internally the directed edges have to be stored in both the node of their origin and of their ending because the walker has to traverse them in both directions.

After resolving references nodes representing declarations might have several parents (see figure 4.9). Hence a walker wandering the graph has several possibilities to reach a node. On account of this the program maintains a directed spanning tree. The initial AST is already a spanning tree. Therefore each edge which is redirected in the process of resolving references is marked as not being part of the spanning tree. Whether a edge belongs to spanning tree or not is expressed in the program by the *belongsToTree* attribute of each edge. In a spanning tree there is exactly one path to each node, so the walker can simply follow the spanning tree ensuring a unique visiting order of the vertices.

An example for a graph with its spanning tree is illustrated in figure 4.9.

Symbol table

Some of the nodes in the graph will be used frequently and thus are cached by the symbol table (cf. [ASU86]). This is done by the *table* attribute of the

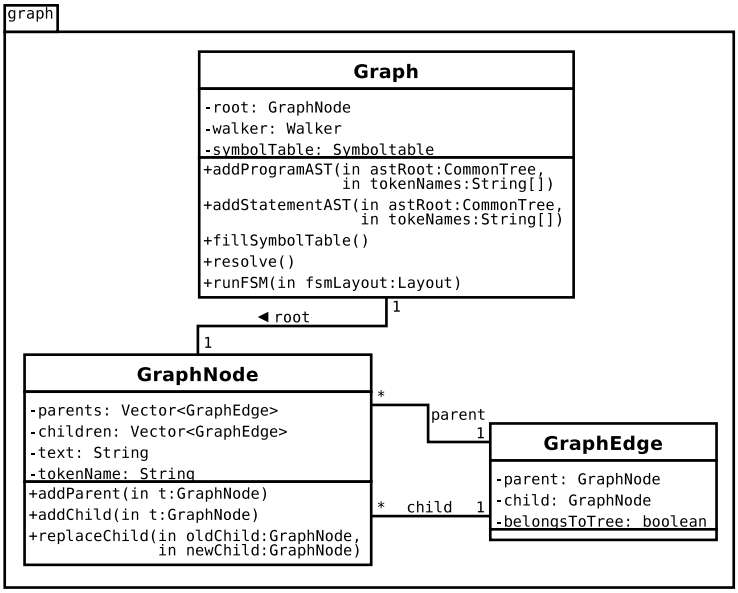

Figure 4.4: UML diagram of the graph data structure in jCreol leaving out set methods. The *text* and the *tokenName* attributes of *GraphNode* store the actual program snippet and its token name, respectively. The *fillSymbolTable* and *resolve* methods of *Graph* launch the walker with the corresponding behavior. *runFSM* accepts a finite state machine layout and runs it on the graph.

SymbolTable class referring to figure 4.5. The nested HashMaps distinguish between the types class, interface, data type and function on the first level and by identifier on the second level. The vector is necessary because there might be several instances with the same identifier. It stores the nodes of their declaration.

The look up of variables by their name uses nested scopes since a variable declared in a class might be hidden by a variable declared in a method for example. The symbol table stores a reference to the most inner scope e.g. of a method. Outer scopes can be reached via the *parent* attribute of the *Scope* class. *EnterScope* and *LeaveScope* creates a new inner scope and deletes the most inner scope, respectively.

Walker

The walker wanders on the graph data structure implementing the visitor design pattern (cf. [GHJV95]). It performs a left-depth-first search (similar to section 22.3 of [CLRS01]) on the spanning tree using two stacks to store

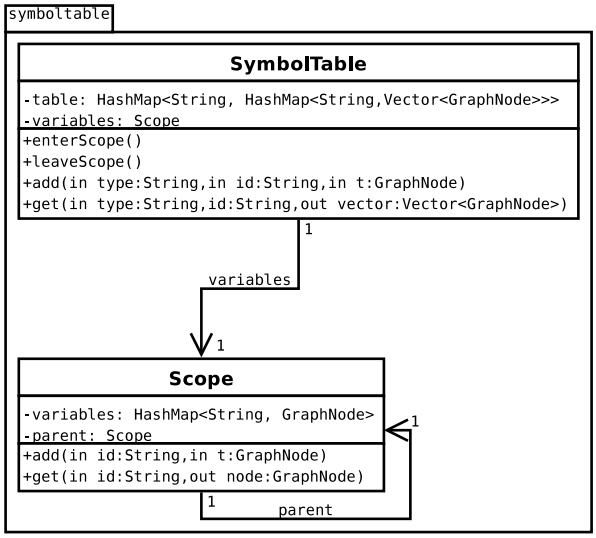

Figure 4.5: UML diagram of the symbol table in jCreol. The *get* and *add* methods of *Scope* are only called by their corresponding methods in *SymbolTable* which hence offers an unified interface to the other parts of the program.

which child is next to visit and from which parent it came from[3].

The walker can be supplied with different behaviors implementing the interface *Behavior* of figure 4.6. This simplifies the process of reusing jCreol for other purposes due to the fact that any non context sensitive operation to be performed on the graph can be implemented by defining a single new class.

Let us have a look at the two behaviors supplied with jCreol.

FillSymbolTable adds a reference to all nodes in the graph declaring a class, interface, data type, or function to the symbol table.

Walker2FSM forwards the actions taken by the walker to the finite state machine.

Finite state machine

The finite state machine is used to track the context the walker is in on the graph and to execute actions on specific nodes of the graph. A UML diagram of the package is available in figure 4.7. In the following the finite state machine used by the program is specified which is an adapted version of a Mealy machine probably named after [Mea55].

[3]As each node has exactly one parent in the spanning tree, one could store this information in the node as well.

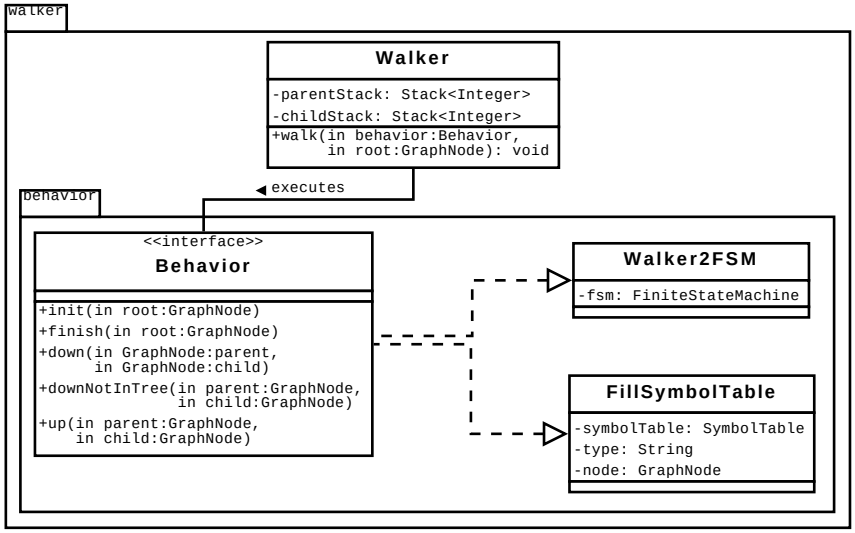

Figure 4.6: UML diagram of the walker package in jCreol. *init* and *finish* are activated by the *Walker* class before and after the run, respectively. *down* and *up* are called if moving from parent to child and vice versa. *downNotInTree* is executed when there is an edge which is not in the spanning tree.

Definition 4.1.1. *A* finite state machine *is a tuple* $F = (S, s_0, \Sigma_1, \Sigma_2, \Gamma, T, G)$ *where*

- S *is a finite set of states*
- s_0 *is the start state.* $s_0 \in S$
- $\Sigma_1 = \{up, down, downNotInTree\}$ *is an input alphabet*
- Σ_2 *is an input alphabet which contains all possible tokens*
- Γ *a set of actions containing at least the empty action* ϵ
- $T : (S \times \Sigma_1 \times \Sigma_2) \rightarrow S$ *a transition relation mapping a state and input to a consequent state*
- $G : (S \times \Sigma_1 \times \Sigma_2) \rightarrow \Gamma$ *an output relation naming the action to be performed on a transition*

As each transition is determined by two input symbols provided by the *Walker2FSM* behavior of the walker, the *transition* attribute of the *State* class is implemented as a nested HashMap. The outer HashMap distinguishes between *up*, *down* and *downNotInTree*. *up* denotes that the walker is moving from a child to a parent. *down* expresses that the walker is changing from a parent to a child. *downNotInTree* is a notification when there is a

child which is connected by an edge not belonging to the spanning tree. The inner HashMap identifies the next state for a given token, delivered as an input for the finite state machine, which is the token belonging to the node the walker is approaching. The *actions* attribute follows the same scheme.
Alike the extensible design of the walker the finite state machine supports

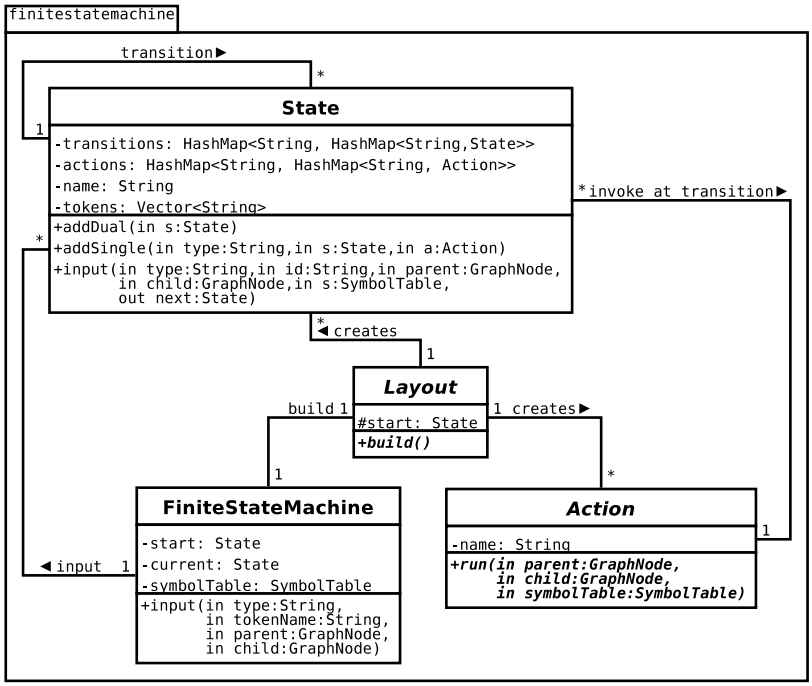

Figure 4.7: UML diagram of the finite state machine in jCreol. The *FiniteStateMachine* class forms the interface to the other packages of the program. The *input* method of *FiniteStateMachine* forwards its calls to the *input* method of the current state which returns the next state and runs the action of the transition if available. The *add** methods of *State* are called by the *build* method of the class *Layout* to create the transitions.

the existence of several layouts achieved by the *Layout* class.

Resolve This layout resolves references of classes, data types, functions, interfaces and variables by replacing each identifier by a non spanning tree edge to its declaration node available in the symbol table (see figure 4.9 for an example). An exemplary part of the finite state machine can be found in figure 4.8.

4.2 Using jCreol

jCreol currently supports only Linux platforms.

As external libraries log4j[4], junit[5] and ANTLR3 are requisite to compile and run. For plotting functionalities currently graphviz[6] and the image viewers feh and eog[7] are utilized where the latters are exchangeable by any other image viewing software.

There are different possibilities to invoke jCreol. Either only the parser is run separately or with resolving of references where in both cases the resulting graph can be drawn. Additionally the layout of the finite state is displayable.

External programs Besides the main class the *jCreolExternal* class can run all parts of jCreol. It is designed to be used by external programs and supports loading of sequences of statements or complete Creol programs.

When invoking jCreol externally a layout for a finite state machine has to be supplied which is the last to be applied to the AST. To accomplish this, the *Layout* and the *Action* classes have to be inherited and thereby adjusted by the external program. In this stage of the development it is particularly important to visualize the results of jCreol to be able to design the layout of the finite state machine.

KeY, for instance, uses jCreol to parse the content of modalities and to create the related KeY AST.

Hints for developers To further design the grammar the ANTLRWorks[8] framework is a helpful tool since it checks grammars for errors and provides debugging facilities.

The log4j library is heavily used if the level is set to debug. Errors occurring during execution are printed by the logger and do not necessarily crash the program.

A number of test cases for the grammar as well as the complete program are in the folder *test* and are runnable automatically simplifying the process of detecting bugs.

4.3 Limitations and further work

The resolving of references in the present version is limited in the sense that it does not support overloading of methods. To achieve these extended functionalities the symbol table has to store information about the parameters of each method.

[4] logging.apache.org/log4j/
[5] www.junit.org
[6] www.graphviz.org
[7] www.gnome.org/projects/eog/
[8] www.antlr.org/works/

The present version of jCreol does not provide type checking or similar support to check the validity of Creol code. While parsing a complete program, references to classes, interfaces, or variables are resolved and throw errors if not declared. Except for the resolving all code is parsed which is valid according to the grammar available in Appendix A. However, the grammar allows non-legal statements. To keep the generative approach of jCreol one could supply the AST tree grammar walker available in the sources with checks regarding properties of the AST. Another possibility would be to invoke the actual Creol compiler before processing a Creol program and reject it on compiling errors. This approach would connect the accepted programs more closely to development of the Creol language itself avoiding forming another Creol branch accepted by jCreol.

The present grammar (see Appendix A) accepts a superset of the Creol language. It is extended by KeY specific rules for schema variables (section 8.1) and the special return statement (section 7.3). To get rid of those features in the grammar, one would have to create a second separate grammar. The reason is that there is currently no inheritance mechanism for grammars in ANTLR3[9]. The other option would be to allow it in the grammar and apply a check against those features which can be turned on and off.

[9]see ANTLR3 wiki. Accessed 11.05.2009

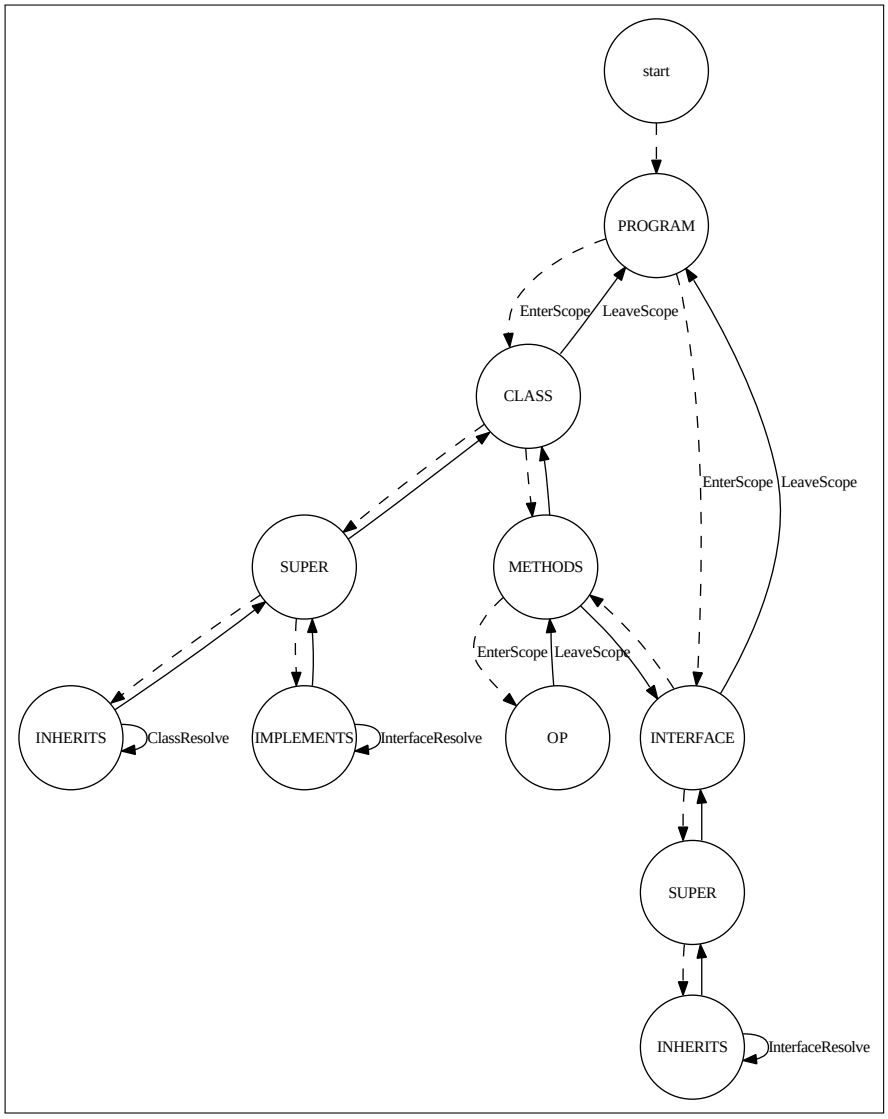

Figure 4.8: Parts of the finite state machine for resolving references. Dashed arrows embody transitions $T : (S \times down \times \Sigma_2) \to S$. Continuous arrows stand for transitions $T : (S \times up \times \Sigma_2) \to S$. As every state is always reached using the same token of Σ_2 it is printed inside the state. Actions are written next to transitions.

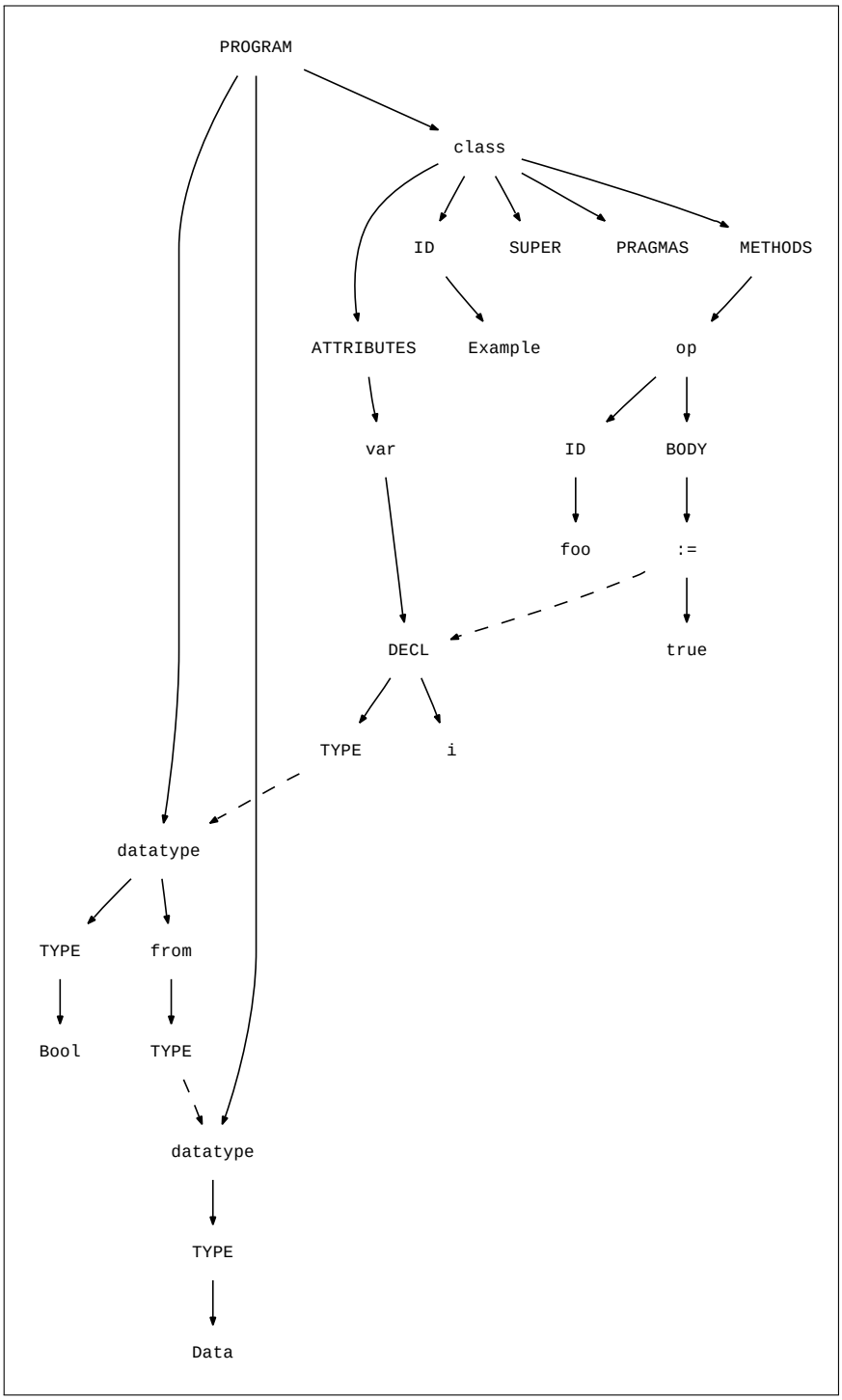

Figure 4.9: Graph with resolved references (e.g. the variable i or the type *Bool*) generated out of listing 4.1. Dashed arrows specify edges of the graph not belonging to the spanning tree. The data types *Bool* and *Data* belong to the Creol library and were added automatically. Unrelated parts of the library were left out in this figure. Nodes named in capital letters are added by (cosmetic) rewriting rules. Names in lower case show code snippets occurring in listing 4.1 or the library.

Chapter 5

Overview of the KeY tool

This chapter provides an overview of the KeY tool [ABB+05, BHS07] in general. Its adaption for Creol is described in chapter 8. The theory behind the KeY tool is elaborated in [BHS07] and our adaption for Creol is available in the chapters 6 and 7.

The KeY tool is a development tool which supports the formal analysis of software. It aims to be a user-friendly software, thus lowering the entry level of formal methods into the software development process. The software can be launched without installation using the Java Webstart[1] version on http://www.key-project.org. It is written in Java and published under the GNU general public license.

At the same time the analysis process is rooted in a well-founded theory. It uses symbolic execution with induction (originally by [Bur74]). Symbolic execution follows the control flow of a program, but instead of using explicit values it uses symbolic values. The effect of a statement is applied according to its operational semantics. Thereby one run of a symbolic execution can stand for infinitely many ordinary runs of a program. For instance, while symbolically executing the statement $x := x + 1$ we would consider all possible values of x in one step. Therefore a version of dynamic logic [Pra77, Har79] is used which is a logic extended by types (similar to types in programs) and modalities containing the code to be symbolically executed. The run of a symbolic execution checks source code against a given specification. The single steps in a symbolic execution run are encoded in a special language called taclets [Hab00a, Hab00b] which is described in section 8.1. Eventually, all source code will be executed symbolically. Then (also already during symbolic execution) the remaining logical formulae are simplified with respect to arithmetic and logical properties using the internal prover or external decision procedures like Yices [DdM06], CVC [SBD02], and Simplify [DNS05], until the specification has been shown. The internal theorem prover of KeY uses a sequent calculus [Gen35]. During a run the

[1] http://java.sun.com/javase/technologies/desktop/javawebstart/index.jsp

symbolic state is kept in updates [Bec01], which essentially are postponed logical substitutions and provide an additional stage of simplification. More details about updates can be found in chapter 6.

Originally KeY was developed for the verification of Java Card[2] where the specifications could be written in the Java Modelling Language (JML) [BCC+05], the Object Constraint Language (OCL) [WK99], or directly in the logic used by the software. Recently, adaptions of the KeY tool for other purposes including the verification of the C programming language [Mür08] and of hybrid systems [PQ08] were developed.

Research using the KeY tool covered different topics in the software verification area, e.g. the creation of finite counter examples [Rüm05], the verification of concurrent Java [BK07], the generation of Junit test cases [EH07], and proof visualization [Bau06].

[2] http://java.sun.com/javacard/

Chapter 6

Creol dynamic logic

A first-order logic extended by sorts and modalities which contain program code enabling reasoning about programs is introduced in this chapter. We use a dynamic logic [Pra77, Har79] which is similar to Hoare Logic [Hoa69] or the weakest precondition calculus [Dij75].

Let us start with a simple example to show what we are up for: Knowing x equals zero we can prove that after executing the incrementing statement x := x+1, it will equal one: $x \doteq 0 \rightarrow \langle \texttt{x := x+1} \rangle x \doteq 1$. In this case $\langle\ \rangle$ is the modality, which contains the statement `x := x+1`. $\rightarrow$ is the logical implication meaning if the left hand side, $x \doteq 0$, holds then the right hand side of the implication has to hold. The modality creates a state update which changes the interpretation of x to 1. So the formula $x \doteq 1$ after the modality holds.

Some basic understanding of first-order logic will be required. There are many textbooks with elaborated introductions, e.g. [Sch00],[Fit90].

The chapter is divided into four different parts. Section 6.1 establishes sorts. Thereafter section 6.2 describes the syntax of logical formulae. The corresponding semantics are discussed in section 6.3. Finally section 6.4 presents a sequence calculus for the logical formulae.

The definitions of this chapter are taken from [BHS07] chapters 2 and 3 and are slightly adapted. Another similar introduction to a dynamic logic calculus for C can be found in [Mür08].

6.1 Sorts

First, we extend classical first-order logic by sorts. The type hierarchy of section 3 will be represented by a sort hierarchy in the logic. Similar to the type system we distinguish between static and dynamic sorts. The theoretical background of this section was developed in [Gie05].

Definition 6.1.1. *A* sort hierarchy *is a tuple* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_s, \sqsubseteq)$ *where*

- $\mathcal{T}_d$ *a finite set of dynamic sorts*
- $\mathcal{T}_s$ *a finite set of static sorts*
- $\mathcal{T}$ *is a finite set of sorts* $\mathcal{T}_d \cup \mathcal{T}_s = \mathcal{T}$
- $\sqsubseteq$: $\mathcal{T} \times \mathcal{T}$ *is a sub sort relation, introducing a partial order on* $\mathcal{T}$
- $Bottom \in \mathcal{T}_d$, $Top \in \mathcal{T}_s$ *with* $\forall A \in \mathcal{T} : Bottom \sqsubseteq A \sqsubseteq Top$
- $\mathcal{T}$ *is closed under greatest lower bounds w.r.t.* $\sqsubseteq$

We write $A \sqcap B$ for the greatest lower bound of A and B and $A \sqcup B$ for the least upper bound if it exists. An example for the greatest lower bound using figure 6.1 is: $Bool \sqcup intDom = Data$.

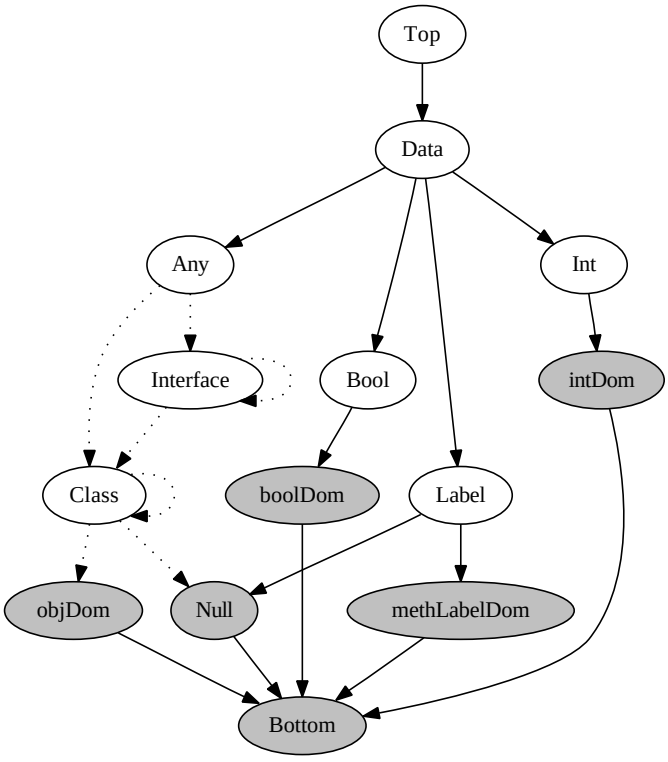

Figure 6.1: Parts of the sort hierarchy used by the KeY system. An arrow from A to B represents $A \sqsubseteq B$. Sorts with gray background are dynamic sorts. A sort with white background is a static sort. Depending on the analyzed program there might be no or many interfaces with sub sorting between them and there might be classes with sub sorting between them and sub sorting of their implemented interfaces illustrated by the dashed arrows (see Def. 6.2.4).

We will use static sorts for terms and dynamic sorts for the domains the terms will be interpreted in. As the *Bottom* sort is sub sort of all sorts every term can be interpreted. Using the minimal sort hierarchy $\mathcal{T}_s = \{Top\}$ and $\mathcal{T}_d = \{Bottom\}$ the well known first-order logic emerges.

Example 6.1.1. Let us instantiate $\mathcal{T}_s$ by $\{Top, Int\}$ and $\mathcal{T}_d$ by $\{intDom, Bottom\}$. In addition let $Bottom \sqsubseteq intDom \sqsubseteq Int \sqsubseteq Top$. Our resulting sort hierarchy is:

$$(\{Top, Int, intDom, Bottom\}, \{Bottom, intDom\}, \{Int, Top\}, \sqsubseteq)$$

This is the sort hierarchy to be used in the following examples.

The partial order $\sqsubseteq$ is transitive meaning $A \sqsubseteq B$ and $B \sqsubseteq C$ implies $A \sqsubseteq C$. Therefore we are not able to express direct sub sorts. This requires another definition.

Definition 6.1.2. *For a sort hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_s, \sqsubseteq)$ *the* direct sub sort relation $\sqsubseteq^0 \subseteq \mathcal{T} \times \mathcal{T}$ *is defined for any* $A, B \in \mathcal{T}$ *as:*
$A \sqsubseteq^0 B$ *iff* $A \sqsubseteq B$ *and* $A \neq B$ *and*
for any $C \in \mathcal{T}$ *with* $A \sqsubseteq C \sqsubseteq B$ *it follows* $A = C$ *or* $B = C$

Example 6.1.2. In the sort hierarchy of Example 6.1.1 it holds $Int \sqsubseteq^0 Top$ and $intDom \sqsubseteq^0 Int$.

As Creol has predefined types we need to define a minimal sort hierarchy which contains their correspondent. The translation of the type hierarchy of section 3 to the sort hierarchy is done by an injective function which maps each type to a sort with exactly the same name. The difference is that we need more sorts than types to create a well-founded sort hierarchy.

Definition 6.1.3. *A* CreolDL sort hierarchy *is a sort hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_s, \sqsubseteq)$ *such that*

- $\mathcal{T}_d$ *contains all domains of appendix B.8.*
- $\{Data, Int, Bool, Label, Any, Interface, Class\} \subseteq \mathcal{T}_s$ *fulfilling the relations of figure 6.1*
- $\{History, ObjectHistory, SendingHistory\} \subseteq \mathcal{T}_s$ *fulfilling the relations of figure 7.3*
- $\{Method, Message, TermLabel, NewLabel\} \subseteq \mathcal{T}_s$ *fulfilling the relations of figure 7.2*

The described sort hierarchy is drawn in the figures 6.1, 7.3 and 7.2. From now on a sort hierarchy will always be a CreolDL sort hierarchy unless stated otherwise.

6.2 Syntax

To construct dynamic logic formulae different sets of names will be necessary and those are kept in the signature. The set of functions is split into rigid and non-rigid symbols where the first have the same meaning in every program state whereas the meaning of the latter can change, e.g. because of an assignment (see Def. 6.3.10). All predicates are rigid.

Definition 6.2.1. *Given a sort hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_s, \sqsubseteq)$ *a* signature *is* $\Sigma = (VSym, FSym_r, FSym_{nr}, PSym, \alpha)$ *where:*

- *the set of variables* $VSym$*, the sets of function symbols* $FSym_r$*,*$FSym_{nr}$ *and the set of predicate symbols* $PSym$ *are pairwise disjoint*
- α *is the sort function such that*
  - $\forall v \in VSym : \alpha(v) \in \mathcal{T}_s$
  - $\forall f \in FSym_r \cup FSym_{nr} : \alpha(f) \in \mathcal{T}_s^* \times \mathcal{T}_s$
  - $\forall p \in PSym : \alpha(f) \in \mathcal{T}_s^*$

From now on the abbreviation $FSym := FSym_r \cup FSym_{nr}$ will be used. Furthermore, we will write $f : A_1, \ldots, A_n \to A$ instead of $\alpha(f) = ((A_1, \ldots, A_n), A)$ and $P : A_1, \ldots, A_n$ instead of $\alpha(P) = (A_1, \ldots, A_n)$.

Example 6.2.1. For the sort hierarchy of Example 6.1.1 a signature could be

$$\Sigma = (\{x\}, \{1, 2, 3, +, -\}, \{a, b\}, \{\doteq\}, \alpha)$$

such that $\alpha(x) = Int$, $\alpha(a) = \alpha(b) = ((), Int)$, $\alpha(+) = \alpha(-) = ((Int, Int), Int)$ and $\alpha(1) = \alpha(2) = \alpha(3) = ((), Int)$. Or using the abbreviating notation: $+ : Int, Int \to Int$ and $\doteq \; : Int, Int$. The sets are disjoint: It holds that $FSym_{nr} \cap FSym_r = \emptyset$, for example.

Alike the sort hierarchy we need some special functions and predicates to reason about Creol. They can be looked up in the appendices B.9 and B.10. We call a signature containing them a CreolDL signature, which will be all signatures mentioned from now on.

The next definitions depend mutually on each other but to improve readability they are distributed to the Definitions 6.2.2, 6.2.3 and 6.2.5. We will define terms (Def. 6.2.2) which are the parameters of predicates. Formulae (Def. 6.2.5) consist out of predicates, modalities with code and logical symbols connecting them. Updates[1] (Def. 6.2.3) representing state updates of processed code can prefix both terms and formulae.

Definition 6.2.2. *Given a sort hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_s, \sqsubseteq)$ *and a signature* Σ *the sets of* terms $\{Term_A\}_{A \in \mathcal{T}_s}$ *are inductively defined as the least system of sets fulfilling*

[1] Updates are postponed substitutions

- $x \in Term_A$ *for any variable* $x \in VSym$ *with* $\alpha(x) = A$
- $f(t_1, \ldots, t_2) \in Term_A$ *for any function symbol* $f : A_1, \ldots, A_n \to A \in FSym$ *with* $t_i \in Term_{A'_i}$ *where* $\alpha(t_i) = A'_i \sqsubseteq A_i$ $(1 \leq i \leq n)$
- $(\mathit{if}\ \phi\ \mathit{then}\ t_1\ \mathit{else}\ t_2) \in Term_A$ *for all* $\phi \in Formulae(\Rightarrow$ *Def. 6.2.5),* *for any* $t_1 \in Term_{A_1}$, $t_2 \in Term_{A_2}$ *with* $A = A_1 \sqcup A_2$
- $\{u\}t \in Term_A$ *for all updates* $u \in Updates(\Rightarrow$ *Def. 6.2.3) and all terms* $t \in Term_A$
- *(ifExMin* $x\ \phi$ *then* t_1 *else* t_2*)*$\in Term_A$ *for all variables* $x \in VSym$*, all formulae* $\phi \in Formulae(\Rightarrow$ *Def. 6.2.5) and any terms* $t_1 \in Term_{A_1}$, $t_2 \in Term_{A_2}$ *with* $A = A_1 \sqcup A_2$

Logical constants are treated as functions of arity zero.

Example 6.2.2. Continuing the previous examples we can write the term $+(x, a)$ using prefix form or as $a - 2$ in infix form. Please note that there is no meaning connected to the symbols. We just defined the syntax.

We turn our attention towards state updates of our logic which are described in detail in [Bec01]. An update is a postponed substitution on the formula or term it prefixes[2]. Updates are often caused by an assignment in the program.

Definition 6.2.3. *Given a signature* Σ *for a sort hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_s, \sqsubseteq)$ *the set of* updates *is the least set fulfilling:*

- *Functional update:* $(f(t_1, \ldots, t_n) := t) \in Updates$ *for all terms* $f(t_1, \ldots, t_n) \in Term_A$ *with* $f \in FSym_{nr}$ *and* $t \in Term_{A'}$ *such that* $A' \sqsubseteq A$
- *Parallel update:* $(u_1 \parallel u_2) \in Updates$ *for all* $u_1, u_2 \in Updates$
- *Quantified update:* $(\mathit{for}\ x; \phi; u) \in Updates$ *for all* $u \in Updates$*,* $x \in VSym$ *and* $\phi \in Formulae(\Rightarrow$ *Def. 6.2.5)*
- *Update application:* $(\{u_1\}u_2) \in Updates$ *for all* $u_1, u_2 \in Updates$

Note that in a functional update f is non rigid implying that its meaning can change (see Def. 6.3.10).

[2]Hoare logic [Hoa69] or classical dynamic logic [HKT00] work as a backwards calculus in resolving the last statement first. A possibly created substitution is applied on the post condition.

In contrast the logic of this chapter processes the first statement first, keeps updates and applies them together after no statements are left. The notion of updates provide another stage in the logic where they can be simplified before application.

Example 6.2.3. Building on the previous examples we can write $\{a := a + 2\}$ or $\{a := b \parallel b := a\}$. We observe that $a, b \in FSym_{nr}$. It will be of particular importance in the next section.

Before we define formulae we consider Creol programs because they can occur in formulae. They have to be restricted as we do not want to deal with incorrect programs like not compiling ones. We will assume that programs have unique identifiers. This is not a restriction on programs as one can easily come up with new names by adding numbers for example. Finally, we ensure that all classes and interfaces are typed correctly.

Definition 6.2.4. *Given a sort hierarchy $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_s, \sqsubseteq)$ and a corresponding signature Σ a* normalized Creol program *p is a set of class and interface declarations satisfying:*

- *p is compile correct*
- *identifiers are unique*
- *for all interfaces I: $I \in \mathcal{T}_s$ with $Null \sqsubseteq^0 I \sqsubseteq^0 Any$ and $I \sqsubseteq^0 I'$ for all interfaces I' it is inheriting*
- *for all classes C: $C \in \mathcal{T}_d$ with $Null \sqsubseteq^0 C \sqsubseteq^0 Top$, $C \sqsubseteq^0 I$ for all interfaces I it is implementing and $C \sqsubseteq^0 C'$ for all classes C' it is inheriting*

We denote the set of normalized Creol programs by Π.

Formulae express logical statements. Unlike in first-order logic, dynamic logic formulae can contain Creol statements in a modality. A modality can either be a box $[p]$ or a diamond $\langle p \rangle$[3]. The diamond and the box are used to express total correctness and partial correctness, respectively (see section 6.3).

Definition 6.2.5. *The set $Formulae$ of* formulae *is defined to be the least set which satisfies:*

- $true, false \in Formulae$
- $P(t_1, \ldots, t_n) \in Formulae$ *for all* $P \in PSym$ *and terms* $t_i \in Term_{A'_i}$ *with* $A'_i \sqsubseteq A_i$ $(1 \leq i \leq n)$
- $\neg\phi, (\phi \wedge \psi), (\phi \vee \psi), (\phi \rightarrow \psi) \in Formulae$ *for any* $\phi, \psi \in Formulae$
- $\forall x.\phi$, $\exists x.\phi$ *for any* $\phi \in Formulae$ *and any* $x \in VSym$
- $\{u\}\phi \in Formulae$ *for all* $\phi \in Formulae$ *and* $u \in Updates$

[3] Diamond and box are similar to Dijkstra's *wp* and *wlp* ([Dij75]), respectively. The Hoare triple $\{\phi\}p\{\psi\}$ ([Hoa69]) is similar to $\phi \rightarrow [p]\psi$.

- $\langle p\rangle\phi, [p]\phi \in Formulae$ *for all* $\phi \in Formulae$ *and any Creol program* $p \in \Pi$.

We will abbreviate the formula $(\phi \to \psi_1) \wedge (\neg\phi \to \psi_2)$ by $if\ \phi\ then\ \psi_1\ else\ \psi_2$.

Example 6.2.4. Resuming the recent examples we can write formulae like:

- $\langle \texttt{a := 2}\rangle(a \doteq 1 + 1)$
- $\{b := a\}true$
- $(\{b := 2\}b - 1 \doteq a) \to a \doteq 1$
- $true \vee \neg a \doteq a + 1$

Later, when the calculus is introduced ($\Rightarrow$ Def. 6.4), we will be interested in closed formulae as known from first-order logic. In a closed formula all variables are bound by a quantifier. A free variable is not bound by a quantifier.

Definition 6.2.6. *We define the set* $fv(t)$ *of* free variables *of a term* t *as*

- $fv(v) = \{v\}$ *for* $v \in VSym$
- $fv(f(t_1, \ldots, t_n)) = \bigcup_{i=1,\ldots,n} fv(t_i)$ *where* $f \in FSym$ *and* t_i *terms*
- $fv(if\ \phi\ then\ t_1\ else\ t_2) = fv(\phi) \cup fv(t_1) \cup fv(t_2)$
- $fv(\{u\}t) = fv(u) \cup fv(t)$
- $fv(ifExMin\ x.\phi\ then\ t_1\ else\ t_2) = ((fv(\phi) \cup fv(t_1))\backslash\{x\}) \cup fv(t_2)$

The set fv *for an update* u *is defined as follows*

- $fv(f(t_1, \ldots, t_n) := t) = fv(t) \cup \bigcup_{i=1}^{n} fv(t_i)$
- $fv(u_1 \mid\mid u_2) = fv(u_1) \cup fv(u_2)$
- $fv(for\ x; \phi; u) = (fv(\phi) \cup fv(u))\backslash\{x\}$

We extend fv *to formulae* ϕ,ψ *by*

- $fv(true) = fv(false) = \emptyset$
- $fv(P(t_1, \ldots, t_n)) = \bigcup_{i=1,\ldots,n} fv(t_i)$ *for* $P \in PSym$ *and* t_i *terms*
- $fv(\neg\phi) = fv(\phi)$
- $fv(\phi \wedge \psi) = fv(\phi \vee \psi) = fv(\phi \to \psi) = fv(\phi) \cup fv(\phi)$
- $fv(\forall x.\phi) = fv(\exists x.\phi) = fv(\phi)\backslash\{x\}$
- $fv(\{u\}\phi) = fv(u) \cup fv(\phi)$
- $fv(\langle p\rangle\phi) = fv([p]\phi) = fv(\phi)$

A formula ϕ *is called* closed *iff* $fv(\phi) = \emptyset$

6.3 Semantics

In this section we assign meaning to the formulae described in the previous section. It is divided in four parts namely the semantics of updates, terms, formulae, predicates, and functions which mutually depend on each other, but are divided for readability.

Predicates and functions

A first-order logic model determines the interpretation of functions and predicates. Though, the interpretation of a division by zero yields a problem: Which value should we assign to 1/0 ? The KeY system follows the underspecification approach characterized in [Häh05]. Instead of using a partial function the function is considered to be total (defined for all inputs), but the value it delivers for critical inputs is not known.

However, we define partial models which fix parts of the interpretation of predicates and functions. The reason is that we will freeze the interpretation of all rigid functions and all predicates by a partial model and leave the interpretation of non rigid functions to models refining the partial model. In Definition 6.3.3 we will show that a model is a special case of a partial model.

Definition 6.3.1. *Given a sort hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_s, \sqsubseteq)$ *and a signature* Σ *a* partial model *is a quintuple* $\mathcal{M} = (\mathcal{T}_0, \mathcal{D}_0, \delta_0, D_0, \mathcal{I}_0)$ *consisting of a set of sorts* $\mathcal{T}_0 \subseteq \mathcal{T}_d$*, a set* $\mathcal{D}_0$ *called the partial domain, a sort function* $\delta : \mathcal{D}_0 \to \mathcal{T}_0$*, a fixing function* D_0 *and a partial interpretation* $\mathcal{I}_0$*, where:*

- $\forall A \in \mathcal{T} : \mathcal{D}_0^A := \{d \in \mathcal{D}_0 | \delta(d) \sqsubseteq A\} \neq \emptyset$
- $\forall f : A_1, \ldots, A_n \to A_0 \in FSym$ *fulfilling for all* i $A_i \in \mathcal{T}_0$*:*
  D_0 *yields a set of tuples of domain elements* $D_0(f) : \mathcal{D}_0^{A_1} \times \ldots \times \mathcal{D}_0^{A_n}$
  and $\mathcal{I}$ *yields a function* $\mathcal{I}(f) : D_0(f) \to \mathcal{D}_0^{A_0}$
- $\forall f : A_1, \ldots, A_n \to A_0 \in FSym$ *where* $\exists i$ $A_i \notin \mathcal{T}_0$*:* $D_0(f) = \emptyset$
- $\forall P : A_1, \ldots, A_n \in PSym$ *fulfilling for all* i $A_i \in \mathcal{T}_0$*:*
  D_0 *yields a set of tuples of domain elements* $D_0(f) : \mathcal{D}_0^{A_1} \times \ldots \times \mathcal{D}_0^{A_n}$
  and $\mathcal{I}$ *yields a subset* $\mathcal{I}_0(P) \subseteq D_0(P)$
- $\forall P : A_1, \ldots, A_n \in PSym$ *where* $\exists i$ $A_i \notin \mathcal{T}_0$*:* $D_0(P) = \emptyset$

This complex definition requires some explanations. A term is interpreted in a domain. $\mathcal{D}_0$ represents the part of the domain which is fixed. δ assigns a sort to each element of the fixed domain. D_0 determines the parameters for which the interpretation for functions or predicates are known. The parameters of a function are related to the output of the function by $\mathcal{I}$. The intended meaning of a predicate is $P(\bar{x})$ is true if $\bar{x} \in \mathcal{I}(P)$ (see Def. 6.3.12).

Example 6.3.1. We proceed with our series of examples by defining a partial model for our sort hierarchy

$$(\{Top, Int, intDom, Bottom\}, \{Bottom, intDom\}, \{Int, Top\}, \sqsubseteq)$$

and a signature $\Sigma = (\{x\}, \{1, 2, 3, /\}, \{a, b\}, \{\doteq\}, \alpha)$:

$$\mathcal{M} = (\{Bottom, intDom\}, \mathbb{Z}, \delta_0, D_0, \mathcal{I}_0)$$

where

- $\mathcal{D}_0^{intDom} = \mathbb{Z}$
- $D_0(/) = \mathbb{Z} \times \mathbb{Z} \backslash \{0\}$
- $D_0(\doteq) = \mathbb{Z} \times \mathbb{Z}$
- $\mathcal{I}_0(1) = 1$, $\mathcal{I}_0(2) = 2$, $\mathcal{I}_0(3) = 3$
- $\mathcal{I}_0(/)(x, y) = z$ such that $0 \leq x - y * z < |y|$

We note that the interpretation for the division is unknown if $y \neq 0$.

As a partial model leaves out some fragment of the interpretation we can imagine another partial model which fixes more parts of the interpretation.

Definition 6.3.2. *A partial model* $\mathcal{M}_1 = (\mathcal{T}_1, \mathcal{D}_1, \delta_1, D_1, \mathcal{I}_1)$ refines *another partial model* $\mathcal{M}_0 = (\mathcal{T}_0, \mathcal{D}_0, \delta_0, D_0, \mathcal{I}_0)$*, if*

- $\mathcal{T}_1 \supseteq \mathcal{T}_0$
- $\mathcal{D}_1 \supseteq \mathcal{D}_0$
- $\forall d \in \mathcal{D}_0 : \delta_1(d) = \delta_0(d)$
- $\forall f \in FSym : D_1(f) \supseteq D_0(f)$
- $\forall f \in FSym : \forall (d_1, \ldots, d_n) \in D_0(f) : \mathcal{I}_1(d_1, \ldots, d_n) = \mathcal{I}_0(d_1, \ldots, d_n)$
- $\forall P \in PSym : D_1(P) \supseteq D_0(P)$
- $\forall P \in PSym : \mathcal{I}_1(P) \cap D_0(P) = \mathcal{I}_0(P)$

A partial model refines another model if it strengthens the constraints on the interpretation and leaves the interpretation of the original model untouched.

Example 6.3.2. Continuing the previous example we extend $D_0(/)$ to $D_1(/) = \mathbb{Z} \times \mathbb{Z}$ and its interpretation to

$$\mathcal{I}_1(/)(x, y) = \begin{cases} z \text{ such that } 0 \leq x - y * z < |y| & \text{if } y \neq 0 \\ \text{some arbitrary but fixed } d \in \mathcal{D}^{Int} & \text{otherwise} \end{cases}$$

Now, terms with a division through zero are interpreted by an unknown value which is the standard approach of KeY.

The notion of models usually used in first-order logic are a special case of partial models.

Definition 6.3.3. *A partial model* $\mathcal{M}_0 = (\mathcal{T}_0, \mathcal{D}_0, \delta_0, D_0, \mathcal{I}_0)$ *is a* model *iff*

- $\forall f : A_1, \ldots, A_n \to A_0 \in FSym$*:* $D_0(f) = \mathcal{D}_0^{A_1} \times \ldots \times \mathcal{D}_0^{A_n}$
- $\forall P : A_1, \ldots, A_n \in PSym$*:* $D_0(P) = \mathcal{D}_0^{A_1} \times \ldots \times \mathcal{D}_0^{A_n}$

Intuitively a model fixes the complete interpretation of functions and predicates. Example 6.3.1 was a partial model not being a model whereas example 6.3.2 is both.

States

The execution of an statement like a := 1 can be seen as an state change such that it holds $a \doteq 1$ in the new state. However, some interpretations should never change, e.g. of the plus operator. To express this in logic we define Kripke structures which relate such states to each other through programs fixing the interpretation of all non program variables.

Definition 6.3.4. *Let a sort hierarchy* $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_s, \sqsubseteq)$ *for a signature* Σ *be given. A Kripke structure is a tuple* $\mathcal{K} = (\mathcal{M}, \mathcal{S}, \rho)$ *consisting of a partial model* $\mathcal{M} = (\mathcal{T}_0, \mathcal{D}_0, \delta_0, D_0, \mathcal{I}_0)$*, a set* $\mathcal{S}$ *of states and a program relation* ρ*, requiring that*

- $\mathcal{D}_0$ *is a partial ordered set with* $\forall \mathcal{D}_{sub} \subseteq \mathcal{D}_0$*:* $\mathcal{D}_{sub}$ *has a minimum*
- $\mathcal{T}_0 = \mathcal{T}$
- $\forall f : A_1, \ldots, A_n \to A \in FSym$
  $D_0(f) = \begin{cases} \emptyset & f \in FSym_{nr} \\ \mathcal{D}_0^{A_1} \times \ldots \times \mathcal{D}_0^{A_n} & f \in FSym_r \end{cases}$
- $\forall P : A_1, \ldots, A_n \in PSym$*:* $D_0(P) = \mathcal{D}_0^{A_1} \times \ldots \times \mathcal{D}_0^{A_n}$
- $\mathcal{S}$ *contains all models refining* $\mathcal{M}$
- $\forall S_1, S_2 \in \mathcal{S}$ *and* $p \in \Pi$ *started in* S_1 *terminates in* S_2*:* $(S_1, p, S_2) \in \rho$

We will abbreviate the tuples of the models $S \in \mathcal{S}$ as $S = (\mathcal{D}, \delta, \mathcal{I})$ since the sorts are fixed and D is unnecessary for models. Note that the interpretation of all rigid symbols is fixed in $\mathcal{M}$ and therefore the same in all states.

Again some special requirements are necessary to reason about Creol programs which are expressed in the following definition.

Definition 6.3.5. *For a given sort hierarchy and a corresponding signature a* CreolDL Kripke structure *is a Kripke structure* $\mathcal{K} = ((\mathcal{T}_0, \mathcal{D}_0, \delta_0, D_0, \mathcal{I}_0), \mathcal{S}, \rho)$ *where*

- $\mathbb{Z} = \mathcal{D}_0^{intDom}$
- $\{tt, ff\} = \mathcal{D}_0^{boolDom}$
- $\{null\} = \mathcal{D}_0^{Null}$
- *and the other domains of Appendix B.8*
- *The order* $\preceq$ *of* $\mathcal{D}_0$ *is given by:*
  - *if* $\delta_0(x) \neq \delta_0(y)$ *then* $\begin{cases} x \preceq y & \text{if } \delta_0(x) \sqsubseteq \delta_0(y) \\ y \preceq x & \text{if } \delta_0(y) \sqsubseteq \delta_0(x) \\ x \preceq y & \text{if } \delta_0(x) \leq_{lex} \delta_0(y) \text{ and neither} \\ & \delta_0(x) \sqsubseteq \delta_0(y) \text{ nor } \delta_0(y) \sqsubseteq \delta_0(x) \end{cases}$
    where $\leq_{lex}$ *is the lexicographic order*
  - *if* $\delta_0(x) = \delta_0(y)$ *then*
    - *if* $\delta_0(x) = booleanDomain$ *then* $x \preceq y$ *iff* $x = ff$
    - *if* $\delta_0(x) = integerDomain$ *then* $x \preceq y$ *iff*
      $x \geq 0$ *and* $y < 0$ *or*
      $x \geq 0$ *and* $y \geq 0$ *and* $x \leq y$, *or*
      $x < 0$ *and* $y < 0$ *and* $x \geq y$
    - *if* $A = \delta_0(x) \sqsubseteq^0 Top$ *then* $x \preceq y$ *iff* $index_A(x) \preceq index_A(y)$
      where $index_T : \mathcal{D}^T \to \mathbb{Z}$ *is an arbitrary but fixed bijective mapping for* $T \in \mathcal{T}_0$
    - *if* $\delta_0(x) = Null$ *then* $x = y$

The *intDom* sort are the mathematical integers $\mathbb{Z}$ as one would expect. *boolDom* has just two elements and $Null$ one. The remaining domains of a CreolDL Kripke structure will be explained in chapter 7. The rather complicated definition of the partial order on $\mathcal{D}_0$ is necessary to resolve clashing updates uniquely (see Def. 6.3.11). All Kripke structures to be used in the following text are considered to be CreolDL Kripke structures.

Terms

Parts of the meaning of terms are already fixed, namely functions, by the model of the given state. However, variables were not mentioned. Hence we establish a variable assignment.

Definition 6.3.6. *Given a partial model* $\mathcal{M}$ *a* variable assignment *is a function* $\beta : VSym \to \mathcal{D}$ *fulfilling* $\beta(x) \in \mathcal{D}^{\alpha(x)}$ *for all* $x \in VSym$. *The modification* β_x^d *of a variable assignment* β *for any variable* $x \in VSym$ *and any domain element* $d \in \mathcal{D}^{\alpha(x)}$ *is*

$$\beta_x^d(y) := \begin{cases} d & if\ x = y \\ \beta(y) & otherwise \end{cases}$$

The variable assignment provides a domain element for each variable.

Example 6.3.3. Pursuing the previous examples we can assign an value to x by writing $\beta(x) = 17$. If we want to change the value of x to 0 we express it as $\beta^0_x(x) = 0$.

The interpretation of the model of the current state, the variable assignment, the semantics of formulae and updates are linked together to specify the meaning of terms.

Definition 6.3.7. *Given a signature for a sort hierarchy for a Kripke structure $\mathcal{K} = (\mathcal{M}, \mathcal{S}, \rho)$ and a variable assignment β for any state $S \in \mathcal{S}$ the* valuation function *val_S for terms is inductively defined as:*

- $val_{S,\beta}(x) = \beta(x)$ *for any* $x \in VSym$
- $val_{S,\beta}(f(t_1, \ldots, t_n)) = \mathcal{I}(f)(val_{S,\beta}(t_1), \ldots, val_{S,\beta}(t_n))$ *for any* $f \in FSym$ *and terms* $t_1, \ldots, t_n$
- $val_{S,\beta}(\mathit{if}\, \phi\, \mathit{then}\, t_1\, \mathit{else}\, t_2) = \begin{cases} val_{S,\beta}(t_1) & \text{if } S, \beta \models \phi \ (\Rightarrow \text{Def. 6.3.12}) \\ val_{S,\beta}(t_2) & \text{if } S, \beta \not\models \phi \ (\Rightarrow \text{Def. 6.3.12}) \end{cases}$ *for all formulae* ϕ *and terms* t_1,t_2
- $val_{S,\beta}(\{u\}t) = val_{S_1,\beta}(t)$ *with* $S_1 = val_{S,\beta}(u)(S)$ *(see Def. 6.3.11) for any update* u *and any term* t
- $val_{S,\beta}(ifExMin\ x\ \phi\ then\ t_1\ else\ t_2) =$
  $\begin{cases} val_{S,\beta^d_x}(t_1) & \exists d \in \mathcal{D}^{\alpha(d)} \text{ such that } S, \beta^d_x \models \phi (\Rightarrow Def.6.3.12) \text{ and} \\ & \forall d' \in \mathcal{D}^{\alpha(d)} \text{ with } S, \beta^{d'}_x \models \phi : d \preceq d' \\ val_{S,\beta^d_x}(t_2) & otherwise \end{cases}$

Example 6.3.4. Given the signature and the model of example 6.3.2 and the variable assignment of the previous example we can evaluate the term $1 + x$:

$val_{S,\beta}(1 + x) = \mathcal{I}(+)(val_{S,\beta}(1), val_{S,\beta}(x)) = \mathcal{I}(f)(1, 0) = 1 + 0 = 1$

Updates

To capture the semantics of updates we will need four definitions. An update will only change the interpretation $\mathcal{I}$ of the model of the current state. First, we define a semantic update to capture the semantics of a functional update.

Definition 6.3.8. *Given a signature Σ for a sort hierarchy $(\mathcal{T}, \mathcal{T}_d, \mathcal{T}_s, \sqsubseteq)$ a* semantic update *is a triple $(f, (d_1, \ldots, d_n), d_0)$ such that*
$f : A_1, \ldots, A_n \to A_0 \in FSym_{nr}$, $d_i \in \mathcal{D}^{A_i}$ $(0 \leq i \leq n)$

Once again, please note that the function f is non-rigid so its semantics are modifiable.

Example 6.3.5. The update $\{a := 2\}$ is represented as a semantic update as $(a, (), 2)$.

A problem considering updates are clashes, namely we have two updates changing the interpretation of the same function in different ways. However, we introduce a set which does not contain such clashes and show a transformation of all possible updates to this set later.

Definition 6.3.9. *A set CU of semantic updates is called* consistent *if* $\forall (f, (d_1, \ldots, d_n), d), (f', (d'_1, \ldots, d'_m), d') \in UC$*:*
$f = f' \wedge n = m \wedge \forall i \in \{1, \ldots, n\} : d_i = d'_i \rightarrow d = d'$

From now on we write $\mathcal{CU}$ for the set of all sets of consistent semantic updates.

Example 6.3.6. The set $\{(a, (), 2), (a, (), 3)\}$ of semantic updates is not consistent since it updates the location a with two different values.

A set of consistent updates does not contain clashes by definition so we can specify the modification of the model.

Definition 6.3.10. *Given a sort hierarchy, a signature, a Kripke structure and for a model* $S = (\mathcal{D}, \delta, \mathcal{I}) \in \mathcal{S}$*. For any set* $CU \in \mathcal{CU}$ *of consistent semantic updates, the* modification $CU(S) = (\mathcal{D}', \delta', \mathcal{I}')$ *where*

- $\mathcal{D} = \mathcal{D}'$
- $\delta = \delta'$
- $\mathcal{I}'(f)(d_1, \ldots, d_n) = \begin{cases} d & \text{if } (f, (d_1, \ldots, d_n), d) \in CU \\ \mathcal{I}(f)(d_1, \ldots, d_n) & \text{otherwise} \end{cases}$

Intuitively a set of semantic updates will change the interpretation of the functions in the set.

Now, we are ready to link (syntactical) updates with semantical updates by a rather complicated function. The possibly occurring clashes are resolved. In parallel updates the last update for a function wins, e.g. $\{a := 1 \parallel a := 2\}$ is equivalent to $\{a := 2\}$. For quantified updates the update assigning the element being the least with respect to $\preceq$ will be applied. This is the reason for the existence of the $ifExMin$ term (see Def. 6.2.2).

Definition 6.3.11. *Let a sort hierarchy, a signature, a Kripke* $\mathcal{K} = (\mathcal{M}, \mathcal{S}, \rho)$ *and a variable assignment* β *be given. For every state* $S \in \mathcal{S}$ *the* valuation function $val_{S,\beta} : Updates \rightarrow \mathcal{CU}$ *is inductively defined by*

- $val_{S,\beta}(f(t_1, \ldots, t_n) := s) = \{(f, (d_1, \ldots, d_n), d)\}$ *where* $d = val_{S,\beta}(s)$ *and* $d_i = val_{S,\beta}(t_i)$ $(1 \leq i \leq n)$

- $val_{S,\beta}(u_1 \mid\mid u_2) = (U_1 \backslash C) \cup U_2$ *where* $U_1 = val_{S,\beta}(u_1)$, $U_2 = val_{S,\beta}(u_2)$ *and* $C = \left\{ (f,(d_1,\ldots,d_n),d') \;\middle|\; \begin{array}{l} (f,(d_1,\ldots,d_n),d') \in U_1 \text{ and} \\ (f,(d_1,\ldots,d_n),d) \in U_2 \text{ with } d \neq d' \end{array} \right\}$
- $val_{S,\beta}(for\ x; \phi; u) = U$ *where*
  $U = \left\{ (f,(d_1,\ldots,d_n),d) \;\middle|\; \begin{array}{l} \text{there is } a \in \mathcal{D}^{\alpha(x)} \text{ such that} \\ ((f,(d_1,\ldots,d_n),d),a) \in dom \text{ and } b \not\preceq a \\ \text{for all } ((f,(d_1,\ldots,d_n),d'),b) \in dom \end{array} \right\}$
  with $dom = \bigcup_{a \in \{d \in \mathcal{D}^{\alpha(x)} \mid S,\beta^a_x \models \phi\}} (val_{S,\beta}(u) \times \{a\})$
- $val_{S,\beta}(\{u_1\}u_2) = val_{S',\beta}(u_2)$ *where* $S' = val_{S,\beta}(u_1)(S)$.

Example 6.3.7. Let us resolve the clash in the parallel update mentioned above: $\{a := 1 \mid\mid a := 2\}$. In this case $a := 1$ yields the set $U_1 = \{(a,(),1)\}$ and $a := 2$ yields $U_2 = \{(a,(),2)\}$. The complicated set C contains the first update: $C = U_1 = \{(a,(),1)\}$. Thus $val_{S,\beta}(a := 1 \mid\mid a := 2) = val_{S,\beta}(a := 2)$.

Formulae

The only thing left for this section is the meaning of formulae. The notable lines in the following definition are the modalities which are not available in first-order logic.

Definition 6.3.12. *Given a sort hierarchy, a signature, a Kripke structure* $\mathcal{K} = (\mathcal{M}, \mathcal{S}, \rho)$ *and a variable assignment* β *we define the* validity relation $\models$ *for every state* $S = (\mathcal{T}, \mathcal{D}, \delta, D, \mathcal{I}) \in \mathcal{S}$*:*

- $S, \beta \models true$
- $S, \beta \not\models false$
- $S, \beta \models P(t_1,\ldots,t_n)$ *iff* $(val_{S,\beta}(t_1),\ldots,val_{S,\beta}(t_n)) \in \mathcal{I}(P)$
- $S, \beta \models \neg\phi$ *iff* $S, \beta \not\models \phi$
- $S, \beta \models \phi \wedge \psi$ *iff* $S, \beta \models \phi$ *and* $S, \beta \models \psi$
- $S, \beta \models \phi \vee \psi$ *iff* $S, \beta \models \phi$ *or* $S, \beta \models \psi$ *(or both)*
- $S, \beta \models \phi \rightarrow \psi$ *iff* $S, \beta \not\models \phi$ *or* $S, \beta \models \psi$ *(or both)*
- $S, \beta \models \exists x\ \phi$ *iff* $S, \beta^d_x \models \phi$ *for at least one* $d \in \mathcal{D}^{\alpha(x)}$
- $S, \beta \models \forall x\ \phi$ *iff* $S, \beta^d_x \models \phi$ *for all* $d \in \mathcal{D}^{\alpha(x)}$
- $S, \beta \models \{u\}\phi$ *iff* $S_1, \beta \models \phi$ *with* $S_1 = val_{S,\beta}(u)(S)$
- $S, \beta \models \langle p \rangle \phi$ *iff there exists at least one state* $S \in \mathcal{S}$ *with* $(S, p, S') \in \rho$ *and* $S', \beta \models \phi$

- $S, \beta \models [p]\phi$ *iff for all states* $S \in \mathcal{S}$ *with* $(S, p, S') \in \rho$ *and* $S', \beta \models \phi$

where ϕ, ψ *are formulae,* x *is a variable,* u *is an update and* $t_1, \ldots, t_n$ *are terms.*

Example 7.2.4 uses the above definition.

Validity

Using the semantics of formulae we can establish the logical satisfiability and validity similar to first-order logic.

Definition 6.3.13. *Given a signature for a sort hierarchy, a formula* $\phi \in$ *Formulae and a normalized program* $p \in \Pi$*:*

For a Kripke structure $\mathcal{K} = (\mathcal{M}, \mathcal{S}, \rho)$*:* ϕ *is* satisfiable *if there is some state* $S \in \mathcal{S}$ *and some variable assignment* β *such that* $S, \beta \models \phi$

Given a Kripke structure $\mathcal{K} = (\mathcal{M}, \mathcal{S}, \rho)$*:* ϕ *is* $\mathcal{K}$-valid *if for all states* $S \in \mathcal{S}$ *and for all variable assignments* β*:* $S, \beta \models \phi$

ϕ *is* logically valid *if* ϕ *is* $\mathcal{K}$*-valid for all Kripke structures* $\mathcal{K}$

Partial and total correctness

When verifying programs there are two different notions of correctness, namely partial and total correctness.

Definition 6.3.14. *For a sort hierarchy for a signature for a Kripke structure* $\mathcal{K}$ *the triple* $(\phi, p, \psi) \in$ *Formulae* $\times \Pi \times$ *Formulae is called*

- partially correct *if* $\phi \rightarrow [p]\psi$ *is* $\mathcal{K}$*-valid.*
- totally correct *if* $\phi \rightarrow \langle p \rangle \psi$ *is* $\mathcal{K}$*-valid.*

The definition of the box modality does not require a final state to exist. Therefore non terminating programs are partially correct.

Example 6.3.8. To show that a program p terminates we use the tuple $(true, p, true)$ which corresponds to the formula $true \rightarrow \langle p \rangle true$.

6.4 Sequent calculus

The section introduces a calculus consisting of rules to formalize calculating with the logic. Sequent calculi were first used by Gentzen in [Gen35].

Definition 6.4.1. *A* sequent *is pair of sets of closed formulae*

$$\phi_1, \ldots, \phi_n \Longrightarrow \psi_1, \ldots, \psi_m$$

To simplify we group some formulas in $\Delta := \phi_1, \ldots, \phi_j$ and $\Gamma := \psi_1, \ldots, \psi_k$. The sequent above could be rewritten to $\Delta, \phi \Longrightarrow \psi, \Gamma$ for instance. The intended meaning of a sequent is $(\phi_1 \wedge \ldots \wedge \phi_n) \rightarrow (\psi_1 \vee \ldots \vee \psi_m)$. To express the logical validity of a formula ϕ we write $\Longrightarrow \phi$.

We need a standardized notion to express rules changing sequents. A sequent rule transforms one or more sequents in a logically sound way.

Definition 6.4.2. *A* sequent rule *contains $n \in \mathbb{N}_0$ sequents as premises and one as conclusion.*

$$\frac{\overbrace{\Gamma_1 \Longrightarrow \Delta_1 \ \ldots \ \Gamma_n \Longrightarrow \Delta_n}^{premises}}{\underbrace{\Gamma \Longrightarrow \Delta}_{conclusion}}$$

To reason about soundness one shows: If all premises are valid the conclusion is valid. The application of a rule is the other way round: In order to prove a goal matching the conclusion prove its premises. Let us have a look at four rules (non-exhaustive) in propositional logic:

$$\mathsf{implicationRight}\ \frac{\Gamma, \psi \Longrightarrow \phi, \Delta}{\Gamma \Longrightarrow \psi \rightarrow \phi, \Delta} \qquad \mathsf{closeGoal}\ \frac{}{\Gamma, \phi \Longrightarrow \phi, \Delta}$$

$$\mathsf{andRight}\ \frac{\Gamma \Longrightarrow \phi, \Delta \qquad \Gamma \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \wedge \psi, \Delta} \qquad \mathsf{closeTrue}\ \frac{}{\Gamma \Longrightarrow \mathit{true}, \Delta}$$

The *implicationRight* rule moves the bottom ψ to the left side of the sequent arrow $\Longrightarrow$ as the intended meaning of a sequence proposes it. By identifying the tautology $\phi \rightarrow \phi$ the *closeGoal* rule closes a goal. *andRight* is valid as if the premises are given the conclusion must hold. If *true* occurs on the right side of the sequent arrow the right side turns true which makes the implicit implication of the sequent arrow logically valid.

Pure first-order logical sequent rules like *closeGoal* and *implicationRight* or rules handling the typed logic are not subject to this thesis. They can be looked up in chapter 2 of [BHS07]. We will present the Creol specific rules in the following sections of this chapter.

We define a proof consisting of many applied rules.

Definition 6.4.3. *A* proof *of a formula ϕ is a tree (⇒ Def.2.2.4) where each node is annotated by a sequent. The root is annotated by $\Longrightarrow \phi$. Every other node is additionally annotated by a sequent rule relating the conclusion of its sequent with its parent and the premises with its descendants.*

Example 6.4.1. For $P \in PSym$ the formula $P \rightarrow P$ is logically valid since $P \rightarrow P = \neg P \vee P = \mathit{true}$. Thus we should be able to prove it with our sequent rules. Our proof obligation is:

$$\Longrightarrow P \rightarrow P$$

The direction of application of sequent rules is from bottom to top:

$$\dfrac{\dfrac{\text{closed}}{P \Longrightarrow P}}{\Longrightarrow P \to P}$$

where we applied first the *implicationRight* rule and afterwards *closeGoal*.

Completeness and Soundness

A calculus is called complete if all valid statements are derivable. Soundness expresses the property that all derivable statements are valid. Because of Gödels Incompleteness theorem [Göd31] it is impossible to create a first-order calculus with arithmetic which is both complete and sound. Therefore we focus on sound calculus. However, it was shown in [Pla04] for a related calculus, ODL [BP06], which captures the essence of CreolDL that it is relative complete. A relative complete calculus does not add any new incompleteness in comparison to a first-order calculus with arithmetic. Relative completeness was defined in [Coo78].

Notation

In the following sections we will write U for a possibly empty sequence of updates. p, q will be normalized Creol programs out of Π. A possibly empty sequence of following statements will be denoted by ; ω. To improve readability a general rule of the form:

$$\dfrac{\begin{array}{l}\Gamma, U\phi_1^1, \ldots, U\phi_{m_1}^1 \Longrightarrow U\psi_1^1, \ldots, U\psi_{n_1}^1, \Delta \\ \vdots \\ \Gamma, U\phi_1^k, \ldots, U\phi_{m_k}^k \Longrightarrow U\psi_1^k, \ldots, U\psi_{n_k}^k, \Delta\end{array}}{\Gamma, U\phi_1, \ldots, U\psi_m \Longrightarrow U\psi_1, \ldots, U\psi_n, \Delta}$$

will be abbreviated by:

$$\dfrac{\begin{array}{l}\phi_1^1, \ldots, \phi_{m_1}^1 \Longrightarrow \psi_1^1, \ldots, \psi_{n_1}^1 \\ \vdots \\ \phi_1^k, \ldots, \phi_{m_k}^k \Longrightarrow \psi_1^k, \ldots, \psi_{n_k}^k\end{array}}{\phi_1, \ldots, \psi_m \Longrightarrow \psi_1, \ldots, \psi_n}$$

If the context U,Δ,Γ is of importance it will be mentioned and the rule will be equipped with its context.

There are (rewriting) rules where even the context of the sequent arrow is unnecessary since they can be applied in any context. In this case we will only write:

$$\dfrac{\phi_1', \ldots, \phi_m'}{\phi_1, \ldots, \phi_n}$$

Modalities

Rules not known from first-order calculi address modalities. The simplest case is that the modality is empty. During a proof involving modalities we will be in this situation at least one time, namely when all statements of the modality have been processed (e.g. converted into updates). If the modality is empty it does not relate two different states of the Kripke structure anymore, so we remove it.

$$\frac{\Longrightarrow \phi}{\Longrightarrow \langle\rangle\phi} \qquad \frac{\Longrightarrow \phi}{\Longrightarrow []\phi}$$

Chapter 7

Reasoning about Creol

In this chapter the dynamic logic of the previous chapter is extended to reason about Creol. We begin with the rules addressing sequential Creol without method calls in section 7.1. Thereafter the theory for reasoning about concurrency and method calls is described in section 7.2. How the knowledge of these two sections is used to reason about a complete Creol program is specified in section 7.3. Finally the chapter is completed by section 7.4 which identifies limitations and further work.

7.1 Sequential calculus

We begin with control flow statements in the first paragraph and in the second part of this section we cover assignments and expressions.

Statements

Being familiar with rules handling modalities in general we can fill the modalities with Creol statements. To reach the point where the modality is empty from which we can proceed in the well understood first-order reasoning we must be able to treat all Creol statements.

To start off easy we will look at the *skip* statement. The skip statement has no effect by definition. What do we with it then? We remove it:

$$\frac{\Longrightarrow \langle \omega \rangle \phi}{\Longrightarrow \langle \texttt{skip; } \omega \rangle \phi}$$

Example 7.1.1. So let us use this rule in our first prove about the simplest Creol program:

```
1 skip
```

We want to show that after executing *skip* the program does terminate. Thus we have to use the diamond as it expresses total correctness:

$$\Longrightarrow \langle \texttt{skip} \rangle true$$

A proof using a sequent calculus is written from bottom to top because the rules are applied in this manner:

$$\dfrac{\dfrac{\dfrac{\text{closed}}{\Longrightarrow true}}{\Longrightarrow \langle\rangle true}}{\Longrightarrow \langle \texttt{skip} \rangle true}$$

First, we apply the just introduced rule for the *skip* statement. Then we have an empty modality which is removed by the corresponding rule introduced in section 6.4. What is left is a sequent arrow with *true* on the right side which closes our proof.

We were using the diamond expressing total correctness. To reason about partial correctness of programs containing the *skip* statement, we would have to compose exactly the same rule using the box. We will avoid this unnecessary overhead by writing the diamond iff it holds for partial and total correctness and the box if the rule is only valid under partial correctness. The only exception is the *block* statement (see section 3), where the rules hold only for the used modalities:

$$\dfrac{\Longrightarrow true}{\Longrightarrow [\texttt{block; } \omega]\phi} \qquad \dfrac{\Longrightarrow false}{\Longrightarrow \langle \texttt{block; } \omega \rangle \phi}$$

It is impossible to reach the next state after the *block* statement. For the box this means that the formula ϕ is not required to hold. The diamond guarantees the existence of a successor state which contradicts the *block* statement. So it is replaced by false.

The block statement was a special one so let us turn towards more usual ones. The concept of control flow statements like *if-then-else* and *while* (or any other loop) are wide spread in programming languages. We begin with *if-then-else*. Intuitively we should try to find out whether the condition of the *if* statement is true or false and then we symbolically execute only the remaining branch:

$$\dfrac{\Longrightarrow if\ texp \doteq TRUE\ then\ \langle p\texttt{; } \omega \rangle \phi\ else\ \langle q\texttt{; } \omega \rangle \phi}{\Longrightarrow \langle \texttt{if } texp \texttt{ then } p \texttt{ else } q \texttt{ end; } \omega \rangle \phi}$$

where *texp* is a variable or a literal. Please note that $TRUE$ is a term and together with *texp* it is a parameter of the predicate $\doteq$. If we cannot deduce whether *texp* is interpreted as $TRUE$ or $FALSE$ the calculus will split the logical *if* construct into two branches.

Example 7.1.2. Given the program which terminates because the condition of the if statement is fulfilled:

```
1 if true
2   then skip
3   else block
4 end
```

We can prove that it actually terminates by writing:

$$\langle \texttt{if true then skip else block end} \rangle true$$

We calculate using the *if-then-else* introduced rule:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{closed}}{\Longrightarrow true}}{\Longrightarrow \langle\rangle true}}{\Longrightarrow \langle \texttt{skip} \rangle true}}{\Longrightarrow if\ true\ then\ \langle \texttt{skip} \rangle true\ else\ \langle \texttt{block} \rangle true}}{\Longrightarrow if\ TRUE \doteq TRUE\ then\ \langle \texttt{skip} \rangle true\ else\ \langle \texttt{block} \rangle true}}{\Longrightarrow \langle \texttt{if true then skip else block end} \rangle true}$$

At first the *if-then-else* rule is applied which translates the code to a logical formula containing both branches as modalities. Thereafter the equality predicate $\doteq$ evaluates to true as both arguments are the same. Subsequently the logical *if* is substituted by its *then* branch what brings us in the same situation as in the previous example.

Why did we require that *texp* is a variable or a literal and did not allow all Boolean expressions? There is a pitfall with expressions in Creol. As mentioned in section 3 a division by zero leads to a blocking behavior. Because a division by zero can occur in a Boolean expression, e.g. $\frac{2}{0} > 1$, we have to unfold all expressions[1]. Applying this to the *if-then-else* statement we create a new Boolean variable v and assign the expression *exp* to it:

$$\dfrac{\Longrightarrow \langle v\ \texttt{:=}\ exp\ \texttt{;}\ \texttt{if}\ v\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end;}\ \omega \rangle \phi}{\Longrightarrow \langle \texttt{if}\ exp\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end;}\ \omega \rangle \phi}$$

where *exp* matches all expressions, but literals and variables. So considering matchings $texp \cap exp = \emptyset$. For the newly created v := exp the rules for assignments discussed in the next paragraph will unfold the expression even

[1] if we reasoned only about partial correctness the unfolding would be unnecessary. From this point of view the given calculus is incomplete. Using a transformation function from Creol functions (e.g. +) to logical functions the development of the corresponding partial correctness rules should be straightforward.

more to look for divisions by zero. Example 7.1.3 contains an instance of such an unfolding process where the matchings of *texp* and *exp* are explicitly mentioned.

The given sequents do not match a *if-then* statement having no else branch. The reason is the lack of a optional statement of the taclet language used (see section 8.1). Thus we write the two analogous rules by just leaving out the *else* branch.

$$\frac{\Longrightarrow if\ texp \doteq TRUE\ then\ \langle p;\ \omega\rangle\phi\ else\ \langle\omega\rangle\phi}{\Longrightarrow \langle \texttt{if}\ texp\ \texttt{then}\ p\ \texttt{end;}\ \omega\rangle\phi}$$

$$\frac{\Longrightarrow \langle v\ \texttt{:=}\ exp\ \texttt{;}\ \texttt{if}\ v\ \texttt{then}\ p\ \texttt{end;}\ \omega\rangle\phi}{\Longrightarrow \langle \texttt{if}\ exp\ \texttt{then}\ p\ \texttt{end;}\ \omega\rangle\phi}$$

We proceed with the analysis of *while* statements. The easiest, but incomplete technique, to handle the *while* statement is unrolling the loop. Thereby we transform the *while* loop by checking the condition once by a *if* statement,and executing the body once. Afterwards the *while* statement is executed leading to a equivalent program transformation:

$$\frac{\Longrightarrow \langle \texttt{if}\ texp\ \texttt{then}\ p\ \texttt{;}\ \texttt{while}\ texp\ \texttt{do}\ p\ \texttt{end end;}\ \omega\rangle\phi}{\Longrightarrow \langle \texttt{while}\ texp\ \texttt{do}\ p\ \texttt{end;}\ \omega\rangle\phi}$$

$$\frac{\Longrightarrow \langle \texttt{v :=}\ exp\texttt{; while}\ v\ \texttt{do}\ p\texttt{; v:=}exp\ \texttt{end;}\ \omega\rangle\phi}{\Longrightarrow \langle \texttt{while}\ exp\ \texttt{do}\ p\ \texttt{end;}\ \omega\rangle\phi}$$

These rules do not cover the case of unbounded loops such as *while true* because this rule would be applicable infinitely often without resolving the loop. To cope with such loops we can use loop invariants. In the context of Creol loop invariants have to be treated with caution because of the method calls the loop might contain. Hence loop invariants will be explained in section 7.2.

A typical Creol statement is the box or non deterministic choice between two branches (see section 3). Only one of the branches is executed. But as we do not know in general which one, we have to symbolically execute both of them:

$$\frac{\begin{array}{c}\Longrightarrow \langle stmt_2\rangle\phi \\ \Longrightarrow \langle stmt_1\rangle\phi\end{array}}{\Longrightarrow \langle stmt_1\ \texttt{[]}\ stmt_2\rangle\phi}$$

Our rule does not capture cases in which a statement is ready for execution and the other is not where we could neglect one branch. However, this affects only completeness and not correctness. A more complete rule would have to look ahead which statement is ready. Though, in general boxes could be nested arbitrarily deep complicating the identification of ready branches.

The last statement to be accounted in this section is the *prove* statement. It is an assertion in the code which must hold. Thus we simply create another prove obligation:

$$\frac{\begin{array}{l}\Longrightarrow \langle\omega\rangle\phi \\ \Longrightarrow \ \text{texp} \doteq TRUE\end{array}}{\Longrightarrow \langle \texttt{prove}\ texp;\ \omega\rangle\phi} \qquad \frac{\Longrightarrow \langle v\ \texttt{:=}\ exp;\ \texttt{prove}\ v;\ \omega\rangle\phi}{\Longrightarrow \langle \texttt{prove}\ exp;\ \omega\rangle\phi}$$

Expressions

As mentioned in section 3 variables are initialized implicitly in our dialect of Creol. Thus the rules for variable declarations convert the declaration to an update with the standard value:

$$\frac{\Longrightarrow \{i := 0\}\langle\omega\rangle\phi}{\Longrightarrow \langle \texttt{var i : Int;}\ \omega\rangle\phi} \qquad \frac{\Longrightarrow \{b := false\}\langle\omega\rangle\phi}{\Longrightarrow \langle \texttt{var b : Bool;}\ \omega\rangle\phi}$$

$$\frac{\Longrightarrow \{o := null\}\langle\omega\rangle\phi}{\Longrightarrow \langle \texttt{var o : Any;}\ \omega\rangle\phi} \qquad \frac{\Longrightarrow \{l := null\}\langle\omega\rangle\phi}{\Longrightarrow \langle \texttt{var l : Label[];}\ \omega\rangle\phi}$$

General expressions can contain divisions by zero which forces us to take apart the expressions. The following rules will address all cases where the right hand side is an expression. A special case is the assignment of a literal or a variable (matched by *texp*) which is side effect free:

$$\frac{\Longrightarrow \{a := texp\}\langle\omega\rangle\phi}{\Longrightarrow \langle \texttt{a :=}\ texp;\ \omega\rangle\phi}$$

Another simple case is if a expression is surrounded by parenthesis:

$$\frac{\Longrightarrow \langle \texttt{v:=}exp;\ \omega\rangle\phi}{\Longrightarrow \langle \texttt{v:=(}exp\texttt{);}\ \omega\rangle\phi} \qquad \frac{\Longrightarrow \langle \texttt{v:=}texp;\ \omega\rangle\phi}{\Longrightarrow \langle \texttt{v:=(}texp\texttt{);}\ \omega\rangle\phi}$$

For each integer operator of arity two we need four different rules because there are four combinations of *exp* and *texp*. All non-terminal-expressions are converted by assigning them to a new variable:

$$\frac{\Longrightarrow \langle \texttt{v':=}exp_2\texttt{;}\ \texttt{v:=}texp_1\texttt{+v';}\ \omega\rangle\phi}{\Longrightarrow \langle \texttt{v:=}texp_1\texttt{+}exp_2\texttt{;}\ \omega\rangle\phi}$$

$$\frac{\Longrightarrow \langle \texttt{v':=}exp_1\texttt{;}\ \texttt{v:=v'+}texp_2\texttt{;}\ \omega\rangle\phi}{\Longrightarrow \langle \texttt{v:=}exp_1\texttt{+}texp_2\texttt{;}\ \omega\rangle\phi}$$

$$\frac{\Longrightarrow \langle \texttt{v':=}exp_1\texttt{;}\ \texttt{v'':=}exp_2\texttt{;}\ \texttt{v:=v'+v'';}\ \omega\rangle\phi}{\Longrightarrow \langle \texttt{v:=}exp_1\texttt{+}exp_2\texttt{;}\ \omega\rangle\phi}$$

Eventually all expressions use the terminal rule which creates an update:

$$\frac{\Longrightarrow \{v := texp_1 + texp_2\}\langle\omega\rangle\phi}{\Longrightarrow \langle \texttt{v:=}texp_1\texttt{+}texp_2\texttt{; } \omega\rangle\phi}$$

For the functions $-$ and $*$ the corresponding rules are analogous to $+$. So they are left out for brevity.

Example 7.1.3. We turn towards an example for an assignment using some integer calculations to clarify the difference between *texp* and *exp*:

```
1 i := 1 + 3 * 0
```

The calculation results in $i = 1$. So we express this in a proof obligation:

$$\Longrightarrow [\texttt{i:=1+3*0}]i \doteq 1$$

First the most upper operator is a plus. The left parameter of the plus is 1 which is a literal and therefore matched by *texp*. The right parameter is $3 * 0$ is neither a variable nor a literal. That is why it is matched by *exp*. Applying the corresponding rule we create a new variable:

$$\frac{\frac{\dots}{\Longrightarrow [\texttt{v':=3*0; i:=1+v'}]i \doteq 1}}{\Longrightarrow [\texttt{i:=1+3*0}]i \doteq 1}$$

Now, the first statement is the assignment of v' involving two instances of *texp*. Thus it is converted to an update which is simplified in the next step:

$$\frac{\frac{\frac{\dots}{\Longrightarrow \{v' := 0\}[\texttt{i:=1+v'}]i \doteq 1}}{\Longrightarrow \{v' := 3 * 0\}[\texttt{i:=1+v'}]i \doteq 1}}{\Longrightarrow [\texttt{v':=3*0; i:=1+v'}]i \doteq 1}$$
$$\frac{}{[\texttt{i:=1+3*0}]i \doteq 1}$$

The only statement left is the assignment of i. But in contrast to the beginning of the proof it does not contain instances of *exp* any more, so it can be converted to an update. Thereafter the first update is applied on the second one:

$$\frac{\frac{\frac{\frac{\dots}{\{v' := 0 \mid\mid i := 1 + 0\}i \doteq 1}}{\{v' := 0\}\{i := 1 + v'\}i \doteq 1}}{\Longrightarrow \{v' := 0\}[\texttt{i:=1+v'}]i \doteq 1}}{\Longrightarrow \{v' := 3 * 0\}[\texttt{i:=1+v'}]i \doteq 1}$$
$$\frac{}{\Longrightarrow [\texttt{v':=3*0; i:=1+v'}]i \doteq 1}$$
$$\frac{}{[\texttt{i:=1+3*0}]i \doteq 1}$$

The update is simplified again and applied to the formula. Then equality is replaced by true what allows us to close the proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{closed}}{\Longrightarrow true}}{\Longrightarrow 1 \doteq 1}}{\Longrightarrow \{v' := 0 \parallel i := 1\} i \doteq 1}}{\Longrightarrow \{v' := 0 \parallel i := 1 + 0\} i \doteq 1}}{\ldots}$$

Handling a division works in way except for the update creation. If a division through zero occurs the statement has to be replaced by *block* what is the reason for the unfolding of expressions:

$$\dfrac{\Longrightarrow if\ texp_2 \neq 0\ then\ \{v := texp_1/texp_2\}\langle\omega\rangle\phi\ else\ \langle\texttt{block;}\ \omega\rangle\phi}{\Longrightarrow \langle\texttt{v:=}texp_1\texttt{/}texp_2\texttt{;}\ \omega\rangle\phi}$$

Unary minus is simply reduced to the minus of arity two by rewriting it:

$$\dfrac{\Longrightarrow \langle\texttt{v:=0-w;}\ \omega\rangle\phi}{\Longrightarrow \langle\texttt{v:=-w;}\ \omega\rangle\phi}$$

Rules covering Boolean operations follow the scheme of the integer rules. There are no special cases leading to a blocked object.

7.2 Concurrent calculus

The rules discussed so far look very similar for every programming language. But we will use several special techniques to reason about Creol. Those are informally introduced in the following paragraphs.

Each object of a Creol program is assumed to have its own processor (see section 3). The scheduling policy of the threads (instances of a method) sharing the processor is cooperative assuring that the thread can only loose the processor on a release-point. Continuing after a release point the values of all global variables could possibly have been modified by other threads. Hence, a sound verification calculus has to assume that the value of global variables is unknown after each release-point. This would lead to a highly incomplete calculus as nearly every program relies on some assumptions about the global variables. To counter this problem a *class invariant* will be used which serves as a contract between all threads about the global variables.

Asynchronous communication between different objects with possible message overtaking is an important property of Creol. To enable us showing

some properties of such a system a *history* which records all the communication between the objects of the system is used. It is challenging to reason about a history which is shared by a number of objects working concurrently because it might be unknown for certain events which happened first. To avoid the necessity of proving all permutations of such sets of events (for example faced while verifying concurrent Java [BK07]) we will verify one class at the time. Thereby we consider only a part of the history namely the communication of the current object. Afterwards a proof of the composability of all those local histories has to be done to verify the complete system.

In basic program verification method calls are replaced by their implementation (inlining). However, a method can be called an arbitrary number of times leading to several proofs of the same code. To avoid this one can use *contracts* which specify assumptions about the input for a method and guarantees about properties of the output. Using this technique one has to prove every implementation of a method exactly once and can just use this contract every time the method is called.

There are several publications about verifying Creol using the described concepts for a *wp* [Dij75] calculus in [DJO08a] or a Hoare logic [Hoa69] in [DJO06, Bla08]. My calculus will mainly be based on them. Composing of histories is described in [DJO08b, JO02]. The notion of a history or trace was classically used in [Hoa83, Dah77]. Contracts are a form of a rely guarantee style [Jon81].

We will follow a bottom up approach by first introducing the domains being specific for reasoning about Creol. Thereafter we will see terms and predicates which are interpreted on those domains.

The section is structured as follows. First the representation of object identifiers in the logic is discussed. Thereafter method identifiers and labels relating invocation messages to completion messages are established. The next step, namely the introduction of messages, builds on the previous notions of this section. Sequences of messages form histories which are the subject of the following paragraphs. Then release-points, loop invariants, and method calls make use of histories.

For brevity we will assume during this section that a sort hierarchy for a CreolDL signature for a CreolDL Kripke structure is given in all definitions. An overview of all domains, rigid functions, and predicates is in Appendix B.

Object identifiers

During a run of a Creol program an object hierarchy is created. As every object (except the first one) must have been created by another already existing object the layout of the type hierarchy should look like a tree if viewed as graph. We will capture this in the definition of the domain of

object identifiers.

Definition 7.2.1. *The* domain of object identifiers *$objDom \in \mathcal{T}_d$ is a set:*

$$\bigcup_{i=0}^{\infty} objDom^i$$

where $objDom^0 := \{p(0)\}$
$objDom^i := \{p(o,i) | i \in \mathbb{N}_0, o \in objDom^{i-1}\}$

The intention behind this definition is that every object numbers its children with an increasing number according to the order they are created in. The integer in the object identifiers does not enumerate the number of created objects. The only assumption is that an object created later has an higher integer. How this is achieved is described in Definition 7.2.6.

Example 7.2.1. In figure 7.1 the integers of the parent relation p are printed for each object. The object identifier of the bottom left object would be $p(p(p(0),2),0)$ whereas the object on the middle right would be named by $p(p(0),25)$.

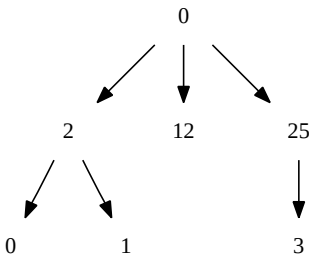

Figure 7.1: Example for an object hierarchy

To express the parent relation in form of terms we define the *parent* function.

Definition 7.2.2. *The* parent function *$parent : Any, Int \rightarrow Any \in FSym_r$ is interpreted as $\mathcal{I}_0(parent)(o,i) = p(o,i)$.*

Method identifiers

To talk about messages invoking a certain method of an object we need method identifiers. The domain is represented by a set of names which correspond to the allowed identifiers in Creol.

Definition 7.2.3. *The* method domain *$methDom \in \mathcal{T}_d$ is described by the following expression*

$$methDom := \{\_, a, b, \ldots, z\} \times \{\_', a, b, \ldots, z, A, B, \ldots, Z, 0, 1, \ldots, 9\}^*$$

Labels

A history consists of messages of method invocations, method completions and messages indicating that a new object was created. To relate an invocation message with its completion message in a history we will use a label uniquely identifying the communication procedure. A label for method calls relates two objects, namely the caller and the callee, a method and an integer (similar to section 3). The integer is used as sequence number as the tuple $(caller, callee, method)$ is not unique if the same method is called several times. The labels for method calls will be kept in the $methLabelDom$ domain. For object creation messages we define similar labels which contain the created object and an integer. Its parent is implicitly encoded in its identifier. The class of the created object does not have to be mentioned as the identifier will be typed by the class.

Definition 7.2.4. *The* domain of method labels $methLabelDom \in \mathcal{T}_d$ *is*

$$methLabelDom := \left\{ \langle o_1, o_2, m, i\rangle \;\middle|\; \begin{array}{l} o_1, o_2 \in objDom, m \in methDom \\ i \in intDom \end{array} \right\}$$

The domain of new object labels $newLabelDom \in \mathcal{T}_d$ *is*

$$newLabelDom := \{\langle p(o,i), i\rangle | o \in objDom, i \in intDom\}$$

So far we illustrated the label domains. But they are just sets terms are interpreted in. Thus we continue with the introduction of functions for labels. We start with a function *com* which is interpreted as a label talking about communication in form of message calls and a function *new* which is interpreted as a label out of $newLabelDom$. Additionally, we need some functions to access the attributes of *com* and *new*.

Definition 7.2.5. *The* communication label function $com : Any, Any, Method, Int \to Label \in FSym_r$ *is interpreted as* $\mathcal{I}_0(com)(o_1, o_2, m, i) = \langle o_1, o_2, m, i\rangle$. *Its parameters can be accessed by the functions:*

- $toCaller : Label \to Any \in FSym_r$
  with $\mathcal{I}_0(toCaller)(\langle o_1, o_2, m, i\rangle) = o_1$
- $toCallee : Label \to Any \in FSym_r$
  with $\mathcal{I}_0(toCallee)(\langle o_1, o_2, m, i\rangle) = o_2$
- $toMethod : Label \to Method \in FSym_r$
  with $\mathcal{I}_0(toMethod)(\langle o_1, o_2, m, i\rangle) = m$
- $toId : Label \to Int \in FSym_r$ *with* $\mathcal{I}_0(toId)(\langle o_1, o_2, m, i\rangle) = i$

The new object label function $new : Any, Int \to NewLabel \in FSym_r$ *is interpreted as* $\mathcal{I}_0(new)(o, i) = \langle o, i\rangle$. *Its parameters can be accessed by the functions:*

- $toNew : NewLabel \to Any \in FSym_r$ *with* $\mathcal{I}_0(toNew)(\langle o, i\rangle) = o$
- $toIdNew : NewLabel \to Int \in FSym_r$ *with* $\mathcal{I}_0(toIdNew)(\langle o, i\rangle) = i$

Under the fixed interpretation we give the related rewriting rules:

$$\frac{o_1}{toCaller(com(o_1, o_2, m, i))} \qquad \frac{o_2}{toCallee(com(o_1, o_2, m, i))}$$

$$\frac{m}{toMethod(com(o_1, o_2, m, i))} \qquad \frac{i}{toId(com(o_1, o_2, m, i))}$$

$$\frac{o}{toNew(new(o, i))} \qquad \frac{i}{toIdNew(new(o, i))}$$

where $o_1, o_2 \in Term_{Any}$, $m \in Term_{Method}$ and $i \in Term_{Int}$.

To guarantee that the integer used in a label is unique during a proof we add a ghost class attribute. It will contain the next integer to be used for a sent invocation or new object message.

Definition 7.2.6. *The* sequence number $\mathcal{L} :\to Int \in FSym_{nr}$ *is a non rigid function.*

During a proof we will increment it after each sent message.

Messages

The last step before building up the actual history are messages. Every message contains a label and some data. In the case of a invocation message the data are the parameters of the invoked method. For a completion message the return values are saved in the data field. A new object message has the parameters for the *init* method of the class.

Definition 7.2.7. *The* message domain $msgDom$ *is defined as*

$$\{\langle t, l, \bar{d}\rangle | t \in \{invoc, comp\}, l \in methLabelDom, \bar{d} \in domData^*\}$$

$$\cup\{\langle new, l, \bar{d}\rangle | l \in newLabelDom, \bar{d} \in domData^*\}$$

where the data domain $domData := \bigcup_{dom \in \mathcal{T}_d, dom \sqsubseteq Data} dom$

The first parameter of a message determines its type.

Example 7.2.2. The message $\langle invoc, \langle o_1, o_2, m, 10\rangle, (4, 3)\rangle$ is a invocation message send from object o_1 to o_2 where the method m was invoked with the parameters 4 and 3. Assuming that m computes the sum the corresponding completion message $\langle comp, \langle o_1, o_2, m, 10\rangle, (7)\rangle$ has exactly the same label and the return value 7.

Having discussed the domains of messages we define a function for each message type.

Definition 7.2.8. *The* message functions *are defined as follows:*

- $msgInvoc : Label, Data^* \to Message \in FSym_r$ *with*
  $\mathcal{I}_0(msgInvoc)(\langle o_1, o_2, m, i\rangle, \overline{d}) = \langle invoc, \langle o_1, o_2, m, i\rangle, \overline{d}\rangle$
- $msgComp : Label, Data^* \to Message \in FSym_r$ *with*
  $\mathcal{I}_0(msgComp)(\langle o_1, o_2, m, i\rangle, \overline{d}) = \langle comp, \langle o_1, o_2, m, i\rangle, \overline{d}\rangle$
- $msgNew : NewLabel, Data^* \to Message \in FSym_r$ *with*
  $\mathcal{I}_0(msgNew)(\langle o, i\rangle, \overline{d}) = \langle new, \langle o, i\rangle, \overline{d}\rangle$

In figure 7.2 the sorts and domains involving messages are pictured.

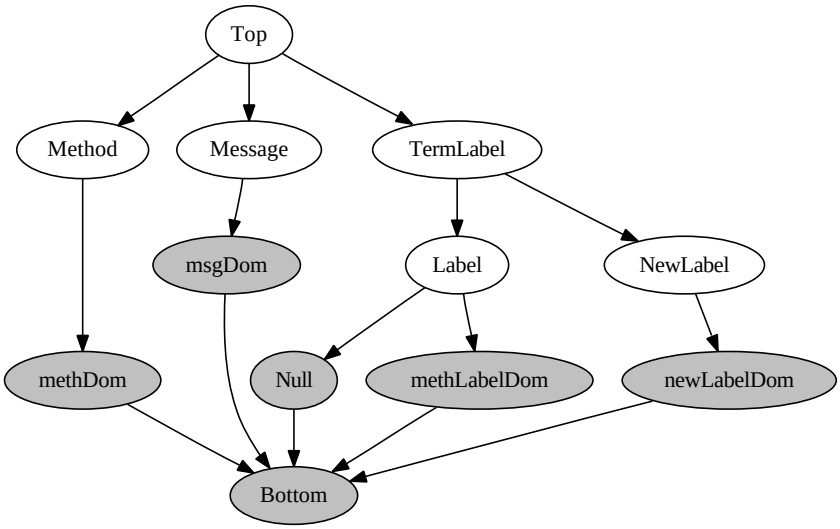

Figure 7.2: Sort hierarchy of the messages

Histories

Now, we can build up a history being a sequence of messages. The history domain contains all those sequences. Thereby we will use the symbol $^\frown$ for concatenation two sequences of messages, so appending a single message is a special case of it. ϵ is the empty sequence such that for any history h : $h^\frown\epsilon = h$ and $\epsilon^\frown h = h$.

Definition 7.2.9. *The* history domain *histDom is defined as*

$$histDom := \bigcup_{i=0}^{\infty} histDom^i$$

$$\textit{where} \quad \begin{array}{l} histDom^0 := \{\epsilon\}, \\ histDom^i := \{h^\frown m | h \in histDom^{i-1}, m \in msgDom\} \end{array}$$

While reasoning about a Creol program using histories the central question is what is the intended meaning of the history. As there are no assumptions about the underlying network (see section 3) the only known order of messages is the sending order. During transmission they might have been reordered such that we cannot assume anything about the order they are arriving in.

A system wide global history speaking about the sending time of each message is easy to imagine. But representing the history from an object view[2] leads to some uncertainty as the actual point in time when a message was sent by another object is unknown. The consequence is that the logic handling the history of an object is blown up to handle the uncertainty. But as this notion is the well understood one (calculi using them in [DJO06, DJO08a], work on composing them in [DJO08a, JO02]) we will create our calculus in this setting. Other approaches are discussed in section 7.4.

The composition of histories is not covered in this work. The prior theoretical investigations require adaption and refinement to be applicable to the approach of this thesis. The problem of composition is undecidable in the general case like most of the issues we are dealing with.

Let us turn to the local histories of objects. The representation of the history as a simple list of messages in form of terms is not possible. First, because the point in time a message from another object was sent is unknown. Second, to make things worse, not even the time of arrival is known. For example if we reach a statement l?(x) in execution the corresponding message might have arrived some time before. Third, we never know whether new method invocations are arriving from other objects.

The last point raises another issue. In general, during execution of any statement, messages could arrive which would require us to note this in all rules concerning statements. We get around this problem by lazily extending the corresponding history on an history access. This view is equivalent to the eager extension of histories if one restricts oneself to the verification of complete methods.

The good news is that this uncertainty covers only messages sent by other objects. About messages sent by the object itself the sending time and hence order is known. This is the motivation to keep two histories of each object. The first one is the sending history which will have to sort *SendingHistory* and will essentially be a list of messages represented by terms. The second history is the complete history of the object whose terms will have the sort *ObjectHistory*. Properties about the object history will be expressed by predicates ensuring the occurrence of certain messages only. Therefore the interpretation of the object history is never completely fixed which allows incoming messages for example.

[2] Just considering messages which either are send by the object or to the object: $h \restriction o$, see Def. 7.2.10

Having a sending and a object history implies that all messages being sent must be mentioned twice. This leads to a overhead as we will have to check the consistency of both histories frequently.

So far a history $h \in histDom$ displays communication between all objects of the system. To express a history which just consists of messages concerning a certain object we define the projection operator $h \restriction o$ which removes all messages not being sent to or from a object o. It will be used to interpret object histories. To reflect only messages which are sent by a object o we will write $h \restriction o_{\rightarrow}$ which is necessary to interpret sending histories. Both operators are projections in the sense that applying them twice has the same effect as applying them once.

Definition 7.2.10. *For a history $h \in histDom$ the projection $h \restriction o$ to the messages sent by or to an object $o \in objDom$ is defined as:*

$$\epsilon \restriction o = \epsilon$$

$$h^\frown\langle t, \langle o_1, o_2, m, i\rangle, \bar{d}\rangle \restriction o = \begin{cases} h \restriction o & \text{if } o \neq o_1, o \neq o_2 \\ (h \restriction o)^\frown\langle t, \langle o_1, o_2, m, i\rangle, \bar{d}\rangle & \text{otherwise} \end{cases}$$

$$h^\frown\langle new, \langle o_1, i\rangle, \bar{d}\rangle \restriction o = \begin{cases} h \restriction o & \text{if } o \neq o_1, p(o,i) \neq o_1 \\ (h \restriction o)^\frown\langle new, \langle o_1, i\rangle, \bar{d}\rangle & \text{otherwise} \end{cases}$$

For a history $h \in histDom$ the projection $h \restriction o_{\rightarrow}$ to the messages sent by an object $o \in objDom$ is defined as:

$$\epsilon \restriction o_{\rightarrow} = \epsilon$$

$$h^\frown\langle t, \langle o_1, o_2, m, i\rangle, \bar{d}\rangle \restriction o_{\rightarrow} = \begin{cases} h \restriction o_{\rightarrow} & \text{if } o \neq o_1 \\ (h \restriction o_{\rightarrow})^\frown\langle t, \langle o_1, o_2, m, i\rangle, \bar{d}\rangle & \text{otherwise} \end{cases}$$

$$h^\frown\langle new, \langle o_1, i\rangle, \bar{d}\rangle \restriction o_{\rightarrow} = \begin{cases} h \restriction o_{\rightarrow} & \text{if } p(o,i) \neq o_1 \\ (h \restriction o_{\rightarrow})^\frown\langle new, \langle o_1, i\rangle, \bar{d}\rangle & \text{otherwise} \end{cases}$$

So we are talking about the history of an object, but we never introduced the object formally, right? For this reason we will use *this* when we need a reference to the object we are currently verifying.

Definition 7.2.11. *A reference to the object subject to verification is always given by this $:\rightarrow Any \in FSym_r$ where the interpretation is fixed during a proof by $\mathcal{I}_0(this) \in objDom$.*

Now we are capable of introducing the domains the *SendingHistory* and *ObjectHistory* will be interpreted in:

Definition 7.2.12. *The* sending history domain *$sendHistDom \in \mathcal{T}_d$ is:*

$$sendHistDom := \{h \restriction \mathcal{I}_0(this)_{\rightarrow} | h \in histDom\}$$

The object history domain *$objHistDom \in \mathcal{T}_d$ is:*

$$objHistDom := \{h \restriction \mathcal{I}_0(this) | h \in histDom\}$$

In figure 7.3 is a picture of all domains and sorts involving the history. $objHistDom \subset histDom$ and $sendHistDom \subset histDom$ which makes them sub sorts of $histDom$.

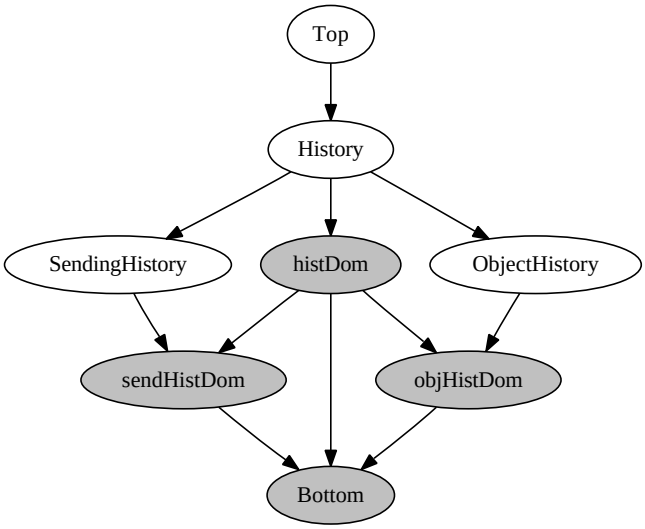

Figure 7.3: Sort hierarchy of the history

To access the histories naturally during our proofs both histories will be additives ghost class attributes of the object. The meaning of the additional class attributes, both histories, can change during a verification procedure. Hence they have to be non-rigid functions.

Definition 7.2.13. *The* sending history $\mathcal{H}_S :\rightarrow SendingHistory \in FSym_{nr}$ *and the* object history $\mathcal{H}_O :\rightarrow ObjectHistory \in FSym_{nr}$ *are non-rigid functions.*

As mentioned before the sending history will be a list in form of terms. We know already about the message functions from Definition 7.2.8 the only thing missing is a function which appends a message to a given sending history. The *hist* function will do so.

To check the consistency of the sending history with a object history we need to fix a common history where both started at. The *startsAt* function which attaches a sending history to a object history will do so.

Definition 7.2.14. *The* history function $hist : SendingHistory, Message \rightarrow SendingHistory \in FSym$ *appends a Message to a history by*

$$\mathcal{I}_0(hist)(h, msg) = h^\frown msg$$

The starts at *function* $startsAt : ObjectHistory \rightarrow SendingHistory \in FSym$ *relates an object history with a sending history:*

$$\mathcal{I}_0(startsAt)(h) = h \restriction \mathcal{I}_0(this)_\rightarrow$$

We introduced all necessary tools express a sending history as a list which leads us to the following example:

Example 7.2.3. For $H_O \in Term_{ObjectHistory}$, $L \in Term_{Label}$ and $D \in Term_{Data}$ the term

$$hist(startsAt(H_O), msgInvoc(L, D))$$

describes a history which is extended by a invocation message. We can check this by calculating the *val* function of Definition 6.3.12:

$val_{S,\beta}(hist(startsAt(H_O), msgInvoc(L, D)))$
$= \mathcal{I}_0(hist)(val_{S,\beta}(startsAt(H_O)), val_{S,\beta}(msgInvoc(L, D)))$
$= \mathcal{I}_0(startsAt)(val_{S,\beta}(H_O))^\frown \mathcal{I}_0(msgNew)(val_{S,\beta}(L), val_{S,\beta}(D))))$
$= val_{S,\beta}(H_O)^\frown \langle invoc, val_{S,\beta}(L), val_{S,\beta}(D)\rangle$
We have no more knowledge about H_O, L and D so the evaluation ends here.

The object history is described by different predicates. One of the most intuitive ones is *Comp* which is evaluated to true if a given history contains a completion message for a given label. *Invoc* and New assure the same issue for invocation and new object messages.

Definition 7.2.15. *There are three predicates evaluating to true iff a message of a given label is contained in a history:*

- new object message predicate $New : ObjectHistory, Any \in PSym$ *with*

$$\mathcal{I}_0(New)(h, o) = \left\{ (h, o) \in \begin{array}{l} objHistDom \\ \times objDom \end{array} \;\middle|\; \begin{array}{l} \exists \bar{d} \in domData^* \\ \exists i \in intDom : \\ \langle new, \langle o, i\rangle, \bar{d}\rangle \in h \end{array} \right\}$$

- *The* invocation message predicate $Invoc : ObjectHistory, Label \in PSym$ *with*

$$\mathcal{I}_0(Invoc)(h, l) = \left\{ (h, l) \in \begin{array}{l} objHistDom \times \\ methLabelDom \end{array} \;\middle|\; \begin{array}{l} \exists \bar{d} \in domData^* : \\ \langle invoc, l, \bar{d}\rangle \in h \end{array} \right\}$$

- *The* completion message predicate $Comp : ObjectHistory, Label \in PSym$ *with*

$$\mathcal{I}_0(Comp)(h, l) = \left\{ (h, l) \in \begin{array}{l} objHistDom \times \\ methLabelDom \end{array} \;\middle|\; \begin{array}{l} \exists \bar{d} \in domData^* : \\ \langle comp, l, \bar{d}\rangle \in h \end{array} \right\}$$

To clarify the definition let us consider an example:

Example 7.2.4. For $l \in Term_{Label}$, $o \in Term_{Any}$ and $d_i \in Term_{Data}$ the history: $h := \langle invoc, l, d_1\rangle^\frown\langle comp, l, d_2\rangle$ describes a method invocation and its completion message. Assuming the constants H_O, O_2 and L are interpreted as $\mathcal{I}(H_O) = h, \mathcal{I}(O_2) = o$ and $\mathcal{I}(L) = l$, we can write the formula $\neg New(H_O, O_2) \wedge Comp(H_O, L)$ which should be valid. Let us inspect this by evaluating:

$S, \beta \models \neg New(H_O, O_2) \wedge Comp(H_O, L)$
iff $S, \beta \models \neg New(H_O, O_2)$ and $S, \beta \models Comp(H_O, L)$
iff $S, \beta \not\models New(H_O, O_2)$ and $S, \beta \models Comp(H_O, L)$
iff $(val_{S,\beta}(H_O), val_{S,\beta}(O_2)) \notin \mathcal{I}(New)$ and $(val_{S,\beta}(H_O), val_{S,\beta}(L)) \in \mathcal{I}(Comp)$
iff $(h, o) \notin \mathcal{I}(New)$ and $(h, l) \in \mathcal{I}(Comp)$
iff
$$(h,o) \notin \left\{ (h,o') \in \begin{array}{l} objHistDom \\ \times Any \end{array} \;\middle|\; \begin{array}{l} \exists \bar{d} \in domData^* \; \exists i \in intDom \\ \langle new, \langle o, i\rangle, \bar{d}\rangle \in h \end{array} \right\}$$
$$(h,l) \in \left\{ (h,l') \in \begin{array}{l} objHistDom \\ \times methLabelDom \end{array} \;\middle|\; \begin{array}{l} \exists \bar{d} \in domData^* : \\ \langle comp, l', \bar{d}\rangle \in h \end{array} \right\}$$
iff $(h, o) \notin \emptyset$ and $(h, l) \in \{(h, l)\}$

During the verification procedure of a class it will be necessary to extend existing object histories. Thus we define the $Prefix$ predicate meaning that a given history starts with all the messages of another given history.

Definition 7.2.16. *The* prefix predicate

$$Prefix : ObjectHistory, ObjectHistory \in PSym$$

is interpreted as follows:

$$\mathcal{I}_0(Prefix) = \left\{ (h_1, h_2) \in \begin{array}{l} objHistDom \times \\ objHistDom \end{array} \;\middle|\; \begin{array}{l} \exists h_3 \in objHistDom : \\ h_1^\frown h_3 = h_2 \end{array} \right\}$$

Each history is a prefix of itself (choose $h_3 = \epsilon$). So if the prefix predicate receives the same history twice as its parameters it is always true. The corresponding rule for a history $H \in Term_{ObjectHistory}$ is:

$$\frac{true}{Prefix(H, H)}$$

The prefix relation is a transitive relation which can be checked in the above definition by concatenating the two obtained h_3 to a new one. We give a rule which expresses the transitivity for histories $H_1, H_2, H_3 \in Term_{ObjectHistory}$:

$$\frac{Prefix(H_1, H_2), Prefix(H_2, H_3), Prefix(H_1, H_3) \Longrightarrow}{Prefix(H_1, H_2), Prefix(H_2, H_3) \Longrightarrow}$$

We note that given $Prefix(H_1, H_2) \wedge Invoc(H_1, L)$ we should be able to conclude $Invoc(H_2, L)$. The reason is that all messages of H_1 are in H_2 as well. So the invocation message with label L must be in H_2, too. The same holds for the $Comp$ and the New predicates. We capture the behavior in the following rules ensuring the monotonicity:

$$\frac{Invoc(H_1, L), Prefix(H_1, H_2), Invoc(H_2, L) \Longrightarrow}{Invoc(H_1, L), Prefix(H_1, H_2) \Longrightarrow}$$

$$\frac{Comp(H_1, L), Prefix(H_1, H_2), Comp(H_2, L) \Longrightarrow}{Comp(H_1, L), Prefix(H_1, H_2) \Longrightarrow}$$

$$\frac{New(H_1, O), Prefix(H_1, H_2), New(H_2, O) \Longrightarrow}{New(H_1, O), Prefix(H_1, H_2) \Longrightarrow}$$

where $H_1, H_2 \in Term_{ObjectHistory}$, $L \in Term_{Label}$ and $O \in Term_{Any}$.

We can apply similar reasoning to find out that a message not contained in a history must not be contained in its prefixes. Therefore we write the following rules:

$$\frac{\neg Invoc(H_2, L), Prefix(H_1, H_2), \neg Invoc(H_1, L) \Longrightarrow}{\neg Invoc(H_2, L), Prefix(H_1, H_2) \Longrightarrow}$$

$$\frac{\neg Comp(H_2, L), Prefix(H_1, H_2), \neg Comp(H_1, L) \Longrightarrow}{\neg Comp(H_2, L), Prefix(H_1, H_2) \Longrightarrow}$$

$$\frac{\neg New(H_2, O), Prefix(H_1, H_2), \neg New(H_1, O) \Longrightarrow}{\neg New(H_2, O), Prefix(H_1, H_2) \Longrightarrow}$$

H_1, H_2, L, O chosen as in the previous rules.

The object history domain contains a lot of histories which are invalid with respect to program execution. For example there might be a completion message in the history for a method which was never invoked. To restrict the histories we use the predicate Wf[3]. It will ensure that for every completion message there is an invocation message in the history. Furthermore the history has to start with the creation message of the object *this*. Furthermore we require that every sent message has a higher sequence number in its label than all previously sent messages (the second parameter of wf).

Definition 7.2.17. *The* well-formed *predicate* $Wf : ObjectHistory, Int \in PSym$ *evaluates to true according to* $\mathcal{I}_0(Wf) = \{(h, o, i) \in objHistDom \times intDom | wf(h, i) = 1\}$ *where* $wf : objHistDom \times intDom \rightarrow \{0, 1\}$ *such that*

[3]Similar to the inReachableState predicate of KeY for Java

- $wf(h^\frown\langle invoc, \langle o_1, o_2, m, i\rangle, \bar{d}\rangle, j) = \begin{cases} wf(h,i) & \text{if } \mathcal{I}_0(this) = o_1, i < j \\ wf(h,j) & \text{if } \mathcal{I}_0(this) = o_2 \\ 0 & \text{otherwise} \end{cases}$

- $wf(h^\frown\langle comp, \underbrace{\langle o_1, o_2, m, i\rangle}_{l}, \bar{d}\rangle, j) = \begin{cases} wf(h,i) & \text{if } \mathcal{I}_0(this) = o_2 \text{ and} \\ & \exists \bar{d}'\ \exists \langle invoc, l, \bar{d}'\rangle \in h \\ wf(h,j) & \text{if } \mathcal{I}_0(this) = o_1, i < j, \\ & \exists \bar{d}'\ \exists \langle invoc, l, \bar{d}'\rangle \in h \\ 0 & \text{otherwise} \end{cases}$

- $wf(h^\frown\langle new, \langle o, i\rangle, \bar{d}\rangle, j) = \begin{cases} 1 & \text{if } \mathcal{I}_0(this) = o \text{ and } h = \epsilon \\ wf(h,i) & \text{if } p(\mathcal{I}_0(this), i) = o \text{ and } i < j \\ 0 & \text{otherwise} \end{cases}$

Please note that the above definition does not allow the occurrence of two invocation messages for a given completion message because one would have to occur after other one which contradicts $i < j$. This fairly complicated definition requires an example.

Example 7.2.5. Given the object history $h := msg_1^\frown msg_2^\frown msg_3$ with:

- $msg_1 = \langle new, \langle \mathcal{I}_0(this), 3\rangle, 5\rangle$
- $msg_2 = \langle invoc, \langle \mathcal{I}_0(this), o_1, m_1, 2\rangle, 17\rangle$
- $msg_3 = \langle invoc, \langle o_2, \mathcal{I}_0(this), m_2, 1\rangle, -2\rangle$

where $o_i \in objDom$ and $m_i \in methDom$. The history should be well-formed such that $wf(h, 18) = 1$ because it starts with the object creation message of *this*, then a invocation message send by *this* which satisfies $17 < 18$ and at last an invocation message received from another object. Hence neither a invocation message for a completion message is missing nor the sequence number is increasing. Let us calculate it. By applying the second case of invocation messages where the receiver is *this*, we obtain:

$$wf(h, 18) = wf(msg_1^\frown msg_2^\frown \langle invoc, \langle o_2, \mathcal{I}_0(this), m_2, 1\rangle, -2\rangle, 18)$$

Now, we use the first case of invocation messages:

$$= wf(msg_1^\frown \langle invoc, \langle \mathcal{I}_0(this), o_1, m_1, 2\rangle, 17\rangle, 18)$$

and finally the first case of new object messages leads us to:

$$= wf(\langle new, \langle \mathcal{I}_0(this), 3\rangle, 5\rangle, 17) = 1$$

After the example about the interpretation of the Wf predicate we turn our attention towards sequent rules which express properties of the interpretation. We will use H for terms of sort *ObjectHistory*, I for integer terms, O for terms of sort *Any*, L for terms with sort *Label* and M for methods.

First, we know that the messages a well-formed history consists of are either sent to or by the object *this*. Therefore the following rules are valid:

$$\frac{Comp(H,L), Wf(H,I), toCaller(L) = this \vee toCallee(L) = this \Longrightarrow}{Comp(H,L), Wf(H,I) \Longrightarrow}$$

$$\frac{Invoc(H,L), Wf(H,I), toCaller(L) = this \vee toCallee(L) = this \Longrightarrow}{Invoc(H,L), Wf(H,I) \Longrightarrow}$$

$$\frac{New(H, parent(O,I_2)), Wf(H,I), O = this \vee parent(O,I_2) = this \Longrightarrow}{New(H, parent(O,I_2)), Wf(H,I) \Longrightarrow}$$

Second, for a completion message there should be an invocation message:

$$\frac{Wf(H,I), Comp(H,L), Invoc(H,L) \Longrightarrow}{Wf(H,I), Comp(H,L) \Longrightarrow}$$

Third, a label which was not used for a invocation must not be used for a completion:

$$\frac{Wf(H,I), \neg Comp(H,L), \neg Invoc(H,L) \Longrightarrow}{Wf(H,I), \neg Invoc(H,L) \Longrightarrow}$$

Next, if we have to show that a history is well-formed, it is allowed to strengthen the well-formed predicate:

$$\frac{\Longrightarrow \exists I'.(Wf(H,I') \wedge I' \leq I)}{\Longrightarrow Wf(H,I)}$$

Where I' must neither occur in H nor in I. Additionally, a well-formed history contains the creation message of the object *this*:

$$\frac{Wf(H,I), New(H,this) \Longrightarrow}{Wf(H,I) \Longrightarrow}$$

Finally, we can impose conditions on the integers of the parent relation, if the message is sent by *this* and the label is given explicitly:

$$\frac{Wf(H,I), Invoc(H, com(this,O,M,I_2)), I_2 < I \Longrightarrow}{Wf(H,I), Invoc(H, com(this,O,M,I_2)) \Longrightarrow}$$

$$\frac{Wf(H,I), Comp(H, com(O,this,M,I_2)), I_2 < I \Longrightarrow}{Wf(H,I), Comp(H, com(O,this,M,I_2)) \Longrightarrow}$$

$$\frac{Wf(H,I), New(H, new(parent(this,I_2),I_2)), I_2 < I \Longrightarrow}{Wf(H,I), New(H, new(parent(this,I_2),I_2)) \Longrightarrow}$$

Now, the object history and the sending history have been discussed. What is left to do, are the predicates involving both of them. To check whether all the messages of the sending history are included in the object history we use the *Cons* predicate. In addition, it ensures the well-formed properties of the sending history.

Definition 7.2.18. *The* consistent predicate*:*

$$Cons : SendingHistory, ObjectHistory, Int \in PSym$$

has the following interpretation:

$$\mathcal{I}_0(Cons) = \left\{ (h_s, h_o, i) \in \begin{array}{l} sendHistDom \times \\ objHistDom \times \\ intDom \end{array} \;\middle|\; \begin{array}{l} \exists h_1 \in sendHistDom \\ h_1 \frown h_s = h_o \restriction this_{\rightarrow} \\ \text{and } wf(h_o, i) \end{array} \right\}$$

We need some rules which check the consistency of the predicate. Thereby the *sending history* is processed recursively and for each message its occurrence in the *object history* is checked:

$$\frac{\Longrightarrow Cons(H_S, H_O, J) \wedge Invoc(H_O, com(this, callee, M, J)) \wedge J < I}{\Longrightarrow Cons(hist(H_S, msgInvoc(com(this, callee, M, J), \overline{D})), H_O, I)}$$

$$\frac{\Longrightarrow Cons(H_S, H_O, J) \wedge New(H_O, new(parent(this, J))) \wedge J < I}{\Longrightarrow Cons(hist(H_S, msgInvoc(new(parent(this, J), J), \overline{D})), H_O, I)}$$

To ensure well-formedness we add the *Invoc* predicate to the succedent:

$$\frac{\Longrightarrow Cons(H_S, H_O, I) \begin{array}{l} \wedge Invoc(H_O, com(caller, this, M, J)) \\ \wedge Comp(H_O, com(caller, this, M, J)) \end{array}}{\Longrightarrow Cons(hist(H_S, msgComp(com(caller, this, M, J), \overline{D})), H_O, I)}$$

Finally, if the *sending history* starts at a *object history* we have to make sure that the initial *object history* contains the other one as a prefix:

$$\frac{\Longrightarrow Wf(H_{O,2}, I) \wedge Prefix(H_{O,2}, H_O)}{\Longrightarrow Cons(hist(startsAt(H_{O,2})), H_O, I)}$$

Release-points

At a release-point the processor is handed over to other processes (see section 3). Thus all class attributes might have been overwritten when continuing after a release-point what makes it difficult to reason about it. Our approach uses the class invariant to express properties of the class attributes. We require the class invariant to hold at every release-point of all methods. Thus we do not need to consider other threads while verifying a method.

Let us start with a notation for the class attributes declared in the code:

Definition 7.2.19. *The vector of* class attributes *is denoted by* $\overline{\mathcal{W}} \in FSym^*_{nr}$.

Please not that we have three additional class attributes, namely $\mathcal{H}_O$, $\mathcal{H}_S$ and $\mathcal{L}$ which are not contained in $\overline{\mathcal{W}}$.

To overwrite all class attributes we use anonymous updates (cf. [BHS07], chapter 3) which replace the non-rigid function by an arbitrary new one nothing is known about:

Definition 7.2.20. *Let a CreolDL signature for a sort hierarchy, a sequent* $\Gamma \Longrightarrow \Delta$ *and a vector* $\bar{v}$ *(or set* v*) of program variables be given. For every* $f_i : A_1, \ldots, A_{n_i} \to A \in \bar{v}$ $(0 \leq i \leq n)$*: let* $f' : A_1, \ldots, A_{n_i} \to A \in FSym_r$ *be a fresh w.r.t* $\Gamma \cup \Delta$ *rigid function symbol, then the update:*

$$u_1 \mid\mid \ldots \mid\mid u_n$$

with

$$u_i = for\ x^i_1;\ true;\ \ldots\ for\ x^i_{n_i};\ true;\ f_i(x^i_1, \ldots, x^i_{n_i}) := f'_i(x^i_1, \ldots, x^i_{n_i})$$

is called anonymizing update *for the sequent* $\Gamma \Longrightarrow \Delta$ *and denoted by* $A_{\bar{v}}$.

Given the above definition we write $A_{\overline{\mathcal{W}}}$ to anonymize the class attributes.

Anonymizing the object history is a bit more complicated because we want to relate the old history with the new one by a prefix. Therefore we have to save the old history before anonymizing:

Definition 7.2.21. *For a CreolDL signature for a sort hierarchy and a sequent* $\Gamma \Longrightarrow \Delta$ *containing* ϕ *the* object history anonymizing formula $F_{\mathcal{H}}$ *is defined as:*

$$F_{\mathcal{H}_O}(\phi) := \forall H_{O,pre}.(\mathcal{H}_O \doteq H_{O,pre} \to \forall H_{O,new}.(\{\mathcal{H}_O := H_{O,new}\}\phi))$$

where H_{pre}, H_{new} *is new w.r.t* Γ *and* Δ.

Example 7.2.6. In our proofs we will be interested in extensions of the history which are prefixed by the old history. We can express this as $F_{\mathcal{H}}(Prefix(H_{O,pre}, H_{O,new}))$.

To anonymize the sequence number $\mathcal{L}$ we will proceed analogous to the history:

Definition 7.2.22. *For a CreolDL signature for a sort hierarchy and a sequent* $\Gamma \Longrightarrow \Delta$ *containing* ϕ *the* sequence number anonymizing formula $F_{\mathcal{L}}$ *is defined as:*

$$F_{\mathcal{L}}(\phi) := \forall L_{pre}.(\mathcal{L} \doteq L_{pre} \to \forall L_{new}.(\{\mathcal{L} := L_{new}\}\phi))$$

where L_{pre}, L_{new} *is new w.r.t* Γ *and* Δ.

A class invariant has to be supplied with each class. In particular we are always interested in well-formed histories. Hence, we add the well-formed predicate to all given class invariants by:

Definition 7.2.23. *A given class invariant* $Inv_{C,program}$ *is extended to* $Inv_C := Inv_{C,program} \wedge Wf(\mathcal{H}_O, \mathcal{L})$.

Finally, we finished all necessary definitions to discuss the rules involving release-points. At a release-point we have to show that the class invariant holds when arriving there and we can assume it when continuing afterwards:

$$\frac{\begin{array}{l}\Longrightarrow Inv_C \wedge Cons(\mathcal{H}_S, \mathcal{H}_O, \mathcal{L}) \\ \Longrightarrow \begin{array}{l} A_{\overline{\mathcal{W}}}(F_{\mathcal{L}}(F_{\mathcal{H}_O}(\{\mathcal{H}_S := startsAt(\mathcal{H}_O)\} \\ (Prefix(H_{pre}, H_{new}) \wedge Inv_C \wedge L_{pre} < L_{new} \rightarrow [\omega]\phi)))) \end{array}\end{array}}{\Longrightarrow [\texttt{release; } \omega]\phi}$$

The upper branch ensures both that the class invariant holds and that the two histories are consistent. The lower branch allows other threads to have modified all class attributes. The sending history is reinitialized (see Def 7.2.21). While reasoning about Creol we will typically define new predicates which express certain properties of the history. Those are checked at the sending history between two release-points and usually are contained in the class invariant. For an example see chapter 9.

We can handle the *await* statement similar to the *release* statement. The guard can is either *wait* or *l?* or a expression. In the case of *await wait* we can equivalently rewrite it to *release*:

$$\frac{\Longrightarrow [\texttt{release; } \omega]\phi}{\Longrightarrow [\texttt{await wait; } \omega]\phi}$$

If the guard is an expression, which in general could enclose a division by zero, we proceed analogous to integer arithmetic by decomposing it:

$$\frac{\Longrightarrow [v\texttt{:=}exp\texttt{; await } v\texttt{; } \omega]\phi}{\Longrightarrow [\texttt{await } exp\texttt{; } \omega]\phi}$$

where v is a new variable. This rule is sound in the context of Creol as there will be no processor release during the decomposition of the expression. When we arrive at a *await* statement with a terminal expression there are two possibilities. First, the guard could be true, so the processor does not release. Second, the processor releases because the guard was false.

$$\frac{\begin{array}{ll} & if\ texp \doteq TRUE \\ & then\ [\omega]\phi \\ \Longrightarrow & \begin{array}{ll} & Inv_C \wedge Cons(\mathcal{H}_S, \mathcal{H}_O, \mathcal{L}) \\ else & \wedge A_{\overline{\mathcal{W}}}(F_{\mathcal{L}}(F_{\mathcal{H}_O}(\{\mathcal{H}_S := startsAt(\mathcal{H}_O)\} \\ & \left(\begin{array}{l} Prefix(H_{pre}, H_{new}) \wedge Inv_C \\ \wedge L_{pre} < L_{new} \wedge texp \doteq TRUE \rightarrow [\omega]\phi \end{array}\right)))) \end{array}\end{array}}{\Longrightarrow [\texttt{await } texp\texttt{; } \omega]\phi}$$

The invariant and the consistency check are in the else-branch as they only have to hold if the processor releases.

await l? can be handled similarly. The difference is the check whether the corresponding completion message has arrived and whether the label is not *null*.

$$\frac{\begin{array}{l}\Longrightarrow l \neq null \\ \quad if\ Comp(\mathcal{H}_O, l) \\ \quad then\ [\omega]\phi \\ \Longrightarrow \quad Inv_C \wedge Cons(\mathcal{H}_S, \mathcal{H}_O, \mathcal{L}) \\ \quad else\ \ \wedge A_{\overline{\mathcal{W}}}(F_{\mathcal{L}}(F_{\mathcal{H}_O}(\{\mathcal{H}_S := startsAt(\mathcal{H}_O)\} \\ \qquad \left(\begin{array}{l} Prefix(H_{pre}, H_{new}) \wedge Inv_C \wedge L_{pre} < L_{new} \\ \wedge \neg Comp(H_{pre}, l) \wedge Comp(H_{new}, l) \rightarrow [\omega]\phi \end{array}\right)))\end{array}}{\Longrightarrow [\texttt{await l?; } \omega]\phi}$$

Loop invariants

To verify loops which do not terminate or where the number of iterations is parameterized we use loop invariants. The principle is similar to induction. We show that the invariant holds in the beginning and after each iteration:

$$\text{incorrect}\ \frac{\begin{array}{l}\Longrightarrow Inv_{Loop} \\ \Longrightarrow Inv_{Loop} \wedge texp \rightarrow [p]Inv_{Loop} \\ \Longrightarrow Inv_{Loop} \wedge \neg texp \rightarrow [\omega]\phi\end{array}}{\Longrightarrow [\texttt{while } texp \texttt{ do } p \texttt{ end; } \omega]\phi}$$

The above rule is incorrect because updates (from before the loop) about possibly modified variables in p are available in the lower two branches. Of course, we can anonymize them as in the previous section but first we have to define which variables are modified in a program.

Definition 7.2.24. *A* modifier set *Mod is a set of pairs* $(\phi, f(t_1, \ldots, t_n))$ *with* $\phi \in Formulae$, $f(t_1, \ldots, t_n) \in Term_{Top}$ *and* $f \in FSym_{nr}$.

Because of the formula we can use dynamic logic to describe which variables are modified. Otherwise a description is sometimes impossible. We continue with assigning meaning to a modifier set.

Definition 7.2.25. *A modifier set is correct for program* $p \in \Pi$, *if for all state pairs* $(S_1, p, S_2) \in \rho$*):*

$$(S_1, S_2) \models Mod$$

where $(S_1, S_2) \models Mod$ *iff*

$$\textit{for all } f : A_1, \ldots, A_n \rightarrow A \in FSym_{nr}$$

for all $(d_1, \ldots, d_n) \in \mathcal{D}^{A_1} \times \cdots \times \mathcal{D}^{A_n}$

$\mathcal{I}_1(f)(d_1, \ldots, d_n) \neq \mathcal{I}_2(f)(d_1, \ldots, d_n)$ *implies there is* $(\phi, f(d_1, \ldots, d_n))$ $\in Mod$ *and a variable assignment* β *such that*

$$d_i = val_{S_1,\beta}(t_i) \quad (0 \leq i \leq n) \qquad \textit{and} \qquad S_1, \beta \models \phi$$

We note that a modifier set can be a super set of the actual modified variables.

For a correct loop invariant rule we do not only need to overwrite the modifier set of the loop body but also the histories and the sequence number. The reason is that there could be method calls inside a loop. Now, we assume a modifier set of p is M:

$$\frac{\begin{array}{l}\Longrightarrow Inv_{Loop} \wedge Wf(\mathcal{H}_O, \mathcal{L}) \wedge Cons(\mathcal{H}_O, \mathcal{H}_S) \\ \quad A_M(F_{\mathcal{H}_O}(F_{\mathcal{L}}(\{\mathcal{H}_S := startsAt(H_{new})\} \\ \Longrightarrow \left(\begin{array}{l}(Prefix(H_{pre}, H_{new}) \wedge Wf(H_{new}) \wedge L_{pre} < L_{new} \wedge Inv_{Loop}) \\ \wedge(texp \rightarrow [p]Inv_{Loop} \wedge Wf(\mathcal{H}_O, \mathcal{L}) \wedge Cons(\mathcal{H}_O, \mathcal{H}_S)) \\ \wedge(\neg texp \rightarrow [\omega]\phi)\end{array}\right)))\end{array}}{\Longrightarrow [\texttt{while}\ texp\ \texttt{do}\ p\ \texttt{end;}\ \omega]\phi}$$

The two lower branches of the incorrect loop invariant rule are merged into one due to the fact that they need the same updates. To create a new history prefixing the old one we have to check for well-formedness and consistency. So this is done for the initial loop invariant and after execution of the body.

Method calls

In Creol all methods declared in interfaces are supplied with a contract. A contract contains two formulae where the first is a condition on the in parameters of a method and the second is a condition on the out parameters.

Definition 7.2.26. *A* method contract *is a tuple* $(Pre, Post, mod, label)$ *such that*

- $Pre := Pre' \wedge Invoc(\mathcal{H}_O, label) \in Formulae$ *is the precondition containing only the in parameters as program variables*
- $Post := Post' \wedge Invoc(\mathcal{H}_O, label) \wedge Comp(\mathcal{H}_O, label) \in Formulae$ *is the postcondition containing only the out parameters as program variables*
- Pre', $Post'$ *do not contain histories*[4]

[4] Allowing histories in pre- and post conditions leads to the problem of history composition which is unsolved in this work.

- $mod \in \{box, diamond\}$ *is set to diamond iff the contract requires termination*
- $label \in Term_{Label}$ *describing the label belonging to the method call*

An init method contract *is a tuple* $(Pre_{init}, mod, object)$ *where*

- $Pre_{init} := Pre \wedge New(\mathcal{H}_O, o) \in Formulae$ *is the precondition containing only the class parameters as program variables*
- Pre *does not contain histories*
- $mod \in \{box, diamond\}$ *is set to diamond iff the contract requires termination*
- $object \in Term_{Any}$ *describing the object identifier belonging to the object*

The idea for a method invocation rule is to extend the history by a sending message and to prove the precondition of the corresponding method.

$$\frac{\begin{array}{l}\Longrightarrow Wf(\mathcal{H}_O, \mathcal{L}) \wedge o \neq null \\ \quad \{l := com(this, o, m, \mathcal{L})\}\{\mathcal{H}_S := hist(\mathcal{H}_S, msgInvoc(l, \bar{x}))\}(F_{\mathcal{H}_O} \\ \Longrightarrow \quad (\{\mathcal{L} := \mathcal{L} + 1\} \left(\begin{array}{lll} Wf(H_{new}, \mathcal{L}) \wedge Prefix(H_{pre}, H_{new}) & & \\ \wedge Invoc(H_{new}, l) \wedge \neg Invoc(H_{pre}, l) & \rightarrow & \langle\omega\rangle\phi \\ \wedge \neg Comp(H_{pre}, l) & & \wedge Pre \end{array} \right))\end{array}}{\Longrightarrow \langle \texttt{l!o.m(}\bar{x}\texttt{); } \omega\rangle\phi}$$

To check for consistency we write $o \neq null$ in the upper branch. If the old object history would not be well-formed, the new object history would be malformed as well. Therefore Wf is in the upper branch. The sequence number $\mathcal{L}$ is incremented to ensure that the new label is unique. The manipulation of the object history is not straight forward. An overview is given in figure 7.2. The newly sent message must not occur in the old history but is in the new history. Also the completion message must not been seen before the method invocation.

The completion statement just adds the completion predicate to the object history. The sending history is untouched. For correctness we have to show that the method was actually invoked and the label does not equal *null*.

$$\frac{\begin{array}{l}\Longrightarrow l \neq null \wedge Wf(\mathcal{H}_O, \mathcal{L}) \wedge Invoc(\mathcal{H}_O) \\ \Longrightarrow A_{\bar{y}}(F_{\mathcal{H}_O} \left(\begin{array}{ll} Wf(H_{new}, \mathcal{L}) \wedge Prefix(H_{pre}, H_{new}) & \\ \wedge Comp(H_{new}, l) \wedge Post & \rightarrow \langle\omega\rangle\phi \end{array} \right))\end{array}}{\Longrightarrow \langle \texttt{l?(}\bar{y}\texttt{); } \omega\rangle\phi}$$

$H_{O,1}$	$H_{O,2}$	$H_{O,3}$	$H_{O,4}$
$\neg Invoc(H_{O,1}, l)$	$Invoc(H_{O,2}, l)$	$Invoc(H_{O,3}, l)$	$Invoc(H_{O,4}, l)$
$\neg Comp(H_{O,1}, l)$			$Comp(H_{O,4}, l)$
H_S	$hist(H_S, msgInvoc)$		
$\ldots;$	$l!obj.meth();$	$\ldots;$	$l?(x);$

Figure 7.4: The intention behind the method call rules. Upper lines: Object history where $Prefix(H_{O,i}, H_{O,i+1})$ for $1 \leq i \leq 3$. Middle line: Sending history omitting parameters of $msgInvoc$. Bottom: Code

The *new* statement is very similar to a method call as it implicitly invokes the *init* method of the newly created object. In general a *init* method has to be supplied with a precondition as well.

$$\frac{\begin{array}{l}\Longrightarrow Wf(\mathcal{H}_O, \mathcal{L}) \\ \quad \{o := parent(this, \mathcal{L})\}\{\mathcal{H}_S := hist(\mathcal{H}_S, msgNew(o, \mathcal{L}))\}F_{\mathcal{H}_O}(\\ \Longrightarrow \quad \{\mathcal{L} := \mathcal{L} + 1\}(\left(\begin{array}{ll} Wf(H_{new}, \mathcal{L}) \wedge Prefix(H_{pre}, H_{new}) & \\ \wedge \neg New(H_{pre}, o) \wedge New(H_{new}, o) & \end{array} \rightarrow \begin{array}{l} Pre_{init} \\ \wedge \langle\omega\rangle\phi \end{array}\right)))\end{array}}{\Longrightarrow \langle \texttt{o := new C(}\bar{x}\texttt{); } \omega\rangle\phi}$$

Like for a method call we extend both histories by the corresponding new object message. The object reference is initialized as child of the current object *this*.

7.3 Verifying a Creol program

In order to verify a complete Creol program we have to verify all methods of all classes and then show the composability of those proofs. Hence, the verification consists of a series of proofs of methods.

Verifying a method

In the following we consider a method with the vectors $\bar{x}$, $\bar{y}$ as parameters and *body* a sequence of statements:

```
1 op m (in x; out y) == body
```

Additionally assuming that a contract of the method is given the proof obligations are:

$$\Longrightarrow \{\mathcal{H}_S := startsAt(\mathcal{H}_O)\}(Pre \wedge Inv_C \rightarrow \langle body\texttt{; return}\rangle Post \wedge Inv_C)$$

The *return* statement was never mentioned before. It is an artificial place holder for the end of the method since a empty modality can occur in other

cases as well (e.g. loop invariant rule). When the *return* statement is processed in a proof the completion message of the method is sent.

$$\frac{\begin{array}{l}\Longrightarrow Cons(\mathcal{H}_S,\mathcal{H}_O,\mathcal{L}) \wedge Invoc(\mathcal{H}_O,label_m) \wedge Wf(\mathcal{H}_O,\mathcal{L}) \\ \quad F_{\mathcal{H}_O}(\{\mathcal{H}_S := hist(\mathcal{H}_S, msgComp(label_m,\bar{x}_{out}))\}\{\mathcal{L} := \mathcal{L}+1\} \\ \Longrightarrow \left(\begin{array}{l} Wf(\mathcal{H}_O,\mathcal{L}) \wedge Prefix(H_{pre},H_{new}) \\ \wedge\neg Comp(H_{pre},label_m) \wedge Comp(H_{new},label_m) \end{array} \rightarrow \langle\rangle\phi\right))\end{array}}{\Longrightarrow \langle \texttt{return} \rangle \phi}$$

where $label_m$ is the label identifying the method call processed in the method we verified. Similarly x_{out} are the out parameters of the method.

Investigating an *init* method is slightly different from a usual method:

```
1 op init == body
```

We do not allow release-points or method calls in a init method body (for which the class invariant must be shown), but require that the class invariant is established when *init* terminates.

$$\Longrightarrow \begin{array}{l}\{\mathcal{H}_S := startsAt(\mathcal{H}_O)\}\{\mathcal{L} := 0\} \\ (Pre_{init} \wedge Wf(\mathcal{H}_O,\mathcal{L}) \rightarrow \langle \texttt{body} \rangle Inv_C \wedge Cons(\mathcal{H}_O,\mathcal{H}_S))\end{array}$$

We may assume that the initial *object history* is well-formed as it consists only of the object creation message of *this*.

7.4 Limitations and further work

History For the present the calculus subject to this thesis is a proof of concept but no more. The main problem with the introduced calculus is its complexity which results from the two separate representations of the history.

To simplify the calculus the object history could be discarded and instead all messages could be added in the order of appearance in the code to a list. But thereby the unsolved problem of the composition of histories gets even more complicated since histories from different objects have different orders in its list. To overcome this issue one probably has to use equivalence classes of histories. This will presumably investigated by Owe et al.

A major limitation of this work is that histories are not allowed in pre- and post conditions. Again the reason is the composition of histories. When proving a method there would be two different histories namely the one described by the class invariant and the other one described by the precondition. For a pure object history with no additional predicates this is not a problem. But for instance when counting messages there will be different information about the history from different object viewpoints. Related theoretical investigations have been done in [JO02, JO04].

Statements There are several statements of Creol which are currently not available in the calculus. A minor missing statement is the parallel assignment, e.g. swapping variables by a,b:=b,a, which seems to be fitting to parallel updates on first sight, but as a division by zero could occur on the right hand side it must be fractioned for reasoning about total correctness.

The rule for the non deterministic choice statement [] is incomplete in its current form. For a step towards completeness one can have a look at [Bla08]. However, the look ahead in case of nested non deterministic choice statements complicates the implementation.

In some Creol publications there is an interleave statement ||| which behaves similar to the non deterministic choice statement except for the fact that both branches have to be executed. The execution is interleaved because at each release statement the active branch can be switched. The verification of this statement is a tedious task since the number of combinations increases exponentially with each release-point.

Several abbreviating statements like synchronous method calls are not covered by this work. Nevertheless only the rules rewriting the statements equivalently are absent.

Data structures Sets, lists, tuples and strings are not handled by the current calculus. The reason is that only finite mathematical sets are supported by the KeY system. Strings support is ongoing work in KeY. Lists should be the easiest task as the can be modelled similar to the *hist* function. Tuples lead to a problem with the sorts as they can in general contain different sorts for each entry.

Floating points Floating points in the mathematical sense are currently only partly supported by the KeY system and therefore not subject to this work.

Chapter 8

KeYCreol: A verification tool

KeYCreol is the realization of the logic presented in the two previous chapters. It is part of the KeY project[1].

This chapter starts with the illustration of the architecture of the KeYCreol software in section 8.1. How the software can be obtained and used is the topic of section 8.2. The discussion of inherent limitations and intended features in section 8.3 completes the chapter.

8.1 Architecture

This section provides a non-exhaustive documentation of the key.lang.creol package. Thereby the focus lies on the relation between the different classes. Comments describing the purpose of single functions or attributes are in the code. The documentation was composed in May 2009 and information might have changed already. The package is based on key.lang.common created by Oleg Mürk (documentation in chapter 5 of [Mür08]) which provides interfaces and rudimentary implementations of base classes to be used by non-Java adaptions of KeY.

In the UML diagrams used in this chapter some classes and interfaces lack a field for attributes or methods. Those were left out to improve readability and to shrink the diagrams to a reasonable size.

Generative approach

In comparison to KeY for Java and to KeYC a more generative approach for the implementation of KeYCreol consisting of only a few classes was chosen. Most of the necessary data structures are created on start up. Their layout is defined in the code or parsed in from a text file. The objects composing the

[1] http://www.key-project.org

data structure can be distinguished by a class attribute. This approach was used for the abstract syntax tree, the type hierarchy, the finite state machine creating the abstract syntax tree and the schema variable hierarchy. The idea is not new as the taclets for example are parsed in on startup as well.

The approach leads to a slightly slower program as the dynamic creation consumes some computing power. Additionally, the heavy reuse of classes cuts down some specific methods. E.g. it is not possible to access the condition of an *if* statement by *If.getCondition()* because both the AST nodes holding *if* and its condition are instances of the same class.

On the other hand the program code was significantly simplified using this approach. On June 1st 2009 the different packages had the following numbers in lines of code (leaving out the strategy):

package	lines of code
key.java	49906
key.lang.clang	25155
key.lang.creol	2752

The *java* package is necessarily the biggest as parts of it are reused by the other two packages. However, in the *clang* package the same design principles as in *key.java* are used and it has about nine times more lines of code than *lang.creol*.

One could argue that the package presented in this section does lack some features in contrast to the other packages which leads to the reduction in lines of code.

An analysis of the different implementations of the abstract syntax tree of *clang* and *creol* does not support this argument. A comparison of the packages yields a ratio of one to ten[2] in lines of code considering *lang.creol.program* and *lang.clang.program*. Adding new statements to the supported subset of Creol does not require a single new line in the abstract syntax tree defined in *lang.creol.program* and therefore the ratio will remain constant. For a new statement at most five lines have to be added to the *loader* package where the ration is one to eleven[3]. To summarize a rather pessimistic guess is that the lines of code were reduced by more than 80%.

In addition, the design approach advocated in this thesis is easier to adapt to new features as the layout of any data structure can be changed within a few lines.

Abstract syntax tree

The central work in adapting a new programming or modeling language to the KeY system is the creation of the abstract syntax tree because it is

[2]On June 1st 2009: *lang.clang.program*: 4473 lines verses *lang.creol.program*: 454 lines

[3]June 1st 2009: *lang.clang.loader*: 9557 lines of code. *lang.creol.loader*: 844 lines of code

language specific whereas most logic features can be reused. The abstract syntax tree (AST) is used to represent Creol statements appearing in modalities. A UML diagram of the architecture of *lang.creol.program* can be found in figure 8.1. All classes occurring in an AST of *key.lang.common* have to

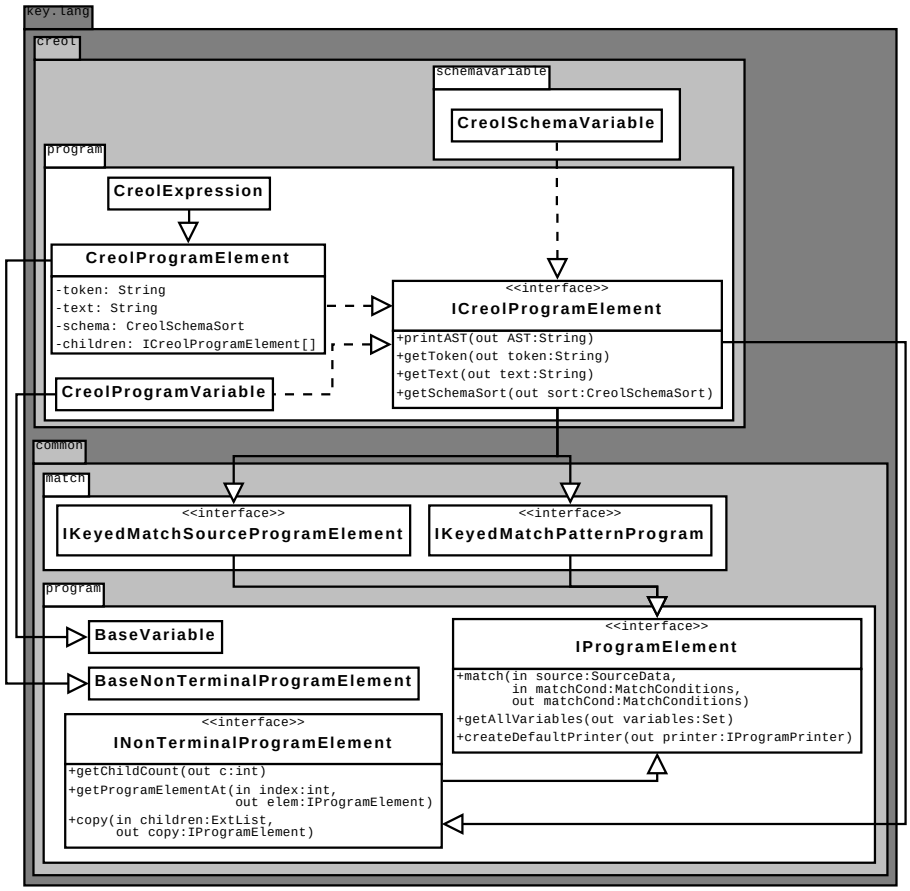

Figure 8.1: UML diagram: key.lang.creol.program

implement *IProgramElement*. In *key.lang.common* the AST is divided in terminal and non-terminal elements. As the first are a special case of the latter only *INonTerminalProgramElement* is inherited by *ICreolProgramElement* which is the interface to be implemented by all classes representing the AST. *INonTerminalProgramElement* provides all necessary methods for traversing an AST. Which taclets are applicable is determined via the functions provided by *IKeyedMatchSourceProgramElement* and *IKeyedMatchPatternProgram*. This process is a comparison of two ASTs, the one of the taclet possibly containing schema variables and the other one of the given modality. The most common AST node is a *CreolProgramElement*. Its text attribute contains the piece of code represented by it. The token attribute is the corresponding token which is reused from jCreol (chapter 4). As the

KeY core checks for some implemented interfaces *CreolProgramVariable* and *CreolExpression* exist (more details in figure 8.6).

There are different walkers wandering the AST. The corresponding UML diagram is in figure 8.2. The *BaseWalker* is supplied *by key.lang.common*. It

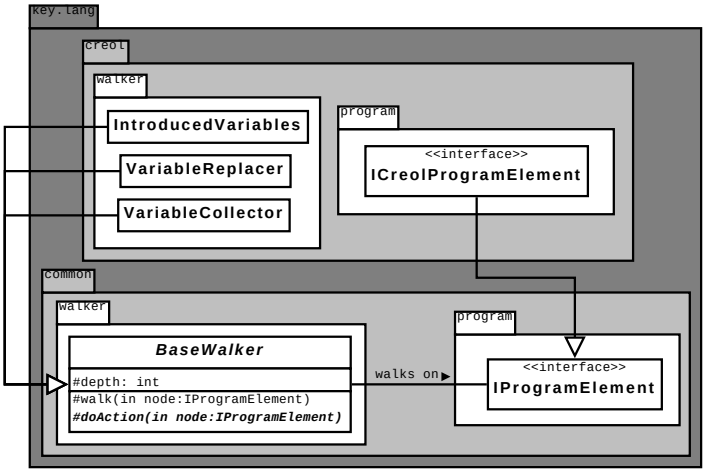

Figure 8.2: UML diagram: key.lang.creol.walker

traverses ASTs implementing *IProgramElement* which is inherited by *ICreolProgramElement* and therefore available in all AST nodes. The different walkers run through the tree and collect information like the used variables or introduced variables. The *VariableReplacer* replaces occurrences of variables by other ones in the AST. There is another walker which exchanges all schema variables of an AST by its current instantiations. However this walker does not inherit *BaseWalker* and it is therefore not in figure 8.2.

The AST is initially created by the loader to be depainted in the next paragraph.

Loader

The loader is one of the most important parts of the package. It invokes an instance of jCreol (see chapter 4) which parses the given statements. Afterwards the resulting jCreol AST is translated to the KeY AST of the previous paragraph. The UML diagram in figure 8.3 provides an overview about the architecture of *key.lang.creol.loader*. The class *CreolLoader* receives all parsing requests and forwards them to an instance of *jCreolExternal* which launches jCreol. Before jCreol finishes it launches another finite state machine (section 4.1) that is passed to it by *CreolLoader*. Its purpose is the creation the KeY AST. Its layout is defined in *ASTLayout*. The different *Action*s are called if the walker handling the finite state machine goes up, down or visits a edge not belonging to the spanning tree (referring to section 4.1). All *Action*s cause certain actions in the *Translator* to build the AST.

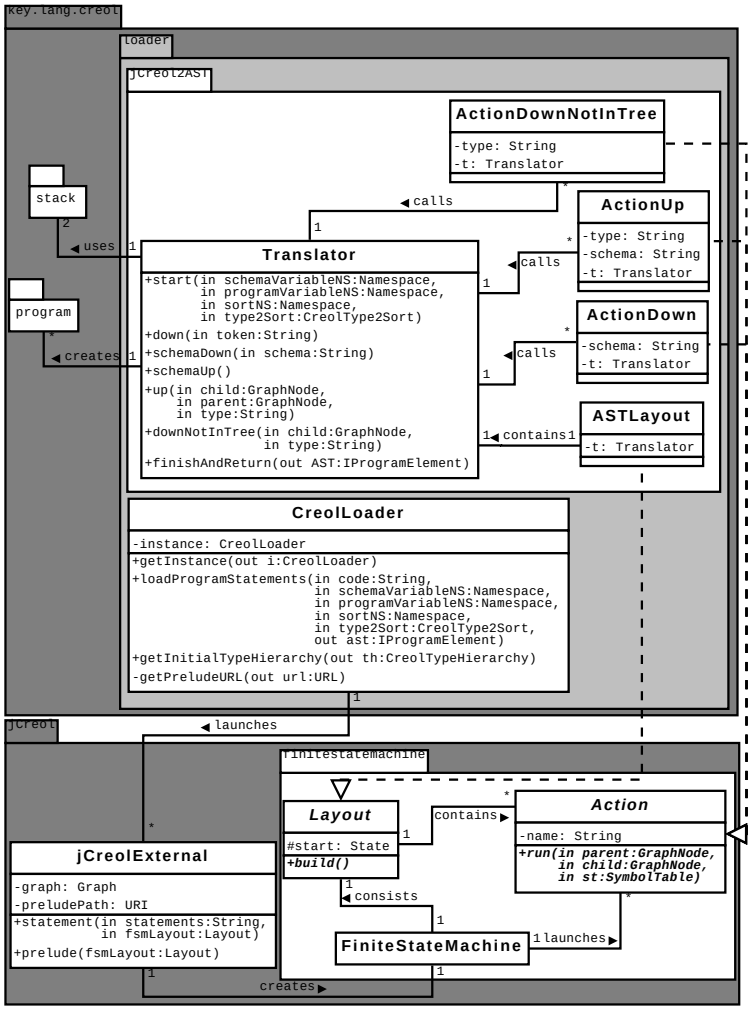

Figure 8.3: UML diagram: key.lang.creol.loader

The construction of the AST proceeds in a bottom up manner. The reason is that AST nodes are generally not modifiable in KeY (e.g. no children can be added after creation of an AST node). The stacks in *Translator* are necessary to store to AST nodes which do not have a parent yet and to store the current schema sort.

lang.creol.loader contains another loader which parses the initial type hierarchy. It follows the same principles as the described one and therefore was omitted.

Taclets and schema variables

The rules of the calculus presented in chapter 7 are represented in the KeY system by a high level language called taclets (except for some spe-

cial cases which exceed their expressive power). This section will provide a brief introduction into the taclet language to explain the role of schema variables. Taclets were first introduced in [Hab00a, Hab00b]. Elaborated explanations about the taclet language used in KeY can be found in chapter 4 of [BHS07].

Taclets The conclusion of a rule is divided into $\backslash find()$ and $\backslash assume()$ where the first contains a logical formula or term and the latter a list of logical formulae or terms. All the formulae or terms have to be matched to make the rule applicable. The difference between both clauses is that a applicable rule will show up in the interface while the cursor is placed on the formula or term matched by $\backslash find()$, but not the other way round. The part matched by $\backslash find()$ can be replaced in using the $\backslash replacewith()$ statement or a logical formula can be added to the sequence with $\backslash add()$. Furthermore a taclet can contain information about variables, e.g. $x \notin fv(\phi)$, expressed by the $\backslash varcond()$ clause.

We presume with an example. Given the rule below:

$$\frac{Invoc(H_1, L), Prefix(H_1, H_2), Invoc(H_2, L) \Longrightarrow}{Invoc(H_1, L), Prefix(H_1, H_2) \Longrightarrow}$$

we can write it as the following taclet:

```
1 Invocation_monoton {
2   \assumes(Prefix(#H1,#H2) ==>)
3   \find(Invoc(#H1,#L) ==>)
4   \add(Invoc(#H2,#L) ==>)
5 };
```

In this case the rule with show up as applicable in the graphical interface if the pointer is above the predicate matching $Invoc(\#H1, \#L)$ If we do not write the sequent arrow ==> in the clauses we obtain a rewriting rule. The taclets can be supplied with the $\backslash heuristics()$ which relates the automatic application of the rule to a specific strategy. In the simplest case a strategy can apply some rules before other. The specific strategy used in KeYCreol is characterized in a following paragraph of this section.

All KeYCreol specific taclets can be found in the folder:
"resources/de/uka/ilkd/key/proof/rules/lang/creol"

Schema variables The formulae or terms to be matched by a taclet can contain schema variable which represent a category of terms, formulae or program statements. This concept ensures that the taclet matches a set of sequents.

Example 8.1.1. In the above listing $\#H1$ is a schema variable matching any term with sort $ObjectHistory$.

To reduce the number of taclets and to limit the number of applicable taclets there are different schema sorts (categories of schema variables) which can be found in figure 8.4. In the figure an arrow from A to B means

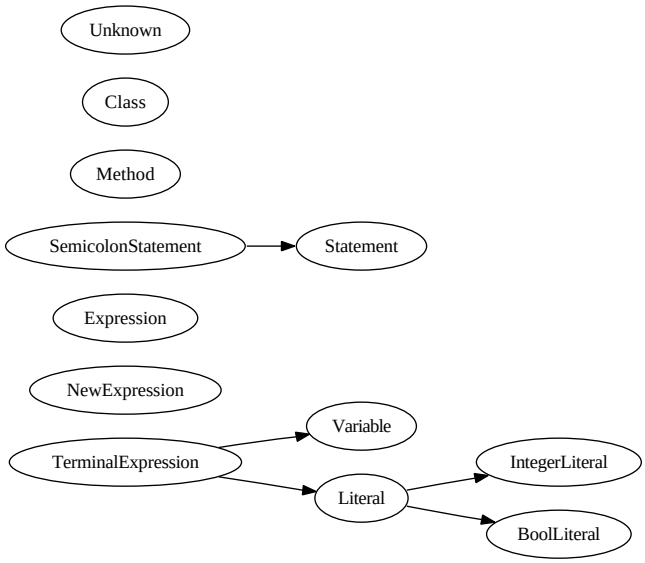

Figure 8.4: Schema sorts

that A will also match B. *Unknown* matches nothing, but exists for initialization purposes. Identifiers of a *Class* or a *Method* are equipped with the corresponding schema sorts. *SemicolonStatement* is the equivalent to ; ω of section 6.4. All statements are matched by *Statement*. *Expression* is the schema sort of all AST nodes which contain a operator connecting expressions. *NewExpression* is applied to expressions containing an object creation. A *TerminalExpression* represents only literals and variables, which is essentially the same as *texp* in section 7.1.

A UML diagram abstracting from the implementation can be looked up in figure 8.5. The concepts of schema variables and its sorts are available in the *key.lang.common* package and inherited or implemented by the Creol classes. A schema variable (*CreolSchemaVariable*) can be part of the AST as it is contained in the code of a modality of a taclet. Therefore *ICreolProgramElement* is implemented by *CreolSchemaVariable*. All the schema sorts are kept in the singleton *CreolSchemaSorts* which the loader uses to obtain a *CreolSchemaSort*. Because *key.lang.common* does distinguish between schema sorts for a variable and other ones (and this is checked in some parts of the KeY system) the empty class *CreolSchemaSortVariable* exists. Each schema variable stores a reference to its sort which provides the methods to match it. An AST node (figure 8.1) matches a schema variable if its schema sort is a sub sort or the same sort as the sort of the schema variable.

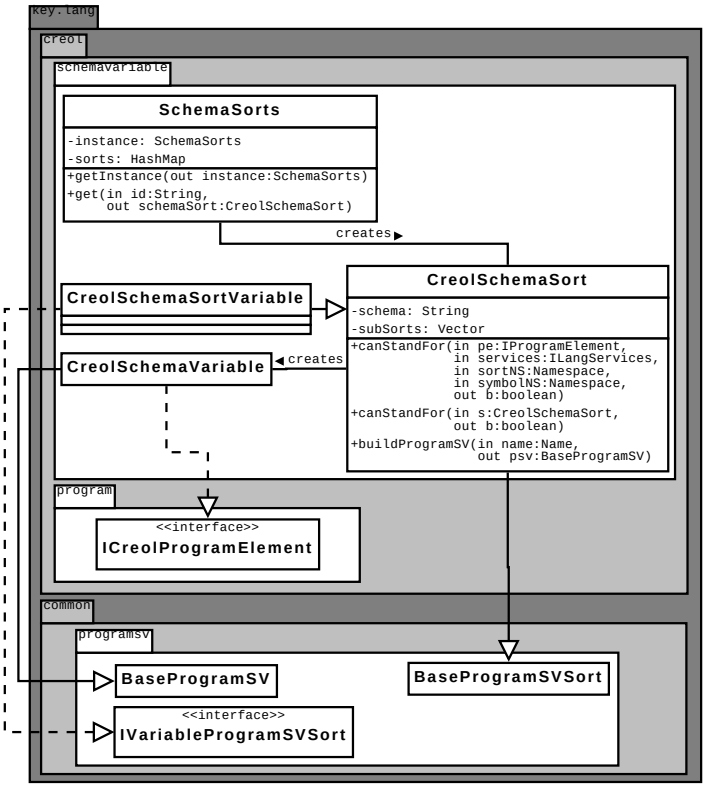

Figure 8.5: UML diagram: key.lang.creol.schemavariable

Types and sorts

The distinction between types (of a Creol program, section 3) and sorts (of the logic, section 6.1) is done in the KeY system as well. Each AST node which is part of an expression contains a sort-type-pair. Other AST nodes are not typed. The associated classes are pictured in the UML diagram in figure 8.6.

The classes *CreolExpression* and *CreolProgramVariable* occur in the AST and both implement the interface *ITypedProgramElement*. This interface provides a function to retrieve the type pair which is represented by the *KeYJavaType* class. *CreolType* inherits the corresponding base class and implements the type interfaces *of key.lang.common* and *key.java.abstraction*. All instances of *CreolType* are unique during run time which is ensured by *CreolTypeHierarchy* being the only class where types can be looked up. *CreolType2Sort* stores an reference to the type hierarchy and uses it to look up the types necessary for the type pairs the loader requires. The loader package creates the initial type hierarchy by parsing *prelude.creol*.

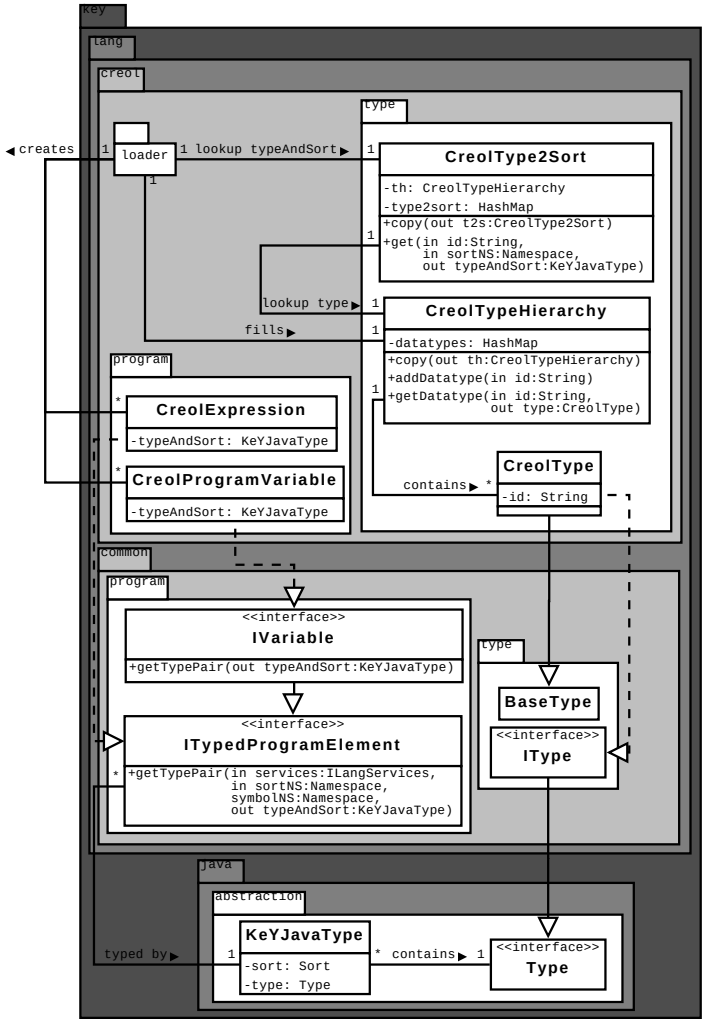

Figure 8.6: UML diagram: key.lang.creol.type

Proof strategy

The current implementation of the strategy is very basic. It comprises only three different levels which differ in their priority. The highest priority have rules simplifying the problem or consuming statements. If the search space has been explored exhaustively the next step is unfolding of predicates, e.g. the class invariant, that hides a formula. Afterwards rules with higher prior are usually applicable again. In the case of having tried all rules of both described levels the well-formed predicate is used to create more predicates. In well-formed histories a invocation predicate can be generated for a given completion predicate, for instance.

8.2 Using KeYCreol

KeYCreol is available from www.key-project.org upon request. It is licensed under the GNU general public license.

The current version is a prototype implying that it does provide neither a full coverage of all Creol statements nor freedom from error.

To compile and run KeYCreol, besides the requirements of KeY 1.4, jCreol in form of a *jar* file is a prerequisite. An appropriate script which creates and copies the file into the library folder of KeY ships with jCreol.

To invoke the interactive prover execute the script *runProver* residing in the folder *bin* with *creol* as the first argument. An optional further argument is the path of a *key* file containing a proof obligation.

The files forming the case studies of this thesis and embodying the rules concerning histories are in the path:
"examples/lang/creol/hardCodedRules/"
During the development some achievements were captured in test cases for the system. These files are in the folder: "examples/_textcase/lang/creol/"

For further information consult the read-me or write to the KeY project.

Development

As noted before there are test cases which can be run after development steps to check them against bugs. The test cases are runnable automatically using the *runAllProofs* script in the *bin* folder.

Turning the settings of log4j[4] to debug for *key.lang.creol* will produce detailed logs. The logger settings are read from a file named *logger.props* in the *.key* folder of your home directory.

The system has the capability of printing the layout of the used finite state machines to a *graphviz* file what from a picture of the layout can be generated. To enable this feature a line has to be uncommented in the *CreolLoader* file.

In general all methods invoked by parts of the system outside of the *lang.creol* package are logged. Missing implementations for additional features should result in an adequate log entry.

The added extensions to the KeY system try within their limits to check input from other parts of the system by assertions. These restrictions might be to harsh in future uses.

8.3 Limitations and further work

This section describes implementation features which are not available in current versions of KeYCreol, but it does not mention missing features re-

[4] Apache Logging Services Project - Apache log4j

lated to theoretical issues. Those are listed in section 7.4.

Loading complete Creol programs The main implementation issue left to do is the loading of complete Creol programs, which should manage open proof obligations for each method of each class for example.

So far it is only possible to load the bodies of methods. Hence, in order to verify a program for each method a separate proof obligation has to be created by hand and some rules have to be adapted. This is because release-points overwrite all class attributes (see section 7.2) and it is impossible to write taclets matching on a list of unknown size. Therefore all rules concerning the history have to be implemented as built-in-rules.

The loading mechanism for complete Creol programs is prepared in the code, but the corresponding layout of the finite state machine for building up classes and interfaces is missing.

Having established a loading procedure for complete programs the Creol compiler could be invoked on given Creol programs to ensure that incorrect programs are rejected.

Consistency of the AST The design of the AST nodes as one single class has the disadvantage that node specific consistency checks are difficult to do. A solution would be to supply the finite state machine creating the AST with checks on the transitions going up in the graph. For each kind of AST node there is a state so this would be the right part of the code to use. A simple check could be the expected number of children, their Schema sorts and in case of expressions their type.

Missing statements Some statements like synchronous method calls cannot be parsed by KeYCreol right now. This is a minor problem as the architecture supports adding statements by only few lines of code.

Pretty-printing The current version does not incorporate any pretty-printing. All newly created variables and constants in a proof are named after a standard hardly readable scheme. A first step could be to generate names according to the sort of the term which would improve readability.

A more advanced topic is the pretty printing of predicates guaranteeing the existence of a message in a history. Since these predicates are monotonous on prefixed histories, a number of them is collected during a proof. In that case an analysis which of them are useful should lead to a procedure of hiding unnecessary information.

Additionally, the predicate names could be pretty-printed. For instance, the *Prefix* predicate could be abreviated by $\leq$ and thereby infix form could be used.

Proof strategy The rudimentary proof strategy developed within this thesis can be extended to make large automatic proofs feasible. To accomplish this additional characteristic proofs of Creol programs must be done which in a second step could be analyzed to create a better heuristic to guide the automated search.

Taclet language There is a mechanism for writing taclets matching both the box and the diamond as modalities using schema variables. This reduces the number of taclets by nearly one half as one would need two taclets for all statements otherwise. One could introduce a similar tool for logical operators like $-$ or $+$. This would allow handling several integer rules with one taclet.

Currently there is no possibility to match on a variable of a specific type (or of any type but one). However, it is possible to write logical rules which are only valid for expressions of a certain type. Again, such a feature would reduce the number of taclets.

Chapter 9

Case studies

This chapter applies the theory derived in chapter 7 to two different examples using the verification tool described in chapter 8. Thereby the focus lies on Creol specific issues leaving out well-understood issues like integer calculus. First, we will consider a simple bank account where the interaction between different methods is described by a simple class invariant (adapted from [Bla08], section 7.1). The second example is about a buffer which can be written and read from (code inspired from [Bla08], section 4.1). The first instance is provable fully automatically whereas the later needs a few interactions.

For simplicity the *run* methods of the used classes are omitted since they are empty.

9.1 Bank account

An bank account should have at least two different operations, namely depositing money on the account and paying a bill by doing a transaction. To model this in Creol we define an interface containing these operations.

```
1 interface BankAccount
2 begin
3   with Any
4     op deposit(in am : Int)
5     op payBill(in am : Int)
6 end
```

Listing 9.1: Bank account interface

What happens if somebody tries to deposit a negative amount of money? We want to avoid this situation so we define the pre-condition of the *deposit* method as:

$$Pre_{deposit} := am \geq 0$$

There is no return value, so we do not use a post-condition. The same issue applies to *payBill* therefore we use a similar pre-condition and no post-condition:

$$Pre_{payBill} := am \geq 0$$

Let us turn to the implementation of the interface. We need a class attribute *bal* for saving the balance of the account. The methods of the interface modify the balance. A call of *deposit* simply adds the amount to the balance. The *payBill* method checks whether there is enough money on the account before subtracting. If not the payment will be delayed. Finally, the balance is initialized by zero in the *init* method.

```
1 class BankAccount implements BankAccount
2 begin
3   var bal : Int;
4   op init == bal := 0
5   with Any
6     op deposit(in am : Int) == bal := bal + am
7     op payBill(in am : Int) == await bal >= am ;
8                                bal := bal - am
9 end
```

Listing 9.2: Bank account class

We note that because of the delayed payments there are no negative values possible for the balance. Additionally, the balance should equal the sum of the amounts deposited minus the sum of the amounts of payed invoices. We express this in the class invariant:

$$Inv_C := bal \geq 0 \wedge Sum_O(\mathcal{H}_O) = bal$$

The predicate Sum_O needs to be defined. As we have the notion of object and sending histories we need two different predicates $Sum_O : ObjectHistory$ and $Sum_S : SendingHistory$ to avoid sort errors. To handle the Sum_S predicate of a given history we need special rewriting rules. If a completion message of *deposit* finishes the sending history the amount is added to the sum:

$$\mathsf{compDeposit}\ \frac{Sum_S(H_S) + am}{Sum_S(hist(H_S, msgComp(label(O, this, deposit, I), am)))}$$

In case of a *payBill* completion message exactly the opposite is done:

$$\mathsf{compPayBill}\ \frac{Sum_S(H_S) - am}{Sum_S(hist(H_S, msgComp(label(o, this, payBill, I), am)))}$$

Other messages are handled without changing the sum:

$$\text{startsAt}\ \frac{Sum_O(H_O)}{Sum_S(startsAt(H_O))}$$

$$\text{invocation}\ \frac{Sum_S(H_S)}{Sum_S(hist(H_S, msgInvoc(L, D)))}$$

$$\text{new}\ \frac{Sum_S(H_S)}{Sum_S(hist(H_S, msgNew(L, D)))}$$

$$\text{compOther}\ \frac{Sum_S(H_S)}{Sum_S(hist(H_S, msgComp(label(O, O_2, Meth, I), D)))}$$

In the above rules $H_S \in Term_{SendingHistory}$, $H_O \in Term_{ObjectHistory}$, $L \in Term_{Label}$, $D \in Term_{Data}$, $O, O_2 \in Term_{Any}$, $I, am \in Term_{Int}$ and $Meth \in Term_{Method}$ such that $\mathcal{I}(Meth) \neq \mathcal{I}(payBill)$, $\mathcal{I}(Meth) \neq \mathcal{I}(deposit)$.

There are three open goals to verify which correspond to the methods of the class in listing 9.2 which are automatically provable by the KeY system and are available in the KeYCreol sources.

9.2 Buffer

A buffer is used to store content to be sent from one to another object. We distinguish between the *writable buffer* interface and the *readable buffer* interface:

```
1 interface WritableBuffer
2 begin
3   op put(in x : Any)
4 end
```

Listing 9.3: WritableBuffer interface

The *writeable buffer* interface provides a *put* which intuitively allows to put an element into the buffer. We want to be sure that no empty elements are in the buffer. Therefore we require the precondition:

$$Pre_{put} := \neg x \doteq null$$

From classes implementing the *readable buffer* interface we can obtain elements stored in the buffer by using the method *get*:

```
1 interface ReadableBuffer
2 begin
3   op get(out y : Any)
4 end
```

Listing 9.4: ReadableBuffer interface

As we assume non-emptiness for all elements in the buffer we can guarantee that the received element is not *null*:

$$Post_{get} := \neg y \doteq null$$

Let us proceed with a simple implementation of a buffer storing at maximum one element. The class implements both interfaces. It has a attribute where the element is stored. Both methods start with an *await* statement so that they continue execution only if the buffer is ready.

```
1 class Buffer implements WriteableBuffer, ReadableBuffer
2 begin
3   var cell : Any;
4   op init == skip
5   with Any
6     op put(in x : Any) == await cell=null; cell := x
7     op get(out y : Any) == await cell!=null;
8                            y := cell; cell := null
9 end
```

Listing 9.5: Buffer class

Considering the listing we realize that *put* and *get* will send a completion message alternatingly beginning with *put*. In other words the sending history is always a prefix of $(PUT\ GET)^*$ (w.r.t completion messages).

Therefore we introduce the predicate $sPrefix$:

- $sPrefix : SendingHistory \in PSym$ with $\mathcal{I}_0(sPrefix) =$

$$\left\{ h \in sendHistDom \,\middle|\, h \in \left(\begin{array}{l} \langle comp, \langle o, \mathcal{I}_0(this), put, i \rangle, \rangle^\frown \\ \langle comp, \langle o, \mathcal{I}_0(this), get, i \rangle, d \rangle \end{array} \right)^* \right\}$$

Encoding this in a class invariant is not sufficient as we need to know what the last message in the history was. If the *cell* does not equal *null* it must have been a *put* completion message.

$$(\neg cell \doteq null) \leftrightarrow (sPut(\mathcal{H}_S) \wedge \neg sGet(\mathcal{H}_S) \wedge \neg sEmpty(\mathcal{H}_S))$$

where $sPut(\mathcal{H}_S)$ is a predicate interpreted as true iff $\mathcal{H}_S$ ends with a *put* completion message, $sGet(\mathcal{H}_S)$ is the analogous for *get* completion messages and $sEmpty(\mathcal{H}_S)$ means that the sending history is empty.

Formally we define the predicates as:

- $sPut : SendingHistory \in PSym$ with $\mathcal{I}_0(sPut) =$

$$\left\{ h \in sendHistDom \,\middle|\, \begin{array}{l} \exists h_1 \in sendHistDom : \\ h = h_1^\frown \langle comp, \langle o, \mathcal{I}_0(this), put, i \rangle, \rangle \end{array} \right\}$$

- $sGet : SendingHistory \in PSym$ with $\mathcal{I}_0(sGet) =$

$$\left\{ h \in sendHistDom \;\middle|\; \begin{array}{l} \exists h_1 \in sendHistDom : \\ h = h_1^\frown \langle comp, \langle o, \mathcal{I}_0(this), get, i \rangle, d \rangle \end{array} \right\}$$

- $sEmpty : SendingHistory \in PSym$ with
  $\mathcal{I}_0(sEmpty) = \{h \in sendHistDom | h = \epsilon\}$

The intended interplay between the predicates is given in figure 9.1.

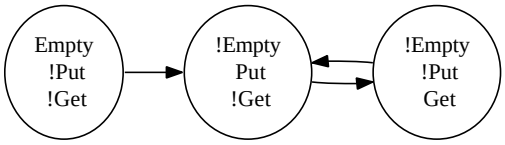

Figure 9.1: Finite state machine relating the predicates with the sending history. The exclamation mark means negation. The class is in the left state after executing *init*. The middle state holds after finishing the *put* method. We note that there are two transitions yielding the middle state which results in the part of the class invariant where *cell* is not null. The history ends in the right state after completing *get*.

If the *cell* equals null it is a bit more complicated: Either a *get* completion message completes the sending history or it is the empty history.

$$(cell \doteq null) \leftrightarrow (\neg sPut(\mathcal{H}_S) \wedge (sGet(\mathcal{H}_S) \not\leftrightarrow sEmpty(\mathcal{H}_S)))$$

Let us compose the class invariant as:

$$\begin{array}{ll} & ((\neg cell \doteq null) \leftrightarrow (sPut(\mathcal{H}_S) \wedge \neg sGet(\mathcal{H}_S) \wedge \neg sEmpty(\mathcal{H}_S))) \\ Inv_{C,S} := & \wedge((cell \doteq null) \leftrightarrow (\neg sPut(\mathcal{H}_S) \wedge (sGet(\mathcal{H}_S) \not\leftrightarrow sEmpty(\mathcal{H}_S)))) \\ & \wedge sPrefix(\mathcal{H}_S) \end{array}$$

For technical reasons the class invariant $Inv_{C,S}$ can only be applied if we have to establish the class invariant. For assuming the class invariant we need $Inv_{C,O}$ which contains exactly the same predicates which talk about the object history, though.

The verification of the *Buffer* class is essentially a mathematical induction, where the begin is the proof of the *init* method and the steps are the proofs of *put* and *get*.

We are yet missing the rules involving the predicates we introduced in this section. For the following paragraph we use the notations $I \in Term_{Int}$, $H_S \in Term_{SendingHistory}$, $O, D \in Term_{Any}$ and $H_O \in Term_{ObjectHistory}$. The $sPut$ predicate is true if the sending history ends with a *put* completion message:

$$\frac{true}{sPut(hist(H_S, msgComp(O, this, put, I), \bar{D}))}$$

and false if a *get* completion message concludes the sending history:

$$\frac{false}{sPut(hist(H_S, msgComp(O, this, get, I), \bar{D}))}$$

If the sending history just relates to the object history we check the object history:

$$\frac{oPut(H_O)}{sPut(startsAt(H_O))}$$

The rules for *sGet* are inverted. The last message being a *put* completion message cause it to be false:

$$\frac{false}{sGet(hist(H_S, msgComp(O, this, put, I), \bar{D}))}$$

The *get* completion message evaluates it to true:

$$\frac{true}{sPut(hist(H_S, msgComp(O, this, get, I), \bar{D}))}$$

Again, if the sending history is relating to an object history we replace the predicates:

$$\frac{oGet(H_O)}{sGet(startsAt(H_O))}$$

Handling the *sEmpty* predicate is false for non-empty histories:

$$\frac{false}{sEmpty(hist(H_S, Msg))} \qquad \frac{oEmpty(H_O)}{sEmpty(startsAt(H_O))}$$

The $sPrefix$ predicate has to check the alternating sequence of completion messages. To accomplish this we need two additional predicates $sPrefixPut$, $sPrefixGet$ which are prefixes expecting the corresponding completion message:

- $sPrefixGet : SendingHistory \in PSym$ with $\mathcal{I}_0(sPrefixGet) =$

$$\left\{ h \in sendHistDom \;\middle|\; \begin{array}{l} h \in \left(\begin{array}{l} \langle comp, \langle o, \mathcal{I}_0(this), put, i \rangle, \rangle^\frown \\ \langle comp, \langle o, \mathcal{I}_0(this), get, i \rangle, d \rangle \end{array} \right)^* \\ \text{and } \exists h_1 \in sendHistDom : \\ h = h_1^\frown \langle comp, \langle o, \mathcal{I}_0(this), get, i \rangle, \rangle \end{array} \right\}$$

- $sPrefixPut : SendingHistory \in PSym$ with $\mathcal{I}_0(sPrefixPut) =$

$$\left\{ h \in sendHistDom \;\middle|\; \begin{array}{l} h \in \left(\begin{array}{l} \langle comp, \langle o, \mathcal{I}_0(this), put, i \rangle, \rangle^\frown \\ \langle comp, \langle o, \mathcal{I}_0(this), get, i \rangle, d \rangle \end{array} \right)^* \\ \text{and } \exists h_1 \in sendHistDom : \\ h = h_1^\frown \langle comp, \langle o, \mathcal{I}_0(this), put, i \rangle, \rangle \end{array} \right\}$$

The $sPrefix$ predicate is exchanged for the associated predicate. If the history ends with a completion message of *put* it is *sPrefixGet*:

$$\frac{sPrefixGet(H_S)}{sPrefix(hist(H_S, msgComp(com(O, this, put, I), \bar{D})))}$$

And the other way round:

$$\frac{sPrefixPut(H_S)}{sPrefix(hist(H_S, msgComp(com(O, this, get, I), \bar{D})))}$$

The relation of two histories translates the predicate accordingly:

$$\frac{oPrefix(H_O)}{sPrefix(startsAt(H_O))}$$

$sPrefixPut$ and $sPrefixGet$ alternate the sending history:

$$\frac{sPrefixGet(H_S)}{sPrefixPut(hist(H_S, msgComp(com(O, this, put, I), \bar{D})))}$$

$$\frac{sPrefixPut(H_S)}{sPrefixGet(hist(H_S, msgComp(com(O, this, get, I), \bar{D})))}$$

and finish with supplying the correct state:

$$\frac{oPrefix(H_O) \land (\neg oPut(H_O) \land (oGet(H_O) \nleftrightarrow oEmpty(H_O)))}{sPrefixGet(startsAt(H_O))}$$

$$\frac{oPrefix(H_O) \land (oPut(H_O) \land \neg oGet(H_O) \land \neg oEmpty(H_O))}{sPrefixPut(startsAt(H_O))}$$

The three methods of *Buffer* are provable interactively by KeYCreol and reside in the repository with its saved proofs.

Chapter 10

Conclusions

This thesis presented the theory and a prototypic implementation of a verification system for Creol rooted in a theoretically well-founded environment based on the KeY software and its dynamic logic. Its functionality was successfully shown in two Creol-specific examples where statements about the interaction of different threads inside an object could be proved close to automatically. It shows that all the objectives of section 1.1 have been met and this work represents to my best knowledge the first time such proofs were achieved by a verification tool for Creol.

The axiomatization of the communication history proved itself to be a challenging but feasible task. In this work, a hybrid approach between explicit and implicit representations of the history capturing all communication knowledge from the perspective of a single object was instantiated. The drawback of frequent consistency checks between both representations can only be bypassed by restricting the calculus to one representation with its associated incompleteness in terms of (un)certainty in the history. As such a decision is still desirable, the next step must be a rigorous formalization of the composition of communication histories where a trade between treatment of the ambiguity in the representation and dealing with it in the composition can simplify matters.

While working on the case-studies the know fact was reinforced that automated program verification is more reliable than proofs of correctness by hand. Humans tend to overlook certain details of the problem. On the contrary, a computer strictly explores all possibilities defined by the underlying calculus. Collaterally, mechanic proofing of apparently small problems turns out to be a cumbersome task. Using an automated prover, one ends up in an development cycle for the verification conditions which starts with the refinement of previous thoughts and ends with a proof attempt. It is thus highly beneficial to know the underlying proof system as it helps to find the flaw in a failed proof attempt and therefore decreases the time used by a development cycle.

The main open issue to be investigated after this work is the composition of communication histories. Without the composition, method contracts cannot embody statements about histories which does not affect correctness but completeness of the calculus. Such updated contracts would enable reasoning about methods to be called only in particular contexts of the communication history. Additionally, the correctness of the total system is not guaranteed by the verification of the methods of all classes without a proof of composability since intrinsic deadlocks or race conditions might still remain.

There have been theoretical investigations in [JO02, JO04], but they need to be extended to the framework presented within this thesis.

Further detail in regards to discussions on the inherent limitations of this work and suggestions for advancing beyond it can be found in the sections 4.3, 7.4 and 8.3.

Appendix A

Creol Grammar

```
1  start -> declaration*
2
3  declaration -> interface_decl | datatype_decl
4    | function_decl | class_decl
5
6  class_decl -> CLASS CLASS_IDENTIFIER
7    var_decl_no_init_argument? super_decl_rw
8    pragma_rw BEGIN class_attributes class_methods END
9
10 class_attributes -> (attribute SEMICOLON?)*
11
12 pragma_rw -> pragma*
13
14 pragma -> PRAGMA CLASS_IDENTIFIER parameter_list?
15   SEMICOLON?
16
17 super_decl_rw -> super_decl*
18
19 super_decl -> (CONTRACTS | IMPLEMENTS | INHERITS)
20   class_list
21
22 class_list -> (class_element (KOMMA class_element)*)
23
24 class_element -> CLASS_IDENTIFIER parameter_list?
25
26 attribute -> VAR var_decl_list
27
28 class_methods -> ( anon_with_def? with_def*)
29
30 anon_with_def -> method_with_body+ invariant*
```

```
31
32 with_def -> WITH^ type method_with_body+ invariant*
33
34 var_decl_no_init_argument -> LPAREN
35   var_decl_no_init_list? RPAREN
36
37 parameter_list -> LPAREN expr_list? RPAREN
38
39 interface_decl  : INTERFACE CLASS_IDENTIFIER
40   var_decl_no_init_argument? interface_super pragma_rw
41   BEGIN invariant? interface_methods END
42
43 interface_methods -> with_decl*
44
45 interface_super -> interface_inherits*
46
47 interface_inherits -> INHERITS class_list*
48
49 datatype_decl -> DATATYPE type from? pragma*
50
51 from -> FROM type_list?
52
53 function_decl -> FUNCTION
54   id_or_op var_decl_no_init_argument? COLON type
55   pragma* function_body
56
57 function_body -> DOUBLE_EQUAL ( expr | extern )
58
59 id_or_op -> (IDENTIFIER | operator)
60
61 operator -> IN | NOT | EQUIVALENCE | IMPLICATION | XOR
62   | OR | AND | EQUALITY | INEQUALITY | comp_op
63   | PROJECTION | CONCAT | PREPEND | APPEND | PLUS
64   | MINUS | MULT | DIV | MOD | POW | NUMBER_SIGN
65
66 with_decl -> WITH type method_decl+ invariant*
67
68 method_with_body -> OP^ IDENTIFIER method_param?
69   requires? ensures? pragma* method_body
70
71 method_body -> DOUBLE_EQUAL ( decl_statement | extern )
72
73 extern -> EXTERNAL STRING
74
```

```
75  method_decl -> (OP IDENTIFIER method_param? requires?
76    ensures? pragma*)
77
78  requires -> REQUIRES expr
79
80  ensures -> ENSURES expr
81
82  method_param -> (LPAREN method_param_in?
83    method_param_out? RPAREN)
84
85  method_param_in -> (IN? var_decl_no_init_list)
86
87  method_param_out -> (SEMICOLON? OUT
88    var_decl_no_init_list)
89
90  var_decl_list -> var_decl (KOMMA var_decl)*
91
92  var_decl -> var_decl_no_init (ASSIGN expr_or_new)?
93
94  var_decl_no_init -> expr COLON type
95
96  var_decl_no_init_list -> var_decl_no_init
97    (KOMMA var_decl_no_init)*
98
99  start_statement -> (statement | attribute
100   ( SEMICOLON^ decl_statement)? )?
101
102 decl_statement -> statement | attribute SEMICOLON
103   decl_statement
104
105 statement -> choice_stmt (INTERLEAVE^ statement)?
106
107 choice_stmt -> seq_stmt (BOX choice_stmt)?
108
109 seq_stmt -> basic_stmt (SEMICOLON seq_stmt )?
110
111 basic_stmt-> small_stmt | control_flow_stmt | expr_stmt
112
113 expr_stmt-> expr ((KOMMA expr)* ASSIGN expr_or_new_list
114   | (EXCLENATION_MARK expr))?
115
116 expr_or_new-> expr | NEW (CLASS_IDENTIFIER | SCHEMAVAR)
117   parameter_list?
118
```

```
119 expr_or_new_list -> expr_or_new
120   (KOMMA expr_or_new_list)?
121
122 control_flow_stmt -> while_stmt | do_while_stmt
123
124 while_stmt -> WHILE expr invariant? measure?
125   DO statement END
126
127 measure -> MEASURE expr BY id_or_op
128
129 invariant -> INV expr
130
131 do_while_stmt -> DO statement invariant? measure?
132   WHILE expr
133
134 small_stmt -> SKIP | RELEASE | BLOCK | RETURN
135   | BEGIN statement END | ( ASSERT | AWAIT |
136   EXCLENATION_MARK | POSIT | PROVE ) expr
137
138 expr ->  equivalence_expr (IN equivalence_expr)*
139
140 equivalence_expr -> impl_expr (EQUIVALENCE impl_expr)*
141
142 impl_expr -> xor_expr (IMPLICATION xor_expr)*
143
144 xor_expr -> or_expr (XOR or_expr)*
145
146 or_expr -> and_expr (OR and_expr)*
147
148 and_expr -> not_expr (AND and_expr)?
149
150 not_expr -> (NOT not_expr) | equals_expr
151
152 equals_expr -> comp_expr ((EQUALITY|INEQUALITY)
153   comp_expr)?
154
155 comp_expr -> projection_expr (comp_op projection_expr)?
156
157 comp_op -> LESS_THAN | GREATER_THAN | LESS_OR_EQUAL
158   | GREATER_OR_EQUAL
159
160 projection_expr-> concat_expr (PROJECTION concat_expr)*
161
162 concat_expr -> prepend_expr (CONCAT prepend_expr)*
```

```
163
164 prepend_expr -> append_expr (PREPEND prepend_expr)?
165
166 append_expr -> add_expr (APPEND add_expr)*
167
168 add_expr -> mult_expr ((PLUS|MINUS) mult_expr)*
169
170 mult_expr -> power ((MULT | DIV | MOD) power)*
171
172 power -> factor (POW factor)*
173
174 factor -> MINUS factor | NUMBER_SIGN factor
175   | atom_with_trailer
176
177 atom_with_trailer: atom (
178  ((AT type) | DOT method_call (as_type)? | bounds
179   | arg_list (as_type)?
180   | (QUESTION_MARK lhs_list_paren? ) | AS type
181  ) trailer_rest? )?
182
183 trailer_rest -> (DOT (IDENTIFIER | SCHEMAVAR) arg_list
184   (as_type)?) | bounds
185
186 method_call -> IDENTIFIER arg_list | SCHEMAVAR arg_list
187
188 atom -> TRUE | FALSE | INTEGER | FLOAT | STRING
189   | IDENTIFIER | THIS | CALLER | NULL | NIL | NOW
190   | HISTORY | LBRACK expr_list? RBRACK
191   | LBRACES expr_in_braces RBRACES
192   | LPAREN expr_in_paren? RPAREN
193   | IF expr THEN statement (ELSE statement)? END
194   | SCHEMAVAR
195
196 arg_list -> (LPAREN (expr_list)? (SEMICOLON lhs_list?)?
197   RPAREN)
198
199 bounds -> BOUND_L CLASS_IDENTIFIER
200     (BOUND_R CLASS_IDENTIFIER)? arg_list (as_type)?
201   | BOUND_R CLASS_IDENTIFIER
202     (BOUND_L CLASS_IDENTIFIER)? arg_list (as_type)?
203
204 lhs_list_paren -> LPAREN lhs_list? RPAREN
205
206 lhs_list -> lhs (KOMMA lhs)*
```

```
207
208 lhs -> IDENTIFIER (AT type)? | UNDERSCORE as_type
209   | SCHEMAVAR
210
211 as_type -> AS type
212
213 expr_in_paren -> expr_list | (FORALL|EXISTS|SOME)
214   var_decl_no_init COLON expr
215
216 expr_list -> (expr (KOMMA expr)*)
217
218 expr_in_braces -> ( setmaker | VERTICAL binding_list
219   VERTICAL)?
220
221 setmaker -> expr (
222   ((KOMMA expr)*) | (COLON expr VERTICAL expr))
223
224 binding_list -> binding (KOMMA binding)*
225
226 binding -> expr MAPSTO expr
227
228 type -> CLASS_IDENTIFIER type_list_brackets?
229   | type_bracket | APOSTROPHE IDENTIFIER
230
231 type_bracket -> LBRACK type_list RBRACK
232
233 type_list_brackets -> LBRACK type_list? RBRACK
234
235 type_list -> (type (KOMMA type)*)
```

Listing A.1: Creol Grammar

Appendix B

Glossary of symbols

B.1 Sets

Notation	Meaning
$\in$	is a member of
$\notin$	is not a member of
$\emptyset$	empty set with no members
$\{a\}$	set with one member a
$\{a, b, c\}$	set with members a, b and c
$\{x \mid F(x)\}$	set of all x such that $F(x)$ holds
$A \subseteq B$	A subset of B
$A \cup B$	union of A and B
$A \cap B$	intersection of A and B
$A \backslash B$	all members of A which are not in B
$\|A\|$	number of members in A
$A \times B$	$\{(a, b) \| a \in A, b \in B\}$
$\bar{a}$	tuple
A^*	all tuples with elements of A
$\mathbb{Z}$	integers
$\mathbb{N}$	$\{1, 2, 3, \dots\}$

B.2 Graphs

Term	Meaning
graph	set of nodes with connections between the nodes
directed graph	graph where the connections are not symmetric
path	a sequence of nodes following connections in a graph
acyclic	no paths forming a circle exist
tree	acyclic graph where each node is reachable by exactly one path
leaf	node in tree with no leaving connections
spanning tree	subgraph being a tree and containing all nodes

B.3 Sorts

Notation	Meaning
$\mathcal{T}$	all sorts
$\mathcal{T}_d$	dynamic sorts for domain elements
$\mathcal{T}_s$	static sorts for terms
$\sqsubseteq$	subsort relation between two sorts
$\sqsubseteq^0$	direct subsort
$\sqcup$	greatest lower bound of two sorts
$\sqcap$	least upper bound of two sorts

The sort hierarchy containing all sorts is pictured in the figures 6.1, 7.2 and 7.3.

B.4 Syntax

Notation	Meaning
Σ	signature containing symbols
$VSym$	set of variable symbols
$FSym$	set of function symbols
$FSym_r$	set of rigid function symbols
$FSym_{nr}$	set of non-rigid function symbols
$PSym$	Set of predicate symbols
α	sort function of symbols
$Term_A$	set of terms of sort A
$Formulae$	set of logical formulae
ϕ, ψ	formulae
$\{a := 1\}$	functional update
$\|\|$	parallel update
$Updates$	set of updates
p, q	programs
Π	set of normalized Creol programs
$\langle p \rangle$	diamond modality
$[p]$	box modality
$\neg\phi$	ϕ is not true
$\phi \wedge \psi$	both ϕ and ψ are true
$\phi \vee \psi$	both ϕ or ψ or both are true
$\phi \rightarrow \psi$	ϕ or ψ or both are true
$\phi \leftrightarrow \psi$	$(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$, both true or both false
$\phi \not\leftrightarrow \psi$	$(\neg\phi \leftrightarrow \psi)$, exactly one true and one false
$if\ \phi\ then\ \psi_1\ else\ \psi_2$	$(\phi \rightarrow \psi_1) \wedge (\neg\phi \rightarrow \psi_2)$
$\exists x.\phi$	there exists x such that ϕ
$\forall x.\phi$	for all x such that ϕ
fv	free variable function

B.5 Semantics

Notation	Meaning
$\mathcal{D}$	domain
$\mathcal{D}^A$	domain of sort A
δ	sort function of domain elements
D	function fixing parameters of partial interpretation
$\mathcal{I}$	interpretation of terms and predicates
$\mathcal{K}$	Kripke structure
$\mathcal{M}$	partial model
$\mathcal{S}$	set of states or models
S	state $S \in \mathcal{S}$
ρ	state transition relation
β	variable assignment
β_x^d	variable assignment, update value of x by d
$val_{S,\beta}$	valuation function
$S, \beta \models \phi$	ϕ is valid under β and S

B.6 Sequent calculus

Notation	Meaning
$\phi_1, \ldots, \phi_n \Longrightarrow \psi_1, \ldots, \psi_m$	$(\phi_1 \wedge \cdots \wedge \phi_n) \rightarrow (\psi_1 \vee \cdots \vee \psi_m)$
Δ	$\phi_1, \ldots, \phi_n$
Γ	$\psi_1, \ldots, \psi_m$
U	updates
$;\omega$	possibly empty sequence of statements

Sequent rule

$$\frac{\overbrace{\Gamma_1 \Longrightarrow \Delta_1 \; ... \; \Gamma_n \Longrightarrow \Delta_n}^{premises}}{\underbrace{\Gamma \Longrightarrow \Delta}_{conclusion}}$$

B.7 Reasoning about Creol

Notation	Meaning
exp	matches expressions with top level operator
$texp$	matches literals and variables
$\mathcal{L}$	sequence number
$\frown$	concatenation of messages
ϵ	empty history
$h \upharpoonright o$	history projected to communication of o
$h \upharpoonright o_{\rightarrow}$	history projected to messages sent by o
$\mathcal{H}_S$	sending history
$\mathcal{H}_O$	object history
$\overline{\mathcal{W}}$	class attributes
$A_{\bar{v}}$	anonymizing update for variables $\bar{v}$
$F_{\mathcal{H}_O}(\phi)$	object history anonymizing formula
$F_{\mathcal{L}}(\phi)$	sequence number anonymizing formula
Inv_C	class invariant
Pre	pre-condition
$Post$	post-condition

B.8 Domains

Name	Set
$Null$	$\{null\}$
$boolDom$	$\{tt, ff\}$
$intDom$	$\mathbb{Z}$
$objDom$	$\bigcup_{i=0}^{\infty} objDom^i$ where $objDom^0 := \{0\}$, $objDom^i := \{p(o,i) \mid i \in \mathbb{N}_0, o \in objDom^{i-1}\}$
$methDom$	$\{\_, a, b, \ldots, z\} \times \{\_', a, b, \ldots, z, A, B, \ldots, Z, 0, 1, \ldots, 9\}^*$
$methLabelDom$	$\{\langle o_1, o_2, m, i\rangle \mid o_1, o_2 \in objDom, m \in methDom, i \in intDom\}$
$newLabelDom$	$\{\langle p(o,i), i\rangle \mid o \in objDom, i \in intDom\}$
$msgDom$	$\{\langle t, l, \bar{d}\rangle \mid t \in \{invoc, comp\}, l \in methLabelDom, \bar{d} \in domData^*\}$ $\cup \{\langle new, l, \bar{d}\rangle \mid l \in newLabelDom, \bar{d} \in domData^*\}$ where $domData := \bigcup_{dom \in \mathcal{T}_d, dom \sqsubseteq Data} dom$
$histDom$	$\bigcup_{i=0}^{\infty} histDom^i$ where $histDom^0 := \{\epsilon\}$, $histDom^i := \{h^\frown m \mid h \in histDom^{i-1}, m \in msgDom\}$
$sendHistDom$	$\{h \upharpoonright \mathcal{I}_0(this)_{\rightarrow} \mid h \in histDom\}$
$objHistDom$	$\{h \upharpoonright \mathcal{I}_0(this) \mid h \in histDom\}$

B.9 Rigid functions

Syntax

function symbol	sorts
$+$	$Int, Int \to Int$
$-$	$Int, Int \to Int$
$*$	$Int, Int \to Int$
$/$	$Int, Int \to Int$
$-$	$Int \to Int$
$\ldots, -1, 0, 1, 2, \ldots$	$\to Int$
$FALSE$	$\to Bool$
$TRUE$	$\to Bool$
$null$	$\to Null$
(A)	$Top \to A$
$parent$	$Any \times Int \to Any$
com	$Any, Any, Method, Int \to Label$
$toCaller$	$Label \to Any$
$toCallee$	$Label \to Any$
$toMethod$	$Label \to Method$
$toId$	$Label \to Int$
new	$Any, Int \to NewLabel$
$toNew$	$NewLabel \to Any$
$toIdNew$	$NewLabel \to Int$
$msgInvoc$	$Label, Data^* \to Message$
$msgComp$	$Label, Data^* \to Message$
$msgNew$	$NewLabel, Data^* \to Message$
$hist$	$SendingHistory, Message \to SendingHistory$
$startsAt$	$ObjectHistory \to SendingHistory$

semantics

semantics
$\mathcal{I}_0(+)(x,y) = x + y$
$\mathcal{I}_0(-)(x,y) = x - y$
$\mathcal{I}_0(*)(x,y) = x * y$
$\mathcal{I}_0(/)(x,y) = \begin{cases} z \text{ such that } 0 \leq x - y * z < \lvert y \rvert & \text{if } y \neq 0 \\ \text{some arbitrary but fixed } d \in \mathcal{D}^{Int} & \text{otherwise} \end{cases}$
$\mathcal{I}_0(-)(x) = -x$
$\mathcal{I}_0(i) = i$ for $i \in \mathbb{Z}$
$\mathcal{I}_0(FALSE) = ff$
$\mathcal{I}_0(TRUE) = tt$
$\mathcal{I}_0(null) = null$
$\mathcal{I}_0((A))(x) = \begin{cases} x & \text{if } \delta_0(x) \sqsubseteq A \\ \text{some arbitrary but fixed } d \in \mathcal{D}^A & \text{otherwise} \end{cases}$
$\mathcal{I}_0(parent)(o,i) = p(o,i)$
$\mathcal{I}_0(com)(o_1,o_2,m,i) = \langle o_1,o_2,m,i\rangle$
$\mathcal{I}_0(toCaller)(\langle o_1,o_2,m,i\rangle) = o_1$
$\mathcal{I}_0(toCallee)(\langle o_1,o_2,m,i\rangle) = o_2$
$\mathcal{I}_0(toMethod)(\langle o_1,o_2,m,i\rangle) = m$
$\mathcal{I}_0(toId)(\langle o_1,o_2,m,i\rangle) = i$
$\mathcal{I}_0(new)(o,i) = \langle o,i\rangle$
$\mathcal{I}_0(toNew)(\langle o,i\rangle) = o$
$\mathcal{I}_0(toIdNew)(\langle o,i\rangle) = i$
$\mathcal{I}_0(msgInvoc)(\langle o_1,o_2,m,i\rangle,\overline{d}) = \langle invoc,\langle o_1,o_2,m,i\rangle,\overline{d}\rangle$
$\mathcal{I}_0(msgComp)(\langle o_1,o_2,m,i\rangle,\overline{d}) = \langle comp,\langle o_1,o_2,m,i\rangle,\overline{d}\rangle$
$\mathcal{I}_0(msgNew)(\langle o,i\rangle,\overline{d}) = \langle new,\langle o,i\rangle,\overline{d}\rangle$
$\mathcal{I}_0(hist)(h,msg) = h^\frown msg$
$\mathcal{I}_0(startsAt)(h) = h \restriction \mathcal{I}_0(this)_\rightarrow$

B.10 Predicates

Syntax

predicate symbol	sort
$<$	Int, Int
$\leq$	Int, Int
$>$	Int, Int
$\geq$	Int, Int
$\doteq$	Top, Top
$quanUpdateLeq$	Top, Top
$\sqsubseteq A$	Top
$Invoc$	$ObjectHistory, Label$
$Comp$	$ObjectHistory, Label$
New	$ObjectHistory, Any$
$Prefix$	$ObjectHistory, ObjectHistory$
Wf	$ObjectHistory, Int$
$Cons$	$SendingHistory, ObjectHistory, Int$

Semantics

semantics
$\mathcal{I}_0(<) = \{(x,y) \in \mathbb{Z} \times \mathbb{Z} \mid x < y\}$
$\mathcal{I}_0(\leq) = \{(x,y) \in \mathbb{Z} \times \mathbb{Z} \mid x \leq y\}$
$\mathcal{I}_0(>) = \{(x,y) \in \mathbb{Z} \times \mathbb{Z} \mid x > y\}$
$\mathcal{I}_0(\geq) = \{(x,y) \in \mathbb{Z} \times \mathbb{Z} \mid x \geq y\}$
$\mathcal{I}_0(\doteq) = \{(x,y) \in \mathbb{Z} \times \mathbb{Z} \mid x = y\}$
$\mathcal{I}_0(quanUpdateLeq) = \{(x,y) \in Top \times Top \mid x \prec y\}$
$\mathcal{I}_0(\sqsubseteq A)(x) = \mathcal{D}^A$
$\mathcal{I}_0(Invoc)(h,l) = \left\{ (h,l) \in \begin{array}{l} objHistDom \times \\ methLabelDom \end{array} \middle\vert \exists \bar{d} \in Data^* : \langle invoc, l, \bar{d} \rangle \in h \right\}$
$\mathcal{I}_0(Comp)(h,l) = \left\{ (h,l) \in \begin{array}{l} objHistDom \times \\ methLabelDom \end{array} \middle\vert \exists \bar{d} \in Data^* : \langle comp, l, \bar{d} \rangle \in h \right\}$
$\mathcal{I}_0(New)(h,o) = \left\{ (h,o) \in \begin{array}{l} objHistDom \times \\ objDom \end{array} \middle\vert \begin{array}{l} \exists \bar{d} \in Data^* \ \exists i \in intDom : \\ \langle new, \langle o, i \rangle, \bar{d} \rangle \in h \end{array} \right\}$
$\mathcal{I}_0(Prefix) = \left\{ (h_1,h_2) \in \begin{array}{l} objHistDom \times \\ objHistDom \end{array} \middle\vert \begin{array}{l} \exists h_3 \in objHistDom : \\ h_1 \frown h_3 = h_2 \end{array} \right\}$
$\mathcal{I}_0(Wf) = \{(h,o,i) \in objHistDom \times intDom \mid wf(h,i) = 1\}$ where $wf : objHistDom \times intDom \to \{0,1\}$: $wf(h \frown \langle invoc, \langle o_1, o_2, m, i \rangle, \bar{d} \rangle, j) = \begin{cases} wf(h,i) & \text{if } \mathcal{I}_0(this) = o_1 \text{ and } i < j \\ wf(h,j) & \text{if } \mathcal{I}_0(this) = o_2 \\ 0 & \text{otherwise} \end{cases}$ $wf(h \frown \langle comp, \underbrace{\langle o_1, o_2, m, i \rangle}_{l}, \bar{d} \rangle, j) = \begin{cases} wf(h,i) & \text{if } \mathcal{I}_0(this) = o_2 \text{ and } \exists \bar{d}' \ \exists \langle invoc, l, \bar{d}' \rangle \in h \\ wf(h,j) & \text{if } \mathcal{I}_0(this) = o_1 \text{ and } i < j \text{ and } \exists \bar{d}' \ \exists \langle invoc, l, \bar{d}' \rangle \in h \\ 0 & \text{otherwise} \end{cases}$ $wf(h \frown \langle new, \langle o, i \rangle, \bar{d} \rangle, j) = \begin{cases} 1 & \text{if } \mathcal{I}_0(this) = o \text{ and } h = \epsilon \\ wf(h,i) & \text{if } p(\mathcal{I}_0(this), i) = o \text{ and } i < j \\ 0 & \text{otherwise} \end{cases}$
$\mathcal{I}_0(Cons) = \left\{ (h_s, h_o, i) \in \begin{array}{l} sendHistDom \times \\ objHistDom \times \\ intDom \end{array} \middle\vert \begin{array}{l} \exists h_1 \in sendHistDom \\ h_1 \frown h_s = h_o \restriction this_{\rightarrow} \\ \text{and } wf(h_o, i) \end{array} \right\}$

Bibliography

[ABB+05] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[ÁdBdRS03] Erika Ábrahám, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. A tool-supported assertional proof system for multithreaded Java. In Susan Eisenbach, Gary T. Leavens, Peter Müller, Arnd Poetzsch-Heffter, and Erik Poll, editors, *Proc. of the Workshop on Formal Techniques for Java-like Programs - FTfJP'2003*, 2003.

[AO97] Krzysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of sequential and concurrent programs (2nd ed.)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[Bal04] Heide Balzert. *Lehrbuch der Objektmodellierung: Analyse und Entwurf mit der UML 2*. Spektrum Akademischer Verlag, 2. edition, 2004.

[Bau06] Markus Baum. Proof Visualization. Studienarbeit, Department of Computer Science, University of Karlsruhe, 2006.

[BCC+05] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe 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, June 2005.

[Bec01] Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors,

Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France, volume 2041 of *LNCS*, pages 6–24. Springer, 2001.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[BK07] Bernhard Beckert and Vladimir Klebanov. A dynamic logic for deductive verification of concurrent programs. In Mike Hinchey and Tiziana Margaria, editors, *Proceedings, 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM), London, UK*. IEEE Press, 2007.

[Bla07] Jasmin Christian Blanchette. Overview over the Creol language. available online at `http://heim.ifi.uio.no/creol/blanchette07essay.pdf`, May 2007. accessed 13-May-2009.

[Bla08] Jasmin Christian Blanchette. Verification of assertions in Creol programs. Master's thesis, University of Oslo, Oslo, Norway, May 2008.

[BP06] Bernhard Beckert and André Platzer. Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In U. Furbach and N. Shankar, editors, *Proceedings, International Joint Conference on Automated Reasoning, Seattle, USA*, volume 4130 of *LNCS*, pages 266–280. Springer, 2006.

[Bur74] Rod M. Burstall. Program proving as hand simulation with a little induction. In *Information Processing '74*, pages 308–312. Elsevier/North-Holland, 1974.

[ByECD+06] Mike Barnett, Bor yuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustanm. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, volume 4111 of Lecture Notes in Computer Science*, pages 364–387. Springer, 2006.

[CDE+08] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narcisco Martí-Oliet, José Meseguer, and Carolyn Talcott. Maude manual (version 2.4). available online at `http://maude.cs.uiuc.edu/maude2-manual/`, October 2008. accessed 24-May-2009.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, USA, 2001.

[Coo78] Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computing*, 7(1):70–90, 1978.

[COR+95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, , and Mandayam Srivas. A tutorial introduction to PVS. April 1995.

[Dah77] O.-J. Dahl. Can program proving be made practical? *Les Fondements de la Programmaion*, pages 57–114, December 1977.

[DdM06] B. Dutertre and L. de Moura. The yices smt solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, August 2006.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[DJO04] Johan Dovland, Einar Broch Johnsen, and Olaf Owe. Reasoning about asynchronous method calls and inheritance. In Chunming Rong, editor, *Proc. of the Norwegian Informatics Conference (NIK'04)*, pages 213–224. Tapir Academic Publisher, November 2004.

[DJO05] Johan Dovland, Einar Broch Johnsen, and Olaf Owe. Verification of concurrent objects with asynchronous method calls. In *Proceedings of the IEEE International Conference on Software Science, Technology & Engineering(SwSTE'05)*, pages 141–150. IEEE Computer Society Press, February 2005.

[DJO06] Johan Dovland, Einar Broch Johnsen, and Olaf Owe. A hoare logic for concurrent objects with asynchronous method calls. Technical Report 315, Department of Informatics, University of Oslo, 2006.

[DJO08a] Johan Dovland, Einar Broch Johnsen, and Olaf Owe. A compositional proof system for dynamic object systems. Technical Report 351, Department of Informatics, University of Oslo, 2008.

[DJO08b] 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.

[DJOS08] Johan Dovland, Einar Broch Johnsen, Olaf Owe, and Martin Steffen. Incremental reasoning for multiple inheritance. Technical Report 373, Department of Informatics, University of Oslo, 2008.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, volume 4963/2008 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin, April 2008.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[dRdBH+01] Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency verification: introduction to compositional and noncompositional methods*. Cambridge University Press, New York, NY, USA, 2001.

[EH07] Christian Engel and Reiner Hähnle. Generating unit tests from formal proofs. In Yuri Gurevich and Bertrand Meyer, editors, *Proceedings, 1st International Conference on Tests And Proofs (TAP), Zurich, Switzerland*, volume 4454 of *LNCS*. Springer, 2007.

[Fit90] Melvin Fitting. *First-order logic and automated theorem proving*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *Proceedings Symposium for Applied Mathematics, Vol. XIX*, pages 19–32. American Mathematical Society, Providence, R.I., 1967.

[FM07] Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. pages 173–177. 2007.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische schließen. *Mathematische Zeitschrift*, 39:176–210 and 405–431, 1935.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[Gie05] Martin Giese. A Calculus for Type Predicates and Type Coercion. In B. Beckert, editor, *A Calculus for Type Predicates and Type Coercion*, volume 3702 of *LNAI*, pages 123–137, Koblenz, Germany, September 2005. Springer.

[Göd31] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[GvN47] Herman H. Goldstine and John von Neumann. Planning and coding of problems for an electronic computing instrument. Part II, vol. I. Technical report 1, Institute for Advanced Study, April 1947.

[Hab00a] Elmar Habermalz. *Ein dynamisches automatisierbares interaktives Kalkül für schematische theoriespezifische Regeln*. PhD thesis, Universität Karlsruhe, 2000.

[Hab00b] Elmar Habermalz. Interactive theorem proving with schematic theory specific rules. Technical Report 19/00, Fakultät für Informatik, Universität Karlsruhe, 2000.

[Häh05] Reiner Hähnle. Many-valued logic, partiality, and abstraction in formal specification languages. *Logic Journal of the IPGL*, 13(4):415–433, July 2005.

[Har79] David Harel. *First-Order Dynamic Logic*. Springer, 1979.

[HKT00] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hoa83] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1):100–106, 1983.

[Hoa03] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.

[JO02] Einar Broch Johnsen and Olaf Owe. A compositional formalism for object viewpoints. In Bart Jacobs and Arend Rensink, editors, *Proceedings of the 5th International Conference on Formal Methods for Open Object-Based Distributed Systems*

(FMOODS 2002), pages 45–60. Kluwer Academic Publishers, March 2002.

[JO04] Einar Broch Johnsen and Olaf Owe. Object-oriented specification and open distributed systems. In Olaf Owe, Stein Krogdahl, and Tom Lyche, editors, *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, volume 2635 of *Lecture Notes in Computer Science*, pages 137–164. Springer-Verlag, 2004.

[Jon81] C.B. Jones. *Development Methods for Computer Programs including a Notion of Interfence*. PhD thesis, Oxford University, 1981.

[Kya08] Marcel Kyas. Creol tools user guide for version 0.0m. available online at `http://heim.ifi.uio.no/~kyas/creoltools/`, November 2008. accessed 13-May-2009.

[Mea55] George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[Mür08] Oleg Mürk. Deductive verification of c programs in key. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, January 2008.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, Raleigh, 2007.

[Pla04] André Platzer. An object-oriented dynamic logic with updates. Master's thesis, Universität Karlsruhe, Fakultät für Informatik, September 2004.

[PQ08] André Platzer and Jan-David Quesel. KeYmaera: A hybrid theorem prover for hybrid systems. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2008, Sydney, Australia, Proceedings*, volume 5195 of *LNCS*, pages 171–178. Springer, 2008.

[Pra77] Vaughan R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proc. 17th Annual IEEE Symposium on Foundation of Computer Science, Houston, TX, USA*, pages 109–121. IEEE Computer Society, 1977.

[Rüm05] Philipp Rümmer. Generating counterexamples for Java Dynamic logic. In Wolfgang Ahrendt, Peter Baumgartner, and Hans de Nivelle, editors, *Preliminary Proceedings of Workshop on Disproving at CADE 20*, pages 32–44, July 2005.

[SBD02] Aaron Stump, Clark W. Barrett, and David L. Dill. Cvc: a cooperating validity checker. In *In 14th International Conference on Computer-Aided Verification*, pages 500–504. Springer, 2002.

[Sch00] Uwe Schöning. *Logik für Informatiker*. Spektrum Akademischer Verlag, 5. edition, 2000.

[Tur49] Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, Cambridge, 1949.

[WK99] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, Reading/MA, 1999.

[YJO06] Ingrid Chieh Yu, Einar Broch Johnsen, and Olaf Owe. Type-safe runtime class upgrades in Creol. In Roberto Gorrieri and Heike Wehrheim, editors, *Proc. 8th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*, volume 4037 of *Lecture Notes in Computer Science*, pages 202–217. Springer-Verlag, June 2006.