CHALMERS

A Formal Verification Environment for Distributed Object-Oriented Models

Master of Science Thesis

GIAMPIERO BAGGIANI

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, January 2011

A Formal Verification Environment for Distributed Object-Oriented Models

GIAMPIERO BAGGIANI

Examiner: Wolfgang Ahrendt

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

Department of Computer Science and Engineering
Göteborg, Sweden, January 2011

Abstract

Distributed systems are gaining increasing interest in the research community. The growing adoption of such systems for safety-critical structures demands for a high reliability and thus, for in-depth functional verification.

This thesis contributes to the development of a formal environment for the verification of Creol models. Creol is an executable modelling language featuring many aspects which make it very suitable for its employment in distributed, concurrent applications.

The major contributions of this work are: the design of a formal specification language for Creol (CSL), the implementation of a front-end supporting inline specifications and its integration in the theorem prover KeY.

CSL focuses on providing the user with an abstract way of expressing properties on communication traces between objects. It relies on a compositional proof system which allows the independent verification of object's methods against invariants and operation contracts.

Keywords: formal verification, specification language, distributed systems, concurrency, communication history

Preface

This work presents the results of a project undertaken in fulfilment of a 30-credit Master's thesis for the "Secure and dependable computer systems" Master's program at the Computer Science and Engineering department of Chalmers University of Technology. The thesis has been carried out under the supervision of Prof. Wolfgang Ahrendt as part of a research project within the Software Engineering using Formal Methods group.

I would like to thank all the members of the SEFM group for welcoming me and for making me feel part of them. I'm very thankful to Wolfgang Ahrendt for supporting me during this period and for showing constant interest in my work, ideas and opinions. It's been a real pleasure having him first as a lecturer and then as my supervisor. A special thanks also goes to Richard Bubel for sharing with me his infinite knowledge about the KeY system and to Maximilian Dylla for always being willing to clarify aspects of his previous work on this topic.

Last but not least I want to thank my family and all my friends who filled this period of studies with unforgettable and irreplaceable moments.

Dedicato a coloro i quali mi hanno dato
il supporto, la forza, il coraggio e la libertà
di arrivare fin qui: i miei genitori.

Grazie!

Contents

Chapter 1

Introduction

In a world where computer systems are a fundamental part of the society, the reliability of the underlying algorithms and their implementation is a necessity.

The research in fields related to the verification of object-oriented programs is witnessing a substantial growth. Several specification languages and verification tools are in continuous development, but the main target of such systems is prevalently *sequential* applications. In an environment where distributed systems play an important role there is the lack of tools able to verify the functionality of *concurrent* applications.

This thesis embodies a contribution in the development of formal verification of distributed, concurrent systems. Its corner stones are represented by two research projects, KeY and Creol. Both are described here briefly.

KeY is a research project undertaken at Chalmers University of Technology in collaboration with the University of Karlsruhe in Germany. It targets the verification of object-oriented software, in particular Java. The KeY system aims to integrate design, implementation, formal specification, and formal verification of object-oriented software as seamlessly as possible. The core of the KeY system is a novel theorem prover for a program logic that combines a variety of automated reasoning techniques. The KeY prover is distinguished from most other deductive verification systems in that symbolic execution of programs, first-order logic reasoning, arithmetic simplification, decision procedures, and symbolic state simplification are interleaved.

Creol is an European research project with the University of Oslo as the main partner. The goal of the project is to develop a formal framework for reasoning about dynamic and reflective modifications in object-oriented open distributed systems, ensuring reliability and correctness of the overall system. The core of the project is the language Creol (Concurrent Reflective Object-oriented Language). It is object-oriented in that classes are the fun-

damental structuring unit and that all interaction between objects occurs through method calls. What sets Creol apart from popular object-oriented languages such as C++ or Java is its concurrency model: In Creol, each object executes on its own virtual processor. This approach leads to increased parallelism in a distributed system, where objects may be dispersed geographically.

In a previous master thesis work [3] by Maximilian Dylla, a program logic and a proof calculus have been designed and implemented for the Creol language following KeY's methodologies. Lately, this work has been carried on by Wolfgang Ahrendt and Maximilian Dylla leading to the completion of a journal paper [2] where the calculus and the logic have been redesigned and broadened and a denotational semantics for Creol has been introduced.

1.1 Project goals

The goal of this thesis project can be generalized as providing a verification environment for Creol models which, on top of a pure theorem prover, offers a user front-end supporting specifications of components and their communication. The primary landmarks of the work are:

- the design and implementation of a specification language for Creol
- the automatic generation of proof obligations
- the improvement of the calculus
- the increase of usability and automation

Additionally, the achievements of the work are evaluated in a case study and complementary tests.

The contributions of the work are thus concerning theoretical aspects as well as practical applications. On the theoretical side, the effort lies in the underlying logical concepts needed for the improvement of the Creol calculus and the definition of a specification language. The actual implementation of a grammar for such language, its integration with a Creol parser and the adaption of the KeY tool for the support of Creol programs with inline specifications, represent the practical aspects of this thesis work.

1.2 Thesis outline

This thesis is targeted to students and researchers interested in verification of computer systems. Knowledge of object-oriented programming paradigms and first-order logic is recommended.

The structure of this report proceedes as follows:

- chapter 2 introduces the Creol language highlighting relevant aspects for this thesis
- chapter 3 gives an overview of the KeY prover
- chapter 4 presents a denotational semantics for Creol
- chapter 5 describes the developed specification language for Creol
- chapter 6 explains the aim of proof obligations and shows the construction of a proof obligation for Creol
- chapter 7 contains the calculus used for reasoning about Creol
- chapter 8 illustrates the implemented verification system
- chapter 9 uses the concept illustrated in the previous chapters in a case study
- chapter 10 summarizes the limitation of this thesis and includes suggestions for future development
- chapter 11 draws the conclusions and recapitulates the results of the project

Chapter 2

Overview of Creol

In this chapter the Creol language is introduced. Many references on Creol can be found on the project's Web page[1]. Here, only the main aspects relevant for this thesis are presented. The focus lies on the concurrency and on the communication between objects. Features like dynamic class upgrades, multiple inheritance, and non-determinism are not considered; literature covering this topics can be found in [4, 5, 6].

Creol is still in a experimental phase and several dialects of this language exist. This thesis follows the version used in the previous works [3, 2] on which the project is based. As we will see in chapter 5 the syntax has been here extended to allow inline specifications.

2.1 General aspects

Creol is a modelling language targeting distributed, concurrent systems. It is an object-oriented language, meaning that objects, instantiating classes and interfaces, are used as data structure and their internal state is hidden and accessible only via method calls.

The major characteristic that sets Creol apart from other object-oriented languages is its inherent support for concurrency. First, all the objects are thought to be running on a separate (possibly virtual) processor, thus no memory or computational resource is shared between different instances. On the other hand, methods belonging to each object are run as instances of threads sharing the processor. In this manner, Creol provides for the modelling of parallel as well as interleaved execution. Additionally, as another feature that makes Creol a very suitable language for distributed environments, the communication between objects, i.e. method calling, is asynchronous: the invocation of a method may not coincide with the actual execution of the method's body and analogously the completion of a called method is separate from the request of its return value (if any). This enforces

[1] heim.ifi.uio.no/creol/

the analogy to the communication in distributed systems which occurs by asynchronous message passing.

2.2 Data types

A narrowed simple type hierarchy is used in this work. The `Data` type represents the top of the type hierarchy, all other types are its subtypes.

The other primitive types are **`Int`**, **`Bool`**, and **`Any`**. The latter represents the basic interface which is by default inherited by all the user-defined interfaces, therefore implemented by all the classes.

Creol includes an additional built-in data type, **`Label`**, used for labelling method invocation and completion requests. We will see how to use labels later on in this chapter.

Interfaces and classes

Creol allows for the definition of classes and interfaces. Interfaces specify the outside vision of objects, meaning they contain the declaration of methods accessible by other interface instances. Classes, on the other hand, contain the implementation of the methods and encapsulate an internal state being the class attributes, which are private by default, i.e. not directly accessible by any other objects.

To clarify this, let's see some actual code:

Creol

```
interface I
begin
  with Any
    op meth1(in i:Int)
    op meth2(in j:Int, out b:Bool)

  with I2
    op meth3()
end
```

Creol

The above listing contains the definition of the interface `I`. As we can see it contains the declaration, i.e. the signature, of its methods, but not their implementation. Method bodies can not be implemented in interfaces. The signature of the methods specifies, of course, the assigned identifier (`meth1`, `meth2`, and `meth3` for this example) and the (optional) *in* and *out* parameters.

The major characteristic that distinguish Creol from other languages is the presence of a co-interface. All the methods declared in an interface must be inside a **with**-block asserting which interfaces the contained methods are visible to. This means that, for instance, the method `meth3` in the example can be called only by an object whose class implements the interface `I2`, while the other two methods can be potentially called by any object.

Now, let's see how to define a class `C` implementing the interface `I`:

— Creol —

```
class C(x:Int) implements I
begin
  var attr:Int;
  var obj:I;

  op init
    == attr:=x;
       obj:=new C(x)
  op run
    == body_0

  with Any
    op meth1(in i:Int)
      == body_1
    op meth2(in j:Int, out b:Bool)
      == body_2

  with I2
    op meth3()
      == body_3
end
```

— Creol —

The above class implements all the methods declared in the interface `I`; besides, it contains two local methods, `init` and `run`. They are special methods implicitly contained in all classes and assumed to be empty if not stated otherwise. The method `init` is called on object creation and the class parameters (`x` in this example) are its parameters. Afterwards `run` is called to perform actions on the object. A class can define additional local methods which are only internally visible. The body of the methods is defined after a double-equal symbol.

The class attributes `attr` and `obj` are visible to all the class methods, but not externally. It is important to notice how the type of the object `obj` is an interface. In Creol objects can *only* be typed by interface, classes are not

belonging to the type hierarchy. This enforces information hiding between different objects which, as is the case for distributed systems scenarios, have a partial view of each other.

2.3 Statements

The body of methods contains the effectively executable code. The syntax of Creol statements follows the standard patterns embraced by any other programming language; the main aspects characterising Creol and which are very relevant for this thesis, are the statements used for communication between objects and for the control of the execution flow.

Object communication

In contrast to common programming languages, Creol separates method invocation from the retrieval of the return value (completion) of a method. Here is where labels comes into play: they are used to connect the invocation of a method to its completion. To better understand consider the following example:

— Creol —

```
var l1:Label;
var l2:Label[Bool];
var z:Bool;
l1!obj.meth1(7);
l2!obj.meth2(10);
...;
l2?(z);
l1?()
```

— Creol —

Here, the statements including the exclamation mark (`!`) represent the invocation of a method. The request for method completion (marked by `?`) can appear at any point in the code following the invocation and assigns the return value (if any) to the specified variable. In the case the called method has not returned, the completion statement cause the current thread to be set in a *busy waiting* state until the reply is available. We will see how to avoid this in the next section.

Note that, since the communication is asynchronous and no assumption is made on the underlying communication network, the order in which invocation messages are delivered to the called object may not coincide to the order in which they are actually sent by the callee.

Control flow

The scheduling process between concurrent threads, i.e. instances of methods belonging to the same objects, is ruled *only* by *explicit* release points. This means that there is no preemption policy and a running thread will never release the processor unless it terminates or it reaches one of the following statements:

`release`: this represents the simplest statement to allow another ready thread to start executing.

`await` *guard*: this is a conditional release point. The control will be released iff the guard evaluates to false. The guard can be a boolean expression as well as a completion statement (`l?()`). In the latter case the guard is considered true if a reply for the corresponding label is available, false otherwise. If the control is released the thread will be rescheduled only when the guard is true.

Other peculiar Creol statements are **`skip`**, which corresponds to the "no operation" command, and **`block`**, which corresponds to an infinite busy waiting. This two last statements are mainly practical for verification purposes.

Exceptions are not supported by Creol. In case of errors, like a division by zero or an invocation on a **`null`** reference, the object is blocked.

Chapter 3

The KeY prover

One of the cardinal points of this work is the KeY prover. It is a formal method tool originally thought for the verification of Java with JML (Java Modelling Language) as specification language, aimed to integrate design, implementation, formal specification and formal verification of Object-Oriented software.

The project surrounding this tool, the KeY Project, is a collaboration mainly between two European universities: Chalmers University of Technology (Gothenburg, Sweden) and University of Karlsruhe (Germany). It's possible to find more information about the project on the Web page[1] where it is also possible to download the KeY tool or directly run it via Java Web Start.

3.1 Overview of KeY

The KeY tool offers a simple GUI which helps users which are not familiar with formal methodologies, to go through the process of formal verification with the support of a reliable tool which provide, among many others features, highly automated translation from specifications to program logic, generation of proof obligations, and interactive application of rules.

The usual user input to KeY consists of a program source file with annotations in the pertinent specification language (e.g. JML for Java code). The specifications are then translated to KeY's program logic and, from the given specifications, different proof obligations arise which are to be discharged, i.e. a proof has to be found. To this ends, the program is symbolically executed with the resulting changes to program variables stored in so-called *updates*. Once the program has been processed completely, there remains a first-order logic proof obligation.

[1] www.key-project.org

At the heart of the KeY system lies a first-order theorem prover based on *sequent calculus*, which is used in the whole process of closing the proofs. Interference rules are mainly captured in so called *taclets* which consist of a simple language to describe changes to a sequent.

3.2 Dynamic Logic

The core of KeY is a theorem prover for formulas in dynamic logic (DL) where the code is part and parcel of the formulas. That is, the logic allows for formulas like $\phi \rightarrow [\alpha]\psi$, which intuitively means that the condition ψ must hold in all program states reachable by executing the program α in any state that satisfies the condition ϕ.

This is a very frequent pattern of DL formulas (as we will see in chapter 5) for the verification of method implementations w.r.t. their contract; here, ϕ is generally called *precondition* and ψ is referred to as *postcondition* of the method.

The previously mentioned formula corresponds to the triple $\{\phi\}\alpha\{\psi\}$ in Hoare calculus [8] if ϕ and ψ are purely first order formulas. However, DL extends Hoare logic in that formulas may contain nested programs, or that quantification over formulas which contain programs is possible. Further, the concept of *modalities* is added. The box modality we saw in the above formula, $[\cdot]$, expresses that the formula following it must hold in *all* terminating run of the contained program; note that this imply that a formula $[\alpha]\psi$ is always true if there is no terminating run of α. On the other hand, we have the diamond modality, $\langle\cdot\rangle$, which requires termination, i.e. the formula $\langle\alpha\rangle\psi$ is true if there exists a terminating run of α resulting in a state where ψ holds.

Another main characteristic of KeY's DL is the handling of substitutions in formulas caused by the resolution of assignments. The used approach is to delay the effect of the substitutions by means of accumulating *updates* representing the effects of program execution.

An elementary *update* represents thus an *explicit substitution* and is expressed in the form of $x := e$ where x is a *location* (e.g. a local variable or attribute) and e is a side-effect free expression. When updates are accumulated during symbolic execution of the program they are combined into simultaneous updates like $x_1 := e_1 \mid x_2 := e_2$, where e_1 and e_2 are evaluated in the same state.

Updates are connected to DL formulas using the modality $\{\cdot\}$, to form expressions like $\{\mathcal{U}\}\langle\alpha\rangle\psi$, where $\mathcal{U}$ is an arbitrary update. The actual substitution expressed by the updates arises only once, when the modality have been fully eliminated by sequent calculus transformations.

3.3 Deduction system

The theorem prover used to reduce proof obligations to axioms is based on a sequent calculus. A sequent is of the form of $\Gamma \vdash \Delta$, where Γ and Δ are sets (possibly empty) of formulas respectively called *antecedent* and *succedent* and representing the assumptions and the propositions of the proof.

A sequent is considered valid if:

$$\bigwedge_{\gamma \in \Gamma} \gamma \rightarrow \bigvee_{\delta \in \Delta} \delta$$

meaning that the validity of all the formulas constituting the antecedent implies the validity of at least one of the formulas in the succedent. Applying deductive transformations, an initial sequent representing the proof obligation is shown to be constructable from just fundamental first-order axioms.

The process of reduction of the initial sequent to an axiom is performed by consecutively applying *sequent rules*. The application of a rule causes the replacement of the sequent it is applied on (the *conclusion*) with one or more sequents representing its *premises*. When an axiom is reached the corresponding proof can be closed.

A sequent rule is expressed as:

$$\frac{\Gamma_1 \vdash \Delta_1 \ \ldots \ \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}$$

where the upper part shows the n premises, which validity implies the validity of the conclusion, in the lower part. Thus, the application of a rule reduces the provability of the conclusion to the provability of its premises.

The applicability of some rules may not be depending on the side of the sequent the conclusion is, and can even be applied to sub-formulas. In this case we use the following syntax:

$$\frac{\phi'}{\phi}$$

where ϕ and ϕ' are single formulas. Such syntax denotes a rule where the *only* premise is formed replacing any occurrence of ϕ in the conclusion with ϕ'.

For a complete description of KeY in all its aspects we strongly recommend to refer to [1].

Different research projects in the software verification area have been employing the KeY tool, such as the verification of concurrent Java [16] and the generation of JUnit test cases [17].

Moreover, variants of the KeY system have been implemented to adapt its functionalities to different fields. For instance *KeY-Hoare* features a Hoare calculus with state updates used to exemplify formal methods in undergraduate classes; also *KeYmaera* is a deductive verification tool for hybrid systems based on a calculus for the differential dynamic logic [19].

This thesis work, as we will see in the next chapters, represents a step forward in the implementation of a KeY version supporting Creol programs with inline specifications.

Chapter 4

Creol Denotational Semantics

This chapter introduces a denotational semantics for Creol proposed by Wolfgang Ahrendt and Maximilian Dylla in [2]. The definitions shown here are entirely taken from this paper and slightly adapted to highlight the most relevant points for this thesis. Here, how Creol is modelled will be shown and the basis to provide a compositional verification framework for this language will be explained.

The basic idea for the semantics of Creol models follows the principle, originally introduced by Zwiers in [14], of considering each existing process independently and later compose them to model the whole program. To this end, each process is modelled with the aid of *histories*, being sequences of messages marking significant program events. Whenever interaction with other processes occurs, the effect on the current process results in the construction of non-deterministic histories embodying all their possible behaviours. Then, when composing the processes, the concordant histories are merged and the other rejected.

To better understand this, we begin by giving the definition of single Creol statements and then, following a bottom-up approach, we arrive at the semantics of methods, objects, and finally the complete program.

The semantics makes use of a function $\mathcal{M}$ defined as follows:

$$\mathcal{M} : PROG \to (\Sigma \to 2^{\Sigma \times H})$$

where $PROG$ is the set of programs, Σ represents the set of all program states, and H is the set of all histories.

This function draws the connection between a program and a function which associates every initial state with a set of pairs containing the possible states and histories the program terminates with.

A program state, which in the following will be denoted by σ, relates local variables and object attributes with their current values; histories, as mentioned before, contain the sequence of messages representing the execution of the program in subject and will be denoted by θ.

The simplest Creol statement is **skip**, which causes no change. Its semantics will thus be the following:

$$\mathcal{M}(\textbf{skip})(\sigma) = \{(\sigma, \langle\rangle)\}$$

This means that, if executing the **skip** statement in a state σ, the resulting state after its termination will correspond to the initial one and the constructed history is empty (symbolized by $\langle\rangle$).

As another example, consider the **block** statement:

$$\mathcal{M}(\textbf{block})(\sigma) = \{\}$$

This statement causes the execution to never terminate, therefore no state is ever reached and thus its semantics corresponds to the empty set.

Next, we continue with assignments. In this case, the initial state is obviously required to change, but no event is recorded in the history:

$$\mathcal{M}(x := e)(\sigma) = \{(\sigma', \langle\rangle) \mid \exists v.\ v = \mathcal{E}(e)\sigma, \sigma' = (\sigma : x \to v)\}$$

With $\mathcal{E}(e)\sigma$ we indicate the result of the evaluation of the expression e in the state σ. $\mathcal{E}$ is a partial function returning no value if an error (e.g. a division by zero) is encountered in the evaluated expression. The resulting state σ' corresponds to the initial one except for the value of the assigned variable x which is updated to the value of v. This is what is expressed by $\sigma' = (\sigma : x \to v)$.

Now we switch the focus to statements dealing with thread concurrency and object communication. Here is where the reader should pay particular attention for a better understanding of the next chapters.

The **release** statement allows another ready thread to use the processor. Thus, the non-determinism caused by other threads here comes into play:

$$\mathcal{M}(\textbf{release})(\sigma) = \{(\sigma', \langle \mathit{yield}(\sigma|_a)\rangle^\frown\langle \mathit{resume}(\sigma'|_a)\rangle) \mid \sigma'|_l = \sigma|_l\}$$

where $\sigma|_l$ and $\sigma|_a$ are respectively used to restrict the preimage of the state to the local variables or the object attributes.

Here the history is marked with a $\mathit{yield} - \mathit{resume}$ pair to indicate a point where a thread switch is allowed in the composition of histories. As an effect on the local thread, the state is unchanged with respect to the local

scope, but no assumption is made on global attributes which may have been changed by other running threads.

Very relevant for the modeling of the object communication is, of course, the semantics of the invocation statement:

$$\mathcal{M}(l\,!\,o.m(\bar{x}))(\sigma) =$$

$$\left\{ (\sigma_1, \theta) \;\middle|\; \begin{array}{l} \exists oid.\ \exists v.\ \exists i.\ oid = \mathcal{E}(o)\sigma, v = \mathcal{E}(\bar{x})\sigma, \\ \quad \sigma_1 = (\sigma : l \to ((\mathcal{E}(this)\sigma, \mathcal{E}(me)\sigma), (oid, (m, i)))), \\ \quad \theta = \langle invoc(\mathcal{E}(l)\sigma_1, v) \rangle \end{array} \right\}$$

In the updated state, the label is assigned with a tuple containing the identity of the current thread and of the receiving one (i.e. an instance of the called method) and, together with the input parameter values, the event is recorded in the history with an *invocation* message.

Dual to the invocation, we have the completion statement:

$$\mathcal{M}(l?(\bar{y}))(\sigma) = \left\{ (\sigma_1, \theta) \;\middle|\; \begin{array}{l} \exists \bar{v}.\ \exists lv.\ lv = \mathcal{E}(l)\sigma, \\ \sigma_1 = (\sigma : \bar{y} \to \bar{v}), \\ \theta = \langle comp(lv, \bar{v}) \rangle \end{array} \right\}$$

The label, initialized in the invocation, is here recalled and recorded in a *completion* message together with the return parameters of the called method. The state is accordingly updated.

Another event marked in the history is the creation of an object as instance of a class C. The instance is identified by a unique ID and a "*new*" message holding the identity of the creator as well is added to the history:

$$\mathcal{M}(o := \textbf{new}\ C)(\sigma) = \left\{ (\sigma_1, \theta) \;\middle|\; \begin{array}{l} \exists i.\ \sigma_1 = (\sigma : o \to (C, i)), \\ \quad \theta = \langle new(\mathcal{E}(this)\sigma, \mathcal{E}(o)\sigma_1) \rangle \end{array} \right\}$$

Now we can introduce the sequential composition of statements. Here the effects of sequential statements (S_1 and S_2) are merged together:

$$\mathcal{M}(S_1\,;\,S_2)(\sigma) = \left\{ (\sigma_2, \theta_1 \frown \theta_2) \;\middle|\; \begin{array}{l} \exists \sigma_1.\ (\sigma_1, \theta_1) \in \mathcal{M}(S_1)(\sigma), \\ (\sigma_2, \theta_2) \in \mathcal{M}(S_2)(\sigma_1) \end{array} \right\}$$

where $\frown$ represents the concatenation of histories. Clearly, a statement can, in turn, be a sequence of statements itself; therefore, the above semantics can describe the entire body of a method.

We can now introduce the semantics of a single thread being an instance of a method m with its *body* representing a sequence of statements:

$$\mathcal{M}(\textbf{op}\ m(\textbf{in}\ \bar{x};\ \textbf{out}\ \bar{y}) == body)(\sigma) =$$

$$\left\{ (\sigma_2, \theta_1 \frown \theta \frown \theta_2) \;\middle|\; \begin{array}{l} \exists \bar{v}.\ \exists \sigma_1.\ \exists o.\ \exists o_2. \\ \quad \sigma_1 = (\sigma : \bar{x} \to \bar{v}), (\sigma_2, \theta) \in \mathcal{M}(body)(\sigma_1), \\ \quad o = \mathcal{E}(caller)\sigma, o_2 = (\mathcal{E}(this)\sigma, \mathcal{E}(me)\sigma), \\ \quad \theta_1 = \langle resume(\sigma|_a) \rangle \frown \langle begin((o, o_2), \bar{v}) \rangle, \\ \quad \theta_2 = \langle end((o, o_2), \mathcal{E}(\bar{y})\sigma_2) \rangle \frown \langle yield(\sigma_2|_a) \rangle \end{array} \right\}$$

It is important to notice how the history created by the body of the method is framed with additional histories (θ_1 and θ_2) used to mark the actual beginning and ending of the thread execution, respectively. The messages *begin* and *end* are the dual part of *invoc* and *compl* added to the history by the caller. They are all used to model the asynchronism of the communication: the invocation of a method doesn't necessarily correspond to the actual execution of the called method and the same is valid for the completion.

Following similar reasoning, we can define the semantics of methods, objects and finally the complete Creol program. Here, we omit a formal description of their semantics since it is not strictly relevant for this thesis, but we strongly recommend to refer to [2] for a complete and more detailed description of the whole semantics.

Roughly speaking, methods are semantically described as the union of all possible numbers of running threads being instances of the considered method; objects are modeled as the combination of all methods of the class instantiated by such object; the semantics of classes is presented as the composition of all their instances as objects.

Finally, we arrive to the semantics of a complete program where the different sets of histories are merged in a global one consistent with all the possible communication patterns from different threads. For instance, the final history will be required to have *yield* messages followed by *resume* ones having the same global state as parameter; this means that we had a consistent switch between thread execution. Further, we will require that invocation messages contain the identification of the actual communication partner and, moreover, they must be followed by the related *begin* messages. Therefore, the semantics of a program will consist of a "well formed" history describing the execution flow and its communication traces.

In the next chapter we will see how to use this modeling of Creol programs to express properties about its components by introducing a *specification language*.

Chapter 5

CSL: Creol Specification Language

This chapter gives a general overview about specifications and concepts related to this topics. After this, CSL is introduced and a detailed description of its syntax and semantics is provided.

5.1 Specifications

When working on any kind of project, one of the most tedious tasks is to give a complete and unambiguous description of how each component of the whole framework is supposed to behave. Natural language can of course come to the aid of this difficult task, but unfortunately it can easily fail even in simple situations.

According to the Oxford English Dictionary, the 500 words used most in the English language each have an average of 23 different meanings; this represent what is usually called *lexical ambiguity* of the language. Another pitfall of natural language is called *structural ambiguity* which arises when sentences or clauses can be interpreted in different ways when in different contexts or when read from different people. The first examination of this issue when dealing with software specifications is contained in [7].

It is then clear how natural language can't be a reliable tool for such a sensible task, especially when involved in safety-related problems. Unclear specifications can lead to substantial flaws in the system, hard to detect and possibly expensive to fix when in a late stage of the development process.

Introducing a formal language which gives the possibility to describe the behaviour of all the units constituting the system is a cardinal step to assure a reliable implementation.

A *specification* is a general term which includes different sub-concepts such as *operation contracts* and *invariants*. Following is an introduction to such concepts.

5.1.1 Operation contracts

An operation contract can be referred to as the basic block constituting the whole specification. As we are dealing with an object-oriented language, we know that all the procedures are coded inside the body of the methods of each class. Thus, the first thing one could think when willing to give the description of the whole behaviour, is to specify for each single method what it does (note *what* and not *how*) and possibly when it is meaningful to call it. This is exactly the purpose of an operation contract.

We distinguish two different states when calling a method: a pre-state representing the system just before the method invocation and a post-state describing the system when the method completes. The method itself can be thought as the transition connecting the two of them.

An operation contract let us specify which conditions should hold in these two states (i.e. pre- and postconditions), so that we can verify that the method implementation satisfies the postconditions, assumed that the precondition was true when calling the method.

Example 5.1.1. Assume we want to specify that the following method

— Creol —

```
op log(in b:Int, x:Int; out y:Int)
```

— Creol —

calculates the logarithm base `b` of the number `x` and assign this value to `y`.

For this method to represent the logarithm *function* it is necessary that the base `b` is a positive number unequal to 1, additionally `x` has to be positive. If this constraints are met then we can expect the result to exist, thus we could for instance say that `y` is not `null`.

We could also add as a postcondition that `b` raised to the power `y` is equals to `x`, but for this condition to hold some other assumption must be added: it is necessary that the value of `b` and `x` in the post-state is the same as it was when calling the method.

To model this scenario (holding on the assumption that this method has no side effects and does not change the value of the input parameters) we could write the following operation contract:

precondition : $\mathtt{b} \neq 1 \wedge \mathtt{b} > 0 \wedge \mathtt{x} > 0$
postcondition : $\mathtt{y} \neq \mathtt{null} \wedge \mathtt{b}^{\mathtt{y}} = \mathtt{x}$

To be more formal a general definition of the validity of an operation contract is the following:

Definition 5.1.1. *An operation contract for a method is satisfied if: when the method is invoked in any state which satisfies the precondition, then in any terminating state of the method the postcondition holds.*

It is important to realize how weak this definition is:

- No guarantee is given when the precondition is not satisfied. What happens if we call the method `log` with negative parameters?
- Termination of the method is never assumed. The method could run forever so that we don't have a terminating state where the postcondition must hold.
- The contract specifies properties only for some attributes. What about the rest of the system-wide state? How the method can modify it?

This issues must be highly considered during the verification process and will be addressed later in this chapter.

5.1.2 Invariants

Another construct widely used to specify properties of a program or model is the *invariant*.

Although one could expect specifications to mainly deal with what is changed by an operation, the same relevance should be given in expressing what remains unchanged during program execution. This is the duty of the invariant.

In object-oriented programming, invariants are usually "attached" to classes and/or interfaces whose internal state and methods are addressed by the expressed properties.

Trivially speaking, the invariant gives us the the possibility to express properties we can always rely on and, on the other hand, we must always guarantee it is preserved.

Example 5.1.2. Continuing from the previous example, assume we want to implement a class, `Binary`, which implements methods to calculate functions in base 2. It might have an attribute, `base`, which is for instance used as `b` parameter when calling the `log` method of the previous example.

A very simple *invariant* for this class could be the following:

$$invariant : \texttt{b} = 2$$

which simply states that none of the class methods is allowed to change the attribute `base` to a value different than 2.

A more formal definition of invariant is here given:

Definition 5.1.2. *An invariant is preserved by a method of the class owning such invariant if: when the method is invoked in a state satisfying the invariant, the invariant holds in any terminating state.*

As for operation contracts it is important to notice how termination is generally not required. Moreover it is crucial to highlight how invariants are not required to hold during the intermediate states of the method execution.

As we will see later, a stronger definition is generally adopted, as a first approximation we can state that invariants must be true whenever no operation is executing.

Another form of invariant used during the validation process is the *loop invariant*. Informally it states a condition which should be true when entering a loop and it is maintained during each iteration of the loop's body, thus it must hold when exiting the loop.

5.2 Overview of CSL

One of the main tasks of this thesis work has been to design a specification language suitable for the peculiarities of the Creol language. Before this thesis work no front end was available, and users willing to write specifications for any sequence of Creol code were forced to hard-code the problem in a separate KeY file. Moreover the properties had to be expressed in program logic and problem-specific taclets were required to be added and adapted to match specific program statements.

As we have seen in chapter 4, Creol has the characteristic of having a *history* being part of the state of the program. This let us extend the properties we can specify with respect to the program execution. Popular specification languages, such as JML for Java, allow to state predicates mainly in terms of class attributes and method parameters; with CSL, additionally to this, it is possible to express properties on communication traces between objects, i.e. the user can verify that method calls between different interfaces occur following specific patterns and furthermore assert predicates across different runs of the same method.

CSL provides an abstract syntax with strong information hiding which allows the user to keep the ease of expressing complex properties on interface communication without the necessity of being aware of the whole system-wide history.

To better understand the expressiveness of this language, in the following section we present a formal explanation of its syntax and semantics.

5.3 CSL Syntax and Semantics

CSL is completely integrated with the Creol language. Its syntax is designed to maintain consistency in the code and to make it easier for someone familiar with Creol to switch to CSL specifications.

Generally speaking, every Creol boolean expression which does not contain side-effect operations or assignments is a CSL expression. CSL extends Creol with *quantified expressions* and with *history predicates* and *functions*.

Table 5.1 illustrates the mapping between basic CSL expressions and first-order logic formulas where $\texttt{e}_i$ represents a CSL expression and $\overline{e_i}$ is its translation to FOL.

CSL Expression	FOL formula
$\sim\texttt{e}_1$	$\neg\overline{e_1}$
$\texttt{e}_1$ `&&` $\texttt{e}_2$	$\overline{e_1} \wedge \overline{e_2}$
$\texttt{e}_1$ `||` $\texttt{e}_2$	$\overline{e_1} \vee \overline{e_2}$
$\texttt{e}_1$ `=` $\texttt{e}_2$	$\overline{e_1} \doteq \overline{e_2}$
$\texttt{e}_1$ `/=` $\texttt{e}_2$	$\neg(\overline{e_1} \doteq \overline{e_2})$
$\texttt{e}_1$ `>=` $\texttt{e}_2$	$\overline{e_1} \geq \overline{e_2}$
$\texttt{e}_1$ `=>` $\texttt{e}_2$	$\overline{e_1} \rightarrow \overline{e_2}$
$\texttt{e}_1$ `<=>` $\texttt{e}_2$	$\overline{e_1} \leftrightarrow \overline{e_2}$
`(\forall x:`T`;` $\texttt{e}_1$`)`	$\forall\ T$ x. $\overline{e_1}$
`(\exists x:`T`;` $\texttt{e}_1$`)`	$\exists\ T$ x. $\overline{e_1}$
`(\forall x:`T`;` $\texttt{e}_1$`;` $\texttt{e}_2$`)`	$\forall\ T$ x. $(\overline{e_1} \rightarrow \overline{e_2})$
`(\exists x:`T`;` $\texttt{e}_1$`;` $\texttt{e}_2$`)`	$\exists\ T$ x. $(\overline{e_1}\ \&\ \overline{e_2})$

Table 5.1: Partial mapping from CSL to FOL

As we will shortly see, nowhere in the specifications we will directly refer to the history for instance as a parameter of our predicates. Having to deal with the history associated with the whole system could be a very demanding task and would lead to extremely complex specifications; as seen in chapter 4 it includes traces of every created object, each switch of thread execution, and communication messages for every running thread.

To provide a more user-friendly environment the CSL language has been designed so that the user could express properties focusing only on the *local history* of the object we are writing specifications about. This means that, when writing an invariant or an operation contract we are just considering the history traces involving the current object and possibly a caller of its methods.

To be more detailed on this topic we need to get acquainted with some definitions. In the next section we are going to introduce some theoretical ground which will help us clarify the semantics of our specifications.

5.3.1 CSL Histories

In chapter 4 we saw how the history is built during program execution. It is a concatenation of messages representing the creation of objects, the invocation and completion requests of methods between objects, and the execution flow ruled by release points.

Following is a short recall to the structure of some of these messages:

Object creation message : $\langle new(o_1, o_2)\rangle$

Invocation message : $\langle invoc(l, \bar{v})\rangle$

Completion message : $\langle comp(l, \bar{v})\rangle$

where o_i represents an object, $\bar{v}$ is the tuple of parameters, and l appears as a label which holds the identity of the caller and the callee as:

Label : $((o_1, t_1), (o_2, (m, t_2)))$

where o_1 and o_2 represent the caller object and the callee object, respectively; t_1 and t_2 are thread IDs and m is the invoked method.

As a counterpart of this messages we need to introduce a *domain* representing the *specification messages*; later we will see how they are related to each other.

Definition 5.3.1. The specification message domain *specMsgDom* is defined as:

$$specMsgDom = \left\{ type\ o.meth\,(\bar{x}) \;\middle|\; \begin{array}{l} type \in \{\texttt{?}, \texttt{!}\}, \\ o \in \{\textbf{this}, \textbf{caller}\} \cup refObj, \\ meth \in methDom, \\ \bar{x} \in domData^* \end{array} \right\}$$

where $refObj$ is the set of all the objects we have a reference to in the current scope, *methDom* represents the domain of all possible method identifiers and *domData* is the domain of all method parameters sorts. For a more detailed description of the last two domains refer to [3].

To clarify this definition, here is how possible specification messages look like:

CSL

```
!obj.meth(3)
!this.meth1(a,b)
?caller.meth2(true)
```

CSL

Now we can concatenate such messages composing a specification history:

Definition 5.3.2. The specification history domain *specHistDom* is defined as:

$$\begin{array}{rl} & \{<>\} \\ specHistDom = & \cup \{sm \mid sm \in specMsgDom\} \\ & \cup \{sm\hat{}sh \mid sh \in specHistDom,\ sm \in specMsgDom\} \end{array}$$

where <> represents the empty history and the symbol ˆ is used to concatenate specification messages.

A possible specification history can be the following:

— CSL —

```
!obj.meth(3)^!this.meth1(a,b)^?caller.meth2(true)
```

— CSL —

Having set the ground domains for the history specifications we proceed introducing some CSL predicates and functions following a top-down approach which will lead us to a detailed description of their semantics.

5.3.2 CSL Predicates and Functions

The first predicate we're going to introduce is '`\HEmpty`'. Its meaning is pretty obvious: it states that the current history is empty. Following is its semantics:

$$\texttt{\textbackslash HEmpty} \triangleq \mathcal{H}_{loc} \doteq \langle\rangle$$

The above definition shows how, when writing specifications, we are referring to a specific history $\mathcal{H}_{loc}$ which represents the *local history* we have already mentioned before. A more detailed explanation of its semantics is going to come later, by now just assume it is the history we are actually interested in.

The next predicate we are introducing is '`\HEndsOn(`sh`)`' which evaluates to *true* iff the *local history* is composed by a sequence of messages where the last ones (i.e. the most recent chronologically speaking) match the pattern represented by the parameter $sh \in SpecHistDom$:

$$\texttt{\textbackslash HEndsOn(}sh\texttt{)} \triangleq \exists\theta, \theta'.\ sh \approx \theta \wedge \mathcal{H}_{loc} \doteq \theta' \frown \theta$$

where $sh \approx \theta$ indicates that the specification history sh matches the history θ. Its formal definition is postponed to the end of this section.

A bit more general is the predicate '`\HContains(`sh`)`' which requires that the history matching sh is contained in the *local history*, not necessarily at its end:

$$\texttt{\textbackslash HContains(}sh\texttt{)} \triangleq \exists\theta.\ sh \approx \theta \wedge \theta \subseteq \mathcal{H}_{loc}$$

where $\theta_1 \subseteq \theta_2$ iff $\exists\theta', \theta''.\ \theta' \frown \theta_1 \frown \theta'' \doteq \theta_2$.

If we want to specify that in the history a message $\langle m_1 \rangle$ is always preceded by another message $\langle m_2 \rangle$, then we are going to use the predicate '`\before`'. It takes as arguments two specification histories and is defined as:

$$\texttt{\textbackslash before(}sh_1\texttt{,}sh_2\texttt{)} \triangleq \begin{array}{l} \forall\theta_2 \subseteq \mathcal{H}_{loc}. \\ \quad (sh_2 \approx \theta_2 \rightarrow \exists\theta_1, \theta.\ sh_1 \approx \theta_1 \wedge \theta_1 \frown \theta \frown \theta_2 \subseteq \mathcal{H}_{loc}) \end{array}$$

it practically states that whenever there is an occurrence of an history matching the pattern specified by sh_2 then the preceding history contains a history matched by sh_1.

Reversely we define '\after' as:

$$\texttt{\textbackslash after}\,(sh_2, sh_1) \triangleq \begin{array}{l}\forall \theta_1 \subseteq \mathcal{H}_{loc}. \\ (sh_1 \approx \theta_1 \rightarrow \exists \theta_2, \theta.\ sh_2 \approx \theta_2 \wedge \theta_1 \frown \theta \frown \theta_2 \subseteq \mathcal{H}_{loc})\end{array}$$

Note that $\texttt{\textbackslash before}\,(sh_1, sh_2) \not\Leftrightarrow \texttt{\textbackslash after}\,(sh_2, sh_1)$. For instance if we have the history being a sequence like $sh_1 \frown sh_1 \frown sh_1$ then $\texttt{\textbackslash before}\,(sh_1, sh_2)$ holds, while $\texttt{\textbackslash after}\,(sh_2, sh_1)$ doesn't.

Next is a function which returns the number of occurrences in the local history of a message matching the passed specification message $sm \in specMsgDom$:

$$\texttt{\textbackslash count}\,(sm) \triangleq count(sm, \mathcal{H}_{loc})$$

where

$$\begin{array}{rl} count(sm, \langle\rangle) & = 0 \\ count(sm, \langle m \rangle \frown \theta) & = \begin{cases} 1 + count(sm, \theta) & \text{if } sm \approx \langle m \rangle \\ count(sm, \theta) & \text{otherwise} \end{cases} \end{array}$$

It's now time to draw the connection between specification histories and actual histories and thus clarify when $sh \approx \theta$ is valid.

When writing specification histories, we are abstracting from labels or thread IDs which are contained in actual histories so that we can focus only on the communicating objects, the called methods, and the parameters. We start defining the correspondence between *specification messages* and *history messages* ($sm \approx \langle m \rangle$):

$$!\,o.meth\,(\bar{x}) \approx \langle msg \rangle \Leftrightarrow \begin{array}{l}\exists o', th, th'. \\ (msg \doteq invoc(((o', th'), (o, (meth, th))), \bar{x}))\end{array}$$

$$?o.meth\,(\bar{x}) \approx \langle msg \rangle \Leftrightarrow \begin{array}{l}\exists o', th, th'. \\ (msg \doteq compl(((o', th'), (o, (meth, th))), \bar{x}))\end{array}$$

where o' is an object and th and th' are thread IDs.

This means that a specification message corresponds to every possible invocation or completion message where the called object, the method name and the parameters match. Then the matching messages are further restricted in our specifications (as we saw in the previous definitions) by requiring them to be contained in the local history.

As we will see in the next paragraph this will imply the object *this* being either the caller or the callee.

Based on the last definition we can now define the correspondence between *specification histories* and actual *histories* ($sh \approx \theta$):

$$\begin{aligned} sh \approx \langle\rangle &\Leftrightarrow sh = <> \\ sm\hat{}sh \approx \langle m\rangle^\frown\theta &\Leftrightarrow sm \approx \langle m\rangle \wedge sh \approx \theta \end{aligned}$$

It's now time to specify which history is the subject of our specifications.

Local History

As stated earlier in this section, when writing specifications we want to reduce our focus on a subset of the whole system-wide history. Let's see what exactly is the history we referred to as $\mathcal{H}_{loc}$ in the semantics of CSL predicates and functions.

We will use the projection operator "/" which, applied to a history, returns a subset of it with all the messages having the original relative order.

Initially we want to filter from the system-wide history only the messages representing an invocation or a completion:

Definition 5.3.3. We define the communication history $\mathcal{H}_{com}$ as:

$$\mathcal{H}_{com} = \mathcal{H}/_{?!}$$

where $\mathcal{H}$ is the system-wide history and the operation '$/_{?!}$' is defined as:

$$\begin{aligned} \langle\rangle/_{?!} &= \langle\rangle \\ (\langle m\rangle^\frown\theta)/_{?!} &= \begin{cases} \langle m\rangle^\frown(\theta/_{?!}) & \text{if } m = invoc(.) \text{ or } m = compl(.) \\ \theta/_{?!} & \text{otherwise} \end{cases} \end{aligned}$$

So far in $\mathcal{H}_{com}$ we obtained a list of messages (wich we will call *communication messages* from now on) containing only requests of methods invocation and completion between all the objects involved in the system. This is the only kind of messages we are dealing with in our specifications.

Now we need to further filter out all the objects we are not interested in while writing specifications. To do this we need to introduce the projection operation on objects:

Definition 5.3.4. We define the projection '$/o$' to an object o with respect to a *communication history* as:

$$\begin{aligned} \langle\rangle/o &= \langle\rangle \\ (\langle m\rangle^\frown\theta)/o &= \begin{cases} \langle m\rangle^\frown(\theta/o) & \text{if } o = toCaller(m) \text{ or } o = toCallee(m) \\ \theta/o & \text{otherwise} \end{cases} \end{aligned}$$

where *toCaller()* and *toCallee()* are two functions which have a *communication message* as parameter and respectively return the caller object or the called object in the message.

Roughly speaking when projecting a history to an object we obtain the history which is somehow related to the object at hand.

Another operation which will be later used in the semantics of CSL is the projection to an object as a caller. With this we will obtain the communication history of an object where it is playing the role of the callee:

Definition 5.3.5. We define the projection to an object o as callee and we write '$/o_{\leftarrow}$' with respect to a *communication history* as:

$$\begin{aligned} \langle\rangle/o_{\leftarrow} &= \langle\rangle \\ (\langle m\rangle^\frown\theta)/o_{\leftarrow} &= \begin{cases} \langle m\rangle^\frown(\theta/o_{\leftarrow}) & \text{if } o = toCallee(m) \\ \theta/o_{\leftarrow} & \text{otherwise} \end{cases} \end{aligned}$$

where *toCallee()* is the function introduced in the previous definition.

It is clear how the history resulting from the projection '$/o_{\leftarrow}$' is a subset of the one resulting from '$/o$' if applied to the same communication history.

Finally, we are now ready to answer to the question "Which history are we actually talking about when writing specifications?". Unfortunately the answer is not unique, so we'll have to go with "It depends!".

In fact, the history under consideration ($\mathcal{H}_{loc}$) is connected with the kind of specification we are writing i.e. whether we are writing an *operation contract* for a method or an *invariant* for an interface or a class we are considering two different histories.

In the following section we are going to further explain this topic and we will go more in the details on how to write specifications.

5.3.3 Operation Contracts in CSL

This section shows how to write CSL operation contracts for methods. Before showing the actual structure of a contact we will finally define the considered local history $\mathcal{H}_{loc}$:

Definition 5.3.6. When writing an *operation contract* we define $\mathcal{H}_{loc}$ as the communication history between the objects held in the special variables *this* and *caller*:

$$\mathcal{H}_{loc} = \mathcal{H}_{com}/this/caller$$

This means that no other object beside the one executing the body of the method at hand (*this*) and the *caller* of such method are involved in operation contracts.

This enforces the meaning of compositional verification which allows us to verify only one method at time abstracting from the rest of the system.

For this reason we can restrict the *specMsgDom* to have only `this` and `caller` as mentionable object in an operation contract.

Following is how a CLS operation contract for the method `sample` looks like:

— CSL ——————————————————————————

```
op sample(in a:Int, b:Int; out z:Int)
requires CSLexpr
ensures CSLexpr
diverges {true, false}
assignable modSet
[ == body ]
```

—————————————————————————— CSL —

where *CSLexpr* is a valid CSL expression and *modSet* is the *modifier set* which we will soon introduce. The *body* is shown between square brackets to indicate that it is not needed in the case the contract belongs to an interface.

The **requires** statement specifies the *precondition* of the operation contract. The *postcondition* is expressed in the **ensures** clause.

The **diverges** keyword states whether we require the method to terminate (i.e. **diverges false**) or if we just want to verify the partial correctness of the implementation and thus allow non-termination (**diverges true**).

Finally the **assignable** clause contains the *modifier set* which expresses the attributes the method is allowed to modify during execution. It is a list of program variables accessible from the current scope or, as special keywords, `\nothing` and `\everything` can be used to respectively indicate that no side effects are allowed to the method or, on the other hand, that the method is allowed to modify all accessible fields.

Whenever one of the above introduced statements is missing, a default value is assigned. In Table 5.2 the default values for each clause are shown.

As precondition of a method, beside the proviously introduced expressions, an extra CSL predicate can be used: '`\firstCall`'. This predicate asserts that the current method must be the first one to be invoked on the current object:

$$\texttt{\textbackslash firstCall} \triangleq \mathcal{H}_{loc\leftarrow} = \langle\rangle$$

Clause	Default value
requires	**true**
ensures	**true**
diverges	**true**
assignable	\everything

Table 5.2: Default values for operation contract clauses

where $\mathcal{H}_{loc\rightarrow}$ is defined as the projection of the local history on *this* as called object:

$$\mathcal{H}_{loc\leftarrow} = \mathcal{H}_{loc}/this_{\leftarrow}$$

5.3.4 Invariants in CSL

In Creol we distinguish mainly between two types of invariants: *interface invariant* and *class invariant*.

The former is used to further specify patterns of interaction and is the only invariant which is visible between different objects. The class invariant, on the other hand, is used to express properties on the internal state of an object and to relate it with the history; it is not visible to other objects, but all the threads of the current object have to take this invariant into account.

Again we have to clarify which history we are dealing with when writing invariants, but once more the answer is not unique. Beside the distinction between class and interface invariant, we divide them in two further categories. Let's see an example code to better illustrate it:

— CSL —

```
interface I
begin
  inv CSLexpr
  with I1
    inv CSLexpr
    op m()
  with I2
    inv CSLexpr
    op n()
end
```

```
class C implements I
begin
  inv CSLexpr
  with I1
    inv CSLexpr
    op m() == body
  with I2
    inv CSLexpr
    op n() == body
end
```

— CSL —

As you can see from the code there are several invariants (marked by the **inv** keyword) involved both in interfaces and in classes. The *global invariants* are the one placed just after the **begin** statement; the properties

expressed by such invariant must be considered by all the methods declared inside the interface or class.

Additionally we can insert in each **with**-block what we will call a *co-invariant* which will deal with all and only the methods contained in the current block.

In *global invariants* it should then be possible to express specifications involving the current object and, when in the case of a class invariant, all the objects we have a reference to when interacting with it. Therefore the *local history* we are going to consider is:

$$\mathcal{H}_{loc} = \mathcal{H}_{com}/this$$

On the contrary, when writing a *co-invariant* we want to focus only on the methods inside the current **with**-block and the caller whose interface is known. Thus the considered history will be:

$$\mathcal{H}_{loc} = \mathcal{H}_{com}/this/caller$$

In this chapter we saw how to write specifications, thus how to give an abstract representation of the behaviour of a program. Now, the new question is how to check if a method implementation respects its specification. This is the topic of the next two chapters.

While the complete semantics can be used for specification purpose and is recognised by the Creol parser (see chapter 8), the verification process is able to handle a subset of it. For operation contracts, only the pre- and postconditions are involved. The implemented calculus only supports the verification of partial correctness of method bodies and the modifier set is so far ignored. Thus methods are considered to possibly modify all accessible fields. Refer to chapter 10 for a further discussion on limitations and future development.

Chapter 6

Proof Obligations

The term *proof obligation* was coined by the Dutch computer scientist E. W. Dijkstra, whose main interest was aimed to applications of formal verification. His idea was to combine specification and implementation so to "develop proof and program hand in hand".

So far we have seen how we can express properties about a program and how they are translated in first-order logic, but no relation has been drawn between this specifications and the implementation of method bodies. This chapter is going to help us understand what is the proof obligation we have to verify to consider a given implementation correct whit respect to its specification.

We had previously given an overview of when to take into account specifications. For instance, we said we want a postcondition to hold after a terminating execution of the method body, given that its precondition was true and we argued that an invariant should hold whenever no operation is running. Here we will unfold this concepts giving a more formal description of how the final proof obligation we want to verify looks like.

For a complete understanding of this and the following chapter, basic knowledge of dynamic logic is recommended. The used dynamic logic is somehow similar to Hoare Logic [8] or the weakest precondition calculus introduced in [9]; a good introduction of first-order logic, which represents the base for this topics, can be found in [10].

Roughly speaking, a dynamic logic is a first-order logic extended with sorts and modalities ($[\cdot]$ and $\langle\cdot\rangle$) surrounding program statements. A complete illustration of the ground on which the used dynamic logic builds on can be found in [3] where the used sort hierarchy and the syntax and semantics of predicates and functions are shown.

We start with expressing the aims of our proof from a semantical point of view.

6.1 Semantics of Creol Dynamic Logic

To compositionally verify the correctness of a program we have to guarantee that each method can rely on the correctness of the other methods. This is achieved by having all the methods of a class preserving the class and interface invariants and co-invariants for the shared variable concurrency to work correctly; in addition to this, when calling a method of any object, its interface invariant and co-invariant must be respected. Moreover, in order to prove the correctness of a method, we must start from the assumption that its precondition is satisfied which will result, for a correct implementation, in the satisfaction of the postcondition.

To give a more formal explanation of what is expressed above we start introducing some terminology. The upcoming definitions follow and extend the definitions used in [2].

For notational simplicity, we are going to assume each method m to belong to exactly one interface, which is a very weak assumption easily obtainable by renaming or qualifying of methods' identifiers.

In the upcoming formulas the following symbols will be used:

$IGlInv(I)$ to indicate the global invariant of interface I;

$ICoInv(W)$ to indicate the interface co-invariant of the with-block W;

$CGlInv(C)$ to indicate the global invariant of the class C;

$CCoInv(m, C)$ to indicate the class co-invariant of the with-block the method m belongs to when implemented in class C;

$IPre(m)$ to indicate the interface precondition of method m;

$IPost(m)$ to indicate the interface postcondition of method m;

$CPre(m)$ to indicate the class precondition of method m;

$CPost(m)$ to indicate the class postcondition of method m.

Further, we will use the symbol $CompInv$ which stands for *"Composed Invariant"* and is defined as:

$$CompInv(m, C) \triangleq \begin{array}{l} CGlInv(C) \wedge\ CCoInv(m, C) \\ \wedge\ \bigwedge_{I \in impl(C)} IGlInv(I)\ \wedge\ ICoInv(with(m)) \\ \wedge\ \bigwedge_{I \in ref(C)} (IGlInv(I)\ \wedge\ \bigwedge_{W \in with(I,C)} ICoInv(W)) \end{array}$$

where $impl(C)$ returns the set of interfaces the class C implements, $with(m)$ returns the with-block m belongs to, $ref(C)$ represents the sets of interfaces potentially invoked by the class C (i.e. the types of the objects

class C has a reference to), and $with(I, C)$ returns the with-blocks of interface I where the co-interface is implemented by C. This last set is used to include all the co-invariants of all the methods class C is able to call.

Note that when talking of any specification (i.e. invariants or operation contracts) we refer to its translation into firs-order logic as showed in the previous chapter. In the following we will refer to the set of all the specifications in a program as *Spec* relatively to which, among other things, we will evaluate the correctness of an implementation.

Based on the above definitions, we can proceed with the base case of dynamic logic formulas, relative to an initial state σ representing an assignment of object attributes and local variables to a value, an initial history θ, an assignment γ of logical variables, and the program specifications *Spec*:

$$(\sigma, \theta, \gamma, Spec, m, C) \models [S]\varphi$$

iff

for all $(\sigma_1, \theta_1) \in \mathcal{M}(S)(\sigma)$:
if

$$\{\theta' \mid \theta' \leq \theta^\frown\theta_1\} \subseteq \begin{array}{l} assume_{IGlInv,ICoInv,\gamma} \\ \cap\ assume_{Post,\gamma} \\ \cap\ rely_{CompInv,m,C,\gamma} \end{array}$$

then

$$\{\theta'' \mid \theta'' \leq \theta_1\} \subseteq \begin{array}{l} commit_{IGlInv,ICoInv,\gamma} \\ \cap\ commit_{Pre,\gamma} \\ \cap\ guarantee_{CompInv,m,C,\gamma} \end{array}$$

and
$(\sigma_1, \theta^\frown\theta_1, \gamma, Spec, m, C) \models \varphi$

where the used sets of histories are defined as:

$$commit_{IGlInv,ICoInv,\gamma} = \left\{ \theta \;\middle|\; \begin{array}{l} \text{if } \theta = \theta_0^\frown\langle invoc(((oid, t), (oid', (m, i))), \bar{v})\rangle \\ \text{then} \\ (\theta, \gamma) \models IGlInv(intf(m)) \wedge ICoInv(with(m)) \end{array} \right\}$$

$$assume_{IGlInv,ICoInv,\gamma} = \left\{ \theta \;\middle|\; \begin{array}{l} \text{if } \theta = \theta_0^\frown\langle comp(((oid, t), (oid', (m, i))), \bar{v})\rangle \\ \text{then} \\ (\theta, \gamma) \models IGlInv(intf(m)) \wedge ICoInv(with(m)) \end{array} \right\}$$

$$commit_{Pre,\gamma} = \left\{ \theta \;\middle|\; \begin{array}{l} \text{if } \theta = \theta_0^\frown\langle invoc(((oid, t), (oid', (m, i))), \bar{v})\rangle \\ \text{then } (\theta_0, \gamma) \models IPre(m) \end{array} \right\}$$

$$assume_{Post,\gamma} = \left\{ \theta \;\middle|\; \begin{array}{l} \text{if } \theta = \theta_0^\frown \langle comp(((oid,t),(oid',(m,i))),\bar{v})\rangle \\ \text{then } (\theta,\gamma) \models IPost(m) \end{array} \right\}$$

$$guarantee_{CompInv,m,C,\gamma} = \left\{ \theta \;\middle|\; \begin{array}{l} \text{if } \theta = \theta_0^\frown \langle yield(\sigma)\rangle \\ \text{then } (\sigma,\theta,\gamma) \models CompInv(m,C) \end{array} \right\}$$

$$rely_{CompInv,m,C,\gamma} = \left\{ \theta \;\middle|\; \begin{array}{l} \text{if } \theta = \theta_0^\frown \langle resume(\sigma)\rangle \\ \text{then } (\sigma,\theta,\gamma) \models CompInv(m,C) \end{array} \right\}$$

Intuitively, this means that S, if executed as body of method m implemented in the context of class C, has to *commit* to the invariants of the interfaces it calls and the preconditions of the called methods, and has to *guarantee* the composed invariant at each release point. Moreover, in case of termination, the formula φ must hold. To accomplish this, S can *assume* the invariants and the postconditions of the replying methods to be true both during and before the execution of S, and it can rely on the other threads of *this* object to establish the composed invariant when releasing control.

As seen in chapter 3, $[S]\varphi$ does not claim termination of any run; in fact the set $\mathcal{M}(S)(\sigma)$ can be empty. We can prove the termination of a sequence of statements by simply claiming the existence of a terminating run and thus writing:

$$(\sigma,\theta,\gamma,Spec,m,C) \models \langle S\rangle\varphi$$

iff

there exists $(\sigma_1,\theta_1) \in \mathcal{M}(S)(\sigma)$ such that:
if

$$\{\theta' \mid \theta' \leq \theta^\frown\theta_1\} \subseteq \begin{array}{l} assume_{IGlInv,ICoInv,\gamma} \\ \cap\ assume_{Post,\gamma} \\ \cap\ rely_{CompInv,m,C,\gamma} \end{array}$$

then

$$\{\theta'' \mid \theta'' \leq \theta_1\} \subseteq \begin{array}{l} commit_{IGlInv,ICoInv,\gamma} \\ \cap\ commit_{Pre,\gamma} \\ \cap\ guarantee_{CompInv,m,C,\gamma} \end{array}$$

and
$(\sigma_1,\theta^\frown\theta_1,\gamma,Spec,m,C) \models \varphi$

In case we have updates in the formula (as described in chapter 3), the following semantics apply:

$$(\sigma, \theta, \gamma, Spec, m, C) \models \{x_1 := e_1 \mid \ldots \mid x_n := e_n\}\varphi$$

iff

$$((\sigma : x_1 \to \mathcal{E}(e_1)\sigma : \ldots : x_n \to \mathcal{E}(e_n)\sigma), \theta, \gamma, m, C) \models \varphi$$

The semantics of Boolean connectives and quantifiers is defined as in first-order logic and are therefore here omitted.

We can now define the *validity* of a formula in the context of method m implemented in class C as:

$$(Spec, m, C) \models \varphi$$

iff

$$\text{for all } \sigma, \theta, \gamma : \ (\sigma, \theta, \gamma, Spec, m, C) \models \varphi$$

6.2 Proof obligation construction

Finally, we arrive to the construction of the "complete" proof obligation for a method m implemented in a class C. Here we want to assert that given the validity of the method precondition in the current state and assuming that its invocation is compliant with the composed invariant, then the execution of its body will result (if terminating) in a state where the method postcondition holds and the composed invariant is preserved.

More formally, for a method declared as **op** m(**in** $\bar{x}$; **out** $\bar{y}$)== *body* the resulting *proof obligation* will be:

$$(Spec, m, C) \models$$
$$Pre(m)(\mathcal{H}, \overline{\mathcal{A}}, \bar{x}) \to \{U^{invoc}_{\mathcal{H}}\}[body;\ \mathbf{return}(\bar{y})]Post(m, C)(\mathcal{H}, \overline{\mathcal{A}}, \bar{x}, \bar{y})$$

where:

$$Pre(m)(\mathcal{H}, \overline{\mathcal{A}}, \bar{x}) = \left(\begin{array}{c} IPre(m)(\mathcal{H}, \bar{x}) \\ \wedge\ CPre(m)(\mathcal{H}, \overline{\mathcal{A}}, \bar{x}) \\ \wedge\ Wf(\mathcal{H}) \end{array}\right)$$

$$Post(m, C)(\mathcal{H}, \overline{\mathcal{A}}, \bar{x}, \bar{y}) = \left(\begin{array}{c} IPost(m)(\mathcal{H}, \bar{x}, \bar{y}) \\ \wedge\ CPost(m)(\mathcal{H}, \overline{\mathcal{A}}, \bar{x}, \bar{y}) \\ \wedge\ CompInv(m, C)(\mathcal{H}, \overline{\mathcal{A}}) \end{array}\right)$$

here, when writing expressions like $IPre(m)(\mathcal{H}, \bar{x})$ we simply mean that the formula $IPre(m)$ may contain statements relative to the current history $\mathcal{H}$ and the in-parameters $\bar{x}$. The symbol $\overline{\mathcal{A}}$ represents the class attributes

which may be addressed in class specifications and $Wf(\mathcal{H})$ holds for well-formed history, meaning we assume to have a history which includes the creation message of **this**, contains invocation messages for all the corresponding invocation ones, and does not have references to objects being **null**. Lastly the abbreviation $U^{invoc}_{\mathcal{H}}$ represents an *update* of the history to a state where it contains an invocation message of method m and the composed invariant holds. Its full form is:

$$\mathcal{H} := \mathsf{some}\ H. \left(\begin{array}{l} \mathcal{H} \leq H \\ \wedge\ \langle invoc(((\mathbf{caller}, i), (\mathbf{this}, (m, j))), \bar{x})\rangle \subseteq H \\ \wedge\ CompInv(m, C)(H, \overline{A}) \end{array} \right)$$

Here and in the following, we use the quantifier some which is not implemented in the logic but better clarify the meaning of the actually expressed formula. For instance, an update formula like $\{\mathcal{H} := \mathsf{some}\ H.(f(H) \wedge \mathcal{H} \leq H)\}\phi$, which we will following refer to as an *anonymizing* update, is rewritten to:

$$\forall H_0.\ (\mathcal{H} \doteq H_0 \rightarrow \forall H_1.\ \{\mathcal{H} := H_1\}((f(H_1) \wedge H_0 \leq H_1) \rightarrow \phi)) \qquad (6.1)$$

where the symbol $\leq$ is used to state that the old history, H_1, is a prefix of the new one, H_0.

Chapter 7

Creol Calculus

The only missing step is to reduce our proof obligation into a logic formula which does not contain any code, so that we can finally show its validity applying the well known properties of first-order logic. This is achieved by progressively applying sequent rules (see chapter 3) to each Creol statement contained inside the modalities, until we arrive to have an empty modality which can be removed.

We are going to start showing the sequent rules handling sequential statements which will provide us with a better understanding about the reasoning process; then, the rules applied on concurrent constructs, such as method invocations and release statements, will be explained. In the latter, the program specifications will play a central role thus our focus will go on such statements and their relation to the construction of the history.

7.1 Sequential Constructs

The most basilar Creol construct is **`skip`** which corresponds to the commonly called "no operation" statement which, by definition, has no effect. The sequent rule handling such statement is the following:

$$\mathsf{skip}\ \frac{\langle\!|\omega|\!\rangle\phi}{\langle\!|\texttt{skip; }\omega|\!\rangle\phi}$$

here and in the following, the $\langle\!|\cdot|\!\rangle$ symbol is used as a replacement for the modality to indicate that the same rule applies both in the context of a box and a diamond modality.

The application of this rule simply delete the **`skip`** statement and leave the rest of the code unchanged. No further modification is made to the state.

Another very simple rule is the one applicable on the **block** statement. This rule highlights the difference between proving total and partial correctness of an implementation:

$$\mathsf{blockBox}\ \frac{\mathit{true}}{[\mathbf{block;}\ \omega]\phi} \qquad \mathsf{blockDia}\ \frac{\mathit{false}}{\langle\mathbf{block;}\ \omega\rangle\phi}$$

The **block** statement causes the program execution to stop, thus a non-terminating program is always partially correct (box modality) but never totally correct (diamond modality).

Now, we turn to declaration statements. The used rules reflect the implicit assignment of variables to their default value:

$$\mathsf{intDecl}\ \frac{\{i := 0\}\langle\!|\omega|\!\rangle\phi}{\langle\!|\mathbf{var}\ i\mathbf{:\ Int;}\ \omega|\!\rangle\phi} \qquad \mathsf{boolDecl}\ \frac{\{b := \mathit{false}\}\langle\!|\omega|\!\rangle\phi}{\langle\!|\mathbf{var}\ b\mathbf{:\ Bool;}\ \omega|\!\rangle\phi}$$

The above example shows the declaration rules for integers and boolean variables, similar rules apply for labels and objects which are implicitly initialized with the *null* value. Moreover, this rules show how updates are generated during the reasoning process. They will eventually be applied in later stages when the involved variables are recalled.

Similar rules are applied in case of assignments. The simplest case occurs when the right side of the assignment is a terminal expression (denoted by te) such as a variable or a literal. On the other hand, when we have an expression, it is recursively unfolded until we get a top level operator applied to terminal expressions. Following, we show such rules for assignment of integers and and the handling of an addition:

$$\mathsf{assign}\ \frac{\{x := te\}\langle\!|\omega|\!\rangle\phi}{\langle\!|x\ \mathbf{:=}\ te\mathbf{;}\ \omega|\!\rangle\phi} \qquad \mathsf{addTerm}\ \frac{\{x := te_1 + te_2\}\langle\!|\omega|\!\rangle\phi}{\langle\!|x\ \mathbf{:=}\ te_1 + te_2\mathbf{;}\ \omega|\!\rangle\phi}$$

$$\mathsf{add}\ \frac{\langle\!|x'\mathbf{:=}\ e_1\mathbf{;}\ x''\mathbf{:=}\ e_2\ \mathbf{;}\ x\mathbf{:=}\ x' + x''\mathbf{;}\ \omega|\!\rangle\phi}{\langle\!|x\ \mathbf{:=}\ e_1 + e_2\mathbf{;}\ \omega|\!\rangle\phi}$$

In case an error occurs, like a division by zero, the behaviour is simulated by blocking the execution of the program. The rule applied to a division distinguishes between the case where the divisor is not zero and thus an update can be added, and the case of a division by zero where a **block** statement is added:

$$\mathsf{DivTerm}\ \frac{(\neg te_2 \doteq 0 \rightarrow \{x := te_1/te_2\}\langle\!|\omega|\!\rangle\phi) \wedge (te_2 \doteq 0 \rightarrow \langle\!|\mathbf{block;}\ \omega|\!\rangle\phi)}{\langle\!|x\ \mathbf{:=}\ te_1/te_2\mathbf{;}\ \omega|\!\rangle\phi}$$

The shown rules give us a wide overview on the reasoning process adopted during verification. The sequent rules for the rest of the Creol sequential statements are here omitted. A complete reference can be found in chapter 7 of [3].

Now, we shift the focus to concurrent statements which hold the main significance for this thesis work and for the understanding of the treatment of operation contracts of called methods and invariants in the verification process.

7.2 Concurrent Constructs

As it has been mentioned several times, Creol supports concurrency in different ways: distinct objects are thought as running on distinct processors, thus they can operate in parallel, moreover each object can process several threads (instances of methods) which execute interleaved following a scheduling process driven by explicit release points. In this section we will see how this concurrency affects the verification process in its lowest level and how specifications are dynamically integrated in the construction of the proof.

We begin dealing with inter-thread communication within an object. Whenever a thread releases the control of the object processor, any other ready thread is allowed to take over and start its execution from the current state which is partially unknown to the new thread. This is because the shared memory (i.e. the class attributes) may have been modified by previously running threads. Anyhow some assumption can still be made, in fact we can rely on each releasing process to fulfil the contract between all the object threads, represented by the composed invariant. In the same way, when the initial thread resumes its execution, the global state will be changed, but the validity of the invariant can still be assumed.

Following is the sequent rule handling the **`release`** statement, in which all the above mentioned considerations will be applied:

$$\text{release}\ \frac{\Gamma \vdash \mathit{CompInv}(m, C)(\mathcal{H}, \overline{\mathcal{A}}), \Delta \qquad \Gamma \vdash \{U_{\mathcal{H},\overline{\mathcal{A}}}\}[\omega]\phi, \Delta}{\Gamma \vdash [\texttt{release;}\ \omega]\phi, \Delta}$$

As you can see, the rule splits the proof into two branches: in the first one we have to show that the composed invariant is preserved, while the second branch models the execution of the code following the **`release`** when the thread resumes. In the latter case, the execution will start from an unknown but fixed state where the composed invariant can be assumed.

This is expressed by the update $U_{\mathcal{H},\overline{\mathcal{A}}}$ which stands for:

$$\mathcal{H}, \overline{\mathcal{A}} := \mathsf{some}\ H, \overline{A}.(CompInv(m, C)(H, \overline{A}) \wedge \mathcal{H} \leq H)$$

Here the system history is extended and the class attributes are *anonymized*, still the obtained state must preserve the invariant.

Note how this rule, as well as the ones which will follow in this section, is strictly related to the context we are executing the code in. For instance here the composed invariant is dynamically loaded depending on which method we are currently verifying and its class.

Similarly to the previous rule, we can handle the **`await`** $l\,?\ (\bar{x})$ statement by splitting the proof into two new branches: one will show that the invariant is established, the other will continue the execution relying on it and with the additional assumption that the method assigned to the label l has completed its execution:

$$\mathsf{awaitLabel}\ \frac{\begin{array}{l}\Gamma \vdash CompInv(m, C)(\mathcal{H}, \overline{\mathcal{A}}), \Delta \\ \Gamma \vdash \{U_{\mathcal{H},\overline{\mathcal{A}}}\}(\exists\ \bar{p}.\ Comp(\mathcal{H}, l, \bar{p}) \rightarrow\ [\omega]\phi), \Delta\end{array}}{\Gamma \vdash [\textbf{\texttt{await}}\ l\,?\ (\bar{x})\ \texttt{;}\ \omega]\phi, \Delta}$$

with the predicate $Comp(\mathcal{H}, l, \bar{p})$ we assert that the history $\mathcal{H}$ contains a completion message with label l and return values $\bar{p}$ which will then be assigned to the variables $\bar{x}$.

By replacing "$\exists\ \bar{p}.\ Comp(\mathcal{H}, l, \bar{p})$" with "$\langle x := b\rangle x \doteq \textbf{\texttt{true}}$" in the previous rule, we obtain the sequent rule handling the **`await`** b statement, where b is a boolean guard. This will assure that, when resuming the execution, the guard evaluates to true without having any error occurred.

It is important to notice how the rules introduced so far enforce what we semantically described in the previous chapter when presenting the validity of formulas. When such statements occur, the history is marked with *yield* and *resume* messages, thus the composed invariant must be taken into account as expressed by the $guarantee_{CompInv,m,C,\gamma}$ and $rely_{CompInv,m,C,\gamma}$ sets.

Similar to the $Comp(\mathcal{H}, l, \bar{y})$ predicate used above, we have $Invoc(\mathcal{H}, l, \bar{x})$ which is going to be used in the next rules and guarantees the existence of an invocation message in $\mathcal{H}$ containing the label l and the in-parameters $\bar{x}$. These predicates are monotonous with respect to the extension of the history, meaning:

$$Comp(H_0, l, \bar{y}) \wedge H_0 \leq H_1 \rightarrow Comp(H_1, l, \bar{y})$$

Every message contained in a prefix of an history is also contained in the extended history, moreover we can state that if a message is not contained in

a history, then it is not contained in any of its prefixes. This properties are heavily used during the verification process since no actual data structure is used to model the history, on the contrary, we base our proof asserting properties on the current history and deductively expand our knowledge in order to obtain more premises on which to base the proof.

Now, we continue with the sequent rules applied to method invocation and completion statements. The used approach is based on the principle of substituting the called method with its operation contract. A different approach, usually called *inlining*, would be to replace the method invocation with its full body. This would imply verifying the complete implementation every time a method is called. Moreover, using inlining would indicate that the method body is executed when the method is called, which is not true in Creol programs since the execution of a method depends on the scheduling policy. The compositional approach adopted in this work is thus here highlighted, since every method implementation needs to be proven correct only once with respect to its specifications which, in turn, can be used as a behavioural description of the method when it is invoked elsewhere.

Assuming we have a valid invocation of a method, we need to show that its precondition is satisfied in the current state, in addition we want the invocation to be compliant with the interface invariants of the involved object:

$$\textsf{invoc}\ \frac{\begin{array}{l}\Gamma \vdash o \doteq \texttt{null} \rightarrow \langle\!|\texttt{block; } \omega|\!\rangle\phi, \Delta \\ \Gamma \vdash \neg o \doteq \texttt{null} \rightarrow \mathit{IPre}(\mathit{mtd})(\mathcal{H}, \bar{x}), \Delta \\ \Gamma \vdash \neg o \doteq \texttt{null} \rightarrow \{l := ((\texttt{this}, \mathit{me}), (o, i))\}\{U^{\mathit{invoc}}_{\mathcal{H}}\} \\ \qquad\qquad (\mathit{IGlInv}(I)(\mathcal{H}) \ \wedge\ \mathit{ICoInv}(\mathit{with}(\mathit{mtd}))(\mathcal{H}), \Delta) \\ \Gamma \vdash \neg o \doteq \texttt{null} \rightarrow \{l := ((\texttt{this}, \mathit{me}), (o, i))\}\{U^{\mathit{invoc}}_{\mathcal{H}}\}\langle\!|\omega|\!\rangle\phi, \Delta\end{array}}{\Gamma \vdash \langle\!|l\,!\,o.\mathit{mtd}(\bar{x})\texttt{; } \omega|\!\rangle\phi, \Delta}$$

The upper branch handles the case where the referred object is **null**, which results in the blocking of the execution; in the succeeding two branches the correctness of the invocation is examined and in the last one the execution is continued. Notice how the precondition is checked before the history is updated with the invocation message (see $\mathit{commit}_{\mathit{Pre},\gamma}$ in the previous chapter), while the invariants are evaluated on the updated history as required in $\mathit{commit}_{\mathit{IGlInv},\mathit{ICoInv},\gamma}$. The interface I is the type of o, remember that in Creol objects are typed by interface. The first update involved in the rule assigns a new value to the label l so that it can be successively recalled to retrieve the out-values of *mtd* (the symbol i represent a newly created value assigned to the new thread ID). Secondly we have the update of the history to an extended history containing the invocation message $\mathit{invoc}(l, \bar{x})$. We can rewrite it as:

$$\mathcal{H} := \textsf{some}\ H.(\mathcal{H} \leq H \wedge \mathit{Invoc}(H, l, \bar{x}))$$

Similarly to the previous rules, when applying this kind of rule, the precondition and the invariants must be dynamically loaded depending on which is the actual object and method we are invoking.

Next, we present the rule for the handling of completion statements like $l?\,(\bar{y})$. Here the consistency of the call is based on checking if a previous invocation has been assigned to the label at hand. If this is not the case, i.e. the label is **null**, the execution is blocked; otherwise the system state is updated and the execution continues:

$$\textsf{comp}\ \frac{\begin{array}{c}\Gamma \vdash l \doteq \texttt{null} \rightarrow [\texttt{block;}\ \omega]\phi, \Delta \\ \Gamma \vdash \neg l \doteq \texttt{null} \rightarrow \{U^{comp}_{\mathcal{H},\bar{y}}\}[\omega]\phi, \Delta\end{array}}{\Gamma \vdash [l?(\bar{y})\texttt{;}\ \omega]\phi, \Delta}$$

The update $U^{comp}_{\mathcal{H},\bar{y}}$, similarly to the one in the previous rule, extends the history adding the pertinent completion message and *assuming* the invariants and the postcondition, fulfilling what expressed in the previous chapter in $assume_{IGlInv,ICoInv,\gamma}$ and $assume_{Post,\gamma}$. Moreover, here, the variables $\bar{y}$ are updated with the return parameters of the method assigned to the label, whose values are unknown but assumed to be compliant with the postcondition of the method. The full form of the update is:

$$\mathcal{H}, \bar{y} := \textsf{some}\ H, \bar{p}.\ \left(\begin{array}{l} \mathcal{H} \leq H \wedge\ Comp(H, l, \bar{p}) \\ \wedge\ IGlInv(I)(H) \wedge\ ICoInv(with(mtd))(H) \\ \wedge\ IPost(mtd)(H, \bar{p}) \end{array}\right)$$

where the interface I and the method mtd are obtained from the label.

The last rule we are going to examine is the one for the **return** statement, which simply adds a completion message for the currently verified method m to the history and, since this is going to be the last executed statement, removes the modality:

$$\textsf{return}\ \frac{\Gamma \vdash \{U^{return}_{\mathcal{H}}\}\phi, \Delta}{\Gamma \vdash \langle\!\langle \texttt{return}\,(\bar{y})\,\rangle\!\rangle\phi, \Delta}$$

where $U^{return}_{\mathcal{H}}$ is rewritten to:

$$\mathcal{H} := \textsf{some}\ H.(\mathcal{H} \leq H \wedge Comp(H, ((\texttt{caller}, i), (\texttt{this}, (m, j))), \bar{y}))$$

Note how in the return rule we do not check the validity of composed invariant and postcondition since this is already included in the proof obligation, thus they will be contained in ϕ.

Chapter 8

System Implementation

A relevant part of this thesis work has been centred around implementing a prototypical version of the verification system for Creol programs with respect to their CSL specifications.

The implementation can be seen as the composition of two main branches: one involving the integration of CSL into a Creol parser and the other consisting in the extension of the KeY tool for supporting the loading of Creol programs and for compatibility with the CSL specification language.

This chapter will give a wide overview of the architecture of the implemented system. The description is not aimed to be fully exhaustive, knowledge of object-oriented programming paradigms and construction of language interpreters is required. Moreover, notions of formalisms as finite state machines (FSMs), graphs and abstract syntax trees (ASTs) will be assumed to be known.

The reader is highly encouraged to have the Java code at hand while reading the chapter to fully understand the description. Additional helpful notes can be found as comments in the source code.

The code is protected under the GNU general public license and can be obtained upon request from the KeY project Web page[1].

In the following the acronym CML (standing for Creol Modelling Language) will be used to refer to the combination of Creol code and CSL specifications.

8.1 Program parsing

In a previous work [3], a parsing library for Creol programs, namely jCreol, has been implemented. It supports the recognition of valid Creol statements and the resolving of program references. This library has been improved and

[1] www.key-project.org

adequately extended to allow the user to write inline CSL specifications and to check their consistency. Moreover functionalities for the collection of the data necessary for the verification process have been added.

We start by giving an overview of the execution flow followed by the system when loading a CML program, then we will describe with more details each performed procedures.

When running jCreol with a CML file as input, the following steps are executed:

- An interpreter is called to validate the input code. If the the code is accepted, i.e. the program is syntactically correct, an AST in generated.
- The AST is translated into a graph data structure.
- The graph is traversed by a *walker* which collects the identifiers of declared classes, interfaces, and other references. Together with the corresponding nodes of the graph, they are stored in a *symbol table*.
- A second walker is launched on the graph and a *program structure* is created: an object is created for each class and interface and internal attributes point to defined methods and with-bocks which, in turn, are represented by other objects with references to the containing class or interface.
- Finally, another walker traverses the graph and modify it resolving used references such as variables and interface identifiers.

Interpreter

The implementation of the interpreter for CML has been carried out with the use of the ANTLR Parser Generator[2] which is a very powerful tool for the construction of interpreters or translators and for the generation of trees. A very exhaustive guide to this tool can be found in [11].

To create a *lexer* and the *parser* for the target language, it is necessary to provide ANTLR with a grammar specifying the structure of the language. A *context free* LL(*) grammar, expressed using Extended Backus Naur Form (EBNF), is required.

The grammar used for CML, which can be found in Appendix A, is an LL(1) grammar. This choice has been made given its better efficiency with respect to grammars with higher values of lookahead. Additionally the grammar has been enriched with rewriting rules for the construction of the resulting abstract syntax tree (AST).

[2]ANTLR: ANother Tool for Language Recognition, www.antlr.org

The file containing the grammar (*Creol.g*) and the source code for the obtained lexer (*CreolLexer.java*) and parser (*CreolParser.java*) can be found in the *antlr* package of the source code of jCreol.

When loading the CML file, first the lexer translates the code in a sequence of tokens, then the parser checks the consistency of the input stream with respect to the grammar. Finally the AST is built following the rewriting rules. In case of syntactic errors, an exception is launched and an error message reports the found inconsistency and its position within the code.

Graph construction

The tree generated by the ANTLR parser is translated into a graph structure. The classes used for this task are all contained in the package *graph*.

First, an instance of *Graph* is created and its method *addAST* is called to perform the translation of the passed AST. Here the tree is walked through and for each encountered node an instance of *GraphNode* is created and initialized with the corresponding text (i.e. code snippet) and token. Moreover a *GraphEdge* is created for each parent-child pair so to maintain the same hierarchy denoted by the AST.

The result is a directed, acyclic graph constituted of data structures which helps in traversing and modifying it.

Filling symbol table

When the graph has been created, its method *fillSymbolTable* is invoked. Here an instance of the class *Walker* (contained in the package *walker*) begins wondering the graph performing a left-depth-first visit of all the nodes.

When initializing the walker, an instance of a class implementing the interface *Behavior* (package *walker.behavior*) must be provided. A *behavior* specifies actions to be performed before the traversing of the graph (*init*), when going *up* or *down* along an edge, and when finishing the traversing (*finish*). An additional action, namely *downNotInTree*, is available which specifies the routine to perform when encountering a modified edge of the original tree (this will occur after the resolving step, as we will shortly see).

For this step the *FillSymbolTable* behavior is used, which collects a reference to all the declared interfaces, classes, functions, and data types and stores them in an instance of the class *SymbolTable* (package *symboltable*) passed as parameter when initialized. For each identifier an entry is added to a hash map contained in the symbol table according to its type; additionally, the node of the graph storing its declaration is saved since it will be used later on when resolving references.

In case an error occurs, e.g. two classes have been defined with the same identifier, or there are references to not defined interfaces, an error message will be shown providing the details.

Program structure

At this point a structure representing the provided program is created; the classes used for this task are collected in the package *programStructure*.

An instance of *CreolProgram* is created initially, then, for each defined class and interface an instance of *CreolProgramClass* and *CreolProgram-Interface* is created respectively and a reference to these objects is saved in the initial *CreolProgram* object.

Additionally, other relevant informations are stored as attributes in the newly created instances, e.g. for each class the implemented interfaces are collected in a vector as well as the class attributes and the invariants.

A reference to instances of *CreolProgramMethod*, representing all the declared and implemented methods, is added to created classes and interfaces; this structure in turn contains references to the method parameters, its body, and its operation contract.

In case there are methods defined inside with-blocks, then an instance of *CreolWith* is created for each co-interface of each class or interface. The reference is stored in the containing class or interface and co-invariants are added.

To accomplish this, another *Walker* is instantiated and initialized with the behavior *Walker2FSM* (in *walker.behavior*). This behavior forwards the actions performed by the walker along the tree to a *Finite state machine* which tracks the context the walker is in and executes appropriate actions when shifting from a state to another. The *Walker2FSM* behavior must be provided with an FSM layout (extending the abstract class *Layout* in the package *finitestatemachine*) which describes the *states* constituting the FSM and the *transitions* between its states. To create a layout it is necessary to define all the different states and assign them the corresponding tokens, further the possible transitions must be added to each state specifying the type (i.e. *up*, *down*, or *downNotInTree*) and the action to perform during the transition (an instance of a class extending the abstract class *Action* must be provided). This approach makes easier to define operations to perform while walking the graph, moreover it is possible to have a graphical representation of the used layout provided by the method *printDot* of the class *FiniteStateMachine* which creates a Graphviz[3] file displayable with any image viewer.

[3]Graphviz: Graph Visualization Software, www.graphviz.org

For the purpose of creating the program structure, the layout *ProgramStructureLayout* from the package *finitestatemachine.structure* has been implemented.

When the structure is complete, a check is run on it to establish if the program is consistent, i.e. all the interfaces implemented by classes exists and all the classes contains all the methods declared in the implemented interfaces and have the same co-interfaces; in addition the interface specifications are added to the implementing classes and relative methods.

Errors are thrown if inconsistencies are found.

The obtained structure allows for easy access to all the elements of a program. For instance, given a method, it is easy to get its pre- and post-condition, obtain the with-block containing it (thus its co-interface and co-invariant), or the class it is defined in, thus its class invariant and the invariants of all the implemented interfaces. As one may now foresee, this is going to be handy during the verification process as we will see in the next section.

Reference resolving

The last performed step checks the consistency of method bodies and specifications with respect to the used identifiers. Whenever a reference is made to a class, an interface, a function, or a data type, it is checked if its identifier is stored in the previously filled symbol table. At the same time nested scopes are used to store declared variables and to resolve them when used in the code.

For this task another walker provided with the previously described *Walker2FSM* behavior is employed. This time the used layout is the *ResolveLayout* defined in the package *finitestatemachine.resolve*. This layout causes the walked graph to be modified: all the nodes corresponding to the use of a reference are replaced by the node holding its declaration. All the newly introduced edges are marked as being not part of the original graph by setting to *false* the attribute *belongsToTree* of the class *GraphEdge*. This distinction allows for maintaining a directed spanning tree so that no loop occurs when walking again the modified tree. This is why the action *downNotInTree* is supported by the walker and the FSM.

8.2 Integration with KeY

The functionalities described above can be both loaded running the *Main* class, or called by an external program using the class *jCreolExternal*. KeY uses this class to interact with jCreol.

As before, the aim of this section is to understand the program flow more then the actual structure of the architecture. Again, we advice to look at

the code for a better understanding of the following description.

The base package for the here described code is *de.uka.ilkd.key* located in the *key/system* source folder. In the following this will be the prefix for all the mentioned packages and thus will be omitted.

Previously to this work the system supported the loading of Creol statements from a key file containing the hard-coded proof obligation together with the problem-specific rules and the desired specification expressed in logic. The details about the implementation of this part can be found in chapter 8 of [3].

The aim of the code introduced here is to add to KeY the capability of loading full Creol programs with inline CSL specifications, translate the specifications, and automatically generate the proof obligation and the needed rules.

Loading of program

The first class called from KeY when loading a Creol program is *CreolServices* in the package *lang.creol*; this class forwards all the calls to the final class *CreolLoader* in *lang.creol.loader* which in turn has a reference to jCreolExternal and acts as a bridge between KeY and jCreol.

At this point jCreol preforms all the steps described in the previous section and, if no errors are encountered, a *CreolProgram* structure is created and accessible.

As next step, all the declared interfaces are introduced in the *sort namespace* so to be recognized as valid types for program variables by KeY. Moreover, program variables representing the methods' identifiers are created: they will be used as parameters for history predicates and functions during the verification process. Basic classes for the instantiation of Creol objects' sorts are available in the package *lang.creol.sort*.

Proof obligation

Now the user will be prompted to select the method to verify from a *proof obligation browser*. The browser shows a tree representing all the defined classes in the program and its methods.

For the implementation of this section, the utilized classes are available in the package *lang.creol.proofObligation.browser*. The code for the implementation of the browser (*CreolPOBrowser*) reuses partially the one used for loading of Java programs; the visualized tree (instance of the class *CreolClassTree*) uses the previously created *CreolProgram* structure for the creation of the tree nodes.

So far the system provides a single proof obligation (see chapter 6) and it is implemented in the *CreolCompletePO* class contained in the package

lang.creol.proofObligation. Other than this class, the *AbstractCreolPO* is provided which implements common functionalities for the generation of proof obligations for future implementation of additional ones.

When the user has selected the target method and started the proof, an instance of *CreolCompletePO* is created and initialized with the pertinent *CreolProgramMethod* instance. From here, with consecutive calls to *CreolLoader*, first the method body is translated to a KeY-specific AST, then the specifications for the method at hand are translated into logical *terms*.

The translation of the body reuses the code previously implemented for the translation of Creol statements, for more details on this section refer to [3]. The translation of specifications puts into effect what semantically described in chapter 5 and it is further explained in the next paragraph.

The last task performed in the *CreolCompletePO* instance, is inserting the translated body into *modalities* and composing the proof obligation as shown in chapter 6.

This is achieved employing the functionalities provided by the class *CmlTermBuilder* in the package *lang.creol.loader.cml2Term*. This class extends the *TermBuilder* class already implemented in the KeY system (package *logic*) which allows for building complex terms by hand and automatically applies basilar simplifications; *CmlTermBuilder* adds Creol specific functionalities and, as we will see next, is one of the central classes for the translation of specifications into terms.

Translation of specifications

In KeY, logical terms and formulas constituting the proof must be an instance of the class *Term* provided in the package *logic*. The aim of this section is indeed the conversion of the *Graph* embodying a CSL specification into a *Term* instance, which can be later composed in a proof obligation as seen before.

The classes utilized for this task are contained in the package *lang.creol.loader.cml2Term*.

A *Walker2FSM* walker, imported from jCreol, is here used again; the employed layout for the FSM is *CmlLayout*. While walking the graph, depending on the FSM transition initiated, an *action* is triggered and the *CmlTermBuilder* instance is invoked. This class, other than the common functionalities inherited by its super-class *TermBuilder*, offers all the necessary operation to translate the CSL graph into a final *Term* and to retrieve the terms corresponding to CSL predicates and functions.

Since the *Term* structure is *immutable*, i.e. cannot be changed after creation, the construction of the whole term translating the CSL graph must be carried out in a bottom-up manner, i.e. starting from the terminal nodes

which will be the sub-terms of their parent graph node which in turn will be a sub-term of the nodes with lower depth.

A stack is thus used to store the terms corresponding to sibling nodes and when we are walking up from their parent node the new term is constructed and stored in the stack. The same process repeats until we reach the root of the graph.

To better understand the process, an example is here given.

Example 8.2.1. Assume we are about to translate the following CSL specification:

— CSL —

```
a > 0 && \HContains(!this.m(true))
```

— CSL —

where `a` is a declared parameter or attribute and `m` is the identifier of a method.

A simplified version of the graph that will be created for this CSL code is shown in Figure 8.1.

The translation will start from the leaves '`a`' and '`0`'; they will be translated into terms and stored in the stack. When going up from the node '`>`' the term corresponding to the "greater than" function will be created and the two sub-terms will be retrieved from the stack and replaced by the new term.

Now the walker will go down to the leaves of the '**`HContains`**' tree and, following the same procedure, stores them in the stack and then composes them to generate an invocation message ('`!`') which will then be the parameter for the '**`HContains`**' function term.

Finally, the stack will contain only the two sub-terms for the '`&&`' function, so they will be fetched and set in conjunction. The obtained term is then stored and available by calling the *getTerm* method of the *CmlTermBuilder* instance.

8.2.1 Rules

For the implementation of the sequent rules described in chapter 7, two different approaches have been followed.

Most of the rules are implemented in the *taclet* language, whereas dynamic rules, such as the one handling Creol concurrent constructs, are built as an implementation of the interface *BuiltInRule* from the package *rule*.

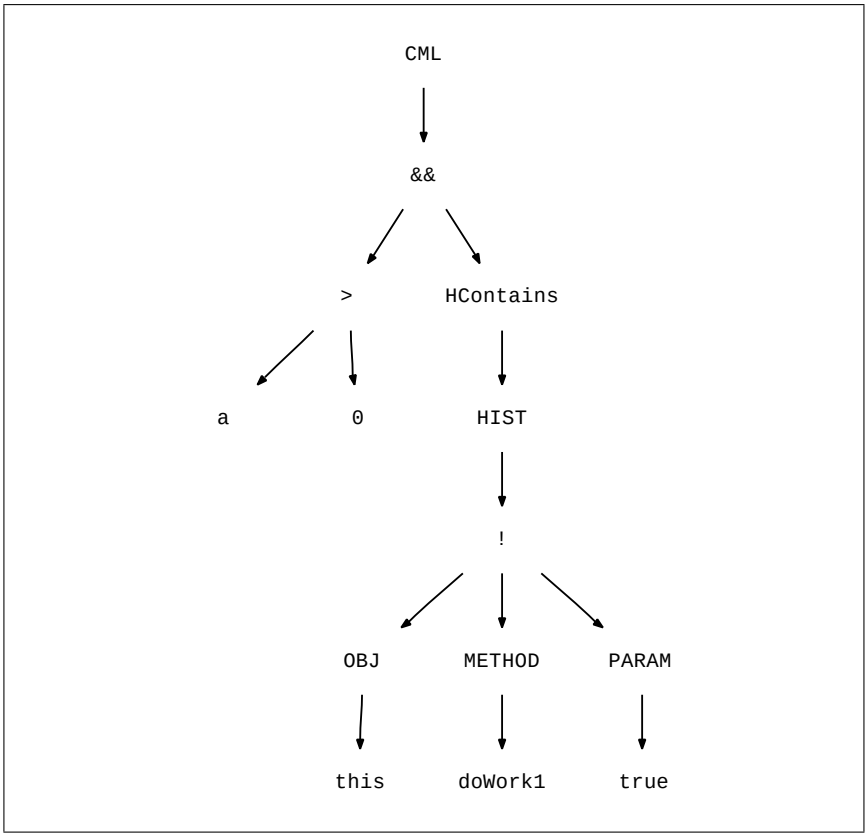

Figure 8.1: Graph representing a CSL expression

Taclets

The *taclet* is a formalism used in the KeY system to easily describe a sequent rule. Mainly it expresses the applicability of a rule, obtained by an AST matching mechanism, and the effect of the application of such rule, that is how the AST will be modified.

Historically the *taclets* have been introduced by E. Habermalz under the name of "Schematic Theory Specific Rules" [18]. An in-depth explanation of the taclet language can be found in chapter 4 of [1], where they are first introduced with a wide range of examples and then a more formal description is provided.

To have an idea of what a taclet looks like, we give an example:

Example 8.2.2. Consider the following rule, impRight, which represents a classical first-order rule commonly used during the verification process:

$$\mathsf{impRight}\ \frac{\Gamma, \phi \vdash \psi, \Delta}{\Gamma \vdash \phi \to \psi, \Delta}$$

The KeY file needed to define such rule will contain the following lines:

KeY

```
\schemaVariables{
      \formula phi, psi;
}
\rules{
      impRight {
               \find(==> phi -> psi)
               \replacewith(==> psi)
               \add(phi ==>)    };
}
```

KeY

Here, first the used *schema variables* are defined; the specified sort defines which expressions the variable can stand for. In the above example `phi` and `psi` represents arbitrary formulas; the taclet language provides for all the sorts necessary to match also variables and terms and allows for defining further sorts for matching program-specific expressions.

The rules are then defined. The **`\find`** clause defines the pattern that must occur for the rule to be applicable; the **`\replacewith`** and **`\add`** clauses describe the alteration caused by the rule when applied. The sequent arrow (`==>`) is used to specify whether a rule is applicable in the *succedent* or the *antecedent* of a proof; if omitted the rule will be applicable in any context.

A taclet used for the simplification of CSL specifications comes next:

— Taclet —

```
HContainsEmpty{
      \find (HContains(cmlEmptyHist,#hist))
      \replacewith (true)
};
```

— Taclet —

which simply states that the empty history is always contained in any history.

The taclet language provides for many more constructs, here omitted, for the implementation of more complex rules; for instance, it is possible to specify conditions on variables and add other needed constraints for the applicability of rules.

All the KeY files containing the CML taclets can be found in the folder *resources/de/uka/ilkd/key/proof/rules/lang/creol/* and its sub-folders.

Built-in rules

Taclets are very easy and intuitive to write, but have the limitation of having a fixed structure which can't dynamically change, for instance we cannot define a general taclets matching the invocation of any method. For this reason, rules needed for the handling of such constructs (i.e. the ones defined in section 7.2) are implemented as *built-in rules*.

The source code of the used classes is located in the package *lang.creol.profile.rules*. The abstract class *AbstractCreolRule* implements common functionalities needed for the implementation of the existing rules and can be reused for future extension of the rule set.

Similarly to taclets, this classes contain a method, namely (*isApplicable*), which performs an AST matching to establish if the rule at hand is applicable; besides, the method *apply* defines the procedures to perform when the rule is applied.

For instance, the class *CreolMethodInvocationRule*, corresponding to the sequent rule invoc from section 7.2, when applied splits the proof, recalls the specifications dealing with the method the rule is applied on, and builds up the structure of the new proofs.

Here the translation procedure described in the previous section is executed again and the *CmlTermBuilder* instance is again employed for the construction of the involved terms.

Chapter 9

An example scenario

This chapter will provide a step by step description of the verification process for an example model.

First we introduce an hypothetical system suitable for an interesting analysis, then we show how to model this system using the Creol language. We add some CSL specifications to the model and finally we highlight meaningful steps in the verification process using the KeY tool.

9.1 The system

Consider the simple system depicted in Figure 9.1. It represents a very common scenario in distributed systems. It is composed of a node (Node A) which resources are shared between two different threads (thr1 and thr2) and the other node (Node B) which embodies a used resource.

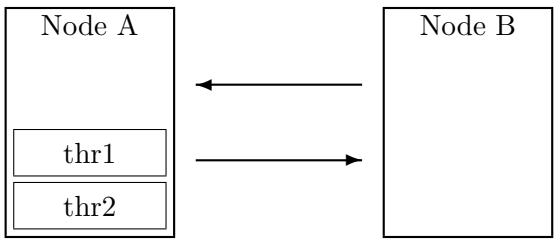

Figure 9.1: Depiction of the example scenario

Here, the threads running in Node A actively call the resource which will provide functionalities to ensure mutual exclusiveness of its usage.

9.2 The model

To model this system we could use the following Creol interfaces:

Creol

```
interface IResource
begin
  with IConsumer
    op lock()
    op use()
    op unlock()
end


interface IWorker
begin
  with Any
    op doWork1()
    op doWork2()
end
```

```
interface IConsumer
begin
  with IResource
    op holdsResource(in b:Bool)
end
```

Creol

The interface `IResource` will of course model the Node B; it offers methods to lock and release the resource itself, and the method `use` which models the actual utilization of it.

Node B will have just a partial view of the node calling its methods: the caller will implement the interface `IConsumer` and offer a method, `holdsResource`, to set its "status".

The last interface, `IWorker`, together with `IConsumer`, will be both implemented by the class modelling Node A. The implementation of this class can be found in the code listing on the next page.

As we can see, the class `NodeA` has a reference to an object of type `IResource` and the class methods, `doWork1` and `doWork2`, perform sequential calls to the same shared object.

This is a very basic implementation which covers some of the main aspects of distributed system modelling, being asynchronous calls and the handling of control switching between threads by explicit release points.

— Creol —

```
class NodeA implements IConsumer, IWorker
begin

  var r : IResource;
  var l : Label;
  var busy : Bool;

  with Any

    op doWork1()
      ==
      l!r.lock();l?();
      l!r.use();
      l!r.unlock();l?();
      release;
      l!r.lock();l?();
      l!r.use();
      l!r.unlock();l?()

    op doWork2()
      ==
      l!r.lock();l?();
      l!r.use();
      l!r.unlock();l?()

  with IResource

    op holdsResource(in b:Bool)
      ==
      busy := b

end
```

— Creol —

9.3 Adding specifications

It's now time to add some specifications to our model.

For instance, we don't have an implementation of the methods of the interface `IResource`; thus, we could add operation contracts to describe their behaviour:

— CML —

```
interface IResource
begin
  with IConsumer
    op lock()
      ensures \HContains(!caller.holdsResource(true))

    op use()
      requires \HContains(!caller.holdsResource(true))

    op unlock()
      requires \HContains(!caller.holdsResource(true))
      ensures \HContains(!caller.holdsResource(false))
end
```

— CML —

Note how the previous operation contracts are far away from modelling the complete behaviour of the three methods, they just assert some desired properties for the pre- and post-state of our methods.

For the `lock` method we don't require any special precondition, we just assert that after its termination the caller of the method (which we know being an instance of `IConsumer`) will hold the resource. This is modelled by stating that at this point the local history will contain an invocation of the method `holdsResource` with parameter **`true`**.

The same property will be a requirement for the method `use`, since we have to check that the object using the resource has set a lock on it to guarantee mutual exclusion.

Finally, for the `unlock` method, we require again the same condition and we add as a postcondition the release of the resource on the caller object.

We can add a further piece of specification in the class `NodeA`. For instance we can insert the following *global invariant*:

— CSL —

```
inv \after(!r.unlock(), !r.lock())
```

— CSL —

This states that whenever the local history contains an invocation of the method `lock` on the common resource `r`, then an invocation of `unlock` must follow.

Again, this is not a complete specification to ensure mutual exclusion of the usage of the shared resource, anyhow it is a desired property that we want to be preserved every time there is a switch between the concurrent threads of the Node A.

9.4 Verification

Now we have our model, and we have some properties to check our implementation against. We can then load it into KeY and start the verification process.

Assume we want to verify the method `doWork1`, so, when loading the program, we select it in the proof obligation browser and we start the proof.

The created proof obligation will be the following:

— KeY PO ———————————————————————————

```
==>
  Wf(HS, mAX_LABEL)
 -> \forall CreolHistory hPre; ( hPre = HS
  -> \forall CreolHistory hNew;
   {HS:=hNew || mAX_LABEL:=mAX_LABEL + 1}
    ( Pf(hPre, hNew)
      & after(invMsg(this, r, unlock, NP),
            invMsg(this, r, lock, NP),
            HS)
      & Invoc(HS,
            com(caller, this, doWork1, mAX_LABEL),
            NP)
     -> \[ body \]
       after(invMsg(this, r, unlock, NP),
           invMsg(this, r, lock, NP),
           HS) ))
```

——————————————————————————————— KeY PO —

This represents a simplified version of the actually displayed PO, some details have been omitted to improve the readability and set the focus on the main details.

In the above listing, *body* is a place-holder for the actual body of the method `doWork1`; the only precondition we have is the history being well-formed followed by an *anonymizing* update (see 6.1) of the history to an extended history containing the invocation message of `doWork1` on **`this`** object from **`caller`**. Additionally, the class global invariant is assumed to hold in the updated history and it is also added as postcondition to be preserved when the program terminates.

As we can notice, this properties are expressed with a different notation than the CSL syntax we saw in chapter 5; this is because the translation process triggered when loading the program converts the CSL specification to the KeY's program logic and also automatically adds parameters the user is not required to specify. The reader is advised to compare the here presented

PO with the one previously described in section 6.2. The predicate `Pf(`H_1`,` H_2`)` corresponds to what we previously denoted with $H_1 \leq H_2$; `Invoc(`H`,` lb`,` x`)` stands for $\langle invoc(lb, x)\rangle \subseteq H$ where lb is a label. In KeY's notation, we expressed a label with `com(`*caller*`,` *callee*`,` m`,` i`)` which contains all the necessary parameters to define a label as seen in our semantics (chapter 4), except for the thread id assigned to **`this`** which in the verification process of a single method would be constant and thus here omitted.

The `mAX_LABEL` is thus used to identify the thread corresponding to the called method and is constantly increased to ensure uniqueness. It is also used as parameter for other predicates (as `Wf(...)` in this example) for verification purposes.

Finally, we shift the attention on how our invariant has been translated from CSL: the corresponding predicate is `after(` msg_1`,` msg_2`,` H`)`. We can notice that the history has been added as parameter and the CSL invocation messages are translated to `invMsg(this, r,` m`, NP)` where NP is used to specify the absence of parameters and the parameter `this` has been automatically added since, as we saw in subsection 5.3.4, the local history we are dealing with is always centred on the current object.

Overall, the syntax used for the implementation in KeY is very similar to our specifications and logical predicates, some modifications have been necessary for the adaptation to the previously adopted code, but users can easily intuit the expressed properties.

Now the user is able to apply sequent rules to discharge the proof obligation. Here we are not going to present the full process for the closure of the whole proof, instead we will highlight some interesting steps.

The first rule we can apply when selecting the modality block in the PO is the **Method invocation** rule corresponding to the invoc rule we saw in section 7.2. The result will be a split of the proof in two branches: one corresponding to the case of having a **`null`** reference, and the other corresponding to a correct invocation. In the latter, the modality block of the initial PO will be replaced by the following formula:

— KeY PO —

```
!r = null
-> {l:=com(this, r, lock, mAX_LABEL + 1)
     || mAX_LABEL:=mAX_LABEL + 1}
 \forall CreolHistory hPreI0; ( hPreI0 = HS
-> \forall CreolHistory hNewI0;
 {HS:=hNewI0}
  ( Pf(hPreI0, hNewI0) & Invoc(hNewI0, l, NP)
    -> \[ stmts \] post )...)
```

— KeY PO —

where *stmts* stands for all the statements following the method invocation `l!r.lock();` in the previous *body*, and *post* contains the formulas following the modality block.

As we can see, here the label `l` is updated to hold a reference to the invoked method, the thread index is increased and the history is extended with the appropriate invocation message.

Since the called method has no precondition specified and the `IResource` interface has no invariants, thus set to **`true`** by default, no further property is required to be proved.

Next, we can continue applying the rule **Method completion** again on the modality block. Again, two branches will be created: the one blocking the execution in case of **`null`** label, and the one corresponding to a valid completion request in which the modality block will be replaced by:

— KeY PO ————————————————————

```
!l = null
-> \forall CreolHistory hPreC0; ( HS = hPreC0
 -> \forall CreolHistory hNewC0;
  {HS:=hNewC0}
   ( Pf(hPreC0, hNewC0)
      & Comp(hNewC0, l, NP)
      & HContains(invMsg(r, this,
           holdsResource, TRUE),HS)
    -> \[ stmts1 \] post )...)
```

———————————————————— KeY PO —

Here, the completion statement `l?();` is removed from the modality block, now containing the remaining code (*stmts*1).

Note how the postcondition of the method previously assigned to label `l`, i.e. `lock()`, is here added as assumption and translated as:

`HContains(invMsg(r, this, holdsResource, TRUE),HS).`

Again, the parameter **`this`** is automatically added as well as the history and the **`caller`** is correctly resolved to the object `r`.

Similarly, we proceed on the following statements until we reach the **`release`** point. Here we can apply the **Release** rule which splits the proof into two additional branches: the first one where we have to show that our class invariant is preserved and the second where we model the case where the execution is resumed.

In the second one, it is interesting to notice how all the global references we have are *anonymized*, i.e. updated to new, unknown values:

— KeY PO —

```
\forall CreolHistory hPreR1;
 ( hPreR1 = HS
  -> \forall CreolHistory hNewR1;
   \forall CreolIResource new_r1;
    \forall CreolLabel new_l1;
     \forall CreolBool new_busy1;
      {busy:=new_busy1 || l:=new_l1 ||
       r:=new_r1 || HS:=hNewR1}
      ( Pf(hPreR1, hNewR1)
         & after(invMsg(this, r, unlock, NP),
                 invMsg(this, r, lock, NP), HS)
        -> \[ stmts2 \] post )...)
```

— KeY PO —

This is because the threads running when the processor was released, could have modified the class attributes.

Following the same procedure, we can step by step remove all the statements contained in the modality block and successively close all the open goals.

This scenario has been completely tested and the code is available in the *key/examples/lang/creol/csl* folder from the KeY source.

Chapter 10

Limitations and future work

To provide a reliable and complete environment for the verification of Creol models with a more expressive specification language, many steps have to be taken. This chapter contains a discussion on some aspects this work is lacking and provides for some suggestions for future development and improvement.

CSL and histories

So far, as we saw in chapter 5, CSL provides for a limited expressiveness which in some situations can be restrictive for complete specifications of complex models. Thus, an extension of the set of CSL predicates and functions can largely contribute to an improvement of this work. The implemented system allows for easy expansion of the syntax and easy integration with KeY: introducing new predicates or functions would require just a minimal modification of the grammar and a few additional lines of code.

Further, the verification process allows for the handling of a subset of the CSL expressions. The main restrictions are the inability of using concatenation of messages as parameter for CSL predicates and functions (only single messages are handled), and the limitation to the number of method parameters to one in-parameter and one out-parameter. These weaknesses are due to the adaptation to the previously implemented system. For instance, as said before, the system doesn't use any data structure for the representation of the history which makes dealing with concatenation of messages a difficult - sometimes impossible - task. The adoption of a data structure for representing the history would probably lead to a much easier definition of specifications dealing with object communication and thus to a faster extension of CSL.

Program consistency

The specified grammar both for Creol and CSL allows for a superset of the actually allowed syntax. Checks are performed when loading the program, but probably not all aspects for a complete syntactical correctness are covered. This problem can be easily overcome by adding further checks on the AST, for example by adding a walker and an FSM with a suitable layout.

Moreover, the parsing procedure doesn't provide type checking, hence errors of this type are only captured in a later stage when loading the code in KeY. Again, it's easy to add this feature to the parser by storing variables' types in a hash map and recall them when needed.

Calculus

The calculus introduced in this work doesn't cover all the possible statements allowed in the Creol language. For instance, Creol supports parallel assignments like "`a,b := b,a`" which are not taken into account in this work. Refinements have also to be made on the handling of non-deterministic composition of statements. As said before, Creol is still in a experimental phase and several dialects exists. Thus, no final assertion can be made on the completeness of the calculus with respect to Creol syntax. Anyhow all the major aspects can be considered covered.

Data Types

Data structures as sets, strings, or tuples are not currently handled. This is because only finite mathematical sets are supported by KeY, the handling of strings is just a recent achievement, and tuples lead to inconsistencies as they can hold different sorts for each entry.

Additionally, floating point representation is also an ongoing work and therefore not addressed in this work.

Proof automation

The available proof strategy covers a limited spectrum of the reasoning process needed to achieve complete automation. Additional characteristic proofs of Creol programs must be undertaken and improvements on the heuristics guiding the strategy must be made.

Readability of proofs

When a proof obligation is generated in KeY, it can be hardly readable. Predicates and functions may have long names and it can be the case that not all of the introduced ones during the verification process are used for the closure of the proof. Pretty-printing can be implemented to improve readability, some predicates can be expressed with infix notation: for instance

the symbol $\subseteq$ could be used to express the presence of a message in a history. Moreover, an analysis can be performed to hide non-relevant information so to improve the user experience while carrying on the verification. For example, in many cases, not all the parts of the frequently involved composed invariant are used to close a proof, an automated process could thus hide them to efficiently enhance readability.

Chapter 11

Conclusions

In this thesis significant aspects of software verification have been raised and discussed. Distributed systems, which are the main target of this work, are an essential domain of research which affects contemporary society from several points of view. The growing demand for communication, cooperation, and automation, brings distributed systems at the centre of our focus.

Reliability of such systems is thus a fundamental issue which leads to the necessity of providing for instruments that guarantee the development of error-free applications. A minimal failure in critical (possibly safety-related) systems may result in inestimable loss.

The study of mathematics and logic has been undertaken for as far back as written records exist, the knowledge on this fields is wider than it is in any other domain. The employment of formal methods for the development and verification of computer systems can be seen, from my point of view, as the reduction of these tasks to the solution of problems concerning a field we have complete knowledge about.

Functional models represent a powerful tool for highlighting the critical aspects during the development of dependable systems. Further, the usage of a formal specification language to express desired properties ensures for an unambiguous description and for the possibility of a rigorous proof of correctness.

Criticism has often been directed to the usage of formal methodologies. The main discussions concern the difficulty of writing formal specifications and the amount of resources needed for formal verification. Writing complete specifications can, indeed, be a difficult task which may require deep understanding of the specification language and of the underlying logic. On the other hand, as learned from personal experience, sometimes formal languages make it much easier to express particular properties (possibly representing common patterns) which may be hard to express otherwise. Moreover, a formal verification procedure for relevant code snippets, even if provided with a minimal set of specifications, can easily show functional errors

which in many cases can be missed by the employment of test cases or other methodologies not exploring all the possible program states.

The Creol language, as we saw in chapter 2, features many characteristics which make it a promising choice for the modelling of concurrent distributed systems. The calculus and denotational semantics elaborated in [3, 2] opened wide perspectives for the verification of Creol models and set a solid basis for this work. The KeY prover is among the state-of-the-art systems for the verification of object-oriented programs. It is in constant development and can lead to a powerful and user-friendly tool for the application of formal methods.

This thesis represents a further step in the creation of a formal verification environment for Creol models. The main contributions have been the refinement and extension of the Creol logic and the design of a specification language, CSL, which combines with the underlying logic emphasizing significant aspects; for instance, the idea of projecting the communication history on the relevant actors has been completed and integrated in the newborn concepts of *global invariant* and *co-invariant*.

As a result CSL provides for a user-friendly syntax which abstracts from logical details and allows, among other things, for the specification of communication traces between concurrent objects and threads. Further, the automation of the verification procedure has been increased and the required input from the user has been highly decreased.

Moreover an easily extendible platform has been built, which simplifies the process of future development of the system.

Several aspects have still to be refined to consider this a complete work. Further research has to be undertaken to prove the soundness and completeness of the Creol calculus, CSL can be widely extended and improved, and the limitations of the prototypical system have to be overcome. Anyhow, the project shows great potential for applicability and research interest, most importantly it contributes in showing the feasibility of the application of formal methods in real scenarios.

Appendix A

CSL Grammar

```
// operation contract ====================

requires
      : REQUIRES cmlExpr
      ;

ensures
      : ENSURES cmlExpr
      ;

diverges
      : DIVERGES ( TRUE | FALSE )
      ;

assignable
      : ASSIGNABLE ( NOTHING | EVERYTHING
            | IDENTIFIER (KOMMA IDENTIFIER)* )
      ;

// invariant ==============================

invariant
      : INV cmlExpr
      ;

// CML expressions ========================

cmlExpr
      : cmlImplicationExpr
            ( EQUIVALENCE cmlImplicationExpr )*
```

```
      ;

cmlImplicationExpr
      : cmlOrExpr ( IMPLICATION cmlOrExpr )*
      ;

cmlOrExpr
      : cmlAndExpr ( OR cmlAndExpr )*
      ;

cmlAndExpr
      : cmlNotExpr ( AND cmlNotExpr )*
      ;

cmlNotExpr
      : ( NOT )? cmlAtomicExpr
      ;

cmlAtomicExpr
      : cmlCompExpr
      | cmlPredicate
      ;

cmlCompExpr
      : cmlAlgExpr ( ( EQUALITY
            | INEQUALITY | comp_op ) cmlAlgExpr)?
      ;

cmlAlgExpr
      : cmlMultExpr ( ( PLUS | MINUS ) cmlMultExpr )*
      ;

cmlMultExpr
      : cmlPowerExpr ( ( MULT | DIV
            | MOD ) cmlPowerExpr )*
      ;

cmlPowerExpr
      : cmlFactor ( POW cmlFactor )?
      ;

cmlFactor
      : MINUS? cmlAtom
      ;
```

```
cmlAtom
      : IDENTIFIER
      | INTEGER
      | FLOAT
      | NULL
      | THIS
      | CALLER
      | TRUE
      | FALSE
      | cmlFunction
      | LPAREN cmlParenExpr RPAREN
      ;

cmlParenExpr
      : cmlQuantifiedExpr
      | cmlNumericalQuantifiedExpr
      | cmlExpr
      ;

cmlQuantifiedExpr
      : cmlQuantifier var_decl_no_init_list
            SEMICOLON cmlExpr (SEMICOLON cmlExpr)?
      ;

cmlQuantifier
      : FORALL | EXISTS
      ;

cmlNumericalQuantifiedExpr
      : cmlNumericalQuantifier var_decl_no_init_list
            SEMICOLON cmlExpr SEMICOLON cmlAlgExpr
      ;

cmlNumericalQuantifier
      : SUM | PRODUCT | MIN | MAX
      ;

cmlMsg
      : ( EXCLENATION_MARK | QUESTION_MARK ) cmlMethod
      ;

cmlHist
      : CML_EMPTY_HIST
```

```
      | cmlMsg (CMLCONCAT cmlMsg)*
      ;

cmlMethod
      : cmlObj DOT cmlMethodId cmlArgList
      ;

cmlObj
      : THIS
      | CALLER
      | IDENTIFIER
      ;

cmlMethodId
      : IDENTIFIER
      ;

cmlArgList
      : LPAREN cmlExprList? RPAREN
      ;

cmlExprList
      : cmlExpr ( KOMMA cmlExpr )*
      ;

cmlFunction
      : COUNT_FUNC LPAREN cmlHist RPAREN
      ;

cmlPredicate
      : HEMPTY
      | FIRSTCALL
      | HCONTAINS LPAREN cmlHist RPAREN
      | HENDSON LPAREN cmlHist RPAREN
      | AFTER LPAREN cmlHist KOMMA cmlHist RPAREN
      | BEFORE LPAREN cmlHist KOMMA cmlHist RPAREN
      ;

cmlOp
      : HEMPTY
      | FIRSTCALL
      | HCONTAINS
      | HENDSON
      | COUNT_FUNC
```

```
      | AFTER
      | BEFORE
      | FORALL
      | EXISTS
      ;

// Predicates ===================

HEMPTY
      : '\\HEmpty'
      ;

FIRSTCALL
      : '\\firstCall'
      ;

HCONTAINS
      : '\\HContains'
      ;

HENDSON
      : '\\HEndsOn'
      ;

AFTER
      : '\\after'
      ;

BEFORE
      : '\\before'
      ;

// Functions ====================

COUNT_FUNC
      : '\\count'
      ;

// Other key-words ==============

CML_EMPTY_HIST
      : '<>'
      ;
```

```
DIVERGES
      : 'diverges'
      ;

ASSIGNABLE
      : 'assignable'
      ;

NOTHING
      : '\\nothing'
      ;

EVERYTHING
      : '\\everything'
      ;

CMLCONCAT
      : '^'
      ;
```

Bibliography

[1] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software. The KeY Approach.* LNCS 4334. Springer-Verlag, 2007.

[2] Wolfgang Ahrendt and Maximilian Dylla. *A System for Compositional Verification of Asynchronous Objects.* Science of Computer Programming. DOI: `http://dx.doi.org/10.1016/j.scico.2010.08.003`. Elsevier, 2010.

[3] Maximilian Dylla. *A Verification System for the Distributed Object-Oriented Language Creol.* Master Thesis, Chalmers University of Technology, Gothenburg, Sweden, June 2009.

[4] I. C. Yu, E. B. Johnsen, and O. Owe. *Type-safe runtime class upgrades in Creol.* In R. Gorrieri and H. 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.

[5] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. *Incremental reasoning for multiple inheritance.* Technical Report 373, Department of Informatics, University of Oslo, 2008.

[6] J. Dovland, E. B. Johnsen, and O. 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.

[7] I. D. Hill. *Wouldn't it be nice if we could write computer programs in ordinary English - or would it? The Computer Bulletin*, June 1972.

[8] C. A. R. Hoare. *An axiomatic basis for computer programming.* Communication of the ACM, vol.12, 1969.

[9] E. W. Dijkstra. *Guarded commands, nondeterminacy and formal derivation of programs.* Communication of the ACM, vol.18, 1975.

[10] E. W. Dijkstra. *First-Order Logic and Automated Theorem Proving (2nd ed.)*. Springer-Verlag New York, Inc., USA, 1996.

[11] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, May 2007.

[12] W. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, November 2001.

[13] J. Hooman, W. de Roever, P. Pandya, Q. Xu, P. Zhou, and H. Schepers. *A Compositional Approach To Concurrency And Its Applications*. unfinished manuscript. Available online at `http://www.informatik.uni-kiel.de/inf/deRoever/books/`, April 2003.

[14] Job Zwiers. *Compositionality, Concurrency and Partial Correctness*. volume 321 of LNCS. Springer-Verlag, 1989.

[15] Colin J. Fidge. *Timestamps in Message-Passing Systems That Preserve the Partial Ordering*. Australian Computer Science Communications, 10:55-66. 1988.

[16] B. Beckert and V. Klebanov. *A dynamic logic for deductive verification of concurrent programs*. In M. Hinchey and T. Margaria, editors, Proceedings, 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM), London, UK. IEEE Press, 2007.

[17] C. Engel and R. Hähnle. *Generating unit tests from formal proofs*. In Y. Gurevich and B. Meyer, editors, Proceedings, 1st International Conference on Tests And Proofs (TAP), Zurich, Switzerland, volume 4454 of LNCS. Springer, 2007.

[18] Elmar Habermalz. *Interactive theorem proving with schematic theory specific rules*. Technical Report 19/00, Fakultät für Informatik, Universität Karlsruhe, 2000.

[19] A. Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. 1st Edition. Springer. September 2010)