UNIVERSITY OF OSLO
Department of Informatics

Verification of Assertions in Creol Programs

Master's Thesis
(60 credits)

Jasmin Christian Blanchette

May 2, 2008

Abstract

Open distributed systems are composed of geographically dispersed components that may be modified at run-time. Such systems are becoming increasingly important, particularly when they are part of the infrastructure used by safety-critical applications. Creol is an experimental object-oriented programming language designed for modeling such systems. Creol objects execute concurrently, each with its own virtual processor and internal process control, and communication takes place using asynchronous (non-blocking) method calls.

To increase the reliability of the systems in which they operate, Creol classes make explicit assumptions about their environment and provide guarantees about their own behavior. The assume–guarantee paradigm enables scalable, compositional verification based on invariance. The goal of this thesis is to implement a tool that verifies assume–guarantee specifications and other program assertions using Maude, an established rewriting logic framework. The tool takes a set of Creol classes and interfaces as input, and uses Hoare logic to generate verification conditions that can be discharged using off-the-shelf theorem provers.

Preface

This master's thesis describes the results of work undertaken in the Precise Modeling and Analysis (PMA) group within the Department of Informatics at the University of Oslo, where my task was to implement the proof system for the experimental programming language Creol using the Maude language.

I would first like to thank Olaf Owe, my main supervisor. Although his time is heavily committed, especially as head of the research group, he always made time for me so that we could discuss my thesis and the Creol language. I greatly benefited from his wide experience and extensive knowledge, and his many reviews of the thesis's chapters lead to hundreds of improvements to the contents and to the exposition.

I had the pleasure to work closely with Marcel Kyas and Johan Dovland, who acted as additional supervisors. Marcel's attention to detail was greatly appreciated, and from Johan I owe much of my understanding of Hoare logic. Also within the PMA group, I received expert advice from Christian Mahesh Hansen, Peter Csaba Ölveczky, and Martin Steffen regarding predicate logic, rewriting logic, and formal semantics.

I also want to thank Willem-Paul de Roever. His views on the relationship between syntax and semantics as well as his insistence on completeness and compositionality deeply influenced my work.

To my delight, my friends Vivi Glückstad Karlsen, Trenton Schulz, and Mark Summerfield agreed to read through the manuscript (twice in Mark's case). Besides improving the quality of the English, their comments greatly contributed to making the thesis accessible to computer scientists with little or no previous experience of formal methods.

I gratefully thank my girlfriend, Anja Palatzke, for her unfailing encouragement and support in this project, as well as Matthias Ettrich and Lars Knoll, my bosses at Trolltech, who encouraged me to pursue this degree and allowed me to continue working part-time.

The contributions of thousands of free and open source software developers made it possible to prepare the manuscript of this thesis and develop the Maude code almost exclusively using open source programs. I want to thank in particular the creators and maintainers of Kubuntu, Debian GNU/Linux, KDE, (LA)TEX, Xfig, KPDF, Xpdf, Ispell, NEdit, Vim, and Maude for making their wonderful programs available to anyone who cares to use them.

Contents

> Normal methods of calculation and proof seem wholly impractical to conduct by hand; and fifteen years of experience suggest that computer assistance can only make matters worse.
>
> — C. A. R. Hoare (1985)

Chapter 1

Introduction

CREOL [PMA08] is an ongoing research project (2004–2008) by the Precise Modeling and Analysis (PMA) group at the University of Oslo. The project aims to develop a formal framework for reasoning about dynamic and reflective modifications in *open distributed systems*, which consist of geographically dispersed components that may be modified at run-time. These systems are becoming increasingly important in society, particularly when they form part of the infrastructure for safety-critical applications.

The cornerstone of the CREOL project is the Creol programming language. Creol is strongly-typed and object-oriented. Like Simula and Java, it supports classes, interfaces, inheritance, and polymorphism. In addition, it provides two intuitive high-level programming constructs for concurrency: *asynchronous* (*non-blocking*) *method calls* and *explicit processor release points*. The reference operational semantics of Creol is expressed in Maude [CDEL+07], a programming and specification language based on rewriting logic. Since Maude specifications are executable, we can use Maude to run Creol programs. This thesis is concerned with the formal verification of Creol components.

1.1 Motivation

One of Creol's main design goals is to ensure the reliability and correctness of open distributed systems. Because of their open nature, such systems generally cannot be verified monolithically; instead, we normally verify each class individually. If the instances of the class make assumptions about the other objects in the system, these must be made explicit; in return, the class can provide explicit guarantees to the other objects in the system. These *assume–guarantee specifications*, as well as other assertions at specific points in the class's implementation, are supplied by Creol programmers directly in their source code. When combining instances of various classes together to build a complex system, we must verify that each class's assumptions are supported by the other classes' guarantees.

Assume–guarantee specifications can be seen as documentation to the users of a class, but since they are expressed in first-order predicate logic they can also be

read by various code validation tools. One such tool is an add-on to the Creol interpreter that monitors the system and aborts it when a class's implementation breaks its specification [AJO04, Axe04]. Another tool performs unit testing on a single class taken in isolation, using a pseudorandom number generator to mimic an arbitrary environment [JOT06]. However, both tools are testing tools and are therefore incomplete; some bugs might go undetected.

Formal verification is more powerful but also much more demanding.[1] Dovland, Johnsen, and Owe [DJO05] developed a simple and compositional proof system for Creol, based on Hoare logic. However, without tool support, the verification conditions identified by their system must be computed and proved by hand, a tedious and error-prone process that defeats the very purpose of verification. It is thus highly desirable to develop a tool that implements the Creol proof system. In addition, any implementation phase is likely to lead to the discovery and correction of flaws in the original design, and to expose the strengths and weaknesses of Creol's reference operational semantics in the context of program verification.

Dovland and his colleagues developed two versions of their proof system for Creol. The first version [DJO05] covers a subset of an earlier Creol dialect and is therefore obsolete. The second version [DJO08] is more up to date but uses a fairly restrictive subset of Creol. To be of greatest utility, the implementation of the proof system should support the same syntax and semantics as the interpreter.

Within the PMA group, there is a long tradition of using term-rewriting systems for the specification and implementation of programs [DO91, Dah92]. In later years, this culminated with the use of Maude for implementing the Creol interpreter [JOA03] and an automated theorem prover [Hol05], for modeling real-time systems [ÖM04], and as the foundation of the introductory formal methods course [Ölv07]. Maude has been appreciated for the conciseness, clarity, and expressiveness of its syntax, as well as for the quality of its implementation. The experience with these various projects suggests that Maude could prove a suitable language for implementing the proof system, and adopting Maude would enable code sharing with other Creol tools.

1.2 Problem Statement

The goal of this thesis to implement a tool, the *assertion analyzer*, that assists the verification of a class's assume–guarantee specification and of any assertions or invariants that appear in its implementation. The tool takes a set of Creol classes and interfaces that define a distributed component as input, and attempts to verify that the component's implementation respects the specified guarantee using Hoare logic. More specifically, it checks that the guarantee holds after initialization of an object, is maintained by all methods, and holds before all processor releases, as long as the assumption holds.

[1] Testing is sometimes considered a "lightweight verification" method, but in this thesis we will reserve the term "verification" for the process of proving the correctness of an implementation with respect to a specification through mathematical means.

The tool's output is a report listing verification conditions that must be proved by hand or using a mechanical theorem prover such as Isabelle/HOL [NPW02] or PVS [CORSS95]. To make its output more readable, the assertion analyzer performs syntactic simplifications on the verification conditions before it displays them.

The thesis tries to answer the following questions:

1. *How can we adapt the existing proof system to fully account for the more challenging aspects of Creol's formal semantics, such as object reentry and the nondeterministic statements?*
2. *How suited are Maude and rewriting logic to implementing Hoare logic?*
3. *To what extent do Creol's reference operational semantics and proof system enable program verification in practice?*

We will come back to these questions in Section 8.1 of the conclusion.

1.3 Related Work

The verification of computer programs was first attempted in the 1940s by Goldstine, von Neumann, and Turing [GvN46, Tur49], but it took another twenty years before this activity was formalized by Floyd [Flo67] (for flowchart programs) and Hoare [Hoa69] (for structured imperative programs). At about that time, James C. King [Kin69] wrote what might be the first mechanized verification condition generator, and since then such tools have been used for a variety of programming languages and contexts, including for interference-freedom checks in shared-memory parallel programs [NPN99] and for Java bytecode verification [BP06].

While most verification condition generators are run as independent programs [Shu01], some are directly embedded in a theorem prover for higher-order logic [Gor89, NPN99]. A promising approach is suggested by the KeY project's prover, which interleaves first-order logic reasoning with symbolic execution of programs, arithmetic simplification, external decision procedures, and symbolic state simplification [BHS07]. In Maude, Hoare-style program verification is embodied by the experimental tool Java+ITP [SM07], which discharges verification conditions using Maude's Inductive Theorem Prover (ITP).

Automatic program verification is still the subject of much research. Inspired by recent advances, Hoare revived the creation of a fully automated "verifying compiler" as a grand challenge for computer science [Hoa03]:

> A verifying compiler uses mathematical and logical reasoning to check the correctness of the programs that it compiles. The criterion of correctness is specified by types, assertions, and other redundant annotations associated with the code of the program. The compiler will work in combination with other program development and testing tools, to achieve any desired degree of confidence in the structural soundness of the system and the total correctness of its more critical components.

The theoretical foundation of concurrent program verification is summarized in Apt and Olderog [AO97] and de Roever et al. [dRdB$^+$01]. The Hoare logic for Creol

[DJO05, DJO08] was developed following the approach advocated by de Boer and Pierik [dBP04]. The proof system is compositional due to the use of communication histories, which were introduced for program verification by Dahl [Dah77].

1.4 Structure of this Thesis

Although this thesis is concerned with program verification, it should be accessible to any computer scientist comfortable with symbolic mathematics. Topics such as operational semantics, Hoare logic, rewriting logic, and Maude are explained when they are first encountered. On the other hand, familiarity with object-oriented programming concepts and terminology is a prerequisite; an excellent introduction can be found in Abadi and Cardelli [AC96].

The rest of this thesis is structured as follows:

- Chapter 2 explains formal systems and first-order predicate logic, which are used throughout the thesis.
- Chapter 3 provides an introduction to rewriting logic and to Maude.
- Chapter 4 presents the syntax and formal semantics of Creol.
- Chapter 5 presents the Hoare-style proof system for Creol that serves as the basis for the assertion analyzer.
- Chapter 6 reviews the assertion analyzer's design and implementation.
- Chapter 7 presents four small case studies of class verification performed using the assertion analyzer.
- Chapter 8 summarizes the results and gives directions for future work.

Readers who are interested in trying out the assertion analyzer for themselves will probably find the appendices useful:

- Appendix A provides a user's guide to the assertion analyzer and the companion tools developed in the thesis.
- Appendix B provides the Maude specifications for the assertion analyzer and the companion tools.
- Appendix C provides the Maude specifications for the case studies presented in Chapter 7.

1.5 Summary of Contributions

The main contribution of this thesis is to develop the assertion analyzer in Maude and use it on concrete examples, allowing us to answer the questions posed in Section 1.2. In particular, we find that Maude's flexible parser makes it possible to parse the Creol language's syntax without needing a parser generator (an improvement over previous Maude-based Creol tools, which work in terms of "Creol

Machine Code" [Arn03, Fje05]) and that rewrite rules are a simple yet powerful way to normalize and simplify assertions.

A second contribution is to strengthen the proof system and to incorporate assume–guarantee reasoning, focusing on soundness and completeness with respect to the language's operational semantics. The main differences between the proof system we present here and the proof system developed by Dovland, Johnsen, and Owe are listed in Section 5.7; they include the axiomatization of object reentry and the nondeterministic statements, as well as the incorporation of assume–guarantee-based reasoning (instead of a single invariant).

To enhance the presentation, we introduce an intermediate semantics to bridge the gap between Creol's traditional operational semantics and the proof system. The resulting "open systems" operational semantics is described in Section 4.4 and in a separate paper [BO08]. The approach was inspired by de Roever et al. [dRdB$^+$01] and might constitute the main distinguishing methodological feature of this thesis.

The thesis provides implementations of two versions of the Creol interpreter: one for closed systems and one for open systems. These tools let us test a program before we subject it to formal verification using the assertion analyzer. Unlike existing Creol interpreters, they expect the same concrete syntax as the assertion analyzer, thus contributing to the establishment of a unified Creol framework. The tools are described in Section 4.5.

Only a masochist would state $A \vee B$ when he knows A.
— Jean-Yves Girard (1995)

Chapter 2

Preliminaries

This chapter explains two basic concepts used throughout this thesis: formal systems and first-order predicate logic. These topics are usually covered as part of a computer science undergraduate curriculum, so the chapter provides only a brief overview, focusing on the terminology and notations used in this thesis. The presentation is based on Dahl [Dah92], Apt and Olderog [AO97], Gallier [Gal03], and Hansen [Han07]. Readers who feel comfortable with these topics are encouraged to skip this chapter.

2.1 Formal Systems

Formal systems are known by many names—deduction systems, derivation systems, inference systems, formal proof systems, logics, calculi—and they play a central role in this thesis. First-order logic (Section 2.2), rewriting logic (Section 3.4), and Hoare logic (Sections 5.1–5.3) are all examples of formal systems, and so are the Maude specifications of the tennis scoring rules (Section 3.3) and of the WHILE interpreter (Section 3.6).

A *formal system* is a structure $\mathcal{S} = (\Sigma, \Phi, A, \mathcal{R})$, where Σ is a set of symbols (the *alphabet*), Φ is a set of strings over Σ (the *formulas*), A is a subset of Φ (the *axioms*), and $\mathcal{R}$ is a set of relations on Φ (the *derivation rules* or *proof rules*). The *theory* associated with $\mathcal{S}$ consists of the axioms in A and all the formulas (*theorems*) that can be derived from the axioms using one or more derivation rules.

To illustrate this, we will specify a formal system $\mathcal{J} = (\Sigma_{\mathcal{J}}, \Phi_{\mathcal{J}}, \{\text{J1}, \text{J2}\}, \{\text{J3–J7}\})$ that infers the type of Java expressions. The alphabet $\Sigma_{\mathcal{J}}$ consists of the Unicode characters that may be used in the source text of a Java program; the formulas in $\Phi_{\mathcal{J}}$ have the form $e : \tau$ (read "e is of type τ"), where e is a syntactically valid Java expression and τ is a legal Java type; and the axioms and derivation rules follow.

Axiom J1. n : `int` for any integer literal n such that $-2^{31} \leq n < 2^{31}$

Axiom J2. $x : \tau$ if x is a variable declared with type τ

Derivation Rule J3. $$\frac{e : \tau}{(e) : \tau}$$

Derivation Rule J4. $\dfrac{e_1 \texttt{ : } \tau \quad e_2 \texttt{ : } \tau}{e_1 \texttt{ + } e_2 \texttt{ : } \tau}$ if $\tau \in \{\texttt{byte}, \texttt{short}, \texttt{int}, \texttt{long}\}$

Derivation Rule J5. $\dfrac{e_1 \texttt{ : } \tau_1 \quad e_2 \texttt{ : } \tau_2}{e_1 \texttt{ == } e_2 \texttt{ : boolean}}$ if $\tau_1, \tau_2 \in \{\texttt{byte}, \texttt{short}, \texttt{int}, \texttt{long}\}$

Derivation Rule J6. $\dfrac{e_1 \texttt{ : boolean} \quad e_2 \texttt{ : } \tau \quad e_3 \texttt{ : } \tau}{e_1 \texttt{ ? } e_2 \texttt{ : } e_3 \texttt{ : } \tau}$

Derivation Rule J7. $\dfrac{e_1 \texttt{ : } \tau_1\texttt{[]} \quad e_2 \texttt{ : } \tau_2}{e_1\texttt{[}e_2\texttt{]} \texttt{ : } \tau_1}$ if $\tau_2 \in \{\texttt{byte}, \texttt{short}, \texttt{int}, \texttt{long}\}$

Axioms J1 and J2 are axiom schemas, whose metavariables (n, x, and τ) can be instantiated to produce specific axioms. For example, J1 and J2 let us derive axioms such as `127 : int` and `a : String[]`, assuming the program of interest declares a variable called `a` of type `String[]`.

Derivation Rules J3–J7 allow us to generate new theorems from existing theorems or axioms. When we instantiate a rule, the formulas above the horizontal bar (the *premises*) must be axioms or theorems themselves; the formula below the bar (the *conclusion*) is then a theorem. Thus, the horizontal bar itself can be seen as an implication. Axiom schemas and derivation rules may have *side conditions* limiting their applicability, displayed next to them.

Using Derivation Rule J7, we can produce the theorem `a[127] : String` from the axioms `a : String[]` and `127 : int`. This can be written as a derivation tree:

$$\dfrac{\dfrac{}{\texttt{a : String[]}}\,\text{J2} \quad \dfrac{}{\texttt{127 : int}}\,\text{J1}}{\texttt{a[127] : String}}\,\text{J7}$$

Derivation trees (or *proof trees*) are drawn with their root at the bottom, like natural trees. The root of the tree is the derived theorem, and each branch ends with an axiom—or with a theorem proved separately, called a *lemma*. Invocations of an axiom or a derivation rule are indicated by a horizontal bar and a label. The derivation tree below for the Java conditional expression `(x == y) ? s : a[127]` involves nearly all the axiom schemas and derivation rules in $\mathcal{J}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{x : int}}\,\text{J2} \quad \dfrac{}{\texttt{y : int}}\,\text{J2}}{\texttt{x == y : boolean}}\,\text{J5}}{\texttt{(x == y) : boolean}}\,\text{J3} \quad \dfrac{}{\texttt{s : String}}\,\text{J2} \quad \dfrac{\dfrac{}{\texttt{a : String[]}}\,\text{J2} \quad \dfrac{}{\texttt{127 : int}}\,\text{J1}}{\texttt{a[127] : String}}\,\text{J7}}{\texttt{(x == y) ? s : a[127] : String}}\,\text{J6}$$

The formal system $\mathcal{J}$ attempts to formalize the typing rules found in the Java language's official specification [GJSB05]. In practice, it is crucial that the theorems of $\mathcal{J}$ are *valid* when interpreted in terms of that specification; in other words, for all theorems $e \texttt{ : } \tau$ of $\mathcal{J}$, the expression e should be of type τ according to the specification. If this is the case, $\mathcal{J}$ is *sound*. Conversely, $\mathcal{J}$ is *complete* if the formula $e \texttt{ : } \tau$ is a theorem whenever e has type τ according to the specification.

With respect to the Java specification, the system $\mathcal{J}$ is sound but not complete. To prove soundness, we would show that the axioms are all valid and that the derivation rules preserve validity downward, meaning that valid premises always lead to valid conclusions. To prove incompleteness, it suffices to consider the counterexample `true : boolean`, which is valid but not a theorem in $\mathcal{J}$.

2.2 First-Order Predicate Logic

First-order predicate logic is a family of languages that let us express assertions over elements of sets. The Hoare-style proof system presented in Chapter 5 and implemented in Chapter 6 builds on first-order logic. Here, we will start by describing the syntax and semantics of first-order logic languages; then we will present the sequent calculus LK ("Logischer Kalkül"), a proof system due to Gentzen [Gen35].

A *first-order language* $\mathcal{L}$ consists of logical symbols and of non-logical symbols. The logical symbols are $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (implies), $\neg$ (not), $\forall$ (for all), $\exists$ (for some), *variables* from a set V, parentheses, commas, and periods. The non-logical symbols are specified by a signature $\Sigma = (C, F, \mathcal{R})$, where C is a set of *constant symbols*, F is a set of *function symbols* with arities (parameter counts), and $\mathcal{R}$ is a set of *relational symbols* with arities. In the context of program verification, the constants typically express values of programming language data types (such as **true**, **false**, 0, and -123), and the function and relational symbols correspond to operators and functions on these data types (such as $\leq$, $+$, and *sqrt*).

The language $\mathcal{L}$ lets us express *assertions* (or *formulas*) that involve *terms*. The set $\mathcal{T}$ of terms associated with Σ is defined inductively to contain all variables and constants from the sets V and C, as well as all elements of the form

$f(t_1, \ldots, t_n)$	function term

where f is an n-ary function symbol from F, $n \geq 1$, and $t_1, \ldots, t_n$ belong to the term language $\mathcal{T}$. The set $\mathcal{A}$ of assertions associated with $\mathcal{L}$ consists of all elements of the forms

$R(t_1, \ldots, t_n)$	relational expression
$\neg\varphi$	logical negation
$\varphi \wedge \psi$	logical conjunction
$\varphi \vee \psi$	logical disjunction
$\varphi \Rightarrow \psi$	implication
$\forall x.\, \varphi$	universal quantification
$\exists x.\, \varphi$	existential quantification
(φ)	parenthesized assertion

where R is a relation symbol, x is a variable, $t_1, \ldots, t_n$ are terms, and φ and ψ are assertions. The operator precedence is as follows: $\neg$ binds more strongly than $\wedge$, which binds more strongly than $\vee$, which binds more strongly than $\Rightarrow$, which binds more strongly than $\forall$ and $\exists$; furthermore, the $\wedge$ and $\vee$ operators are associative and commutative, and $\Rightarrow$ associates to the right. Thus, $a \wedge \forall x.\, \neg b \wedge c \Rightarrow d \Rightarrow e$ is considered the same as $a \wedge (\forall x.\, (((\neg b) \wedge c) \Rightarrow (d \Rightarrow e)))$.

We will also use the abbreviations defined below.

$$
\begin{array}{ll}
\forall x_1, x_2, \ldots, x_n.\ \varphi & \triangleq\ \forall x_1.\, \forall x_2. \cdots \forall x_n.\ \varphi \\
\exists x_1, x_2, \ldots, x_n.\ \varphi & \triangleq\ \exists x_1.\, \exists x_2. \cdots \exists x_n.\ \varphi \\
x_1\ R\ x_2 & \triangleq\ R(x_1, x_2) \\
x_1\ \not{R}\ x_2 & \triangleq\ \neg R(x_1, x_2) \\
\varphi \Leftrightarrow \psi & \triangleq\ (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi) \\
\textbf{if } \varphi \textbf{ then } \chi \textbf{ else } \psi \textbf{ fi} & \triangleq\ (\varphi \wedge \chi) \vee (\neg\varphi \wedge \psi).
\end{array}
$$

Example 2.1. Let $\mathcal{L}_{\mathbb{Z}}$ be the first-order language with the signature $\Sigma_{\mathbb{Z}} = (\mathbb{Z}, F_{\mathbb{Z}}, \mathcal{R}_{\mathbb{Z}})$, where $\mathbb{Z} = \{\ldots, -1, 0, 1, \ldots\}$, $F_{\mathbb{Z}} = \{*, \mathit{square}\}$, and $\mathcal{R}_{\mathbb{Z}} = \{=, <, >, \leq, \geq\}$. Using this language, we can express assertions such as these:

$$
\begin{array}{ll}
\forall x.\, \exists y.\ y > x, & \forall x, y.\ (x < y) \Leftrightarrow (y > x), \\
\forall k.\, \exists n.\ 3 * n > \mathit{square}(k), & \forall x.\ x = x \wedge x \geq 0.
\end{array}
$$

□

A variable occurrence is *bound* if it appears inside the scope of a $\forall$ or $\exists$ quantifier; otherwise, it is *free*. In the assertion $x = 5 \wedge \forall x.\ x \geq 0$, the first occurrence of x is free, whereas the second and third occurrences are bound by the $\forall$ quantifier. The notation φ_t^x represents the result of replacing each free occurrence of x in φ with the term t. If a free variable in t would become bound as a result of the substitution, the bound variables in φ are renamed to avoid clashing. An assertion or term that contains no free variables is said to be *closed*.

So far, we have considered assertions as purely syntactic entities. Our goal is to use them to represent Boolean functions (predicates). To make $\wedge$ mean "and", $\neg$ mean "not", and "$\mathit{square}(k)$" mean k^2, we must attribute a semantics to assertions.

The semantics is parameterized by a model $\mathcal{M}$ and a variable assignment σ. A *model* $\mathcal{M}$ for a first-order language $\mathcal{L}$ consists of a non-empty set $|\mathcal{M}|$ (the *domain*) and an operator $\_^{\mathcal{M}}$ (where '$\_$' is a placeholder for the operand) that interprets all non-logical symbols as follows: If c is a constant symbol, then $c^{\mathcal{M}} \in |\mathcal{M}|$; if f is an n-ary function symbol, then $f^{\mathcal{M}}$ is a function from $|\mathcal{M}|^n$ to $|\mathcal{M}|$; and if R is an n-ary relational symbol, then $R^{\mathcal{M}}$ is a relation on $|\mathcal{M}|^n$. A *variable assignment* σ is a function from the set V of variables to the domain $|\mathcal{M}|$. The semantics of a term t in $\mathcal{M}$ under σ, written $\mathcal{M}[\![t]\!]\sigma$, is defined by the equations

$$
\begin{array}{ll}
\mathcal{M}[\![x]\!]\sigma & \triangleq\ \sigma(x) \\
\mathcal{M}[\![c]\!]\sigma & \triangleq\ c^{\mathcal{M}} \\
\mathcal{M}[\![f(t_1, \ldots, t_n)]\!]\sigma & \triangleq\ f^{\mathcal{M}}(\mathcal{M}[\![t_1]\!]\sigma, \ldots, \mathcal{M}[\![t_n]\!]\sigma).
\end{array}
$$

The semantics of an assertion is given as follows: For an assertion φ, we write $(\mathcal{M}, \sigma) \models \varphi$ when φ is true in $\mathcal{M}$ under σ. The $\models$ relation is specified below.

$$
\begin{array}{ll}
(\mathcal{M}, \sigma) \models R(t_1, \ldots, t_n) & \text{iff } \langle \mathcal{M}[\![t_1]\!]\sigma, \ldots, \mathcal{M}[\![t_n]\!]\sigma \rangle \in R^{\mathcal{M}} \\
(\mathcal{M}, \sigma) \models \neg\varphi & \text{iff } (\mathcal{M}, \sigma) \not\models \varphi \\
(\mathcal{M}, \sigma) \models \varphi \wedge \psi & \text{iff } (\mathcal{M}, \sigma) \models \varphi \text{ and } (\mathcal{M}, \sigma) \models \psi \\
(\mathcal{M}, \sigma) \models \varphi \vee \psi & \text{iff } (\mathcal{M}, \sigma) \models \varphi \text{ or } (\mathcal{M}, \sigma) \models \psi \\
(\mathcal{M}, \sigma) \models \varphi \Rightarrow \psi & \text{iff } (\mathcal{M}, \sigma) \models \varphi \text{ implies } (\mathcal{M}, \sigma) \models \psi \\
(\mathcal{M}, \sigma) \models \forall x.\ \varphi & \text{iff } (\mathcal{M}, \sigma[x \mapsto a]) \models \varphi \text{ for all } a \in |\mathcal{M}| \\
(\mathcal{M}, \sigma) \models \exists x.\ \varphi & \text{iff } (\mathcal{M}, \sigma[x \mapsto a]) \models \varphi \text{ for some } a \in |\mathcal{M}|.
\end{array}
$$

If σ is a variable assignment, $\sigma[x \mapsto a]$ denotes the variable assignment that is identical to σ except that x maps to a. For closed assertions, the variable assignment is irrelevant, so we can simply write $\mathcal{M} \models \varphi$ to express that φ is true in $\mathcal{M}$.

Example 2.2. Let $\mathcal{M}$ be a model such that $|\mathcal{M}| = \mathbb{Z}$, $c^{\mathcal{M}} = c$, $*^{\mathcal{M}} = *$, $square^{\mathcal{M}} = \square^2$, and $>^{\mathcal{M}} = >$; furthermore, let σ be the variable assignment $\{n \mapsto 5, k \mapsto 4\}$. Is the assertion $3 * n > square(k)$ true in $\mathcal{M}$ under σ?

$$
\begin{array}{ll}
& (\mathcal{M}, \sigma) \models 3 * n > square(k) \\
\text{iff} & \langle \mathcal{M}[\![3 * n]\!]\sigma,\ \mathcal{M}[\![square(k)]\!]\sigma \rangle \in >^{\mathcal{M}} \\
\text{iff} & \mathcal{M}[\![3 * n]\!]\sigma > \mathcal{M}[\![square(k)]\!]\sigma \\
\text{iff} & *^{\mathcal{M}}(\mathcal{M}[\![3]\!]\sigma, \mathcal{M}[\![n]\!]\sigma) > square^{\mathcal{M}}(\mathcal{M}[\![k]\!]\sigma) \\
\text{iff} & 3 * \sigma(n) > \sigma^2(k) \\
\text{iff} & 3 * 5 > 4^2 \\
\text{iff} & 15 > 16.
\end{array}
$$

The assertion is false. □

We will now present a sound and complete proof system for first-order logic. The sequent calculus LK operates on formulas of the form $\Gamma \vdash \Delta$, called *sequents*, where Γ (the *antecedent formulas*) and Δ (the *succedent formulas*) are comma-separated multisets of closed assertions. Semantically, a sequent is *valid* if for all models $\mathcal{M}$, at least one succedent formula is true whenever all antecedent formulas are true. The axiom and proof rules of LK are specified below [Gal03, Han07]:

Axiom PA. $\Gamma, \varphi \vdash \varphi, \Delta$

Proof Rule L∧. $\dfrac{\Gamma, \varphi, \psi \vdash \Delta}{\Gamma, \varphi \wedge \psi \vdash \Delta}$ **Proof Rule R∧.** $\dfrac{\Gamma \vdash \varphi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \varphi \wedge \psi, \Delta}$

Proof Rule L∨. $\dfrac{\Gamma, \varphi \vdash \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \varphi \vee \psi \vdash \Delta}$ **Proof Rule R∨.** $\dfrac{\Gamma \vdash \varphi, \psi, \Delta}{\Gamma \vdash \varphi \vee \psi, \Delta}$

Proof Rule L⇒. $\dfrac{\Gamma \vdash \varphi, \Delta \quad \Gamma, \psi \vdash \Delta}{\Gamma, \varphi \Rightarrow \psi \vdash \Delta}$ **Proof Rule R⇒.** $\dfrac{\Gamma, \varphi \vdash \psi, \Delta}{\Gamma \vdash \varphi \Rightarrow \psi, \Delta}$

Proof Rule L¬. $\dfrac{\Gamma \vdash \varphi, \Delta}{\Gamma, \neg\varphi \vdash \Delta}$ **Proof Rule R¬.** $\dfrac{\Gamma, \varphi \vdash \Delta}{\Gamma \vdash \neg\varphi, \Delta}$

Proof Rule L∀. $\dfrac{\Gamma, \forall x.\, \varphi, \varphi^x_t \vdash \Delta}{\Gamma, \forall x.\, \varphi \vdash \Delta}$ **Proof Rule R∀.** $\dfrac{\Gamma \vdash \varphi^x_a, \Delta}{\Gamma \vdash \forall x.\, \varphi, \Delta}$

Proof Rule L∃. $\dfrac{\Gamma, \varphi^x_a \vdash \Delta}{\Gamma, \exists x.\, \varphi \vdash \Delta}$ **Proof Rule R∃.** $\dfrac{\Gamma \vdash \exists x.\, \varphi, \varphi^x_t, \Delta}{\Gamma \vdash \exists x.\, \varphi, \Delta}$

In Proof Rules L∀ and R∃, t is an arbitrary closed term. In Proof Rules L∃ and R∀, a is a constant that doesn't appear in the conclusion (a "fresh" constant).

Example 2.3. Let *nick* and *dime* be constants. Here is a proof tree for the sequent $nick = 5 \Rightarrow dime = 10,\ nick = 5 \vdash dime = 10$:

$$
\dfrac{\dfrac{}{nick = 5 \vdash nick = 5,\ dime = 10}\ \text{PA} \qquad \dfrac{}{dime = 10,\ nick = 5 \vdash dime = 10}\ \text{PA}}{nick = 5 \Rightarrow dime = 10,\ nick = 5 \vdash dime = 10}\ \text{L}\Rightarrow
$$

By soundness of the sequent calculus LK, the assertion $dime = 10$ is true for any model that satisfies the assertions $nick = 5 \Rightarrow dime = 10$ and $nick = 5$. □

Example 2.4. In practice, the assertions Γ on the left-hand side of the sequent symbol $\vdash$ usually specify data types (natural numbers, lists, sets, etc.). In the proof tree below, the assertion $\forall x.\, \exists y.\ y > x$ specifies a well-known property of natural numbers:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\forall x.\, \exists y.\ y > x,\ a > 1000 \vdash \exists k.\, k > 1000,\ a > 1000}\,\mathrm{PA}}{\forall x.\, \exists y.\ y > x,\ a > 1000 \vdash \exists k.\, k > 1000}\,\mathrm{R}\exists}{\forall x.\, \exists y.\ y > x,\ \exists y.\, y > 1000 \vdash \exists k.\, k > 1000}\,\mathrm{L}\exists}{\forall x.\, \exists y.\ y > x \vdash \exists k.\, k > 1000}\,\mathrm{L}\forall$$

□

Assertion validity in first-order logic is a semidecidable problem: There exists an algorithm that derives an LK-proof for any valid formula in a finite amount of steps, but if an invalid formula is supplied to the algorithm, it may run forever without producing any results. Most automated theorem provers rely on more efficient (and usually more complex) calculi than LK [Shu01], but they suffer from the same theoretical limitations as LK.

In the above development, we barely mentioned types. Yet, when using first-order logic for specifying properties of programs, we constantly need to express things like "for all n of type **int**" or "for some s of type **str**", where **int** and **str** are data types from the underlying programming language. There are two main ways to handle this [LP99]:

- We can use an untyped first-order language that caters for an axiomatic set theory (say, Zermelo–Fraenkel). In this context, sets are the only type, and every object (integer, string, etc.) is encoded as a set. This approach is traditionally favored by mathematicians, who generally regard set theory as the foundation of mathematics. To express that every integer is less than its successor, we would write $\forall n.\ n \in \mathbf{int} \Rightarrow n < n + 1$. All of this can be done within the framework presented in this section.
- We can extend the syntax and semantics of first-order logic with types (also called sorts). Variables, constant symbols, function symbols, and relation symbols must then be declared with their types, and only well-typed assumptions are legal. This is the approach traditionally favored by computer scientists and used by most theorem provers, because types allow the machine to catch errors that would otherwise go undetected. Using this approach, we could write $\forall n : \mathbf{int}.\ n < n + 1$, having declared $<$ as a relation symbol on $\mathbf{int} \times \mathbf{int}$.

In the rest of the thesis, we will prefer the second approach, since it is general enough for our purposes and typed assertions are easy to recast to untyped assertions. Furthermore, we will often omit the typing constraints and write, say, $\forall n.\ n < n + 1$ when the types can be inferred from the context.

> The fact that occasionally some particular technical concept mentioned in passing (for example, "the Church–Rosser property") may be unfamiliar should not be seen as an obstacle. It should be taken in a relaxed, sporting spirit.
>
> — *Maude Manual, Version 2.3* (2007)

Chapter 3

Rewriting Logic and Maude

Maude is a programming and specification language based on order-sorted rewriting logic [CDEL+07]. This chapter provides an introduction to rewriting logic in Maude 2, focusing on the features that are used in this thesis. The presentation is based primarily on Ölveczky [Ölv07] and Baader and Nipkow [BN98].

Section 3.1 reviews Maude's equational subset by defining some basic data types: natural numbers, pairs, binary trees, lists, multisets, and sets. Section 3.2 reviews Maude's support for membership equational logic. Section 3.3 introduces rewrite rules by formally specifying the tennis scoring rules and demonstrates Maude's built-in term-rewriting and search capabilities. Section 3.4 presents the formal semantics of rewriting logic specifications. Section 3.5 shows how to manipulate Maude specifications using the `META-LEVEL` module, relying on rewriting logic's reflective nature. Finally, Section 3.6 presents the implementation of an interpreter for a minimalistic imperative programming language, written in Maude.

3.1 Equational Specifications

Data types in Maude are defined by algebraic specifications. As an example, consider the set $\mathbb{N}$ of natural numbers. Following Peano, we can define it by induction: 0 is a natural number, and if n is a natural number, then $\mathrm{S}n$ (the successor of n) is a natural number. In Peano's notation, the number 4 would be written SSSS0. For $m, n \in \mathbb{N}$, we can define addition and multiplication using equations as follows:

Equation N1. $m + 0 \triangleq m$ **Equation N3.** $m * 0 \triangleq 0$

Equation N2. $m + \mathrm{S}n \triangleq \mathrm{S}(m + n)$ **Equation N4.** $m * \mathrm{S}n \triangleq m + (m * n)$

These equations can be used to compute the value of any complex term that contains $+$ or $*$. For example, here is one way to compute the value of $\mathrm{SSS0} * \mathrm{SS0}$:

$$
\begin{aligned}
& \mathrm{SSS0} * \mathrm{SS0} \\
\overset{\mathrm{N4}}{\rightsquigarrow}\ & \mathrm{SSS0} + (\mathrm{SSS0} * \mathrm{S0}) \\
\overset{\mathrm{N4}}{\rightsquigarrow}\ & \mathrm{SSS0} + (\mathrm{SSS0} + (\mathrm{SSS0} * 0)) \\
\overset{\mathrm{N3}}{\rightsquigarrow}\ & \mathrm{SSS0} + (\mathrm{SSS0} + 0)
\end{aligned}
$$

$\overset{N1}{\rightsquigarrow}$ $\mathrm{SSS0} + \mathrm{SSS0}$
$\overset{N2}{\rightsquigarrow}$ $\mathrm{S}(\mathrm{SSS0} + \mathrm{SS0})$
$\overset{N2}{\rightsquigarrow}$ $\mathrm{SS}(\mathrm{SSS0} + \mathrm{S0})$
$\overset{N2}{\rightsquigarrow}$ $\mathrm{SSS}(\mathrm{SSS0} + 0)$
$\overset{N1}{\rightsquigarrow}$ SSSSSS0.

The above algebraic definition can serve as the basis of a Maude module that provides natural numbers:

```
fmod PEANO is
  sort Nat .

  op 0 : -> Nat [ctor] .
  op s_ : Nat -> Nat [ctor prec 1] .

  op _+_ : Nat Nat -> Nat [prec 7 gather (E e)] .
  op _*_ : Nat Nat -> Nat [prec 5 gather (E e)] .

  vars M N : Nat .

  eq M + 0 = M .
  eq M + s N = s (M + N) .

  eq M * 0 = 0 .
  eq M * s N = M + M * N .
endfm
```

The module is called `PEANO` so as not to clash with Maude's predefined `NAT` module, which is part of Maude's automatically loaded `prelude.maude` file. The `fmod` keyword indicates that the module is a *functional* (or *equational*) *module*.

We define a sort (type) `Nat` that corresponds to the set $\mathbb{N}$ of natural numbers. We define two constructor functions, identified by the `ctor` attribute. The first constructor, `0`, is a constant (or nullary function) of sort `Nat`. The second constructor, `s_`, takes a `Nat` and returns a `Nat`. These constructors allow us to specify natural numbers: `0`, `s 0`, `s s 0`, `s s s 0`, etc. The underscore in `s_` indicates that `s` is a prefix operator. Maude supports prefix, postfix, infix, and even "mixfix" operators.

Next, we declare the infix operators `+` and `*`. Unlike `0` and `s`, these operators are not constructors. Terms like `s s 0 + s 0` and `s s s 0 * s s 0` that contain these operators are reduced to constructor terms by equations.

For operators that have leading or trailing underscores, we resolve potential parsing ambiguities by specifying a precedence using the `prec` attribute. Lower numbers denote higher precedences. For example, `s` is given higher precedence than `*` to ensure that `s 0 * 0` is parsed as `(s 0) * 0`, not as `s (0 * 0)`. When constructing terms, we can use parentheses to force a specific precedence order.

The `gather` attribute is used to ensure that the operators are left-associative. For example, `1 + 2 - 3 + 4` is interpreted as `((1 + 2) - 3) + 4`. Right-associativity would be specified as `gather (e E)`.

Maude's `red` (reduce) command simplifies a given term by applying equations in a left-to-right manner until no equation can be applied. If several equations can be

applied on the same term, Maude will choose one. Using `red`, we can compute the value of any well-formed term built using `0`, `s`, `+`, and `*`:

```
red s s s s s 0 .                  *** result Nat: s s s s s 0
red 0 + 0 .                        *** result Nat: 0
red s 0 + s s 0 .                  *** result Nat: s s s 0
red s s s 0 * s s 0 .              *** result Nat: s s s s s s 0
red s (s 0 + s s 0) .              *** result Nat: s s s s 0
```

The result of each command is displayed in a single-line comment introduced by three consecutive asterisks (`***`).

A desirable property of an equational specification in Maude is *termination*, meaning that irrespective of the starting term, only a finite number of reductions can be performed before reaching an irreducible term. The `PEANO` module is terminating, but we can make it nonterminating by adding the equation

```
eq M = M + 0 .
```

Although it looks harmless in its assertion of the obvious, this equation can cause Maude to loop infinitely. Indeed, depending on which equations Maude chooses to execute when, `red 0` can give rise to a looping computation:

`0` $\rightsquigarrow$ `0 + 0` $\rightsquigarrow$ `0` $\rightsquigarrow$ `0 + 0` $\rightsquigarrow$ `0` $\rightsquigarrow$ ...

It can also lead to a non-looping infinite computation:

`0` $\rightsquigarrow$ `0 + 0` $\rightsquigarrow$ `(0 + 0) + 0` $\rightsquigarrow$ `((0 + 0) + 0) + 0` $\rightsquigarrow$ ...

As a rule of thumb, we can obtain termination by ensuring that the right-hand sides of our equations are "simpler" than the corresponding left-hand sides. For example, `0` is simpler than `0 + 0`, because the latter embeds the former as a subterm. Termination is generally undecidable, but there are techniques to prove termination for interesting classes of specifications.

Another desirable property of equational specifications is *confluence* (also called the *Church–Rosser property*). For terminating specifications, confluence means that we always obtain the same irreducible term regardless of which equations are applied in which order. Consider the following extension to `PEANO`:

```
op min{_,_} : Nat Nat -> Nat .

eq min { M, 0 } = 0 .
eq min { 0, N } = 0 .
eq min { s M, s N } = s min { M, N } .
```

The `min{_,_}` operator returns the minimum of its two arguments. If we ask Maude to reduce `min { 0, 0 }`, it will apply either the first equation (with `M` bound to `0`) or the second equation (with `N` bound to `0`)—either way, the result is `0`. The specification is confluent as well as terminating.

Let us continue our review of Maude with the following module.

```
fmod NAT-PAIR is
  including NAT .

  sort NatPair .
```

```
  op <_,_> : Nat Nat -> NatPair [ctor] .

  op _+_ : NatPair NatPair -> NatPair .

  vars M1 M2 N1 N2 : Nat .

  eq < M1, N1 > + < M2, N2 > = < M1 + M2, N1 + N2 > .
endfm
```

The `NAT-PAIR` module defines the sort `NatPair`, which consists of constructor terms of the form `< m, n >`, where *m* and *n* are of the sort `Nat`. In addition to the constructor function, the `NatPair` sort provides a binary `+` operator implemented in terms of `Nat`'s `+` operator. This is possible because Maude supports operator overloading.

The `including NAT` declaration near the top imports Maude's predefined `NAT` module into the current module, so that we can access the built-in `Nat` sort. We could have imported `PEANO` instead of `NAT`, but `NAT` is more efficient and lets us use decimal numbers interchangeably with Peano numbers.

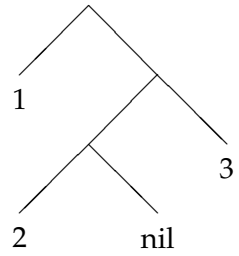
Here are a few examples of `NatPair` in action:

```
red < s 0, s s s s 0 > .          *** result NatPair: < 1, 4 >
red < 3 + 4, 5 * 6 + 7 > .        *** result NatPair: < 7, 37 >
red < 1, 2 > + < 3, 4 > .         *** result NatPair: < 4, 6 >
```

The `NatPair` sort demonstrates how to reuse existing sorts when defining new sorts. The next example takes this idea further:

```
fmod LISP-TREE is
  including NAT .

  sort Leaf .
  subsort Nat < Leaf .

  sort InnerNode .

  sort Tree .
  subsort Leaf < Tree .
  subsort InnerNode < Tree .

  op nil : -> Leaf [ctor] .
  op cons__ : Tree Tree -> InnerNode [ctor] .

  op car_ : InnerNode -> Tree .
  op cdr_ : InnerNode -> Tree .

  vars T1 T2 : Tree .

  eq (car (cons T1 T2)) = T1 .
  eq (cdr (cons T1 T2)) = T2 .
endfm
```

The `LISP-TREE` module defines a `Tree` sort that represents a binary tree, using Lisp notation. For example, `(cons 1 (cons (cons 2 nil) 3))` represents the tree



In addition to `Tree`, the `LISP-TREE` module defines the `Leaf` and `InnerNode` sorts, which represent the leaves and inner nodes of a tree, respectively. The declaration `subsort Nat < Leaf` makes `Nat` a *subsort* of `Leaf`, meaning that all `Nat` terms are also `Leaf` terms. Moreover, as in Lisp, the term `nil` can be used as a leaf. The declarations `subsort Leaf < Tree` and `subsort InnerNode < Tree` state that leaves and inner nodes are trees. The `cons` constructor function takes two trees and returns an inner node. The `car` and `cdr` functions are also modeled after Lisp: They take an inner node and return the first and second subtrees, respectively.[1]

For convenience, most Lisp interpreters provide the functions `caar`, `cadr`, `cdar`, and `cddr`, which correspond to two successive applications of `car` and/or `cdr`. In Maude, we would define them as follows:

```
op caar_ : InnerNode ~> Tree .
op cadr_ : InnerNode ~> Tree .
op cdar_ : InnerNode ~> Tree .
op cddr_ : InnerNode ~> Tree .

var I : InnerNode .

eq (caar I) = (car (car I)) .
eq (cadr I) = (car (cdr I)) .
eq (cdar I) = (cdr (car I)) .
eq (cddr I) = (cdr (cdr I)) .
```

The new functions' signatures use `~>` instead of `->` to indicate that these are partial functions. For example, `(caar (cons 1 2))` is not defined:

```
red (caar (cons 1 2)) .             *** result [Tree]: car 1
```

Maude first expands `(caar (cons 1 2))` to `(car (car (cons 1 2)))`; then it reduces the subterm `(car (cons 1 2))` to `1`. At that point, no equations can be applied, so we are left with the irreducible non-constructor term `(car 1)`. When Maude displays the result, it puts the sort name in square brackets, denoting a "kind" (an error sort). To avoid these undesirable terms, we must ensure that partial functions are used only on values for which they are defined.

Another approach to partial functions is simply to avoid them. Here is how we could define `cadr` as a total function:

```
op cadr_ : InnerNode -> Tree .
```

[1] The names `car` and `cdr` come from the original implementation of Lisp on the IBM 704, whose memory words had a 15-bit "address" part and a 15-bit "decrement" part. In this context, `car` stood for "contents of address part of register" and `cdr` stood for "contents of decrement part of register" [AS96]. Standard ML calls these functions `hd` (head) and `tl` (tail).

```
eq (cadr I) =
   if (cdr I) :: InnerNode then (car (cdr I)) else (cdr I) fi .
```

We start by extracting the second subtree of `I` using `cdr`. If the result is a tree, we extract its first subtree; otherwise, we simply return `(cdr I)`.

The example uses Maude's built-in `if_then_else_fi` and `_::_S` operators. The `if_then_else_fi` operator chooses the `then` or `else` branch based on whether the condition reduces to `true` or `false`. The `_::_S` operator returns `true` if and only if its argument has sort `S`, which may be any sort.

Instead of using `if_then_else_fi`, we can specify a pair of conditional equations:

```
ceq (cadr I) = (car (cdr I)) if (cdr I) :: InnerNode .
ceq (cadr I) = (cdr I)       if not (cdr I) :: InnerNode .
```

Conditional equations are applied only if their side condition is true. (The `if` keyword that introduces the side condition should not be confused with the `if` token of `if_then_else_fi`.) Side conditions are normally used to restrict the applicability of a rule, but they can also be used purely to bind values to variables. For example, the equation below binds `(cdr I)` to `T` and then uses `T` in the equation's right-hand side:

```
var T : Tree .

ceq (cadr I) = if T :: InnerNode then (car T) else T fi
if T := (cdr I) .
```

Lisp programmers usually regard lists as degenerate binary trees. For example, `nil` is an empty list, and `(cons 1 (cons 2 (cons 3 (cons 4 nil))))` represents the list [1, 2, 3, 4]. In Maude, it is usually more convenient to use a dedicated sort to represent lists. Here is a module that defines a `NatList` sort:

```
fmod NAT-LIST is
  including NAT .

  sort NatList .

  op nil : -> NatList [ctor] .
  op __ : Nat NatList -> NatList [ctor] .
endfm
```

The empty list is represented by `nil`. More complex lists are built by prepending a `Nat` to an existing `NatList`; for example, `1 2 nil` represents the two-element list [1, 2] and is constructed by prepending `1` to the list `2 nil`, which in turn is obtained by prepending `2` to the list `nil`. We can make the module more useful, and eliminate the need for trailing `nil`s, by defining the `NatList` sort slightly differently:

```
fmod NAT-LIST is
  including NAT .

  sort NatList .
  subsort Nat < NatList .

  op nil : -> NatList [ctor] .
  op __ : NatList NatList -> NatList [ctor assoc id: nil] .
endfm
```

This time we make `Nat` a subsort of `NatList`, so that plain `Nat` terms such as `7` and `32` can be used as one-element lists. The `__` operator becomes a general concatenation operator that takes two `NatLists`, rather than a `Nat` and a `NatList`. The operator is also declared with the `assoc` (associative) attribute, so that `(1 2) 3` and `1 (2 3)` are considered equal, allowing us to write `1 2 3` with no ambiguity. Finally, the `id: nil` attribute makes `nil` an identity element for concatenation. This ensures that prepending or appending `nil` to a list leaves the list unchanged.

When defining functions on `NatLists`, we usually distinguish two cases: the empty list `nil` and lists of the form *`n l`*, where *`n`* is a `Nat` and *`l`* is a `NatList`. For example:

```
op size : NatList -> Nat .
op reverse : NatList -> NatList .

var L : NatList .
var N : Nat .

eq size(nil) = 0 .
eq size(N L) = 1 + size(L) .

eq reverse(nil) = nil .
eq reverse(N L) = reverse(L) N .
```

Unlike the symbols defined so far, the `size` and `reverse` functions don't specify the position of their arguments using underscores. In such cases, the arguments must be supplied in parentheses after the function name:

```
red size(nil) .                    *** result Zero: 0
red size(1 2 3) .                  *** result NzNat: 3
red reverse(1 2 3) .               *** result NatList: 3 2 1
```

(The `Zero` and `NzNat` sorts are subsorts of `Nat`.)

From an algebraic point of view, the distinctive feature of lists is that concatenation is associative. When we have a list such as `1 2 3 4`, it is immaterial whether this list is the concatenation of `1 2` and `3 4`, or of `1` and `2 3 4`, or of `1 2 3 4` and `nil`: All these interpretations are equally valid when it comes to matching the left-hand side of an equation with an actual term.

If we relax the definition of concatenation so that it is commutative as well as associative, we obtain a different data structure: a multiset, or unordered list. Here is the Maude specification of a `NatMSet` sort:

```
fmod NAT-MULTISET is
  including NAT .

  sort NatMSet .
  subsort Nat < NatMSet .

  op empty : -> NatMSet [ctor] .
  op _++_ : NatMSet NatMSet -> NatMSet
                                [ctor prec 7 assoc comm id: empty] .

  op |_| : NatMSet -> Nat .
  op _in_ : Nat NatMSet -> Bool [prec 9] .
```

```
  var MS : NatMSet .
  vars N N' : Nat .

  eq | empty | = 0 .
  eq | N ++ MS | = 1 + | MS | .

  eq N in empty = false .
  eq N in N' ++ MS = (N == N') or (N in MS) .
endfm
```

The empty multiset is written `empty`. The union of two multisets ms_1 and ms_2 is written ms_1 `++` ms_2. Because `++` is declared with the `comm` attribute, the order of the elements in a multiset is irrelevant. Thus, `4 ++ 7` and `7 ++ 4` denote the same multiset and are considered equal.

The cardinality operator `|_|` is defined recursively on multisets. Because multisets are associative and have an identity element, we need only to consider the cases `empty` and n `++` ms. The cardinality of the multiset `7 ++ 7` is `2`, because every occurrence of an element in a multiset counts.

The `_in_` operator is defined in a similar way. An alternative, equally valid definition would be

```
eq N in N ++ MS = true .
eq N in MS = false [otherwise] .
```

By associativity and commutativity, `N ++ MS` will match any multiset that contains `N`. The `otherwise` attribute tells Maude to use the second equation only when no other equations are applicable for `in`. Here are a few examples:

```
red 4 in 4 .                    *** result Bool: true
red 4 in 7 ++ 4 .               *** result Bool: true
red 9 in 7 ++ 4 .               *** result Bool: false
```

A set, like a multiset, is an associative and commutative data structure, but in addition set union is idempotent. Idempotence means that S `++` S equals S for any set S. To implement this in Maude, we could create a `NatSet` sort identical to `NatMSet` except that it would also include the equation

```
eq N ++ N = N .
```

Now, what is the cardinality of `4 ++ 4`? Depending on which equation is applied first, we obtain `2` or `1`:

```
| 4 ++ 4 | ⇝ 1 + | 4 | ⇝ 1 + | 4 ++ empty | ⇝ 1 + 1 + | empty |
          ⇝ 2 + | empty |  ⇝ 2 + 0 ⇝ 2
| 4 ++ 4 | ⇝ | 4 | ⇝ | 4 ++ empty | ⇝ 1 + | empty | ⇝ 1 + 0 ⇝ 1.
```

Because of the equation `N ++ N = N`, the specification is no longer confluent. The solution is to change the equations associated with `|_|` so that whenever an element `N` is counted, any superfluous occurrences of that element in the remaining set `S` are removed explicitly using the set subtraction operator `_\_`:

```
eq | empty | = 0 .
eq | N ++ S | = 1 + | S \ N | .
```

The set subtraction operator is defined as follows:

```
op _\_ : NatSet Nat -> NatSet [prec 7] .

eq empty \ N = empty .
eq (N ++ S) \ N' = if N == N' then S \ N' else N ++ (S \ N') fi .
```

This is sufficient to make the specification confluent. Confluence is generally undecidable, but for terminating equational specifications there is an algorithm that decides confluence.

3.2 Membership Axioms

Equational logic lets us implement data types by declaring sorts, subsorts, and constructors. Maude extends equational logic with a powerful mechanism called *membership axioms* that lets us specify subsorts based on semantic criteria.

Suppose that we would like to specify a `NatTwins` sort consisting of terms of the form `< n, n >`, where *n* is a natural number. This sort is very similar to the `NatPair` sort presented in the previous section, except that for `NatTwins` the two components of the pair must be identical. Using membership axioms and building upon the `NatPair` sort, we obtain the following specification:

```
fmod NAT-TWINS is
  including NAT-PAIR .

  sort NatTwins .
  subsort NatTwins < NatPair .

  var N : Nat .

  mb < N, N > : NatTwins .
endfm
```

The `NatTwins` sort is declared as a subsort of `NatPair`. The membership axiom `< N, N > : NatTwins` specifies that all `NatPair` terms of the form `< N, N >` qualify as `NatTwins` terms.

Like equations, membership axioms may have a side condition. For example, here is how we could define a `NatNonTwins` sort with terms of the form `< m, n >` where *m* and *n* are distinct:

```
fmod NAT-NON-TWINS is
  including NAT-PAIR .

  sort NatNonTwins .
  subsort NatNonTwins < NatPair .

  vars M N : Nat .

  cmb < M, N > : NatNonTwins if M =/= N .
endfm
```

3.3 Rewrite Rules

Functions on data structures such as binary trees, lists, multisets, and sets should be deterministic and terminating. However, sometimes we want to model systems that are nondeterministic, nonterminating, or both. For example, a program that contains an infinite loop gives rise to a nonterminating process, yet we might still want to model this process using Maude. Similarly, concurrent systems often give rise to nondeterminism.

Besides equations, which let Maude replace equals for equals, we can model change in a system using unidirectional rewrite rules. These rewrite rules are applied in much the same way as equations, but they need not be confluent or terminating.

To illustrate this, we will attempt to model a tennis game in Maude. The standard scoring rules for tennis from the 2007 edition of the International Tennis Federation's *Rules of Tennis* are given below.

> A standard game is scored as follows with the server's score being called first:
>
> No point — "Love"
> First point — "15"
> Second point — "30"
> Third point — "40"
> Fourth point — "Game"
>
> except that if each player/team has won three points, the score is "Deuce". After "Deuce", the score is "Advantage" for the player/team who wins the next point. If that same player/team also wins the next point, that player/team wins the "Game"; if the opposing player/team wins the next point, the score is again "Deuce". A player/team needs to win two consecutive points immediately after "Deuce" to win the "Game".

The following module specifies these rules in Maude:

```
mod TENNIS-GAME is
  including NAT .

  sort Game .

  op _--_ : Nat Nat -> Game [ctor] .
  op deuce : -> Game [ctor] .
  op advantage-server : -> Game [ctor] .
  op advantage-receiver : -> Game [ctor] .
  op game-server : -> Game [ctor] .
  op game-receiver : -> Game [ctor] .

  var N : Nat .

  eq 40 -- 40 = deuce .

  *** the server wins a point
  rl 0 -- N => 15 -- N .
```

```
  rl 15 -- N => 30 -- N .
  rl 30 -- N => 40 -- N .
  crl 40 -- N => game-server if N < 40 .
  rl deuce => advantage-server .
  rl advantage-server => game-server .
  rl advantage-receiver => deuce .

  *** the receiver wins a point
  rl N -- 0 => N -- 15 .
  rl N -- 15 => N -- 30 .
  rl N -- 30 => N -- 40 .
  crl N -- 40 => game-receiver if N < 40 .
  rl deuce => advantage-receiver .
  rl advantage-receiver => game-receiver .
  rl advantage-server => deuce .
endm
```

A term of sort `Game` represents the score of a game. It can be of the form *m* `--` *n*, where *m* is the server's score and *n* is the receiver's score, or it can be one of `deuce`, `advantage-server`, `advantage-receiver`, `game-server`, and `game-receiver`. We let `0` stand for "Love". The score `40 -- 40` is considered the same as `deuce`, thanks to the equation `40 -- 40 = deuce`.

The `mod` keyword at the very beginning of the module indicates that it is a *system module*. Unlike functional modules, system modules may contain rewrite rules. Rewrite rules are specified using the `rl` or `crl` keyword and the `=>` operator between the left- and right-hand sides.

An inspection of the rules shows that from the start state `0 -- 0`, two final states are possible: `game-server` and `game-receiver`. This means that the system is not confluent. It is not terminating either, because from `deuce` it allows infinite looping:

`deuce` ⟶ `advantage-server` ⟶ `deuce` ⟶ ...

To analyze system modules, Maude provides the `rew`, `frew`, and `search` commands. The `rew` command rewrites a term using equations and rewrite rules until none can be applied. More precisely, at each rewrite step, it starts by reducing the term using equations (as `red` would do), then it applies a rewrite rule that matches. At the end, it reduces the term one last time using equations. For example:

```
rew 0 -- 0 .                        *** result Game: game-server
```

We can specify a limit on the number of rewrite steps in brackets:

```
rew [1] 0 -- 0 .                    *** result Game: 15 -- 0
rew [2] 0 -- 0 .                    *** result Game: 30 -- 0
rew [3] 0 -- 0 .                    *** result Game: 40 -- 0
rew [4] 0 -- 0 .                    *** result Game: game-server
```

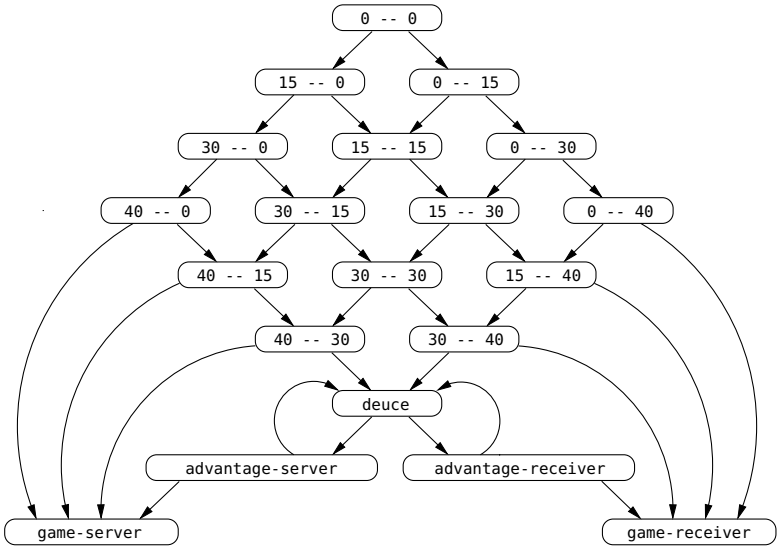
The `frew` command is similar to `rew`, except that it is fairer in its choice of subterm on which to apply a rewrite rule. For the example above, it would make no difference since all the rewrite rules operate on entire terms. A typical example that would benefit from `frew` is that of a multiset of tennis games played in parallel, each represented by a subterm in the multiset.

While the `rew` and `frew` commands find a single path, the `search` command performs a breadth-first search to find all reachable states from a given state. For example, the command

```
search 0 -- 0 =>1 G:Game .
```

asks Maude to return all the terms of sort `Game` that can be reached from the term `0 -- 0` in exactly one rewrite step. Maude finds exactly two solutions: `15 -- 0` and `0 -- 15`. By replacing `=>1` with `=>*` in the `search` command, we ask Maude to list all states reachable in zero or more rewrite steps. This time, Maude finds 20 solutions, including `0 -- 0`. Finally, by using `=>!` in the `search` command, we obtain the reachable states from which no transitions are possible, namely `game-server` and `game-receiver`.

For any state, we can retrieve the list of rewrite rules applied to reach that state using `show path`. This enables us to construct a transition graph for the system:



If only a finite number of states are reachable from the initial state (as is always the case with tennis games), `search` will give us all the possible results, or output "No solution", in a finite amount of time—unless it aborts for some reason. If an infinite number of states are reachable from the initial state, the search will go on forever.

3.4 Logical Semantics of Rewriting Logic Specifications

So far, we have used an informal description of how Maude executes a specification. We intuitively know how to "apply an equation", and we have some notion of whether the left-hand side of an equation or rewrite rule "matches" a term or not. Since Maude is used throughout this thesis, both as a programming language and as a semantic framework for giving executable semantics to Creol, it is appropriate to rigorously specify its semantics.

To keep the presentation as simple as possible, we will focus on a subset of rewriting logic. We will assume that the specification contains only one sort and that equations and rewrite rules are unconditional. Moreover, we will ignore Maude's

attributes (`assoc`, `comm`, `id`, etc.) and will assume that all terms are expressed using a functional syntax.

A *rewriting logic specification* is a triple $\mathcal{R} = (\Sigma, E, R)$ where Σ is a set of function symbols (with arities), E is a set of equations of the form $l(x_1, \ldots, x_p) = r(x_1, \ldots, x_p)$, and R is a set of rewrite rules of the form $l(x_1, \ldots, x_p) \longrightarrow r(x_1, \ldots, x_p)$.

The subtheory $\mathcal{E} = (\Sigma, E)$ of $\mathcal{R}$ constitutes an *equational logic specification*. To express that two terms t and u built using function symbols in Σ are equal by the equations in E, we will write $\mathcal{E} \vdash t = u$. Let $l(x_1, \ldots, x_p) = r(x_1, \ldots, x_p)$ be an equation in E, let $t, u, v, \ldots$ be variable-free terms on Σ, and let f be an n-ary function symbol from Σ. *Equational logic* consists of the following axiom schemas and derivation rules:

Axiom E1 (Reflexivity)

$$\mathcal{E} \vdash t = t$$

Axiom E2 (Substitutivity)

$$\mathcal{E} \vdash l(t_1, \ldots, t_p) = r(t_1, \ldots, t_p)$$

Derivation Rule E3 (Symmetry)

$$\frac{\mathcal{E} \vdash t = u}{\mathcal{E} \vdash u = t}$$

Derivation Rule E4 (Transitivity)

$$\frac{\mathcal{E} \vdash t = u \quad \mathcal{E} \vdash u = v}{\mathcal{E} \vdash t = v}$$

Derivation Rule E5 (Congruence)

$$\frac{\mathcal{E} \vdash t_1 = u_1 \quad \cdots \quad \mathcal{E} \vdash t_n = u_n}{\mathcal{E} \vdash f(t_1, \ldots, t_n) = f(u_1, \ldots, u_n)}$$

The Axioms and Derivation Rules E1–E5 express properties that we can use for reasoning about equality.

Example 3.1. Inspired by the elementary arithmetic result $1 + 2 = 2 + 1$, we would like to derive the formula $\mathcal{E}_+ \vdash \mathrm{S0} + \mathrm{SS0} = \mathrm{SS0} + \mathrm{S0}$. The equational specification $\mathcal{E}_+ = (\Sigma_+, E_+)$ has the alphabet $\Sigma_+ = \{0, \mathrm{S}, +\}$ and the equation set $E_+ = \{m + 0 = m,\ m + \mathrm{S}n = \mathrm{S}(m + n)\}$. Here is the complete derivation:

1. $\mathcal{E}_+ \vdash \mathrm{S0} + \mathrm{SS0} = \mathrm{S}(\mathrm{S0} + \mathrm{S0})$ (by E2)
2. $\mathcal{E}_+ \vdash \mathrm{S}(\mathrm{S0} + \mathrm{S0}) = \mathrm{SS}(\mathrm{S0} + 0)$ (by E2)
3. $\mathcal{E}_+ \vdash \mathrm{S0} + \mathrm{SS0} = \mathrm{SS}(\mathrm{S0} + 0)$ (by E4 on 1 and 2)
4. $\mathcal{E}_+ \vdash \mathrm{SS}(\mathrm{S0} + 0) = \mathrm{SSS0}$ (by E2)
5. $\mathcal{E}_+ \vdash \mathrm{S0} + \mathrm{SS0} = \mathrm{SSS0}$ (by E4 on 3 and 4)
6. $\mathcal{E}_+ \vdash \mathrm{SS0} + \mathrm{S0} = \mathrm{S}(\mathrm{SS0} + 0)$ (by E2)
7. $\mathcal{E}_+ \vdash \mathrm{S}(\mathrm{SS0} + 0) = \mathrm{SSS0}$ (by E2)
8. $\mathcal{E}_+ \vdash \mathrm{SS0} + \mathrm{S0} = \mathrm{SSS0}$ (by E4 on 6 and 7)
9. $\mathcal{E}_+ \vdash \mathrm{SSS0} = \mathrm{SS0} + \mathrm{S0}$ (by E3 on 8)
10. $\mathcal{E}_+ \vdash \mathrm{S0} + \mathrm{SS0} = \mathrm{SS0} + \mathrm{S0}$. (by E4 on 5 and 9)

In the derivation, each instance of Axiom E2 corresponds to the application of an equation from E_+. □

Rewriting logic supplements equational logic with its own set of axiom schemas and derivation rules. To express that the term t can be rewritten into u using the equations and rewrite rules in $\mathcal{R} = (\Sigma, E, R)$, we will write $\mathcal{R} \vdash t \longrightarrow u$. Let $l(x_1, \ldots, x_p) \longrightarrow r(x_1, \ldots, x_p)$ be a rewrite rule in R, let $t, u, v, \ldots$ be variable-free terms on Σ, and let f be an n-ary function symbol from Σ. The following axiom schema and derivation rules define *rewriting logic*:

Axiom R1 (Reflexivity)

$$\mathcal{R} \vdash t \longrightarrow t$$

Derivation Rule R2 (Equality)

$$\frac{\mathcal{R} \vdash t \longrightarrow u \quad \mathcal{E} \vdash t = t' \quad \mathcal{E} \vdash u = u'}{\mathcal{R} \vdash t' \longrightarrow u'}$$

Derivation Rule R3 (Replacement)

$$\frac{\mathcal{R} \vdash t_1 \longrightarrow u_1 \quad \cdots \quad \mathcal{R} \vdash t_p \longrightarrow u_p}{\mathcal{R} \vdash l(t_1, \ldots, t_p) \longrightarrow r(u_1, \ldots, u_p)}$$

Derivation Rule R4 (Transitivity)

$$\frac{\mathcal{R} \vdash t \longrightarrow u \quad \mathcal{R} \vdash u \longrightarrow v}{\mathcal{R} \vdash t \longrightarrow v}$$

Derivation Rule R5 (Congruence)

$$\frac{\mathcal{R} \vdash t_1 \longrightarrow u_1 \quad \cdots \quad \mathcal{R} \vdash t_n \longrightarrow u_n}{\mathcal{R} \vdash f(t_1, \ldots, t_n) \longrightarrow f(u_1, \ldots, u_n)}$$

Logically, what distinguishes an equation from a rewrite rule is that equations are symmetric. While the equation `40 -- 40 = deuce` specifies that the terms `40 -- 40` and `deuce` belong to the same equivalence class and can be used interchangeably, the rewrite rule `N -- 0 => N -- 15` specifies an irreversible transition from one state to another. Operationally, Maude applies equations and rewrite rules in a left-to-right manner. The logical and the operational semantics are equivalent under certain conditions, notably that the equations in E are confluent and terminating.

3.5 Reflection and Metaprogramming

Rewriting logic supports reflection, in the sense that there exists a Maude program that can take an arbitrary Maude program as input and simulate it. Programs that take programs as input are called metaprograms. Metaprograms can themselves be processed by other programs, which can be seen as meta-metaprograms, which in turn are processed by meta-meta-metaprograms, and so on; this gives rise to a "reflective tower" [CDEL+07].

Maude supports metaprogramming through its `META-LEVEL` module, which is declared in `prelude.maude`. The module defines a syntax to represent a Maude module within Maude. In addition, it provides various functions that implement the fundamental Maude algorithms in terms of the metarepresentation of a module: pattern matching, term reduction, term rewriting, and breadth-first search. Finally, the `META-LEVEL` module provides hooks to move Maude terms and modules between reflection levels.

Metaprogramming has many uses:

- Some metaprograms analyze Maude specifications to find out if they have certain properties. Maude's Inductive Theorem Prover (ITP), Church–Rosser Checker (CRC), and Coherence Checker (ChC) fall into this category.
- Metaprogramming can be used to implement custom evaluation strategies, complementing the built-in `red`, `rew`, `frew`, and `search` commands. This approach has been used to monitor object-oriented systems [AJO04, JOT06] and to implement custom proof strategies for first-order logic [Hol05].
- Metaprogramming lets us construct Maude specifications at run-time and execute them on demand.

We will now briefly review metaprogramming, using the `TENNIS-GAME` module from Section 3.3 as our primary example. The metarepresentation of the module follows:

```
mod 'TENNIS-GAME is
  including 'NAT .
  sorts 'Game .
  none
  op '_--_ : 'Nat 'Nat -> 'Game [ctor] .
  op 'deuce : nil -> 'Game [ctor] .
  op 'advantage-server : nil -> 'Game [ctor] .
  op 'advantage-receiver : nil -> 'Game [ctor] .
  op 'game-server : nil -> 'Game [ctor] .
  op 'game-receiver : nil -> 'Game [ctor] .
  none
  eq '_--_['s_^40['0.Zero], 's_^40['0.Zero]] = 'deuce.Game [none] .
  rl '_--_['0.Zero, 'N:Nat] =>
     '_--_['s_^15['0.Zero], 'N:Nat] [none] .
       ⋮
  rl 'advantage-server.Game => 'deuce.Game [none] .
endm
```

The entire code above is a Maude term of sort `SModule` (system module). The `mod` ... `endm` operator is declared as follows:

```
op mod_is_sorts_.____endm :
   Header ImportList SortSet SubsortDeclSet OpDeclSet MembAxSet
   EquationSet RuleSet
   -> SModule [ctor gather (& & & & & & &) format (...)] .
```

A system module consists of a name, a list of import directives, a set of sorts, a set of subsort declarations, a set of operator declarations, a set of membership axioms, a set of equations, and a set of rewrite rules, in that order. Maude also provides an `fmod_is_sorts_.____endfm` constructor for functional modules.

To obtain the metarepresentation of a previously loaded module, we can use the `upModule` function. For example, the metarepresentation of the `TENNIS-GAME` module given above was obtained using the command

```
red upModule('TENNIS-GAME, false) .
```

The second argument indicates whether the imported modules should be metarepresented as well. Other useful conversion functions include `upTerm`, which returns the metarepresentation of a term, and `downTerm`, which returns a term from its metarepresentation. For example:

```
red upTerm(deuce) .                  *** result Term: 'deuce.Game
red downTerm('deuce.Game, 0 -- 0) . *** result Game: deuce
red downTerm('blah.Game, 0 -- 0) .  *** result Game: 0 -- 0
```

The second argument to `downTerm` specifies the expected sort (more precisely, the kind) of the result. If the first argument represents a term of that sort, that term is returned; otherwise, the second argument is returned.

Maude provides metalevel versions of its `red`, `rew`, `frew`, and `search` commands, called `metaReduce`, `metaRewrite`, `metaFrewrite`, and `metaSearch`. For example:

```
red metaRewrite(upModule('TENNIS-GAME, false), upTerm(0 -- 0), 1) .
*** result ResultPair: { '_--_['s_^15['0.Zero], '0.Zero], 'Game }
```

The `ResultPair` term holds the resulting metaterm and its sort. We can extract the metaterm from the `ResultPair` using `getTerm` and convert it to a plain `Game` term using `downTerm`. If we combine everything together, we get

```
red downTerm(getTerm(metaRewrite(upModule('TENNIS-GAME, false),
                                 upTerm(0 -- 0), 1)),
             0 -- 0) .
*** result Game: 15 -- 0
```

This example is contrived, because we could just as well have written

```
rew [1] 0 -- 0 .
```

rather than performing the rewrite step at the metalevel, but it clearly illustrates the relationship between programs and metaprograms.

3.6 Example: A WHILE Interpreter

To conclude, we will use Maude to write an interpreter for WHILE (Weak Hypothetical Imperative Language Example), a minimalistic imperative programming language that provides the following statements:

skip	null statement
$x := e$	assignment
if B **then** S_1 **else** S_2 **fi**	if statement
while B **do** S **od**	while loop
S_1; S_2	sequential composition

In the above, x is a variable name, e is an arithmetic or Boolean expression, B is a Boolean expression, and S, S_1, S_2 are statements. The empty statement list, written ϵ, is an identity element for sequential composition. The WHILE language is a deterministic subset of Creol as defined in Section 4.1.

The following WHILE program uses Euclid's algorithm to compute the greatest common divisor of two positive integers a and b, leaving the result in a:

while $b \neq 0$ **do**
  if $a > b$ **then**
    $a := a - b$
  else
    $b := b - a$
  fi
od

Although the semantics of WHILE programs is fairly obvious from the syntax, it is still worthwhile to spell it out to avoid ambiguities. Following the structural operational semantics approach [Plo04], we represent a program under execution by a pair $\langle S, \sigma \rangle$, where S is the program code to execute and σ is the current state. We can then define the semantics of the WHILE language using four rewrite rules.

Rewrite Rule W1 (Null Statement)

$$\langle \textbf{skip}; S, \sigma \rangle \longrightarrow \langle S, \sigma \rangle$$

If the next statement to execute is **skip**, we can simply eliminate it. The state σ is not affected by **skip**. The statement S represents the rest of the program. By making ϵ an identity element for sequential composition, **skip**; S will even match **skip**, with S bound to ϵ.

Rewrite Rule W2 (Assignment)

$$\langle x := e; S, \sigma \rangle \longrightarrow \langle S, \sigma[x \mapsto \{e\}_\sigma] \rangle$$

The assignment statement leads to a modification of the state. The new state is identical to the old state, except that the variable x should map to the value of e, evaluated in the original state σ. This is written $\sigma[x \mapsto \{e\}_\sigma]$.

Rewrite Rule W3 (If Statement)

$$\langle \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}; S, \sigma \rangle \longrightarrow \textbf{if } \{B\}_\sigma \textbf{ then } \langle S_1; S, \sigma \rangle \textbf{ else } \langle S_2; S, \sigma \rangle \textbf{ fi}$$

If the condition B evaluates to **true**, the **if** statement expands to its **then** branch; otherwise, it expands to its **else** branch. Notice that the first **if** construct in the rule is a WHILE language statement, whereas the second **if** is a conditional expression in rewriting logic.

Rewrite Rule W4 (While Loop)

$$\langle \textbf{while } B \textbf{ do } S \textbf{ od}; S', \sigma \rangle \longrightarrow \textbf{if } \{B\}_\sigma \textbf{ then } \langle S; \textbf{while } B \textbf{ do } S \textbf{ od}; S', \sigma \rangle \textbf{ else } \langle S', \sigma \rangle \textbf{ fi}$$

If the condition B evaluates to **true**, the **while** statement expands to its body followed by another instance of the loop; otherwise, the entire statement is eliminated.

To execute a WHILE program S from a state σ that gives a value to all the variables used in the program, we begin with the configuration $\langle S, \sigma \rangle$ and apply rewrite rules until we reach a configuration of the form $\langle \epsilon, \sigma' \rangle$. By inspecting Rewrite Rules W1–W4, we can prove that the final state σ' is unique if it exists.

We will now review the Maude specification of the WHILE interpreter, module by module. The first modules define data structures for storing WHILE programs under execution, as well as an auxiliary operator to evaluate expressions. Once we have defined these, the interpreter proper can be written in about 30 lines of code.

```
fmod WHILE-PRELUDE is
  including (INT + QID) * (op _xor_ : Nat Nat -> Nat to _xor2_) .
endfm
```

The `WHILE-PRELUDE` module imports Maude's predefined `INT` and `QID` modules and renames the integer version of the `xor` operator to avoid a clash with the Boolean version, which has a different precedence.[1]

```
fmod WHILE-VALUE is
  including WHILE-PRELUDE .

  sort Value .

  subsort Bool < Value .
  subsort Int < Value .
endfm
```

The `WHILE-VALUE` module defines the `Value` sort as a supersort of `Bool` and `Int`.

```
fmod WHILE-EXPRESSION is
  including WHILE-VALUE .

  sort BasicExp .
  subsort Qid < BasicExp .

  op [_] : BasicExp -> BasicExp [ctor] .

  sort AExp .
  subsort BasicExp < AExp .
  subsort Int < AExp .

  op plus_ : AExp -> AExp [ctor prec 3] .
  op minus_ : AExp -> AExp [ctor prec 3] .
```

[1] The clash occurs in the WHILE interpreter because `Bool` and `Int` belong to the same connected component in the subsort graph, which in turn is necessary because we want `Bool` and `Int` to be subsorts of `Value`. This quirk has been reported to the Maude developers.

```
  op _times_ : AExp AExp -> AExp [ctor prec 5 gather (E e)] .
  op _div_ : AExp AExp -> AExp [ctor prec 5 gather (E e)] .
  op _plus_ : AExp AExp -> AExp [ctor prec 7 gather (E e)] .
  op _minus_ : AExp AExp -> AExp [ctor prec 7 gather (E e)] .
  op [_] : AExp -> AExp [ctor] .

  sort BExp .
  subsort BasicExp < BExp .
  subsort Bool < BExp .

  op _eq_ : AExp AExp -> BExp [ctor prec 9] .
  op _ne_ : AExp AExp -> BExp [ctor prec 9] .
  op _lt_ : AExp AExp -> BExp [ctor prec 9] .
  op _gt_ : AExp AExp -> BExp [ctor prec 9] .
  op _le_ : AExp AExp -> BExp [ctor prec 9] .
  op _ge_ : AExp AExp -> BExp [ctor prec 9] .

  op !_ : BExp -> BExp [ctor prec 3] .
  op _&&_ : BExp BExp -> BExp [ctor assoc comm prec 11] .
  op _||_ : BExp BExp -> BExp [ctor assoc comm prec 13] .
  op [_] : BExp -> BExp [ctor] .

  sort Exp .
  subsort AExp < Exp .
  subsort BExp < Exp .
endfm
```

The `WHILE-EXPRESSION` module defines the sorts `AExp` (arithmetic expression), `BExp` (Boolean expression), and `Exp` (arithmetic or Boolean expression). Arithmetic expressions can be quoted identifiers, integer literals, or complex expressions such as `['x plus 'y] div 2`. Boolean expressions are defined similarly. The `Exp` sort is a supersort of `AExp` and `BExp`. For both types of expressions, we provide a constructor for bracketed expressions (`[_]`).

The operators are called `plus`, `minus`, etc., to avoid clashes with the predefined Maude operators: While `1 + 2` is a term of sort `Nat` that reduces to `3`, the term `1 plus 2` is irreducible and its sort is `AExp`. Similarly, parentheses occurring in WHILE expressions are represented by square brackets to distinguish them from Maude parentheses. The `gather` attributes specify left-associativity.

The module also defines a `BasicExp` sort, which provides the `[_]` operator and has `Qid` (quoted identifier) as a subsort. If we attempted to do without it, Maude would display the message

```
Warning: sort declarations for operator '[_'] failed preregularity
    check on 1 out of 13 sort tuples. First such tuple is (Qid).
```

When Maude sees the term `['x]`, it cannot determine whether it is an arithmetic or a Boolean expression. By factoring out what `AExp` and `BExp` have in common in a `BasicExp` sort, we ensure that Maude can assign a unique minimal sort to `['x]`.

```
fmod WHILE-STATE is
  including WHILE-VALUE .

  sort State .
```

```
  op emptyState : -> State [ctor] .
  op [_|->_] : Qid Value -> State [ctor] .
  op __ : State State -> State [ctor assoc prec 1 id: emptyState] .

  var SIGMA : State .
  vars V V' : Value .
  var X : Qid .

  eq [X |-> V] SIGMA [X |-> V'] = SIGMA [X |-> V'] .
endfm
```

The `State` sort encodes a mapping from `Qid`s to `Value`s. For example, `['x |-> 1] ['y |-> 2]['z |-> 3]` denotes the state in which $x = 1$, $y = 2$, and $z = 3$. If a variable occurs several times in the state, only the last occurrence is kept. This behavior is implemented by an equation.

```
fmod WHILE-STATEMENT is
  including WHILE-EXPRESSION .

  sort SingleStmt .

  sort Stmt .
  subsort SingleStmt < Stmt .

  op skip : -> SingleStmt [ctor] .
  op _:=_ : Qid Exp -> SingleStmt [ctor prec 23] .
  op if_th_el_fi : BExp Stmt Stmt -> SingleStmt [ctor] .
  op while_do_od : BExp Stmt -> SingleStmt [ctor] .

  op emptyStmt : -> Stmt [ctor] .
  op _;_ : Stmt Stmt -> Stmt [ctor assoc prec 25 id: emptyStmt] .
endfm
```

The `WHILE-STATEMENT` module defines the `SingleStmt` and `Stmt` sorts, with constructor terms that correspond to the WHILE language syntax. The correspondence between the abstract, mathematical syntax for WHILE statements and the concrete, Maude-compatible syntax is straightforward. The WHILE language's **then** and **else** keywords are written `th` and `el` to avoid conflicts with Maude's predefined `if_then_else_fi` operator.

```
fmod WHILE-PROCESS is
  including WHILE-STATE .
  including WHILE-STATEMENT .

  sort Process .

  op <_,_> : Stmt State -> Process [ctor] .
endfm
```

The `WHILE-PROCESS` module defines the `Process` sort, which stores the code and current state of a process.

```
fmod WHILE-EXPRESSION-EVALUATION is
  including WHILE-EXPRESSION .
  including WHILE-STATE .
```

```
  op {_}_ : Exp State -> Value [prec 3] .

  vars A A1 A2 : AExp .
  vars B B1 B2 : BExp .
  var E : Exp .
  var N : Int .
  var SIGMA : State .
  var V : Value .
  vars X X' : Qid .

  eq {X} SIGMA [X' |-> V] = if X == X' then V else {X} SIGMA fi .

  eq {[E]} SIGMA = {E} SIGMA .

  eq {N} SIGMA = N .
  eq {plus A} SIGMA = {A} SIGMA .
  eq {minus A} SIGMA = - {A} SIGMA .
  eq {A1 times A2} SIGMA = {A1} SIGMA * {A2} SIGMA .
  eq {A1 div A2} SIGMA = {A1} SIGMA quo {A2} SIGMA .
  eq {A1 plus A2} SIGMA = {A1} SIGMA + {A2} SIGMA .
  eq {A1 minus A2} SIGMA = {A1} SIGMA - {A2} SIGMA .

  eq {true} SIGMA = true .
  eq {false} SIGMA = false .
  eq {A1 eq A2} SIGMA = {A1} SIGMA == {A2} SIGMA .
  eq {A1 ne A2} SIGMA = {A1} SIGMA =/= {A2} SIGMA .
  eq {A1 lt A2} SIGMA = {A1} SIGMA < {A2} SIGMA .
  eq {A1 gt A2} SIGMA = {A1} SIGMA > {A2} SIGMA .
  eq {A1 le A2} SIGMA = {A1} SIGMA <= {A2} SIGMA .
  eq {A1 ge A2} SIGMA = {A1} SIGMA >= {A2} SIGMA .
  eq {! B} SIGMA = not {B} SIGMA .
  eq {B1 && B2} SIGMA = {B1} SIGMA and {B2} SIGMA .
  eq {B1 || B2} SIGMA = {B1} SIGMA or {B2} SIGMA .
endfm
```

The `WHILE-EXPRESSION-EVALUATION` module defines the `{_}_` operator, which returns the value of an expression in a given state. The operator is defined recursively on the syntax of expressions, and relies on Maude's predefined operators on `Int` and `Bool`. The `{_}_` operator is given a lower precedence than state concatenation (`__`) to ensure that `{E} SIGMA SIGMA'` is parsed as `{E} (SIGMA SIGMA')`.

```
mod WHILE-INTERPRETER is
  including WHILE-EXPRESSION-EVALUATION .
  including WHILE-PROCESS .

  var B : BExp .
  var E : Exp .
  vars S S' S1 S2 : Stmt .
  var SIGMA : State .
  var X : Qid .

  rl [null-statement] :
  < skip ; S, SIGMA >
   =>
  < S, SIGMA > .
```

```
  rl [assignment] :
  < X := E ; S, SIGMA >
   =>
  < S, SIGMA [X |-> {E} SIGMA] > .

  rl [if-statement] :
  < if B th S1 el S2 fi ; S, SIGMA >
   =>
  if {B} SIGMA then < S1 ; S, SIGMA >
              else < S2 ; S, SIGMA > fi .

  rl [while-loop] :
  < while B do S od ; S', SIGMA >
   =>
  if {B} SIGMA then < S ; while B do S od ; S', SIGMA >
              else < S', SIGMA > fi .
endm
```

Finally, the `WHILE-INTERPRETER` module builds upon the previous modules to specify Rewrite Rules W1–W4 in Maude.

To execute the greatest common divisor program shown earlier, we can use the `rew` command as follows, after having loaded the `WHILE-INTERPRETER` module:

```
rew < while 'b ne 0 do
         if 'a gt 'b th
            'a := 'a minus 'b
         el
            'b := 'b minus 'a
         fi
      od,
      ['a |-> 518]['b |-> 1155] > .
```

The result is `< emptyStmt, ['a |-> 7]['b |-> 0] >`, meaning that the greatest common divisor of 518 and 1155 is 7.

Maude's dual nature as a programming language and as a specification language is made apparent by the WHILE interpreter. On the one hand, the Maude code can be run reasonably efficiently to execute any WHILE program, which suggests that Maude can be used to solve real-world problems normally tackled using imperative, object-oriented, or functional programming languages. On the other hand, the Maude equations and rewrite rules can be read as a logical theory, yielding an elegant formulation of the WHILE language's semantics.

By specifying the WHILE interpreter in Maude rather than in a conventional programming language such as C or Java, we also benefit from Maude's extensible rewrite strategies, its search capabilities, and its built-in model checker. As an additional benefit, Maude's powerful parsing capabilities let us analyze the WHILE syntax without needing a parser generator.

> The 1980s will probably be remembered as the decade in which programmers took a gigantic step backwards by switching from secure Pascal-like languages to insecure C-like languages.
>
> — Per Brinch Hansen (1993)

Chapter 4

Syntax and Semantics of Creol

This chapter presents the syntax and semantics for the Creol dialect used in this thesis. The dialect broadly corresponds to the language understood by the standard Creol interpreter in use at the University of Oslo, with a few extensions to support the proof system of Chapter 5. The following Creol language features are not part of the dialect and so they are not covered here: class upgrades [Ofs05, YJO06], constrained method calls [JO05], and type parameters [JO04b, Fje05].

To simplify the exposition, we will consider only **bool** and **int** as built-in data types, and ignore **char**, **float**, **list**, **pair**, **set**, and **str**. In our setting, the additional data types don't present any particular challenges and can be handled in essentially the same way as **bool** and **int**.

Section 4.1 gives a brief overview of Creol's main features and illustrates them through a concrete example. Section 4.2 explains Creol's syntax and semantics in more detail. Section 4.3 presents a small-step operational semantics for Creol, and Section 4.4 gives an alternative semantics that lays the theoretical foundation for the proof system introduced in Chapter 5. Section 4.5 concludes by explaining how the two semantics are implemented in Maude.

4.1 Overview of the Language

Creol is an experimental object-oriented language that supports interfaces, class inheritance, and dynamic method binding. It is object-oriented in the sense that classes are the fundamental structuring unit and that all interaction between objects occurs through method calls. The name Creol is a pseudoacronym for "Concurrent Reflective Object-Oriented Language" and is usually pronounced "cray-OOL" or [kɾeˈuːl]. What sets Creol apart from popular object-oriented languages such as C++, Java, or C# 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.

Objects have unique identities and communicate using *asynchronous method calls*. When an object A calls a method m of an object B, it first sends an invocation message to B along with arguments. Method m executes on B's processor and sends a

reply to A once it has finished executing, with return values. Object A may continue executing while expecting B's reply. Like in other object-oriented languages, object identities (references) can be passed as parameters, and thanks to Creol's interface concept, method calls are type-safe [JOY06].

Johnsen and Owe [JO07] argue that asynchronous method calls offer a more structured object interaction model than shared variables and message passing, while avoiding the blocking associated with synchronous method calls (also called remote procedure calls or remote method invocation). For convenience, Creol supports synchronous method calls as syntactic sugar for a method invocation followed by a blocking wait for the reply.

Besides asynchronous method calls, the other main distinguishing feature of Creol is its reliance on *explicit processor release points*, which take the form of **await** statements. Since there is only one processor per object, at most one method m may execute at a given time for a given object; any other method invocations must wait until m finishes or explicitly releases the processor by using **await**. This "cooperative" approach to intra-object concurrency ensures that while a method is executing, no other processes are accessing the object's attributes (instance variables), leading to a programming and reasoning style reminiscent of monitors [BH70, Hoa74].

To illustrate how Creol programs are executed, we will study a Creol solution to the classic producer–consumer problem, in which a producer process writes to a shared buffer and a consumer process reads the data from the buffer as it is written. The example consists of two interfaces and four classes. Let us start with the interfaces:

```
interface WritableBuffer
begin
with any:
  op put(in x : int)
end
```

```
interface ReadableBuffer
begin
with any:
  op get(out y : int)
end
```

The *WritableBuffer* interface declares the signature of a *put* method that has an input parameter called x of type **int**. Similarly, the *ReadableBuffer* interface declares the signature of a *get* method that has an output parameter called y of type **int**. The **with any** clauses specify that any object may call these methods.

```
class Producer (buf : WritableBuffer)
begin
  op run is
    var i : int;
    i := 1;
    while true do
      buf.put(i);
      i := i + 1
    od
end
```

```
class Consumer (buf : ReadableBuffer)
begin
  op run is
    var j : int, sum : int;
    sum := 0;
    while true do
      buf.get(; j);
      sum := sum + j
    od
end
```

The *Producer* class takes an object that supports the *WritableBuffer* interface as a context parameter. When instantiating the class, we must pass an object that implements the *WritableBuffer* interface as argument. The class's *run* method repeat-

edly calls *put* on the *WritableBuffer* object with 1, 2, 3, and so on. The *run* method is automatically invoked when the class is instantiated.

The *Consumer* class mirrors *Producer*. Instances of *Consumer* get their data from an object that supports the *ReadableBuffer* interface, one integer at a time, and compute the sum of the data read from the buffer. The semicolon in *buf.get*(; *j*) indicates that *j* is an output argument; in many other languages, we would write *j* := *buf.get*().

```
class Buffer
    implements WritableBuffer, ReadableBuffer
begin
    var value : int, full : bool
with any:
    op put(in x : int) is
        await ¬full;
        value := x;
        full := true
    op get(out y : int) is
        await full;
        y := value;
        full := false
end
```

The *Buffer* class supports the *WritableBuffer* and *ReadableBuffer* interfaces and implements the *put* and *get* methods. It also declares two attributes, *value* and *full*. The buffer's role is to synchronize the producer and the consumer, ensuring that the consumer doesn't read data that the producer hasn't generated yet and that the producer doesn't overwrite data that the consumer hasn't read.

In this example, the buffer can store at most one data item at a time, which partly defeats its purpose. A more realistic (and more efficient) *Buffer* class would use a circular buffer internally [And00]. This could be implemented in *Buffer* without having to change *Producer* or *Consumer*.

```
class Main
begin
    op run is
        var buf : any, prod : any, cons : any;
        buf := new Buffer;
        prod := new Producer(buf);
        cons := new Consumer(buf)
end
```

To launch the program, we must instantiate the *Main* class. Creol provides an explicit syntax for achieving this:

```
bootstrap system Main
```

The *run* method, which is invoked automatically when *Main* is instantiated, starts by creating a *Buffer* instance. Then it creates one *Producer* and one *Consumer* instance, both initialized with a reference back to the buffer. At that point, the pro-

ducer's and the consumer's *run* methods are invoked, giving rise to two nonterminating processes that exchange data through the buffer object.

Producer–consumer synchronization works as follows: If the consumer calls *get* before the producer calls *put*, then *full* is false and the *get* method invocation is suspended on the **await** *full* statement. This enables *put* to execute. When *put* returns, *full* is now true and *get* can resume its execution. If the producer calls *put* twice in a row before the consumer calls *get*, the second *put* invocation is suspended on **await** $\neg$*full*.

One of Creol's main strengths is that it lets us seamlessly combine active and reactive (client and server) behavior in the same object [JOA03]. The *run* method initiates the active component of an object, whereas the other methods embody the object's reactive behavior. Consider the following version of the *Consumer* class:

```
class Consumer (buf : ReadableBuffer)
begin
  var sum : int
  op run is
    var j : int;
    sum := 0;
    while true do
      await buf.get(; j);
      sum := sum + j
    od
with any:
  op getSum(out s : int) is
    s := sum
end
```

In this version of the class, we declare a *getSum* method that returns the sum computed so far, and promote *sum* to an attribute. In addition, the synchronous method call *buf.get*$(; j)$ in *run* has been replaced by the statement **await** *buf.get*$(; j)$, which invokes *get* asynchronously, releases the processor while it waits for a reply, and retrieves the return value. Because **await** releases the processor, the object can service incoming calls to *getSum* while waiting for *buf*'s reply.

While not using all the features found in Creol, the producer–consumer example demonstrates most of Creol's key concepts. Johnsen et al. [JO02, JOA03, DJO05, JOY06, JO07] provide more elaborate examples, including a Creol model of a peer-to-peer network [JO07]. We will also consider many other examples later in this thesis, notably in Chapter 7.

4.2 The Language's Syntax

Interface and class declarations constitute the core of Creol's syntax; in fact, a Creol program is simply a list of interface and class declarations, followed by a **bootstrap** command. Before we look at their syntax, we will introduce a few basic syntactic entities: identifiers, types, typed identifiers, and assertions.

The set *Id* of *identifiers* consists of case-sensitive alphanumeric tokens that start with a letter. Thus, *x*, *hungryCat*, and *R2D2* are valid identifiers, whereas *3DCube* and *Muad'dib* aren't. Identifiers are used to name interfaces, classes, methods, attributes, parameters, local variables, and method call labels. Keywords are not allowed as identifiers.

The set *Type* of types comprises the built-in data types **bool** and **int**. In addition, the name of any interface can be used as a type. For simplicity, we will allow any identifier to act as a type (regardless of whether it is declared or not) by letting $\mathit{Type} \triangleq \mathit{Id} \cup \{\mathbf{bool}, \mathbf{int}\}$. A *typed identifier*, belonging to the set *TypedId*, has the syntax $x : \tau$, where x is an identifier and τ is a type.

Creol programmers can specify first-order logic *assertions* in the program code. These assertions may represent assumptions or guarantees associated with a class or interface, loop invariants, or conditions that must hold at certain points during the program's execution. Their precise syntax is given in Section A.3. The set of assertions is denoted *Assn*.

In many places in the Creol syntax, programmers can supply a comma-separated list of items. For example, the assignment operator can be applied to several variables simultaneously, as follows: $a, b, c := 1, 2, 3$. To represent a comma-separated list of elements from a set X (*Id*, *TypedId*, etc.), we will use an element $\bar{x} \equiv x_1, \ldots, x_n$ (where '$\equiv$' denotes syntactic equality) of the set

$$\mathcal{L}(X) \triangleq \bigcup_{k=1}^{\infty} \{a_1, \ldots, a_k \mid a_i \in X\}.$$

We are now ready to tackle interface and class declarations. An *interface declaration* has the syntax

interface i $\big[(\bar{\mathrm{x}})\big]$
  $\big[$**inherits** $\bar{\jmath}\big]$
begin
with k:
  op μ_1
    $\vdots$
  op μ_n
  $\big[$**asum** $\varphi\big]$
  $\big[$**guar** $\psi\big]$
end

where $i \in \mathit{Id}$ is the name of the interface, $\bar{\mathrm{x}} \in \mathcal{L}(\mathit{TypedId})$ is a list of context parameters, $\bar{\jmath} \in \mathcal{L}(\mathit{Super})$ is a list of inherited interfaces, $\mu_1, \ldots, \mu_n \in \mathit{Sig}$ are method signatures, $k \in \mathit{Id} \cup \{\mathbf{any}\}$ is an interface associated with $\mu_1, \ldots, \mu_n$, $\varphi \in \mathit{Assn}$ is an assumption about the environment, and $\psi \in \mathit{Assn}$ is a guarantee offered by the interface. Square brackets denote optional clauses.

The set *Super* of *supertypes* (superinterfaces or superclasses) consists of elements with the syntax $\tau\big[(\bar{e})\big]$, where τ is the name of a type (interface or class) and $\bar{e}$ is a list of arguments corresponding to τ's context parameters. The arguments can refer to the parameters $\bar{\mathrm{x}}$.

The *cointerface* k of a method is an interface that the calling object is required to implement; using the implicit **caller** parameter, the method can then call back the calling object through interface k. If $k \equiv$ **any**, no cointerface is specified and the methods $\mu_1, \ldots, \mu_n$ can be called from any object.

The **asum** and **guar** clauses, together with the context parameters $\bar{x}$, let the programmer specify an assume–guarantee specification that is inherited by all classes that support the interface. The assumption is a condition upon which classes that implement the interface can rely. The guarantee is a commitment that the interface makes as long as the assumption holds on behalf of the classes that implement it. We will see examples of this later in this chapter.

A *method signature*, belonging to the set *Sig*, has the syntax

$$m(\left[\textbf{in } \bar{x}\right] \left[\textbf{out } \bar{y}\right])$$

with $m \in \mathit{Id}$ and $\bar{x}, \bar{y} \in \mathcal{L}(\mathit{TypedId})$. The variables $\bar{x}$ are input parameters. These are passed by value and are read-only inside the method. Methods don't return a value directly; instead, they may assign values to one or several output parameters declared using the **out** keyword. If the method has neither input nor output parameters, we can drop the parentheses around the empty parameter list.

This completes our review of the syntax of interface declarations. We now proceed with *class declarations*, which have the syntax

class c $\left[(\bar{x})\right]$
  $\left[\textbf{implements } \bar{\imath}\right]$
  $\left[\textbf{contracts } \bar{\jmath}\right]$
  $\left[\textbf{inherits } \bar{q}\right]$
begin
  $\left[\textbf{var } \bar{w}\right]$
  $\left[G_0\right]$
with k_1:
  G_1
  $\vdots$
with k_m:
  G_m
  $\left[\textbf{asum } \varphi\right]$
  $\left[\textbf{guar } \psi\right]$
end

where $c \in \mathit{Id}$ is the class name, $\bar{x} \in \mathcal{L}(\mathit{TypedId})$ is a list of context parameters, $\bar{\imath}, \bar{\jmath} \in \mathcal{L}(\mathit{Super})$ are superinterfaces, $\bar{q} \in \mathcal{L}(\mathit{Super})$ are superclasses, $\bar{w} \in \mathcal{L}(\mathit{TypedId})$ is a list of attributes, $G_0, \ldots, G_m \in \mathit{MtdDeclGroup}$ are groups of methods, $k_1, \ldots, k_m$ are cointerfaces associated with $G_1, \ldots G_m$, and $\langle \varphi, \psi \rangle \in \mathit{Assn} \times \mathit{Assn}$ is an assume–guarantee specification.

A class may declare context parameters $\bar{x}$ and attributes $\bar{w}$. The context parameters must be provided at object creation and behave like read-only attributes. An object's parameters and attributes are not accessible to other objects.

Unlike interfaces, classes provide bodies for the methods they declare. The set *MtdDeclGroup* of *method declaration groups* contains elements with the syntax

op μ_1 **is** β_1
$\vdots$
op μ_n **is** β_n

where $\mu_1, \ldots, \mu_n \in Sig$ are method signatures and $\beta_1, \ldots, \beta_n \in MtdBody$ are method bodies. Methods declared before the first **with** clause can be called only by the object itself.

A class c can be declared with one or several superclasses, from which it inherits all the non-internal methods that it doesn't override. It may also have superinterfaces, specified using the **implements** clause, in which case it must provide implementations for all the methods declared by the interfaces and must respect the interfaces' assume–guarantee specifications.

In Creol, inheritance and subtyping don't coincide: Classes that inherit from class c do not inherit its assume–guarantee specification or that of its superinterfaces $\bar{\imath}$, and are not even considered to support $\bar{\imath}$. This stands in sharp contrast with the concept of inheritance as found in Java or C#. If we want to force a certain interface (and its assume–guarantee specification) upon a class's direct and indirect subclasses, we must use the **contracts** clause instead of **implements**.

The set *MtdBody* of *method bodies* consists of elements with the syntax

$[\mathbf{var}\ \bar{\mathrm{v}};]$
S

where $\bar{\mathrm{v}} \in \mathcal{L}(TypedId)$ is the list of local variables and $S \in Stmt$ is a (simple or compound) statement.

To complete the syntactic definition of class declarations, we must explain what a Creol statement looks like. We will start by defining expressions, which play a major role in the definition of Creol statements; then we will look at statements.

The set *AExp* of *arithmetic expressions* includes the set *Id* of identifiers and contains integer literals of arbitrary size expressed as decimal numbers, such as 0, 1234, and -11235813213455. In addition, if $x \in Id$ is an attribute and $c \in Id$ the name of a superclass, the *qualified identifier* $x@c \in QualifiedId$ denotes the attribute x declared by c, to distinguish it from attributes with the same name inherited from other superclasses. Moreover, for any arithmetic expressions $A, A_1, A_2 \in AExp$ and Boolean expression $B \in BExp$, the following expressions also belong to *AExp*:

$+A$	unary plus
$-A$	unary minus
if B **then** A_1 **else** A_2 **fi**	conditional expression
$A_1 * A_2$	multiplication
A_1 / A_2	integer division (rounding toward zero)
$A_1 + A_2$	addition
$A_1 - A_2$	subtraction
(A)	parenthesized expression

This definition of arithmetic expressions leads to syntactic ambiguities. To resolve these, we follow the standard conventions: The unary operators bind more strongly than the other operators; the multiplicative binary operators ($*$, $/$) bind more strongly than the additive operators ($+$, $-$); and binary operators with the same binding power associate to the left. As a consequence, $x + -y * z$ is read as $x + ((-y) * z)$, and $-a + b - c + d$ is read as $(((-a) + b) - c) + d$.

If τ is the name of an interface, any variable of type τ or of one of its subtypes can be used as an *object expression of type* τ. The set of such variables is denoted $OExp_\tau$. To this set, we add **null** (the null object reference), as well as **self** (a reference to the current object) and **caller** (a reference to the object that invoked the current method), if they support τ. Finally, for any object expressions $O, O_1, O_2 \in OExp_\tau$ and Boolean expression $B \in BExp$, the following expressions also belong to $OExp_\tau$:

if B **then** O_1 **else** O_2 **fi**	conditional expression
(O)	parenthesized expression

For the set $BExp$ of *Boolean expressions*, we consider all identifiers as base elements, as well as the constants **true** and **false**. In addition, for any $A_1, A_2 \in AExp$ and $R \in \{=, \neq, <, >, \leq, \geq\}$, the set $BExp$ contains the expression

$A_1\ R\ A_2$	arithmetic comparison

Similarly, for any $O_1, O_2 \in OExp_\tau$ and $R \in \{=, \neq\}$, the set $BExp$ contains

$O_1\ R\ O_2$	object comparison

Let $B, B_1, B_2 \in BExp$. Then the following expressions also belong to $BExp$:

$\neg B$	logical negation
if B **then** B_1 **else** B_2 **fi**	conditional expression
$B_1 \wedge B_2$	logical conjunction
$B_1 \vee B_2$	logical disjunction
(B)	parenthesized expression

The operator $\neg$ binds more strongly than $\wedge$, which in turn binds more strongly than $\vee$. The $\wedge$ and $\vee$ operators are associative.

In addition to the built-in data types, Creol lets us define custom data types and functions using a formalism similar to Maude's equational sublanguage. Data types are defined inductively, and operations are defined as functions f that take a list of arguments $\bar{e}$ and return a value of a given type. The arguments and return value can be of any type. Let f be a function (or constructor) with n parameters of types $\tau_1, \ldots, \tau_n$ that returns a value of type τ. Then the function application $f(e_1, \ldots, e_n)$, where each expression e_i is of type τ_i, is an expression of type τ.

Example 4.1. A *point2D* data type that stores a pair of integer coordinates would be defined as follows:

functional
begin
  type *point2D* [**default-value**: $point2D(0, 0)$]
  ctor *point2D* : **int** $\times$ **int** $\rightarrow$ *point2D*
  fn *getX* : *point2D* $\rightarrow$ **int**

fn $getY : point2D \to \mathbf{int}$
fn $add : point2D \times point2D \to point2D$
var $x : \mathbf{int}$, $x' : \mathbf{int}$, $y : \mathbf{int}$, $y' : \mathbf{int}$
eq $getX(point2D(x, y)) \triangleq x$
eq $getY(point2D(x, y)) \triangleq y$
eq $add(point2D(x, y), point2D(x', y')) \triangleq point2D(x + x', y + y')$
end

These definitions must be provided to the Creol interpreter in addition to the Creol program to execute. In the program, we could use the new data type as follows:

var $p1 : point2D$, $p2 : point2D$, $width : \mathbf{int}$, $height : \mathbf{int}$
$p1 := point2D(320, 200)$;
$p2 := point2D(1024, 768)$;
$width := getX(p2) - getX(p1)$;
$height := getY(p2) - getY(p1)$ □

As they are defined here, Creol expressions may not contain method calls, assignments, or other constructs with side effects. This design greatly simplifies the formulation of the semantics and of the proof system, and enables us to define the assertion language *Assn* as a superset of the Boolean expression language *BExp*. Using temporary variables, it is always possible to reformulate a program that contains expressions with side effects as an equivalent program that doesn't [Dah92].

In some places in the syntax, an expression of any kind (arithmetic, Boolean, or object) is expected. We will denote the set of all expressions by *Exp*.

We will now define Creol's *statements*, belonging to the set *Stmt*. Creol offers the following trivial statements, where $\varphi \in Assn$:

skip	null statement
abort	abnormal termination
prove φ	inline assertion

The **skip** statement is a "do nothing" statement. The **abort** statement causes the object to terminate all its activity immediately, reporting an error to the user. To quote Ole-Johan Dahl [Dah92]:

> The [**abort**] statement is useful when programming consistency and capacity checks. When developing larger programs we may use **abort** statements to represent unfinished program segments. Thereby the program may remain partially correct throughout the development phase, producing valid results whenever it terminates normally.

The **prove** statement allows the programmer to specify a proof outline as part of the program, which can be formally verified using the assertion analyzer presented in Chapter 6. When the program is executed, **prove** statements are ignored.

Let $\bar{z} \equiv z_1, \ldots, z_n \in \mathcal{L}(QualifiedId)$ be a list of variables and $\bar{e} \equiv e_1, \ldots, e_n \in \mathcal{L}(Exp)$ be a list of expressions such that each e_i is type-compatible with z_i. The assignment statement has the following syntax:

$\bar{z} := \bar{e}$	assignment

If $n > 1$, the assignments to $z_1, \ldots, z_n$ are performed simultaneously. For example, the statement $a, b := b, a$ (unlike $a := b$; $b := a$) swaps the values of a and b. If the same variable is assigned to several times, only the last assignment is applied; thus, $a, a := 1, 2$ sets a to 2.

Since the right-hand side of the := operator is a list of expressions and expressions cannot have side effects, we need an additional syntax rule for assigning a newly created object to a variable. Let $z \in \mathit{QualifiedId}$ be the name of a variable, $c \in \mathit{Id}$ be the name of a class, and $\bar{e} \equiv e_1, \ldots, e_n \in \mathcal{L}(\mathit{Exp})$ be a list of expressions. New instances of class c can be created as follows:

$z := \textbf{new}\ c\big[(\bar{e})\big]$	object creation

The expressions $e_1, \ldots, e_n$ are assigned to the context parameters specified in the class declaration. If the class has a parameterless method called *init*, this method is called immediately. If class c has superclasses, the superclasses' *init* methods are called recursively, before c's version of *init* is run. Moreoever, if a parameterless *run* method is declared or inherited by the class, it is invoked immediately after.

Creol offers two basic flavors of method calls, synchronous (blocking) and asynchronous (non-blocking), with the following syntax:

$\big[l\big]!\big[O.\big]m(\big[\bar{e}\big])$	asynchronous invocation
$l?(\big[\bar{y}\big])$	asynchronous reply
$\big[O.\big]m(\big[\bar{e}\big]\big[;\bar{z}\big])$	synchronous call

Here, $l \in \mathit{Id}$ is a label variable identifying the asynchronous call, $O \in \mathit{OExp}_\tau$ is an object expression of type τ, $m \in \mathit{Id}$ is the name of a method supported by the τ type, $\bar{e} \in \mathcal{L}(\mathit{Exp})$ is a list of input arguments, and $\bar{z} \in \mathcal{L}(\mathit{QualifiedId})$ is a list of output arguments. The input and output arguments must match the input and output parameters declared in the method's signature. In addition, the calling object must implement the method's cointerface if one was specified.

Asynchronous method calls consist of an invocation and a reply. The invocation can be seen as a message from the caller to the called method, with arguments corresponding to the method's input parameters. The reply is a message from the called method, containing the return values for the call. The label l makes it possible to refer to a specific asynchronous call.

Example 4.2. The following code initiates two asynchronous calls, releases the processor while waiting for the replies, and retrieves the return values:

```
var res1 : int, res2 : int;
l1!server.request();
l2!server.request();
await l1? & l2?;
l1?(res1);
l2?(res2)
```
□

For synchronous calls, we distinguish between local and remote calls. Local synchronous calls correspond to the case where O is omitted or evaluates to **self** and are executed immediately, as they would be in Pascal or Java. In contrast, a remote synchronous call $O.m(\bar{e}; \bar{z})$ is implemented as an asynchronous method invocation

$\nu!O.m(\bar{e})$ followed by the reply statement $\nu?(\bar{z})$, where ν is a special identifier that may not occur in the program text. Because there is no **await** statement between the method invocation and the reply statement, the calling object is blocked while the remote method executes. When the reply statement $\nu?(\bar{z})$ terminates, the values assigned to the output parameters are available in $\bar{z}$.

When overriding a method in a subclass, it is often necessary to call the original implementation of the method as well. Creol lets us do that using qualified method calls, which have the syntax

$[l]!m@c([\bar{e}])$	qualified asynchronous invocation
$m@c([\bar{e}][;\bar{z}])$	qualified synchronous call

where $c \in Id$ is the name of a superclass.

We have already seen instances of the **await** statement, which releases the processor conditionally. In general, **await** statements are of the forms

await g	conditional wait
await $[g\ \&]\ l?([\bar{z}])$	conditional wait for reply
await $[g\ \&]\ O.m([\bar{e}][;\bar{z}])$	conditional wait during call
await $[g\ \&]\ m[@c]([\bar{e}][;\bar{z}])$	conditional wait during local call

with $g \in Guard$, $l, m, c \in Id$, $\bar{z} \in \mathcal{L}(QualifiedId)$, $O \in OExp_\tau$, and $\bar{e} \in \mathcal{L}(Exp)$. The statement **await** g releases the processor if the conditional guard g evaluates to **false** and reacquires it at some later time when g is true. The second, third, and fourth versions of the **await** statement are abbreviations:

$$\begin{aligned}
&\textbf{await}\ g\ \&\ l?(\bar{z}) &&\equiv\ \textbf{await}\ g\ \&\ l?;\ l?(\bar{z})\\
&\textbf{await}\ g\ \&\ O.m(\bar{e};\bar{z}) &&\equiv\ \nu!O.m(\bar{e});\ \textbf{await}\ g\ \&\ \nu?;\ \nu?(\bar{z})\\
&\textbf{await}\ g\ \&\ m@c(\bar{e};\bar{z}) &&\equiv\ \nu!m@c(\bar{e});\ \textbf{await}\ g\ \&\ \nu?;\ \nu?(\bar{z}).
\end{aligned}$$

In the above, ν is a fresh label. The set *Guard* of *conditional guards* is constructed as follows:

B	Boolean guard
$l?$	reply guard
wait	unconditional processor release
$g_1\ \&\ g_2$	complex guard

where $B \in BExp$, $l \in Id$, and $g_1, g_2 \in Guard$. The guard $l?$ evaluates to **true** if and only if a reply for the asynchronous call identified by l has arrived. The **wait** guard evaluates to **false** the first time it is encountered, resulting in a processor release, and evaluates to **true** from then on. The complex guard $g_1\ \&\ g_2$ evaluates to **true** if and only if both g_1 and g_2 evaluate to **true**.

The basic statements seen so far can serve as building blocks for compound statements. The first of these is the **if** statement:

if B **then**
  S_1
[**else**
  S_2]
fi

If the condition $B \in BExp$ evaluates to **true**, the statement S_1 is executed; otherwise, S_2 is executed. Omitting the **else** clause is equivalent to letting S_2 be **skip**.

A second type of compound statement is the **while** loop:

$$
\begin{array}{l}
[\textbf{inv}\ \varphi] \\
\textbf{while}\ B\ \textbf{do} \\
\quad S \\
\textbf{od}
\end{array}
$$

If the condition $B \in BExp$ evaluates to **true**, the loop body $S \in Stmt$ is executed, then the condition is reevaluated. If the condition is still true, the body is executed a second time, and so on, only stopping if the condition ever turns false.

The optional **inv** clause lets the programmer specify a *loop invariant* $\varphi \in Assn$ that must hold before entering the loop and after each iteration. The **inv** clause is used by the assertion analyzer presented in Chapter 6.

Loops are currently being phased out of Creol because they can always be replaced by recursion. Nonetheless, it seems appropriate to include **while** loops here for the following reasons:

1. Loops are a well understood concept that is easy to define and reason about and that integrates unproblematically with the rest of the language.
2. Many programs, including the producer–consumer example earlier in this section, can be expressed more simply using **while** than with recursion. (This explains why fairly recent papers [JOY06, JO07] use loops in their examples.)
3. With its invariant, the **while** statement will serve as a stepping stone for axiomatization of more advanced Creol constructs in Chapter 5.

In Chapter 7, we will compare loops and recursion for program verification.

The remaining compound statements resemble expression syntax in that they combine simpler statements together using operators, and parentheses can be used to force a precedence order:

$S_1; S_2$	sequential composition
$S_1 \,\square\, S_2$	nondeterministic choice
$S_1 \,\|\|\|\, \cdots \,\|\|\|\, S_n$	nondeterministic merge
(S)	parenthesized statement

with $S, S_i \in Stmt$. The ; operator binds more strongly than $\square$, and $\square$ binds more strongly than $|||$. The ; and $\square$ operators are associative, allowing us to write things like $A; B; C$ and $A \,\square\, B \,\square\, C$. Furthermore, $\square$ and $|||$ are commutative.

The sequential composition of two or more statements yields a statement. Later in this chapter, to denote any statement except sequential composition, we will use an element s from the set $SingleStmt \triangleq Stmt \backslash \{S_1; S_2 \mid S_i \in Stmt\}$.

We need the following two definitions to describe $\square$ and $|||$: A statement S is *enabled* if it can execute in the current state without releasing the processor immediately. For example, **await** B is enabled if B evaluates to **true**; otherwise it is disabled. If S is enabled and doesn't block immediately, S is said to be *ready*. The statement

$l?(\bar{z})$ is always enabled, but it is ready only if the asynchronous reply associated with l has arrived.

The nondeterministic choice statement $S_1 \,\square\, S_2$ executes either S_1 or S_2. If both branches S_i are ready, $S_1 \,\square\, S_2$ randomly chooses either S_1 or S_2. If only one branch S_i is ready, that branch is executed. If neither branch is ready, $S_1 \,\square\, S_2$ blocks if either S_1 or S_2 is enabled and releases the processor otherwise.

The nondeterministic merge statement $S_1 \,|||\, \cdots \,|||\, S_n$ (also written $|||_{i=1}^{n} S_i$) is similar to $S_1 \,\square\, S_2$ but it always executes all branches S_i. The precise order in which the constituents of $S_1, \ldots, S_n$ are executed is nondeterministic. Like with $\square$, a random branch S_i is chosen to start and is allowed to run until a disabled statement occurs or the branch finishes. If one or more other branches are ready at that point, one of these is randomly chosen to run; otherwise, the merge statement either blocks (if at least one branch is enabled) or releases the processor and tries to continue executing one of the branches later on. This goes on until all branches have terminated.

There is a subtle pitfall with the definition of $|||$. Unlike $\square$, it is not associative; in general, $(S_1 \,|||\, S_2) \,|||\, S_3$ and $S_1 \,|||\, (S_2 \,|||\, S_3)$ mean two different things, and $S_1 \,|||\, S_2 \,|||\, S_3$ also means something else. This will become clearer in the next section.

Example 4.3. The nondeterministic statements are typically used in conjunction with asynchronous method calls. Consider the following method bodies:

var *res* : **int**;	**var** *res1* : **int**, *res2* : **int**;
l1!*server1.request*();	*l1*!*server1.request*();
l2!*server2.request*();	*l2*!*server2.request*();
(*l1*?(*res*) $\square$ *l2*?(*res*));	(*l1*?(*res1*); *processResult1*(*res*1)
processResult(*res*)	$\,|||$ *l2*?(*res2*); *processResult2*(*res*2))

In both code snippets, we initiate two asynchronous method calls: one to *server1* and the other to *server2*. In the code on the left-hand side, we use the nondeterministic choice statement to handle the first reply that arrives, and we ignore the other reply. In the code on the right-hand side, we pick up both replies and process them as they arrive using nondeterministic merge. □

We have now reviewed the syntax of interface and class declarations. In addition to interfaces and classes, a Creol program must also specify an entry point. Let $c \in Id$ be the name of a class, and $\bar{e} \equiv e_1, \ldots, e_n \in \mathcal{L}(Exp)$ be a list of expressions. The following command launches the program by instantiating c:

bootstrap system $c[(\bar{e})]$

Like with the object creation statement, the expressions $e_1, \ldots, e_n$ are assigned to the context parameters specified in the class declaration, and the class's *init* and *run* methods are invoked.

4.3 An Operational Semantics in Rewriting Logic

In this section, we formally define the operational semantics of Creol programs, thereby providing a solid foundation for the rest of the thesis. The operational semantics is expressed using the rewriting logic formalism presented in Chapter 3.

Through its implicit support for concurrency, rewriting logic captures the nondeterministic, concurrent nature of distributed systems in a natural way [JO07, Ölv07]. Appendix B gives a complete Maude listing for a Creol interpreter based on the equations and rewrite rules given here.

The operational semantics of Creol was originally sketched by Einar Broch Johnsen [Joh02]. These ideas were refined by Marte Arnestad [Arn03], who implemented the first version of the Creol interpreter in Maude. Our presentation is primarily based on Johnsen and Owe [JO07].

When a Creol program is executed, it gives rise to an evolving data structure that represents the program's state, called a *system configuration*. A system configuration, belonging to the set *Config*, is a multiset that consists of objects, classes, and messages. In the style of Full Maude [CDEL+07], the terms that belong to the multiset are directly adjoined, with no operator in between. Following a suggestion by Ölveczky [Ölv07], the entire system configuration is enclosed in a pair of braces. These braces allow us to match the entire configuration in rewrite rules, as we will see shortly.

In a system configuration, Creol objects are represented by terms of the form

$$\langle o : c \mid \text{Pr: } S,\ \text{LVar: } \beta,\ \text{Att: } \alpha,\ \text{PrQ: } P,\ \text{MsgQ: } Q,\ \text{LabCnt: } k \rangle,$$

where $o \in \mathit{OId}$ is a unique identity for the object, $c \in \mathit{Id}$ is the object's class, $S \in \mathit{Stmt}$ is the active process's code, $\beta \in \mathit{State}$ is the active process's local variables, $\alpha \in \mathit{State}$ is the current state of the object's attributes, $P \in \mathcal{M}(\mathit{State} \times \mathit{Stmt})$ is a queue of suspended processes, $Q \in \mathcal{M}(\mathit{MsgBody})$ is the incoming message queue, and $k \in \mathbb{N}$ is a counter used to number outgoing invocation messages.

For a given set X, $\mathcal{M}(X)$ denotes the set of all multisets over X (the power multiset of X). We will also write $\bar{x}$ to denote a comma-separated list of elements of a set X. Unlike in the previous section, but in keeping with the algebraic definition of a list, $\bar{x}$ may be the empty list (denoted ϵ).

The set *State* of *variable states* consists of mappings from variables to values. For example, $[x \mapsto 1][y \mapsto 2][z \mapsto 3]$ denotes the state in which $x = 1$, $y = 2$, and $z = 3$. We let $\emptyset$ denote the empty state. The concatenation $\alpha\beta$ of two states α and β gives precedence to β for variables defined by both. Thus, if σ is a state, $\sigma[x \mapsto v]$ denotes the state that is identical to σ except that x maps to v. The $\{\bar{e}\}_\sigma$ function returns the value of an expression list $\bar{e}$ in a state σ; it will be the object of Definitions T12–T17.

The set *Value* of *values* consists of the Boolean constants **true** and **false**, integer literals, and object identities. Expressions such as $1 + 2$ and $x > y$ are not values.

Example 4.4. The term

$$\langle \mathit{Main}\#0 : \mathit{Main} \mid \text{Pr: } \mathit{cons} := \textbf{new } \mathit{Consumer}(\mathit{buf});\ \textbf{return } \epsilon;\ \textbf{continue } 0,\ \text{LVar: } \beta,\ \text{Att: } \alpha,\ \text{PrQ: } \langle \nu?(), [\nu \mapsto 0] \rangle,\ \text{MsgQ: } \emptyset,\ \text{LabCnt: } 1 \rangle$$

with

$$\alpha \equiv [\textbf{self} \mapsto \mathit{Main}\#0]$$
$$\beta \equiv [\textbf{caller} \mapsto \mathit{Main}\#0][\textbf{label} \mapsto 0][\mathit{buf} \mapsto \mathit{Buffer}\#0][\mathit{prod} \mapsto \mathit{Producer}\#0][\mathit{cons} \mapsto \textbf{null}]$$

represents the *Main* instance from the producer–consumer program reviewed at the beginning of this chapter after it instantiated *Buffer* and *Producer*. □

In the rest of this section and in the next section, we will use the following notations to refer to elements of particular sets (resorting to subscripts and primes when necessary):

$c, i, l, m \in \mathit{Id}$	$e \in \mathit{Exp}$	$\varsigma \in \mathit{Super}$
$z \in \mathit{QualifiedId}$	$A \in \mathit{AExp}$	$\tau \in \mathit{Type}$
$\mathrm{W}, \mathrm{X}, \mathrm{Y} \in \mathit{TypedId}$	$B \in \mathit{BExp}$	$\Gamma \in \mathit{Config}$
$\mathrm{Z} \in \mathit{TypedQualifiedId}$	$O \in \mathit{OExp}$	$G \in \mathit{MtdDeclGroup}$
$k, n \in \mathbb{N}$	$\varphi, \psi \in \mathit{Assn}$	$M \in \mathcal{M}(\mathit{Mtd})$
$o \in \mathit{OId}$	$g \in \mathit{Guard}$	$P \in \mathcal{M}(\mathit{State} \times \mathit{Stmt})$
$v, w \in \mathit{Value}$	$S \in \mathit{Stmt}$	$Q \in \mathcal{M}(\mathit{MsgBody})$
$\alpha, \beta, \sigma \in \mathit{State}$	$s \in \mathit{SingleStmt}$	

Creol classes are represented by terms of the form

$$\langle c : \text{Class} \mid \text{Inh: } \bar{\varsigma},\ \text{Param: } \bar{\mathrm{X}},\ \text{Att: } \bar{\mathrm{W}},\ \text{Mtd: } M,\ \text{ObjCnt: } n \rangle,$$

where c is the class name, $\bar{\varsigma}$ is the list of base classes, $\bar{\mathrm{X}}$ is the list of context parameters, $\bar{\mathrm{W}}$ is the list of attributes, M is a multiset of methods, and n is a counter used to generate unique object identities. The following term is an example of a class:

$$\langle \mathit{Producer} : \text{Class} \mid \text{Inh: } \epsilon,\ \text{Param: } \mathit{buf} : \mathit{WritableBuffer},\ \text{Att: } \epsilon,\ \text{Mtd: } \{\ldots\},\ \text{ObjCnt: } 1 \rangle.$$

The set *Mtd* consists of terms of the form

$$\langle m : \text{Method} \mid \text{In: } \bar{\mathrm{X}},\ \text{Out: } \bar{\mathrm{Y}},\ \text{LVar: } \bar{\mathrm{W}},\ \text{Code: } S \rangle,$$

where m is the method's name, $\bar{\mathrm{X}}$ is a list of input parameter names, $\bar{\mathrm{Y}}$ is a list of output parameter names, $\bar{\mathrm{W}}$ is a list of local variables, and S is the method's body. The following term is a method:

$$\langle \mathit{get} : \text{Method} \mid \text{In: } \epsilon,\ \text{Out: } y : \mathbf{int},\ \text{LVar: } \epsilon,\ \text{Code: } \mathbf{await}\ \mathit{full};\ y := \mathit{value};\ \mathit{full} := \mathbf{false} \rangle.$$

Creol objects interact by exchanging messages, which are stored in the system configuration. *Invocation messages* have the form

$$\mathit{Invoke}(o, k, m[@c], \bar{v}),$$

where o is the calling object, k is the sequence number associated with the method call, m (or $m@c$) is the called method, and $\bar{v}$ is a list of input arguments to m. *Reply messages* have the form

$$\mathit{Reply}(k, \bar{w}),$$

where k is the sequence number for the method call and $\bar{w}$ is a list of return values. Invocation and reply messages belong to the set *MsgBody*. When messages are passed around, the receiver object o' is specified by appending **to** o' to the message.

The operational semantics for Creol consists of 24 rewrite rules (labeled S1–S24) that model concurrent execution, object creation, and inter-object communication. It also relies on equations (labeled T1–T17) to perform auxiliary tasks. We assume

throughout that the Creol program is syntactically correct and well-typed. This can be checked using a separate tool, such as the one developed by Jørgen Hermanrud Fjeld [Fje05].

To execute a Creol program, we must first convert its source text into a system configuration that reflects the initial state of the system. Definitions T1–T3 express this conversion within the rewriting logic framework.

Definition T1 (Interface Declaration)

$$\textbf{interface}\ i\ [(\bar{x})]\ \ldots\ \textbf{begin with}\ \tau\colon \ldots\ \textbf{end} \triangleq \emptyset$$

Interfaces are used by the type checker to ensure that when a method m is called on an object o, the method exists and the arguments are well-typed. In the operational semantics, it could be used to bind methods correctly in the presence of dynamic updates. For the purposes of this section, we simply ignore them.

Definition T2 (Class Declaration)

$$\begin{aligned}&\textbf{class}\ c\ [(\bar{x})]\ \ldots\ [\textbf{inherits}\ \bar{\varsigma}]\\ &\textbf{begin}\ [\textbf{var}\ \bar{w}]\ [G_0]\ \textbf{with}\ \tau_1\colon G_1\ \ldots\ \textbf{with}\ \tau_m\colon G_m\ [\textbf{asum}\ \varphi]\ [\textbf{guar}\ \psi]\ \textbf{end}\\ &\quad\triangleq \langle c : \text{Class} \mid \text{Inh: } \bar{\varsigma},\ \text{Param: } \bar{x},\ \text{Att: } \bar{w},\ \text{Mtd: } \mathit{classMethods}(G_0,\ldots,G_m),\\ &\qquad\qquad \text{ObjCnt: } 0\rangle\end{aligned}$$

Class declarations give rise to terms of the form $\langle c : \text{Class} \mid \ldots \rangle$. The *classMethods* auxiliary function expands to a multiset of method terms. For brevity, we will not present it here; see Appendix B for details.

Definition T3 (Synthetic Statements)

$$\begin{aligned}
&!O.m(\bar{e}) && \triangleq \nu!O.m(\bar{e})\\
&!m[@c](\bar{e}) && \triangleq \nu!m[@c](\bar{e})\\
&O.m(\bar{e}\,[;\bar{z}]) && \triangleq \nu!O.m(\bar{e});\ \nu?([\bar{z}])\\
&m[@c](\bar{e}\,[;\bar{z}]) && \triangleq \nu!m[@c](\bar{e});\ \nu?([\bar{z}])\\
&\textbf{await}\ [g\ \&]\ l?(\bar{z}) && \triangleq \textbf{await}\ [g\ \&]\ l?;\ l?(\bar{z})\\
&\textbf{await}\ [g\ \&]\ O.m(\bar{e}\,[;\bar{z}]) && \triangleq \nu!O.m(\bar{e});\ \textbf{await}\ [g\ \&]\ \nu?;\ \nu?([\bar{z}])\\
&\textbf{await}\ [g\ \&]\ m[@c](\bar{e}\,[;\bar{z}]) && \triangleq \nu!m[@c](\bar{e});\ \textbf{await}\ [g\ \&]\ \nu?;\ \nu?([\bar{z}])\\
&\textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{fi} && \triangleq \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else skip fi}
\end{aligned}$$

We use equations to expand statements that were defined as abbreviations. The letter ν is a special identifier that may not appear in actual programs.

Other versions of the interpreter simplify **await** statements further, reducing **await wait** & B & $l?$ to **await wait**; **await** $l?$; **await** B, in that order. This approach leads to simpler equations later on, but it also has subtle effects on the semantics (and thus on the axiomatization) of $S_1 \,\square\, S_2$ and $|||_{i=1}^{n} S_i$ when some of the S_i branches contain **await** statements. Section A.2.9 has the details.

Definition T4 (Residual Statements)

$$\begin{aligned}
&\epsilon := \epsilon \triangleq \textbf{skip}\\
&\epsilon \mathbin{///} S \triangleq S
\end{aligned}$$

Definition T4 eliminates constructs that may appear during program execution, as we will see shortly.

The equations presented so far are concerned with setting up the class terms so that they can be executed. To actually launch the program, we must create a root object using Rewrite Rule S1, presented below. For conciseness, the fields that are not used by a rule are omitted.

Rewrite Rule S1 (System Bootstrapping)

$$\begin{array}{l}\{\ \ \textbf{bootstrap system}\ c[(\bar{v})] \\ \quad \Gamma\ \ \} \\ \quad\longrightarrow \\ \{\ \ \langle c\#0 : c \mid \text{Pr: } \mathit{initialPr}(c(\bar{v}), \Gamma),\ \text{LVar: } \emptyset, \\ \qquad\qquad \text{Att: } \mathit{initialAtt}(c(\bar{v}), \Gamma)[\textbf{self} \mapsto c\#0],\ \text{PrQ: } \emptyset,\ \text{MsgQ: } \emptyset, \\ \qquad\qquad \text{LabCnt: } 0\rangle \\ \quad \mathit{incrementObjCnt}(c, \Gamma)\ \ \}\end{array}$$

If there is a **bootstrap system** command in the system configuration, we create an object term for the specified class and call it c#0. The new object's Pr and Att fields are initialized using the *initialPr* and *initialAtt* auxiliary functions, which traverse the class hierarchy. The *incrementObjCnt* function returns the configuration Γ in which the ObjCnt field of class c has been incremented by 1, to ensure that the object identities remain unique.

The *initialPr*, *initialAtt*, and *incrementObjCnt* functions need to access the terms that represent class c and its superclasses to do their work. The equations that implement *initialPr*, *initialPr*, and *incrementObjCnt* are listed in Appendix B. The braces around the left-hand side ensure that Γ matches all the classes and interfaces, including c and all its superclasses; without the braces, Γ could match any subset of the configuration.[1]

As soon as we have an object in the system, we can start executing its active process. This is what most of the remaining rewriting rules are about.

Rewrite Rule S2 (Null Statement)

$$\begin{array}{l}\langle o : c \mid \text{Pr: } \textbf{skip}; S\rangle \\ \quad\longrightarrow \\ \langle o : c \mid \text{Pr: } S\rangle\end{array}$$

Executing a **skip** statement amounts to eliminating it. An alternative to Rewrite Rule S2 would be the equation

$$\langle o : c \mid \text{Pr: } \textbf{skip}; S\rangle \triangleq \langle o : c \mid \text{Pr: } S\rangle.$$

Both approaches work equally well for executing **skip**, because the statement is terminating and deterministic. Statements that may not terminate (like **while**) or that are nondeterministic (like $\Box$) must be modeled by rewrite rules, to keep the equation system confluent and terminating. For consistency, we will systematically

[1] Another option would have been to use messages and equations to traverse the class hierarchy [JO07]. However, this approach relies on specificities of Maude's operational semantics and leads to incorrect behavior under the logical semantics associated with Maude specifications.

use rewrite rules to model the execution of a statement in an object. It might also be tempting to define an equation $\mathbf{skip} \triangleq \epsilon$ to eliminate all **skip** statements before they are executed; however, this would have subtle consequences on $\Box$ and $|||$, which inspect the very first statement of each branch before committing to a branch.

Rewrite Rule S3 (Abnormal Termination)

$$\begin{array}{l}\langle o : c \mid \text{Pr: } \mathbf{abort}; S\rangle \\ \longrightarrow \\ \emptyset\end{array}$$

The **abort** statement causes the object to disappear. To other objects, an object that has aborted is indistinguishable from an object that is infinitely slow to respond.

Rewrite Rule S4 (Inline Assertion)

$$\begin{array}{l}\langle o : c \mid \text{Pr: } \mathbf{prove}\ \varphi; S\rangle \\ \longrightarrow \\ \langle o : c \mid \text{Pr: } S\rangle\end{array}$$

Like **skip**, the **prove** statement is simply ignored. An alternative would have been to check that the assertion φ (which must then be a Boolean expression) is true each time the statement is executed. In this thesis, we favor an approach where assertions specified by the programmer are formally verified before the program is run, rather than tested at run-time [AJO04, JOT06].

Rewrite Rule S5 (Assignment)

$$\begin{array}{l}\langle o : c \mid \text{Pr: } (z_0, \bar{z}) := (e_0, \bar{e}); S, \text{ LVar: } \beta, \text{ Att: } \alpha\rangle \\ \longrightarrow \\ \mathbf{if}\ z_0\ \mathbf{in}\ \beta\ \mathbf{then}\ \langle o : c \mid \text{Pr: } \bar{z} := \{\bar{e}\}_{\alpha\beta}; S, \text{ LVar: } \beta[z_0 \mapsto \{e_0\}_{\alpha\beta}], \text{ Att: } \alpha\rangle \\ \qquad\qquad \mathbf{else}\ \langle o : c \mid \text{Pr: } \bar{z} := \{\bar{e}\}_{\alpha\beta}; S, \text{ LVar: } \beta, \text{ Att: } \alpha[z_0 \mapsto \{e_0\}_{\alpha\beta}]\rangle\ \mathbf{fi}\end{array}$$

The assignment statement takes a list of variables $z_0, \bar{z}$ and a list of expressions $e_0, \bar{e}$. We start by evaluating e_0 in the state provided by α and β and assigning that value to z_0. If z_0 is a local variable, we update the local state β; otherwise, we update the object state α. Although the assignment statement requires several rewrite steps (one for each variable we assign to), it appears to be simultaneous because it evaluates all assigned expressions during the first step.

The **in** predicate checks whether a qualified identifier z is attributed a value by a state σ. It is defined as follows:

Definition T5 (State Membership)

$$\begin{array}{lll} z\ \mathbf{in}\ \emptyset & \triangleq & \mathbf{false} \\ z\ \mathbf{in}\ \sigma[z' \mapsto v] & \triangleq & z = z' \vee z\ \mathbf{in}\ \sigma \end{array}$$

Example 4.5. Suppose a, b, and c are local variables with initial values A, B, and C, respectively. The following execution shows that the assignment $a, b, c := c, a, b$ effectively shuffles the values of the three variables:

$$\xrightarrow{\text{S5}} \langle o : c' \mid \text{Pr: } a, b, c := c, a, b, \text{ LVar: } \beta[a \mapsto A][b \mapsto B][c \mapsto C] \rangle$$
$$\xrightarrow{\text{S5}} \langle o : c' \mid \text{Pr: } b, c := A, B, \text{ LVar: } \beta[a \mapsto C][b \mapsto B][c \mapsto C] \rangle$$
$$\xrightarrow{\text{S5}} \langle o : c' \mid \text{Pr: } c := B, \text{ LVar: } \beta[a \mapsto C][b \mapsto A][c \mapsto C] \rangle$$
$$\xrightarrow{\text{S2}} \langle o : c' \mid \text{Pr: } \mathbf{skip}, \text{ LVar: } \beta[a \mapsto C][b \mapsto A][c \mapsto B] \rangle$$
$$\langle o : c' \mid \text{Pr: } \epsilon, \text{ LVar: } \beta[a \mapsto C][b \mapsto A][c \mapsto B] \rangle.$$

The reduction of $\epsilon := \epsilon$ to **skip** is handled implicitly by Definition T4. □

Rewrite Rule S6 (If Statement)

$$\langle o : c \mid \text{Pr: } \mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}; S, \text{ LVar: } \beta, \text{ Att: } \alpha \rangle$$
$$\longrightarrow$$
$$\mathbf{if}\ \{B\}_{\alpha\beta}\ \mathbf{then}\ \langle o : c \mid \text{Pr: } S_1; S, \text{ LVar: } \beta, \text{ Att: } \alpha \rangle$$
$$\mathbf{else}\ \langle o : c \mid \text{Pr: } S_2; S, \text{ LVar: } \beta, \text{ Att: } \alpha \rangle\ \mathbf{fi}$$

If the condition B evaluates to **true**, the **if** statement expands to its **then** branch; otherwise, it expands to its **else** branch. Notice that the first **if** construct in the rule above is a Creol statement, while the second **if** is a conditional expression in rewriting logic.

Rewrite Rule S7 (While Loop)

$$\langle o : c \mid \text{Pr: } [\mathbf{inv}\ \varphi]\ \mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}; S', \text{ LVar: } \beta, \text{ Att: } \alpha \rangle$$
$$\longrightarrow$$
$$\mathbf{if}\ \{B\}_{\alpha\beta}\ \mathbf{then}\ \langle o : c \mid \text{Pr: } S; \mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}; S', \text{ LVar: } \beta, \text{ Att: } \alpha \rangle$$
$$\mathbf{else}\ \langle o : c \mid \text{Pr: } S', \text{ LVar: } \beta, \text{ Att: } \alpha \rangle\ \mathbf{fi}$$

If the condition B evaluates to **true**, the **while** statement expands to its body followed by another instance of the loop; otherwise, the entire statement is eliminated.

Rewrite Rule S8 (Guard Crossing)

$$\langle o : c \mid \text{Pr: } \mathbf{await}\ g; S, \text{ LVar: } \beta, \text{ Att: } \alpha, \text{ MsgQ: } Q \rangle$$
$$\longrightarrow$$
$$\langle o : c \mid \text{Pr: } S, \text{ LVar: } \beta, \text{ Att: } \alpha, \text{ MsgQ: } Q \rangle$$
$$\mathbf{if}\ \mathit{satisfied}(g, \alpha\beta, Q)$$

An **await** statement whose guard evaluates to **true** is equivalent to a **skip** statement. The *satisfied* predicate is defined recursively using equations:

Definition T6 (Guard Satisfaction)

$$\begin{aligned}
&\mathit{satisfied}(B, \sigma, Q) && \triangleq \{B\}_\sigma \\
&\mathit{satisfied}(l?, \sigma, Q) && \triangleq \mathit{replied}(\{l\}_\sigma, Q) \\
&\mathit{satisfied}(\mathbf{wait}, \sigma, Q) && \triangleq \mathbf{false} \\
&\mathit{satisfied}(g_1\ \&\ g_2, \sigma, Q) && \triangleq \mathit{satisfied}(g_1, \sigma, Q) \wedge \mathit{satisfied}(g_2, \sigma, Q)
\end{aligned}$$

A Boolean guard is satisfied if it evaluates to **true**. A reply guard $l?$ is satisfied if the reply with label l is in the object's incoming message queue. The label identifier l must be evaluated to yield a sequence number. A **wait** guard is considered **false**. The *replied* predicate is defined below.

Definition T7 (Replied Predicate)

$$\begin{array}{ll} \mathit{replied}(k, \emptyset) & \triangleq \textbf{false} \\ \mathit{replied}(k, \{\mathit{Invoke}(o, k', m@c, \bar{v})\} \cup Q) & \triangleq \mathit{replied}(k, Q) \\ \mathit{replied}(k, \{\mathit{Reply}(k', \bar{v})\} \cup Q) & \triangleq k = k' \vee \mathit{replied}(k, Q) \end{array}$$

The $\mathit{replied}(k, Q)$ predicate determines whether the message queue Q contains the reply associated with the asynchronous message call with sequence number k.

Rewrite Rule S9 (Nondeterministic Choice)

$$\begin{array}{l} \langle o : c \mid \text{Pr: } (S_1 \,\square\, S_2); S, \text{ LVar: } \beta, \text{ Att: } \alpha, \text{ MsgQ: } Q \rangle \\ \longrightarrow \\ \langle o : c \mid \text{Pr: } S_1; S, \text{ LVar: } \beta, \text{ Att: } \alpha, \text{ MsgQ: } Q \rangle \\ \textbf{if } \mathit{ready}(S_1, \alpha\beta, Q) \end{array}$$

The $\square$ statement may choose its left branch if it is ready. By letting $\square$ be commutative in the rewriting logic, the rule may also choose the right branch if that one is ready. This saves us from formulating a separate rule for the right branch. If both branches are ready, the rule may be applied in two different ways.

Rewrite Rule S10 (Nondeterministic Merge)

$$\begin{array}{l} \langle o : c \mid \text{Pr: } \big(|||_{i=1}^{n} S_i\big); S, \text{ LVar: } \beta, \text{ Att: } \alpha, \text{ MsgQ: } Q \rangle \\ \longrightarrow \\ \langle o : c \mid \text{Pr: } \big(S_1 \mathbin{///} |||_{i=2}^{n} S_i\big); S, \text{ LVar: } \beta, \text{ Att: } \alpha, \text{ MsgQ: } Q \rangle \\ \textbf{if } \mathit{ready}(S_1, \alpha\beta, Q) \end{array}$$

The $|||$ operator makes a nondeterministic choice among its branches based on their readiness, to determine which branch should start executing. The actual execution of the branch is performed by the auxiliary operator $///$. To allow the selection of a random branch S_i, we let the branches of $|||$ commute inside the rewriting logic.

Rewrite Rule S11 (Left Merge)

$$\begin{array}{l} \langle o : c \mid \text{Pr: } \big((s; S_1') \mathbin{///} |||_{i=2}^{n} S_i\big); S, \text{ LVar: } \beta, \text{ Att: } \alpha, \text{ MsgQ: } Q \rangle \\ \longrightarrow \\ \textbf{if } \mathit{enabled}(s, \alpha\beta, Q) \textbf{ then} \\ \quad \langle o : c \mid \text{Pr: } s; \big(S_1' \mathbin{///} |||_{i=2}^{n} S_i\big); S, \text{ LVar: } \beta, \text{ Att: } \alpha, \text{ MsgQ: } Q \rangle \\ \textbf{else} \\ \quad \langle o : c \mid \text{Pr: } \big((s; S_1') \mathbin{|||} |||_{i=2}^{n} S_i\big); S, \text{ LVar: } \beta, \text{ Att: } \alpha, \text{ MsgQ: } Q \rangle \\ \textbf{fi} \end{array}$$

The $///$ operator executes its left branch statement by statement until it encounters a disabled statement. At that point, it lets the $|||$ operator make a nondeterministic choice between its n branches again. Because it prefers the left branch over the right branch, $///$ is not commutative. The reduction of $\epsilon \mathbin{///} |||_{i=2}^{n} S_i$ to $|||_{i=2}^{n} S_i$ is handled by Definition T4.

The history of $|||$ is convoluted. In earlier versions of Creol [Arn03, JO04a], $S_1 \mathbin{|||} S_2$ expanded to $(S_1; S_2) \,\square\, (S_2; S_1)$, which lead to very coarse-grained interleaving.

Following a suggestion by Are Husby [Hus05], newer versions of the Creol interpreter use a non-commutative version of $S_1 \,|||\, S_2$ that gives precedence to S_1 if both branches are ready. Our formulation of $|||$ in terms of $///$ is inspired by Johnsen and Owe [JO07]. This commutative definition seems preferable to the non-commutative definition for the following reasons:

1. Commutativity relieves the programmer from the need to choose between $S_1 \,|||\, S_2$ and $S_2 \,|||\, S_1$ where both would work, enhancing the clarity of the program and avoiding an overspecification of the solution [Dij75].
2. The commutative definition leads to a simpler axiomatization.
3. Similar merge operators found in other concurrent languages [AO97, And00] or process calculi [Hoa85, Mil99] are commutative.

Moreover, programmers who desire one of the two other semantics can usually use $\Box$ or $|||$ to achieve it. For example, the non-commutative definition of $S_1 \,|||\, S_2$ can be approximated by (**skip**; S_1) $|||$ (**await wait**; S_2).

The definition of $|||$ as an n-ary operator is original to this thesis. It allows us to distinguish between $(S_1 \,|||\, S_2) \,|||\, S_3$ and $S_1 \,|||\, (S_2 \,|||\, S_3)$ and $S_1 \,|||\, S_2 \,|||\, S_3$ semantically, enabling a cleaner axiomatization.

Rewrite Rule S12 (Parenthesized Statement)

$$\begin{array}{l} \langle o : c \mid \text{Pr: } (S); S' \rangle \\ \quad\longrightarrow \\ \langle o : c \mid \text{Pr: } S; S' \rangle \end{array}$$

Executing a parenthesized statement amounts to executing the statement within the parentheses. To stress the difference between Creol parentheses (which are part of the syntax) and rewriting logic parentheses (which are part of the semantics), Creol parentheses are shown in bold.

Rewrite Rule S13 (Object Creation)

$$\begin{array}{l} \{\ \langle o : c \mid \text{Pr: } z := \textbf{new } c'[(\bar{e})]; S, \text{ LVar: } \beta, \text{ Att: } \alpha \rangle \\ \quad \Gamma\ \} \\ \quad\longrightarrow \\ \{\ \langle o : c \mid \text{Pr: } z := o'; S, \text{ LVar: } \beta, \text{ Att: } \alpha \rangle \\ \quad \langle o' : c' \mid \text{Pr: } \mathit{initialPr}(c'(\{\bar{e}\}_{\alpha\beta}), \Gamma), \text{ LVar: } \emptyset, \\ \qquad\qquad \text{Att: } \mathit{initialAtt}(c', \Gamma)[\textbf{self} \mapsto o'], \text{ PrQ: } \emptyset, \text{ MsgQ: } \emptyset, \text{ LabCnt: } 0 \rangle \\ \quad \mathit{incrementObjCnt}(c', \Gamma)\ \} \\ \textbf{if } o' := \mathit{nextOId}(c', \Gamma) \end{array}$$

A **new** statement creates an instance of a given class. The new object's identity is computed by the *nextOId* auxiliary function and bound to o'; it is of the form $c'\#n$, where c' is the class name and n a sequence number that uniquely identifies this object among c' instances. Like in Rewrite Rule S1, the Pr and Att fields are set up using the *initialPr* and *initialAtt* auxiliary functions, and Γ matches the rest of the system configuration.

In the parent object, creating an object is viewed as an assignment of o' to a variable. In the instantiated class, the object counter is incremented (by *incrementObjCnt*) to ensure that object identities remain unique.

Rewrite Rule S14 (Process Suspension)

$$\langle o : c \mid \text{Pr: } S,\ \text{LVar: } \beta,\ \text{Att: } \alpha,\ \text{PrQ: } P,\ \text{MsgQ: } Q \rangle$$
$$\longrightarrow$$
$$\langle o : c \mid \text{Pr: } \epsilon,\ \text{LVar: } \emptyset,\ \text{Att: } \alpha,\ \text{PrQ: } P \cup \{\langle \mathit{clearWait}(S), \beta \rangle\},\ \text{MsgQ: } Q \rangle$$
$$\textbf{if } \neg \mathit{enabled}(S, \alpha\beta, Q)$$

A statement is enabled unless it would immediately release the processor if executed. If the next statement is not enabled, the current process is put on the process queue, together with its local state. The *enabled* predicate is defined below.

Definition T8 (Statement Enabledness)

$$\begin{array}{lll}
\mathit{enabled}(\textbf{await } g; S, \sigma, Q) & \triangleq & \mathit{satisfied}(g, \sigma, Q) \\
\mathit{enabled}((S_1 \,\square\, S_2); S, \sigma, Q) & \triangleq & \mathit{enabled}(S_1, \sigma, Q) \vee \mathit{enabled}(S_2, \sigma, Q) \\
\mathit{enabled}((|||_{i=1}^{n} S_i); S, \sigma, Q) & \triangleq & \bigvee_{i=1}^{n} \mathit{enabled}(S_i, \sigma, Q) \\
\mathit{enabled}((S); S', \sigma, Q) & \triangleq & \mathit{enabled}(S, \sigma, Q) \\
\mathit{enabled}(S, \sigma, Q) & \triangleq & \textbf{true} \qquad \text{[otherwise]}
\end{array}$$

A statement is enabled unless it would release the processor immediately.

Definition T9 (Wait Guard Clearer)

$$\begin{array}{lll}
\mathit{clearWait}(\textbf{await } g;\ S) & \triangleq & \textbf{await } \mathit{cleared}(g);\ S \\
\mathit{clearWait}((S_1 \,\square\, S_2);\ S) & \triangleq & (\mathit{clearWait}(S_1) \,\square\, \mathit{clearWait}(S_2));\ S \\
\mathit{clearWait}((|||_{i=1}^{n} S_i);\ S) & \triangleq & (|||_{i=1}^{n} \mathit{clearWait}(S_i));\ S \\
\mathit{clearWait}((S);\ S') & \triangleq & (\mathit{clearWait}(S));\ S' \\
\mathit{clearWait}(S) & \triangleq & S \qquad \text{[otherwise]}
\end{array}$$

Definition T10 (Cleared Guard)

$$\begin{array}{lll}
\mathit{cleared}(\textbf{wait}) & \triangleq & \textbf{true} \\
\mathit{cleared}(g_1 \,\&\, g_2) & \triangleq & \mathit{cleared}(g_1) \,\&\, \mathit{cleared}(g_2) \\
\mathit{cleared}(g) & \triangleq & g \qquad \text{[otherwise]}
\end{array}$$

Informally, **wait** evaluates to **false** the first time it is encountered and to **true** afterward. This is achieved formally by letting $\mathit{satisfied}(\textbf{wait}, \sigma, Q) \triangleq \textbf{false}$ and by replacing leading **wait** guards with **true** when the process is suspended.

Rewrite Rule S15 (Process Activation)

$$\langle o : c \mid \text{Pr: } \epsilon,\ \text{LVar: } \beta,\ \text{Att: } \alpha,\ \text{PrQ: } \{\langle S', \beta' \rangle\} \cup P,\ \text{MsgQ: } Q \rangle$$
$$\longrightarrow$$
$$\langle o : c \mid \text{Pr: } S',\ \text{LVar: } \beta',\ \text{Att: } \alpha,\ \text{PrQ: } P,\ \text{MsgQ: } Q \rangle$$
$$\textbf{if } \mathit{ready}(S', \alpha\beta', Q)$$

While a process is suspended, other processes may be activated. It is also possible that the suspended process is reactivated immediately afterward, if it has become enabled in the meantime as a result of an incoming reply or a cleared **wait** guard.

Maude's facilities for associative, commutative, and identity (AC1) matching allow $\{\langle S', \beta' \rangle\}$ to match any process in PrQ. The *ready* predicate is defined below.

Definition T11 (Statement Readiness)

$$
\begin{array}{lll}
\mathit{ready}(l?(\bar{z}); S, \sigma, Q) & \triangleq & \mathit{satisfied}(l?, \sigma, Q) \\
\mathit{ready}(\textbf{await}\ g; S, \sigma, Q) & \triangleq & \mathit{satisfied}(g, \sigma, Q) \\
\mathit{ready}((S_1 \,\square\, S_2); S, \sigma, Q) & \triangleq & \mathit{ready}(S_1, \sigma, Q) \vee \mathit{ready}(S_2, \sigma, Q) \\
\mathit{ready}((|||_{i=1}^{n} S_i); S, \sigma, Q) & \triangleq & \bigvee_{i=1}^{n} \mathit{ready}(S_i, \sigma, Q) \\
\mathit{ready}((S); S', \sigma, Q) & \triangleq & \mathit{ready}(S, \sigma, Q) \\
\mathit{ready}(S, \sigma, Q) & \triangleq & \textbf{true} \qquad \text{[otherwise]}
\end{array}
$$

Intuitively, a statement is ready if it is enabled and doesn't block immediately.

Rewrite Rule S16 (Asynchronous Invocation)

$$
\begin{array}{l}
\langle o : c \mid \text{Pr: } l!O.m(\bar{e}); S, \text{ LVar: } \beta, \text{ Att: } \alpha, \text{ LabCnt: } k \rangle \\
\quad \longrightarrow \\
\langle o : c \mid \text{Pr: } S, \text{ LVar: } \beta[l \mapsto k], \text{ Att: } \alpha, \text{ LabCnt: } k+1 \rangle \\
\mathit{Invoke}(o, k, m, \{\bar{e}\}_{\alpha\beta})\ \textbf{to}\ \{O\}_{\alpha\beta}
\end{array}
$$

Asynchronous method calls lead to the creation of an invocation message that is sent to the called object. If O equals **null** or refers to an object that has aborted, the invocation message will in effect be ignored.

Each method call originated by a given object is identified by a unique sequence number k. This number is assigned to the local variable l, which corresponds to the label l. We assume that there is no local variable or parameter called l that clashes with the label l. This can easily be verified by the type checker. Some versions of Creol require the programmer to declare all labels before using them [Arn03], but since labels may appear only in certain syntactic positions where other variables are not allowed ($l!$ and $l?$), such declarations are superfluous.

Rewrite Rule S17 (Local Asynchronous Invocation)

$$
\begin{array}{l}
\langle o : c \mid \text{Pr: } l!m@c'(\bar{e}); S, \text{ LVar: } \beta, \text{ Att: } \alpha, \text{ LabCnt: } k \rangle \\
\quad \longrightarrow \\
\langle o : c \mid \text{Pr: } S, \text{ LVar: } \beta[l \mapsto k], \text{ Att: } \alpha, \text{ LabCnt: } k+1 \rangle \\
\mathit{Invoke}(o, k, m@c', \{\bar{e}\}_{\alpha\beta})\ \textbf{to}\ o
\end{array}
$$

Local method calls are initiated in essentially the same way as calls to **self**.

Rewrite Rule S18 (Transport of Invocation Message)

$$
\begin{array}{l}
\langle o : c \mid \text{MsgQ: } Q \rangle \\
\mathit{Invoke}(o', k, m@c', \bar{v})\ \textbf{to}\ o \\
\quad \longrightarrow \\
\langle o : c \mid \text{MsgQ: } Q \cup \{\mathit{Invoke}(o', k, m@c', \bar{v})\} \rangle
\end{array}
$$

At some point after an invocation message has been sent, the recipient receives it. There are no guarantees whatsoever as to when this rule is applied. In particular, Rewrite Rule S18 allows *message overtaking*, meaning that messages sent from

object o to object o' might arrive in a different order than they were sent. They may also be delayed forever, if the rule is never applied. AC1 matching applies.

Rewrite Rule S19 (Method Binding)

$$\begin{array}{l}\{\ \ \langle o : c \mid \text{PrQ: } P,\ \text{MsgQ: } \{\mathit{Invoke}(o', k, m@c', \bar{v})\} \cup Q\rangle \\ \quad \Gamma\ \ \} \\ \quad\longrightarrow \\ \{\ \ \langle o : c \mid \text{PrQ: } P \cup \mathit{boundMtd}(\textbf{if } c' = \textbf{none then } c \textbf{ else } c' \textbf{ fi}, o', k, m, \bar{v}, \Gamma), \\ \qquad\qquad \text{MsgQ: } Q\rangle \\ \quad \Gamma\ \ \}\end{array}$$

A pending invocation message gives rise to a new pending process. The *boundMtd* auxiliary function finds the specified method and returns a pair $\langle S, \beta\rangle$ storing the code and initial state of the process. Thanks to the identity axiom $m = m@\textbf{none}$, the pattern $m@c'$ matches both unqualified (m) and qualified ($m@c'$) method names. The initial state consists of the method's local variables and parameters, including the implicit **caller** and **label** parameters. The implementation of *boundMtd* is in Appendix B.

Rewrite Rule S20 (Method Return)

$$\begin{array}{l}\langle o : c \mid \text{Pr: } \textbf{return } \bar{e};\ S,\ \text{LVar: } \beta,\ \text{Att: } \alpha\rangle \\ \quad\longrightarrow \\ \langle o : c \mid \text{Pr: } S,\ \text{LVar: } \beta,\ \text{Att: } \alpha\rangle \\ \mathit{Reply}(\{\textbf{label}\}_\beta, \{\bar{e}\}_{\alpha\beta})\ \textbf{to}\ \{\textbf{caller}\}_\beta\end{array}$$

The **return** statement is added implicitly at the end of every method body by *boundMtd*. It sends a reply message to the caller along with the output arguments.

Rewrite Rule S21 (Transport of Reply Message)

$$\begin{array}{l}\langle o : c \mid \text{MsgQ: } Q\rangle \\ \mathit{Reply}(k, \bar{v})\ \textbf{to}\ o \\ \quad\longrightarrow \\ \langle o : c \mid \text{MsgQ: } Q \cup \{\mathit{Reply}(k, \bar{v})\}\rangle\end{array}$$

A reply message is eventually received by the calling object and added to its incoming message queue.

Rewrite Rule S22 (Asynchronous Reply)

$$\begin{array}{l}\langle o : c \mid \text{Pr: } l?(\bar{z});\ S,\ \text{LVar: } \beta,\ \text{MsgQ: } \{\mathit{Reply}(k, \bar{v})\} \cup Q\rangle \\ \quad\longrightarrow \\ \langle o : c \mid \text{Pr: } \bar{z} := \bar{v};\ S,\ \text{LVar: } \beta[l \mapsto -1],\ \text{MsgQ: } Q\rangle \\ \textbf{if } \{l\}_\beta = k\end{array}$$

A statement $l?(\bar{z})$ may proceed only if the corresponding reply message has arrived. We can find this out by looking for a reply message numbered k, where k is l's value. The output parameter values stored in the reply are assigned to $\bar{z}$.

The next two rules are necessary to avoid internal deadlocking when a local synchronous call is performed:

Rewrite Rule S23 (Local Reentry)

$$\langle o : c \mid \text{Pr: } l?(\bar{z});\ S,\ \text{LVar: } \beta,\ \text{PrQ: } \{\langle S', \beta' \rangle\} \cup P \rangle$$
$$\longrightarrow$$
$$\langle o : c \mid \text{Pr: } S';\ \textbf{continue}\ \{l\}_{\beta},\ \text{LVar: } \beta',\ \text{PrQ: } P \cup \{\langle l?(\bar{z}); S, \beta \rangle\} \rangle$$
$$\textbf{if } \{\textbf{caller}\}_{\beta'} = o \wedge \{\textbf{label}\}_{\beta'} = \{l\}_{\beta}$$

Rewrite Rule S24 (Local Continuation)

$$\langle o : c \mid \text{Pr: } \textbf{continue}\ k,\ \text{LVar: } \beta,\ \text{PrQ: } \{\langle l?(\bar{z}); S, \beta' \rangle\} \cup P \rangle$$
$$\longrightarrow$$
$$\langle o : c \mid \text{Pr: } l?(\bar{z});\ S,\ \text{LVar: } \beta',\ \text{PrQ: } P \rangle$$
$$\textbf{if } \{l\}_{\beta'} = k$$

If the $l?(\bar{z})$ statement corresponds to a pending local method invocation, we temporarily let that method proceed and append a **continue** statement (instead of blocking forever). The **continue** statement gives control back to the process that attempted to execute $l?(\bar{z})$; it shouldn't occur in actual programs.

Example 4.6. Consider the local call $cube(4; n)$. By Definition T3, the statement is first expanded to $\nu!cube(4);\ \nu?(n)$. From there, we have the following execution:

$\xrightarrow{\text{S17}}$ $\langle o : c \mid \text{Pr: } \nu!cube(4);\ \nu?(n),\ \text{LVar: } \beta,\ \text{PrQ: } P,\ \text{MsgQ: } Q,\ \text{LabCnt: } k \rangle$

$\langle o : c \mid \text{Pr: } \nu?(n),\ \text{LVar: } \beta[\nu \mapsto k],\ \text{PrQ: } P,\ \text{MsgQ: } Q,\ \text{LabCnt: } k+1 \rangle$
$Invoke(o, k, cube, 4)$ **to** o

$\xrightarrow{\text{S18}}$ $\langle o : c \mid \text{Pr: } \nu?(n),\ \text{LVar: } \beta[\nu \mapsto k],\ \text{PrQ: } P,\ \text{MsgQ: } Q \cup \{Invoke(o, k, cube, 4)\} \rangle$

$\xrightarrow{\text{S19}}$ $\langle o : c \mid \text{Pr: } \nu?(n),\ \text{LVar: } \beta[\nu \mapsto k],\ \text{PrQ: } P \cup \{\langle y := x * x;\ \textbf{return}\ y;\ \textbf{continue}\ k,\ [\textbf{caller} \mapsto o][\textbf{label} \mapsto k][x \mapsto 4][y \mapsto 0] \rangle\}$
MsgQ: $Q \rangle$

$\xrightarrow{\text{S23}}$ $\langle o : c \mid \text{Pr: } y := x * x;\ \textbf{return}\ y;\ \textbf{continue}\ k,$
LVar: $[\textbf{caller} \mapsto o][\textbf{label} \mapsto k][x \mapsto 4][y \mapsto 0],$
PrQ: $P \cup \{\langle \nu?(n), \beta[\nu \mapsto k] \rangle\},$ MsgQ: $Q \rangle$

$\xrightarrow{\text{S5}}$ $\langle o : c \mid \text{Pr: } \textbf{return}\ y;\ \textbf{continue}\ k, \text{LVar: } [\textbf{caller} \mapsto o][\textbf{label} \mapsto k][x \mapsto 4][y \mapsto 64],$
PrQ: $P \cup \{\langle \nu?(n), \beta[\nu \mapsto k] \rangle\},$ MsgQ: $Q \rangle$

$\xrightarrow{\text{S20}}$ $\langle o : c \mid \text{Pr: } \textbf{continue}\ k, \text{LVar: } [\textbf{caller} \mapsto o][\textbf{label} \mapsto k][x \mapsto 4][y \mapsto 64],$
PrQ: $P \cup \{\langle \nu?(n), \beta[\nu \mapsto k] \rangle\},$ MsgQ: $Q \rangle$

$Reply(k, 64)$ **to** o

$\xrightarrow{\text{S24}}$ $\langle o : c \mid \text{Pr: } \nu?(n),\ \text{LVar: } \beta[\nu \mapsto k],\ \text{PrQ: } P,\ \text{MsgQ: } Q \rangle$

$Reply(k, 64)$ **to** o

$\xrightarrow{\text{S21}}$ $\langle o : c \mid \text{Pr: } \nu?(n),\ \text{LVar: } \beta[\nu \mapsto k],\ \text{PrQ: } P,\ \text{MsgQ: } Q \cup Reply(k, 64) \rangle$

$\xrightarrow{\text{S22}}$ $\langle o : c \mid \text{Pr: } n := 64,\ \text{LVar: } \beta[\nu \mapsto -1],\ \text{PrQ: } P,\ \text{MsgQ: } Q \rangle$

$\xrightarrow{\text{S5}}$ $\langle o : c \mid \text{Pr: } \textbf{skip},\ \text{LVar: } \beta[\nu \mapsto -1][n \mapsto 64],\ \text{PrQ: } P,\ \text{MsgQ: } Q \rangle$

$\xrightarrow{\text{S2}}$ $\langle o : c \mid \text{Pr: } \epsilon,\ \text{LVar: } \beta[\nu \mapsto -1][n \mapsto 64],\ \text{PrQ: } P,\ \text{MsgQ: } Q \rangle.$ □

To complete the semantics of Creol, we will define the $\{e\}_\sigma$ function, which we used in several places to determine the value of expression e in a given state σ.

Definition T12 (Evaluation of Variable)

$$\{z\}_{\sigma[z' \mapsto v]} \triangleq \textbf{if } z = z' \textbf{ then } v \textbf{ else } \{z\}_\sigma \textbf{ fi}$$

Evaluating a variable consists of looking it up in the state. The function is defined only for existing variables. If an undefined variable z is supplied, the result is an irreducible term $\{z\}_\emptyset$ of kind [*Value*].

Definition T13 (Evaluation of Generic Expression)

$$\begin{aligned} \{\textbf{if } B \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ fi}\}_\sigma &\triangleq \textbf{if } \{B\}_\sigma \textbf{ then } \{e_1\}_\sigma \textbf{ else } \{e_2\}_\sigma \textbf{ fi} \\ \{(e)\}_\sigma &\triangleq \{e\}_\sigma \end{aligned}$$

A Creol **if** expression evaluates to a rewriting logic **if** expression, where the condition and the two branches have been evaluated. A parenthesized expression evaluates to the value of the expression in parentheses.

Definition T14 (Evaluation of Arithmetic Expression)

$$\begin{aligned} \{n\}_\sigma &\triangleq n \\ \{+A\}_\sigma &\triangleq \{A\}_\sigma \\ \{-A\}_\sigma &\triangleq -\{A\}_\sigma \\ \{A_1 * A_2\}_\sigma &\triangleq \{A_1\}_\sigma * \{A_2\}_\sigma \\ \{A_1 \,/\, A_2\}_\sigma &\triangleq \{A_1\}_\sigma \,/\, \{A_2\}_\sigma \\ \{A_1 + A_2\}_\sigma &\triangleq \{A_1\}_\sigma + \{A_2\}_\sigma \\ \{A_1 - A_2\}_\sigma &\triangleq \{A_1\}_\sigma - \{A_2\}_\sigma \end{aligned}$$

For evaluating arithmetic expressions, we rely on operators specified algebraically by equations. Tools like Maude already provide the most important built-in operators for unbound integers. The operator precedence specified in Section 4.1 must be respected when evaluating arithmetic expressions.

Definition T15 (Evaluation of Boolean Expression)

$$\begin{aligned} \{\textbf{true}\}_\sigma &\triangleq \textbf{true} \\ \{\textbf{false}\}_\sigma &\triangleq \textbf{false} \\ \{e_1 \, R \, e_2\}_\sigma &\triangleq \{e_1\}_\sigma \, R \, \{e_2\}_\sigma \\ \{\neg B\}_\sigma &\triangleq \neg\{B\}_\sigma \\ \{B_1 \wedge B_2\}_\sigma &\triangleq \{B_1\}_\sigma \wedge \{B_2\}_\sigma \\ \{B_1 \vee B_2\}_\sigma &\triangleq \{B_1\}_\sigma \vee \{B_2\}_\sigma \end{aligned}$$

Boolean expressions are evaluated in an analogous way to arithmetic expressions.

Definition T16 (Evaluation of Object Expression)

$$\{o\}_\sigma \triangleq o$$

Object identities (including the special identity **null**) evaluate to themselves.

Definition T17 (Evaluation of Expression List)

$$\{\epsilon\}_\sigma \triangleq \epsilon$$
$$\{e_0, \bar{e}\}_\sigma \triangleq \{e_0\}_\sigma, \{\bar{e}\}_\sigma \quad \textbf{if } \bar{e} \neq \epsilon$$

Finally, evaluating a list of expressions amounts to evaluating each expression.

The operational semantics defined by Rewrite Rules S1–S24 can be used to simulate the execution of a Creol program. In fact, the Creol interpreter of Appendix B is essentially a Maude specification of the rewrite rules and equations presented here. Such an interpreter simulates an entire system of objects communicating with each other. Using Maude's search capabilities, we can analyze the system in various ways. We can also implement a custom execution strategy to guide Maude when two or more rules are applicable. This can be used to obtain more random (or fairer) executions than are achieved by the built-in `rew` and `frew` strategies [AJO04, JOT06].

There is, however, one significant limitation in what we have done so far: While our semantics correctly captures the behavior of a closed system, it doesn't directly cater for open systems, which are characterized as follows:

1. Objects communicate through interfaces and generally don't have access to each other's implementations.
2. Objects and classes may be added at any time to the system.
3. Classes should be upgradable while the system is running.

Point 1 is a fundamental issue for us. The operational semantics presented in this section captures the execution of a monolithic system, but we have no satisfactory way to simulate the activity of a single process taken in isolation once we abstract away the environment with which it communicates (the other processes executing in the same object and the other objects in the system). Even for a closed system, this is an issue if the system includes a proprietary third-party component whose source code is kept secret.

Point 2 reflects a limitation in Creol's syntax as we have defined it here. For our convenience, we have assumed the existence of a root object that launches the system, but this approach is generally impractical in a distributed setting, where there might be several root objects created independently of each other. To remove this limitation, we would need to extend the Creol syntax to allow different Creol programs to communicate with each other [Kya06].

Point 3 raises a particularly difficult issue. On the one hand, we want to verify formally that a class update is safe before applying it to a system in use; on the other hand, we want to reuse existing correctness proofs as much as possible. This issue is currently being researched [Ofs05, YJO06]; we will not consider it here.

4.4 An Alternative Semantics for Open Systems

In this section, we will define an operational semantics that captures the behavior of a process seen in isolation, paving the way for the compositional proof system

introduced in Chapter 5. Conceptually, the semantics focuses on a single process executing in an unspecified environment. In this setting, a system configuration will always contain exactly one object executing one process.

Following Dahl [Dah77], we introduce the concept of a *communication history*. A communication history records the creation of objects and the messages that are exchanged between objects in a distributed system—in other words, the objects' observable behavior. More formally, a history h is a finite sequence of *communication events*. Let $o, o' \in \mathit{OId}$, $c, m \in \mathit{Id}$, $\bar{v}, \bar{w} \in \mathcal{L}(\mathit{Value})$, and $k \in \mathbb{N}$. The following are communication events:

$[o{\rightarrow}o'.\mathbf{new}\ c(\bar{v})]$	object creation
$[o{\rightarrow}o'.m@c(\bar{v})]^k$	asynchronous invocation
$[o{\leftarrow}o'.m@c(\bar{v};\bar{w})]^k$	asynchronous reply

For invocation and reply events, $\bar{v}$ stores the values passed to the method, and k is the sequence number of the method call. For reply events, $\bar{w}$ stores the return values. To simplify the notation, we will write

$$[o{\leftrightarrow}o'.m@c(\bar{v};\bar{w})]^k$$

as an abbreviation for

$$[o{\rightarrow}o'.m@c(\bar{v})]^k \frown [o{\leftarrow}o'.m@c(\bar{v};\bar{w})]^k,$$

where $\frown$ denotes concatenation of histories. In addition to these inter-object communication events, we also record how processes compete for the processor within a single object. This is captured by the following control events:

$[o.\mathbf{initialized}]$	termination of initial process
$[o.\mathbf{release}]$	processor release
$[o.\mathbf{reenter}]^k$	local reentry

The **initialized** event is recorded when the initial process of an object has terminated. The **release** and **reenter** events correspond to applications of Rewrite Rules S14 (Process Suspension) and S23 (Local Reentry) in the semantics of Section 4.3.

Example 4.7. Consider the producer–consumer program from the beginning of the chapter. The system starts when *Main* is instantiated. Next, the *Main* object's *run* method instantiates *Buffer*, *Producer*, and *Consumer*. At that point, the *Producer* object repeatedly calls *put* on the *Buffer* object, and the *Consumer* object repeatedly calls *get*. The following sequence of events is a possible history for this system when the second value has reached the consumer:

$$\begin{aligned}
&[\mathbf{null}{\rightarrow}M.\mathbf{new}\ \mathit{Main}()] \frown [M{\rightarrow}M.\mathit{run}()]^0 \frown [M.\mathbf{reenter}]^0 \frown \\
&[M{\rightarrow}B.\mathbf{new}\ \mathit{Buffer}()] \frown [M{\rightarrow}P.\mathbf{new}\ \mathit{Producer}(B)] \frown \\
&[M{\rightarrow}C.\mathbf{new}\ \mathit{Consumer}(B)] \frown [M{\leftarrow}M.\mathit{run}()]^0 \frown [M.\mathbf{initialized}] \frown \\
&[P{\rightarrow}P.\mathit{run}()]^0 \frown [C{\rightarrow}C.\mathit{run}()]^0 \frown [P{\leftrightarrow}B.\mathit{put}(1;)]^1 \frown [C{\leftrightarrow}B.\mathit{get}(;1)]^1 \frown \\
&[P{\leftrightarrow}B.\mathit{put}(2;)]^2 \frown [C{\leftrightarrow}B.\mathit{get}(;2)]^2.
\end{aligned}$$

In the above, M, B, P, and C denote *Main*#0, *Buffer*#0, *Producer*#0, and *Consumer*#0, respectively. □

The history represents a snapshot of the system's execution at a given point and is therefore always finite. When designing or analyzing a complex system, we often want to spell out the set of possible histories for that system. Knowing the set of all possible histories allows us to deduce safety properties about the system.

Let us define some notations for histories. For an event v and a history h, the predicate v **in** h is true iff v occurs in h, and v **not in** h expresses the negation of v **in** h. Given two histories h and h', h **ew** h' is true if and only if h ends with h' (that is, h' is a suffix of h). Similarly, h **bw** h', also written $h' \preceq h$, is true if and only if h begins with h'. The *projection* h/A of history h onto the set of events (or *alphabet*) A returns the longest subsequence of h that consists exclusively of events from A. The length of a history h is given by $\#(h)$. The history of length 0 is written ϵ. Event sets are often represented using wildcard patterns. For example:

$$\begin{array}{rcl}
[o{\rightarrow}*] & \triangleq & \{[o{\rightarrow}o'.\mathbf{new}\ c(\bar{v})] \mid o' \in OId, c \in Id, \bar{v} \in \mathcal{L}(Value)\} \\
& & \cup\ \{[o{\rightarrow}o'.m@c(\bar{v})]^k \mid o' \in OId, m, c \in Id, \bar{v} \in \mathcal{L}(Value), k \in \mathbb{N}\} \\
[*{\leftarrow}o] & \triangleq & \{[o'{\leftarrow}o.m@c(\bar{v};\bar{w})]^k \mid o' \in OId, m, c \in Id, \bar{v}, \bar{w} \in \mathcal{L}(Value), k \in \mathbb{N}\}.
\end{array}$$

We will also need the following event sets:

$$\begin{array}{rcl}
in_o & \triangleq & [*{\rightarrow}o] \cup [o{\leftarrow}*] \\
out_o & \triangleq & [o{\rightarrow}*] \cup [*{\leftarrow}o] \\
ctl_o & \triangleq & [o.\mathbf{initialized}] \cup [o.\mathbf{release}] \cup [o.\mathbf{reenter}] \\
o & \triangleq & in_o \cup out_o \cup ctl_o.
\end{array}$$

We are now ready to define an alternative version of Creol's operational semantics that focuses on the execution of a single process. The new "open system" operational semantics uses a communication history to abstract away the environment. The semantics reuses Rewrite Rules S2–S12 and S22 from the previous section, because these rules involve no interaction between objects or between processes within an object. Rewrite Rules S1, S13–S21, S23, and S24 are replaced with a new set of rules that operate on the history. The table below compares the closed system semantics of Section 4.3 with the open system semantics.

Name of Rewrite Rule	**Closed System**	**Open System**
System Bootstrapping	S1	–
Object Bootstrapping	–	S1′α
Method Bootstrapping	–	S1′β
Null Statement	S2	S2
Abnormal Termination	S3	S3
Inline Assertion	S4	S4
Assignment	S5	S5
If Statement	S6	S6
While Loop	S7	S7
Guard Crossing	S8	S8
Nondeterministic Choice	S9	S9
Nondeterministic Merge	S10	S10
Left Merge	S11	S11
Parenthesized Statement	S12	S12
Object Creation	S13	S13′

Name of Rewrite Rule	Closed System	Open System
Process Suspension	S14	} S14′
Process Activation	S15	
Asynchronous Invocation	S16	S16′
Local Asynchronous Invocation	S17	S17′
Transport of Invocation Message	S18	–
Method Binding	S19	–
Method Return	S20	S20′
Transport of Reply Message	S21	–
Asynchronous Reply	S22	S22
Local Reentry	S23	} S23′
Local Continuation	S24	
Parallel Activity	–	S25′

In the open system semantics, Creol objects are represented by terms of the form

$$\langle o : c \mid \text{Pr: } S,\ \text{LVar: } \beta,\ \text{Att: } \alpha,\ \text{MsgQ: } Q,\ \text{Asum: } \varphi,\ \text{Guar: } \psi,\ \text{ROAtt: } \bar{z} \rangle.$$

The attributes α now include a distinguished $\mathcal{H}$ attribute that represents the object's communication history. The history could also have been stored in a separate field, or as a separate object, but making it an attribute will simplify the definition of Hoare-style correctness formulas in Chapter 5.

We omit the LabCnt field because we can deduce its value from the communication history. Technically, the MsgQ field is also redundant now that we record the history; we keep it because Rewrite Rules S8–S11 and S22 rely on it. Finally, since we concentrate on one process's execution, we can also leave out PrQ.

On the other hand, we introduce a pair of fields, Asum and Guar, that store an assume–guarantee specification derived from the **asum** and **guar** clauses supplied by the programmer in the class declaration for c and in the declarations of the interfaces implemented by c [JO02, JO04b]. We also introduce a list of read-only attributes (ROAtt), which we will use to strengthen the guarantee.

The assumption is the responsibility of the objects in the environment. The guarantee is the responsibility of the **self** object, and must hold after initialization of the object, be maintained by all methods, and hold before all processor releases, as long as the assumption holds. Together, the assumption and the guarantee can be seen as an invariant for the class. In Section 5.5, we will see how to prove that a guarantee is valid for a class.

Example 4.8. The following interface provides access to a reference counter:

interface *ReferenceCounter*
begin
with any:
  op *inc*
  op *dec*
  op *getCount*(**out** n : **int**)
  asum $\#(\mathcal{H}/[*\rightarrow\textbf{self}.inc()]) \geq \#(\mathcal{H}/[*\rightarrow\textbf{self}.dec()])$
  guar $\exists x.\ G(\mathcal{H}, x)$
end

The *inc* and *dec* methods are expected to increment and decrement the counter by one, and the *getCount* method retrieves the current value of the counter. The predicate $G(h, x)$ used in the **guar** clause is defined recursively as follows:

$$\begin{array}{lll}
G(\epsilon, x) & \triangleq & x = 0 \\
G(h \frown [o{\rightarrow}\textbf{self}.inc()]^k, x) & \triangleq & x > 0 \wedge G(h, x-1) \\
G(h \frown [o{\rightarrow}\textbf{self}.dec()]^k, x) & \triangleq & G(h, x+1) \\
G(h \frown [o{\leftarrow}\textbf{self}.getCount(;n)]^k, x) & \triangleq & n = x \wedge G(h, x) \\
G(h \frown v, x) & \triangleq & G(h, x). \qquad \text{[otherwise]}
\end{array}$$

The **guar** clause expresses that the reference counter has a current value (represented by x) corresponding to the number of times *inc* has been called, minus the number of times *dec* has been called. It further asserts that *getCount* calls return the current value and that the current value stays positive as long as the assertion holds. The **asum** clause expresses the requirement that the history must at all times contain at least as many calls to *inc* as to *dec*.

The class declaration below implements the methods declared in the interface and respects the assume–guarantee specification:

class *SimpleReferenceCounter*
  implements *ReferenceCounter*
begin
  var *value* : **int**
with any:
  op *inc* **is**
    value := *value* + 1
  op *dec* **is**
    value := *value* − 1
  op *getCount*(**out** *n* : **int**) **is**
    n := *value*
end □

In the closed system semantics, we could ignore interfaces altogether, since we had access to the code for all the classes. In the open system semantics, interface declarations are important because we will need their assume–guarantee specifications. We represent Creol interfaces by terms of the form

$$\langle i : \text{Interface} \mid \text{Inh: } \bar{\varsigma},\ \text{Param: } \bar{\text{x}},\ \text{Asum: } \varphi,\ \text{Guar: } \psi \rangle.$$

Finally, Creol classes are represented by terms of the form

$$\langle c : \text{Class} \mid \text{Impl: } \bar{\varsigma},\ \text{Cntc: } \bar{\varsigma}',\ \text{Inh: } \bar{\varsigma}'',\ \text{Param: } \bar{\text{x}},\ \text{Att: } \bar{\text{w}},\ \text{Mtd: } M,\ \text{Asum: } \varphi,\ \text{Guar: } \psi \rangle.$$

For classes, we now store the assume–guarantee specification provided in the **asum** and **guar** clauses. We also store the interfaces listed in a class's **implements** and **contracts** clauses in the Impl and Ctrc fields; this will allow us to retrieve the superinterfaces' assume–guarantee specifications. The ObjCnt field, which we used in the closed system semantics to give unique names to new objects, is no longer necessary, because we can generate unique names by inspecting $\mathcal{H}$.

We will now review the rewrite rules that are specific to the open system semantics, starting with object creation.

Rewrite Rule S13′ (Object Creation)

$$\begin{aligned}&\langle o : c \mid \text{Pr: } z := \mathbf{new}\ c'(\bar{e});\ S,\ \text{LVar: } \beta,\ \text{Att: } \alpha\rangle\\ &\longrightarrow\\ &\langle o : c \mid \text{Pr: } z := o';\ S,\ \text{LVar: } \beta,\ \text{Att: } \alpha[\mathcal{H} \mapsto h]\rangle\\ &\textbf{if } o' \notin \mathit{objectIds}(\{\mathcal{H}\}_\alpha) \wedge \mathit{parent}(o') = o \wedge h := \{\mathcal{H}\}_\alpha \frown [o{\rightarrow}o'.\mathbf{new}\ c'(\{\bar{e}\}_{\alpha\beta})]\end{aligned}$$

With the open system semantics, a **new** statement allocates a fresh object identity o' and extends the communication history $\mathcal{H}$ with an object creation event. The new object is now part of the environment embodied by the communication history. The objects created by o have arbitrary names o' such that $\mathit{parent}(o') = o$. One option is to call o's children $o\#0$, $o\#1$, $o\#2$, and so on, and to let $\mathit{parent}(o\#n) = o$; however, Rewrite Rule S13 does not mandate any specific naming scheme. Because we require that o' is fresh, the parenthood graph remains acyclic.

Example 4.9. The following rewrite step illustrates Rewrite Rule S13′ in the context of the producer–consumer program:

$$\begin{aligned}&\xrightarrow{\text{S13}'} \langle M : \mathit{Main} \mid \text{Pr: } \mathit{cons} := \mathbf{new}\ \mathit{Consumer}(\mathit{buf});\ \mathbf{return}\ \epsilon,\ \text{Att: } \alpha[\mathcal{H} \mapsto h_0]\rangle\\ &\quad\ \langle M : \mathit{Main} \mid \text{Pr: } \mathit{cons} := C;\ \mathbf{return}\ \epsilon;\\ &\qquad\qquad \text{Att: } \alpha[\mathcal{H} \mapsto h_0 \frown [M{\rightarrow}C.\mathbf{new}\ \mathit{Consumer}(B)]]\rangle,\end{aligned}$$

where

$$\begin{aligned}h_0 \equiv\ &[\mathbf{null}{\rightarrow}M.\mathbf{new}\ \mathit{Main}()] \frown [M{\rightarrow}M.\mathit{run}()]^0 \frown [M.\mathbf{reenter}]^0 \frown\\ &[M{\rightarrow}B.\mathbf{new}\ \mathit{Buffer}()] \frown [M{\rightarrow}P.\mathbf{new}\ \mathit{Producer}(B)].\end{aligned}$$

Following the proposed naming scheme, B, P, and C stand for $M\#0$, $M\#1$, and $M\#2$, respectively. □

Rewrite Rule S16′ (Asynchronous Invocation)

$$\begin{aligned}&\langle o : c \mid \text{Pr: } l!O.m(\bar{e});\ S,\ \text{LVar: } \beta,\ \text{Att: } \alpha\rangle\\ &\longrightarrow\\ &\langle o : c \mid \text{Pr: } S,\ \text{LVar: } \beta[l \mapsto k],\ \text{Att: } \alpha[\mathcal{H} \mapsto h]\rangle\\ &\textbf{if } [o{\rightarrow}*]^k \textbf{ not in } \{\mathcal{H}\}_\alpha \wedge h := \{\mathcal{H}\}_\alpha \frown [o{\rightarrow}\{O\}_{\alpha\beta}.m(\{\bar{e}\}_{\alpha\beta})]^k\end{aligned}$$

Rewrite Rule S17′ (Local Asynchronous Invocation)

$$\begin{aligned}&\langle o : c \mid \text{Pr: } l!m@c'(\bar{e});\ S,\ \text{LVar: } \beta,\ \text{Att: } \alpha\rangle\\ &\longrightarrow\\ &\langle o : c \mid \text{Pr: } S,\ \text{LVar: } \beta[l \mapsto k],\ \text{Att: } \alpha[\mathcal{H} \mapsto h]\rangle\\ &\textbf{if } [o{\rightarrow}*]^k \textbf{ not in } \{\mathcal{H}\}_\alpha \wedge h := \{\mathcal{H}\}_\alpha \frown [o{\rightarrow}o.m@c'(\{\bar{e}\}_{\alpha\beta})]^k\end{aligned}$$

Asynchronous method calls simply extend the history with a new invocation event. The $[o{\rightarrow}*]^k$ **not in** $\{\mathcal{H}\}_\alpha$ condition ensures that k is a fresh sequence number.

Rewrite Rule S20′ (Method Return)

$$\begin{array}{l}\langle o : c \mid \text{Pr: } \mathbf{return}\ \bar{e};\ S,\ \text{LVar: } \beta,\ \text{Att: } \alpha\rangle \\ \longrightarrow \\ \langle o : c \mid \text{Pr: } S,\ \text{LVar: } \beta,\ \text{Att: } \alpha[\mathcal{H} \mapsto h]\rangle \\ \textbf{if } h := \{\mathcal{H}\}_\alpha \frown \mathit{replyEvent}(\{\mathcal{H}\}_\alpha, o, \{\mathbf{caller}\}_\beta, \{\mathbf{label}\}_\beta, \{\bar{e}\}_{\alpha\beta})\end{array}$$

Returning from a method extends the history with a reply event. The auxiliary function *replyEvent* determines the reply event by inspecting the history, which should contain a corresponding invocation event. If $[o' \rightarrow o.m(\bar{v})]^k$ **in** h, then $\mathit{replyEvent}(h, o, o', k, \bar{w})$ is $[o' \leftarrow o.m(\bar{v}; \bar{w})]^k$.

Rewrite Rule S14′ (Process Suspension and Reactivation)

$$\begin{array}{l}\langle o : c \mid \text{Pr: } S,\ \text{LVar: } \beta,\ \text{Att: } \alpha,\ \text{MsgQ: } Q,\ \text{Asum: } \varphi,\ \text{Guar: } \psi,\ \text{ROAtt: } \bar{z}\rangle \\ \longrightarrow \\ \langle o : c \mid \text{Pr: } \mathit{clearWait}(S),\ \text{LVar: } \beta,\ \text{Att: } \alpha',\ \text{MsgQ: } \mathit{replies}(\{\mathcal{H}\}_{\alpha'}, o), \\ \qquad \text{Asum: } \varphi,\ \text{Guar: } \psi,\ \text{ROAtt: } \bar{z}\rangle \\ \textbf{if } \neg \mathit{enabled}(S, \alpha\beta, Q) \wedge \mathit{release}(\varphi, \psi, o, \{\mathbf{caller}\}_\beta, \{\mathbf{label}\}_\beta, \bar{z}, \alpha, \alpha') \\ \quad \wedge \mathit{enabled}(\mathit{clearWait}(S), \alpha'\beta, \mathit{replies}(\{\mathcal{H}\}_{\alpha'}, o))\end{array}$$

If the next statement to execute is disabled, the process is suspended. When it wakes up, it is in a different state in which the statement is enabled. The writable attributes, including the history $\mathcal{H}$, might have changed in the meantime; this is modeled by replacing α with α' in the Att field. In addition, the MsgQ field is updated to reflect the new history. The constraint $\mathit{release}(\varphi, \psi, o, \{\mathbf{caller}\}_\beta, \{\mathbf{label}\}_\beta, \bar{z}, \alpha, \alpha')$ restricts the values that the attributes α' may take. It is defined below.

Definition T18 (Processor Release Predicate)

$$\begin{array}{rcl}\mathit{release}(\varphi, \psi, o, o', k, \bar{z}, \alpha, \alpha') & \triangleq & \mathit{lwf}(\{\mathcal{H}\}_{\alpha'}, o) \\ & & \wedge\ \{\mathcal{H}\}_\alpha \frown [o.\mathbf{release}] \preceq \{\mathcal{H}\}_{\alpha'} \\ & & \wedge\ \mathit{mayAcquireProcessor}(\{\mathcal{H}\}_{\alpha'}, o, o', k) \\ & & \wedge\ \mathit{pending}(\{\mathcal{H}\}_{\alpha'}, o', o, k) \\ & & \wedge\ \mathit{compatibleStates}(\alpha, \alpha') \\ & & \wedge\ \mathit{readOnly}(\bar{z}, \alpha, \alpha') \\ & & \wedge\ \big((\{\mathcal{H}\}_\alpha \frown [o.\mathbf{release}])/(\mathit{out}_o \cup \mathit{ctl}_o) = \\ & & \qquad \{\mathcal{H}\}_{\alpha'}/(\mathit{out}_o \cup \mathit{ctl}_o) \Rightarrow \\ & & \quad \alpha[\mathcal{H} \mapsto \{\mathcal{H}\}_{\alpha'}] = \alpha'\big) \\ & & \wedge\ (\{\varphi \wedge \psi\}_\alpha \Rightarrow \{\varphi \wedge \psi\}_{\alpha'})\end{array}$$

The preceding definition is complex because we want to precisely capture the effect of a processor release on the object's attributes and local history. By restricting the nondeterminism associated with processor releases, each conjunct gives us some potentially useful information about the new attribute state α'.

The first conjunct forces the new history to respect program-independent well-formedness rules, defined below. The second conjunct expresses that the old history must be a prefix of the new history, and requires the presence of a **release** event at the beginning of the history extension. The third conjunct ensures that the new history ends with an event that releases the processor. The fourth conjunct

states that the method call that released the processor should still be pending when the processor is reacquired. Together, these conjuncts capture the idea that we first release the processor, then other processes execute, and finally the processor is released again and we acquire it.

The last four conjuncts state that the new state has the same variables as the old state, that the read-only attributes cannot change, that the writable attributes may only be changed by the object itself, and that the assumption and the guarantee should hold when the processor is reacquired if they held before it was released.

Definition T19 (History Well-Formedness)

$$
\begin{array}{ll}
wf(\epsilon) & \triangleq \textbf{ true} \\
wf(h \frown [o{\rightarrow}o'.\textbf{new } c(\bar{v})]) & \triangleq wf(h) \wedge o' \notin objectIds(h) \wedge parent(o') = o \\
wf(h \frown [o{\rightarrow}o'.m@c(\bar{v})]^k) & \triangleq wf(h) \wedge [o{\rightarrow}*]^k \textbf{ not in } h \\
wf(h \frown [o{\leftarrow}o'.m@c(\bar{v};\bar{w})]^k) & \triangleq wf(h) \wedge pending(h,o,o',k) \\
wf(h \frown [o.\textbf{initialized}]) & \triangleq wf(h) \wedge [o.\textbf{initialized}]^k \textbf{ not in } h \\
wf(h \frown [o.\textbf{release}]) & \triangleq wf(h) \\
wf(h \frown [o.\textbf{reenter}]^k) & \triangleq wf(h) \wedge pending(h,o,o,k) \\
 & \quad \wedge [o.\textbf{reenter}]^k \textbf{ not in } h
\end{array}
$$

Intuitively, a communication history is well-formed if new objects have unique identities, if method invocations and replies match, and if **initialized** and **reenter** events follow certain rules. We also require that a pair (o,k) uniquely identifies a method call originating from an object o. This is all captured by the $wf(h)$ predicate.

For local histories, we have an additional requirement—only events that originate from o or that are sent to o may appear in o's history. This is captured by the $lwf(h,o)$ predicate.

Definition T20 (Local History Well-Formedness)

$$lwf(h,o) \triangleq wf(h) \wedge h = h/o$$

Well-formedness is used to constrain nondeterministic history extensions. Since Creol programs are strongly typed, we could tighten Definition T19 to incorporate type safety.

Definition T21 (Processor Acquisition Predicate)

$$
\begin{array}{l}
mayAcquireProcessor(h,o,o',k) \\
\quad \triangleq h/(out_o \cup ctl_o) \textbf{ ew } ([o.\textbf{initialized}] \cup [o.\textbf{release}]) \\
\quad\quad \vee (o = o' \wedge h/(out_o \cup ctl_o) \textbf{ ew } [o.\textbf{reenter}]^k) \\
\quad\quad \vee \exists k'.\, (h/(out_o \cup ctl_o) \textbf{ ew } [*{\leftarrow}o]^{k'} \wedge [o.\textbf{reenter}]^{k'} \textbf{ not in } h)
\end{array}
$$

The $mayAcquireProcessor(h,o,o',k)$ predicate is true if and only if the process identified by the caller o' and sequence number k running in object o is allowed to acquire the processor at the point in time described by the history h. A process can acquire the processor in any of the following circumstances:

1. When the object initialization code terminates executing, the processor is available to any pending process. This situation is identified by an **initialized** event at the end of the history.

2. Similarly, when a process explicitly releases the processor using **await**, the processor is available to any pending process. This situation is identified by a **release** event at the end of the history.

3. There is also the possibility that the process is at the receiving end of a local reentry. This may occur only for **self** calls (hence the $o = o'$ condition) and is identified by a **reenter** event.

4. Finally, when a method returns, the processor generally becomes available to any process. The exception is when the method was reentered; in that case, the processor is given to the process that performed the local reentry. This corresponds to the execution of a **continue** statement in the closed system semantics of Section 4.3.

Definition T22 (Pending Call Predicate)

$$pending(h, o, o', k) \triangleq [o{\rightarrow}o'.*]^k \textbf{ in } h \wedge [o{\leftarrow}o'.*]^k \textbf{ not in } h$$

The $pending(h, o, o', k)$ predicate is true if and only if there is an unfinished method invocation with sequence number k from o to o'.

Definition T23 (Read-Only Predicate)

$$\begin{aligned} readOnly(\epsilon, \alpha, \alpha') &\triangleq \textbf{true} \\ readOnly((z_0, \bar{z}), \alpha, \alpha') &\triangleq \{z_0\}_\alpha = \{z_0\}_{\alpha'} \wedge readOnly(\bar{z}, \alpha, \alpha') \end{aligned}$$

The $readOnly(\bar{z}, \alpha, \alpha')$ predicate is true if and only if the variables $\bar{z}$ have the same values in states α and α'.

The $compatibleStates(\alpha, \alpha')$ predicate used in Definition T18 is given in Appendix B.

Rewrite Rule S23′ (Local Reentry and Continuation)

$$\begin{aligned} &\langle o : c \mid \text{Pr: } l?(\bar{z});\ S,\ \text{LVar: } \beta,\ \text{Att: } \alpha,\ \text{Asum: } \varphi,\ \text{Guar: } \psi,\ \text{ROAtt: } \bar{z}\rangle \\ &\longrightarrow \\ &\langle o : c \mid \text{Pr: } l?(\bar{z});\ S,\ \text{LVar: } \beta,\ \text{Att: } \alpha',\ \text{Asum: } \varphi,\ \text{Guar: } \psi,\ \text{ROAtt: } \bar{z}\rangle \\ &\textbf{if } pending(\{\mathcal{H}\}_\alpha, o, o, \{l\}_\beta) \wedge reenter(\varphi, o, \{\textbf{caller}\}_\beta, \{\textbf{label}\}_\beta, \{l\}_\beta, \bar{z}, \alpha, \alpha') \end{aligned}$$

If the label l is associated with a call to **self** that hasn't been serviced yet, executing $l?(\bar{z})$ gives the processor directly to the pending process. Control returns to $l?(\bar{z})$ when the pending call is completed. This behavior is captured by the *reenter* predicate, defined below:

Definition T24 (Local Reentry Predicate)

$$\begin{aligned} reenter(\varphi, \psi, o, o', k, k', \bar{z}, \alpha, \alpha') \triangleq\ & lwf(\{\mathcal{H}\}_{\alpha'}, o) \\ & \wedge \{\mathcal{H}\}_\alpha \frown [o.\textbf{reenter}]^{k'} \preceq \{\mathcal{H}\}_{\alpha'} \\ & \wedge \{\mathcal{H}\}_{\alpha'} \textbf{ ew } [o{\leftarrow}o.*]^{k'} \\ & \wedge pending(\{\mathcal{H}\}_{\alpha'}, o', o, k) \\ & \wedge compatibleStates(\alpha, \alpha') \\ & \wedge readOnly(\bar{z}, \alpha, \alpha') \\ & \wedge (\{\varphi \wedge \psi\}_\alpha \Rightarrow \{\varphi \wedge \psi\}_{\alpha'}) \end{aligned}$$

The *reenter* predicate is very similar to *release*. To model the reentry and continuation, we append a **reenter** event to the history before we release the processor, and we require that the history ends with a matching reply event when we obtain the processor again.

Example 4.10. In an unspecified environment, a local call $l!cube(4); l?(n)$ may give rise to the following execution:

$$\begin{aligned}
&\xrightarrow{\text{S17}'} \langle o : c \mid \text{Pr: } l!cube(4); l?(n), \text{ LVar: } \beta, \text{ Att: } \alpha[\mathcal{H} \mapsto h_0], \text{ Asum: } \mathbf{true}, \text{ Guar: } \mathbf{true}\rangle \\
&\xrightarrow{\text{S23}'} \langle o : c \mid \text{Pr: } l?(n), \text{ LVar: } \beta[l \mapsto 2], \text{ Att: } \alpha[\mathcal{H} \mapsto h_0 \frown [o{\rightarrow}o.cube(4)]^2]\rangle \\
&\xrightarrow{\text{S22}} \langle o : c \mid \text{Pr: } l?(n), \text{ LVar: } \beta[l \mapsto 2], \\
&\qquad\qquad \text{Att: } \alpha[\mathcal{H} \mapsto h_0 \frown [o{\rightarrow}o.cube(4)]^2 \frown [o.\mathbf{reenter}]^2 \frown [o{\leftarrow}o.cube(4;666)]^2\rangle \\
&\qquad\quad \langle o : c \mid \text{Pr: } n := 666, \text{ LVar: } \beta[l \mapsto -1]\rangle.
\end{aligned}$$

From the caller's perspective, the *cube* method can return any value, including 666. To remedy this, we would need to supply a guarantee ψ that relates the method's return value to its input. □

Rewrite Rule S25′ (Parallel Activity)

$$\begin{aligned}
&\langle o : c \mid \text{Pr: } S, \text{ Att: } \alpha, \text{ MsgQ: } Q, \text{ Asum: } \varphi\rangle \\
&\quad\longrightarrow \\
&\langle o : c \mid \text{Pr: } S, \text{ Att: } \alpha[\mathcal{H} \mapsto h], \text{ MsgQ: } replies(h, o), \text{ Asum: } \varphi\rangle \\
&\textbf{if } interleave(\varphi, o, \alpha, h)
\end{aligned}$$

Rewrite Rule S25′ lets us extend the history in a nondeterministic way with invocation and reply events originating from the environment at any point during the execution of a process. Notice that the attributes other than $\mathcal{H}$ are left unchanged by the environment.

The nondeterministic extension of the history must abide by the following constraints: The environment must preserve the well-formedness of the history; the environment may only append events to the history; the environment may only produce events that do not originate from o; and the environment may not invalidate the assumption φ. This is formalized by the following definition:

Definition T25 (Parallel Activity Interleaving Predicate)

$$\begin{aligned}
interleave(\varphi, o, \alpha, h) \triangleq\ & lwf(h, o) \\
& \wedge \{\mathcal{H}\}_\alpha \preceq h \\
& \wedge \{\mathcal{H}\}_\alpha / (out_o \cup ctl_o) = h/(out_o \cup ctl_o) \\
& \wedge (\{\varphi\}_\alpha \Rightarrow \{\varphi\}_{\alpha[\mathcal{H}\mapsto h]})
\end{aligned}$$

We have now reviewed all the rules that can be used to simulate the execution of a process. The remaining rules, S1′α and S1′β, serve as starting points for the other rules by introducing an object term in the system configuration.

For the closed system semantics, Rewrite Rule S1 bootstrapped the system by instantiating a root object. Here, we want to instantiate any process from any class. This is achieved by letting the user specify which process to instantiate by supplying a **bootstrap object** or **bootstrap method** command in the Creol program.

The **bootstrap object** command instantiates the initial process running in an object, which initializes the context parameters and calls *init* and *run*. The syntax is

bootstrap object $o := \textbf{new}\ c(\bar{v})$ **with parent** o'

The **bootstrap method** command instantiates a process that starts as a result of an asynchronous method invocation. The syntax is

bootstrap method $o.m@c'(\bar{v})$
with class c, **caller** o', **label** k, **history** h [, **attributes** $\bar{z} := \bar{w}$]

The **history** and **attributes** clauses let us specify the exact initial state of the object at the moment when the method acquires the processor and starts executing.

The semantics of the **bootstrap** commands is given by Rewrite Rules S1′α and S1′β.

Rewrite Rule S1′α (Object Bootstrapping)

$\{$ **bootstrap object** $o := \textbf{new}\ c(\bar{v})$ **with parent** o'
Γ $\}$
$\longrightarrow$
$\{$ $\langle o : c \mid$ Pr: $initialPr(c(\bar{v}), \Gamma)$; $\mathcal{H} := \mathcal{H} \frown [o.\textbf{initialized}]$
LVar: $\emptyset$, Att: $initialAtt(c, \Gamma)[\mathcal{H} \mapsto h][\textbf{self} \mapsto o]$, MsgQ: $\emptyset$,
Asum: φ, Guar: ψ, ROAtt: $classROAtt(c, \Gamma)\rangle$
Γ $\}$
if $h := [o' \rightarrow o.\textbf{new}\ c(\bar{v})]$
$\wedge$ $parent(o) = o'$
$\wedge$ $\langle \varphi, \psi \rangle := classAGSpec(c, \Gamma)$

Rewrite Rule S1′α creates a new instance of class c. The new instance's Pr field is initialized as it was in Rewrite Rule S1, using an auxiliary *initialPr* function, except that we append a history assignment statement to the process to record an $[o.\textbf{initialized}]$ event when the initial process terminates. (In contrast, the other processes terminate with a **return** statement, which generates a reply event.) The Att, Asum, Guar, and ROAtt fields are initialized using auxiliary functions.

Rewrite Rule S1′β (Method Bootstrapping)

$\{$ **bootstrap method** $o.m@c'(\bar{v})$
with class c, **caller** o', **label** k, **history** h [, **attributes** $\bar{z} := \bar{w}$]
Γ $\}$
$\longrightarrow$
$\{$ $\langle o : c \mid$ Pr: S, LVar: β, Att: $[\mathcal{H} \mapsto h][\textbf{self} \mapsto o][\bar{z} \mapsto \bar{w}]$,
MsgQ: $replies(h, o)$, Asum: φ, Guar: ψ, ROAtt: $classROAtt(c, \Gamma)\rangle$
Γ $\}$
if $lwf(h, o)$
$\wedge$ h **bw** $[* \rightarrow o.\textbf{new}\ c(*)]$
$\wedge$ $mayAcquireProcessor(h, o, o', k)$
$\wedge$ $[o' \rightarrow o.m@c'(\bar{v})]^k$ **in** h
$\wedge$ $pending(h, o', o, k)$
$\wedge$ $\langle S, \beta \rangle := boundMtd(\textbf{if}\ c' = \textbf{none then}\ c\ \textbf{else}\ c'\ \textbf{fi}, o', k, m, \bar{v}, \Gamma)$
$\wedge$ $\langle \varphi, \psi \rangle := classAGSpec(c, \Gamma)$

Rewrite Rule S1′β also creates an object term but initializes it differently. The *boundMtd* function finds the method $m@c'$ and expands to the method's code and initial state, which are put into the Pr and LVar fields. The Att field is populated with the history and attributes specified in the **bootstrap** command.

The side condition ensures that the local history is well-formed, that it starts with a **new** event to create o as an instance of c, that it ends with an event that makes the processor available to the call, and that it contains the pending call to $m@c'$.

Example 4.11. Let us simulate the execution of a call to the *put* method on an instance of the *Buffer* class from Section 4.1. To restrict the activity of concurrent processes, we add the following guarantee to the *Buffer* class declaration:

$$\left.\begin{array}{l}\textbf{guar}\ \big(\forall j.\ \mathcal{H}/out_{\textbf{self}}\ \textbf{ew}\ [*{\leftarrow}\textbf{self}.put(j;)] \Rightarrow full \wedge value = j\big)\\ \qquad \wedge\ \big(\mathcal{H}/out_{\textbf{self}}\ \textbf{ew}\ [*{\leftarrow}\textbf{self}.get(;*)] \Rightarrow \neg full\big)\\ \qquad \wedge\ \mathcal{H}/out_{\textbf{self}} \preceq \big([*{\leftarrow}\textbf{self}.put(j;)] \frown [*{\leftarrow}\textbf{self}.get(;j)]\ \textbf{some}\ j\big)^*\end{array}\right\}\ \psi$$

The first conjunct states that if the last method that returned was *put*, then *full* is true and *value* stores the argument passed to *put*. The second conjunct states that if the last method that returned was *get*, then *full* is false. The third conjunct states that the replies for *put* and *get* alternate, and that for each *put*–*get* pair, the argument to *put* matches *get*'s return value. The notation $(\alpha_j\ \textbf{some}\ j)^*$ denotes the set of event sequences of the form $\alpha_{j_1} \ldots \alpha_{j_n}$, with $n \geq 0$ and $j_1, \ldots, j_n \in \mathbb{N}$ [JO02]. If H is a set of event sequences, then $h \preceq H$ if and only if there exists $h' \in H$ such that $h \preceq h'$.

To start the process, we must add a **bootstrap** command at the end of the program:

bootstrap method $B.put(2)$
with class *Buffer*, **caller** P, **label** 2, **history** h_0,
attributes *value*, *full* := 1, **true**

where

$$h_0 \equiv [M{\rightarrow}B.\textbf{new}\ Buffer()] \frown [P{\leftrightarrow}B.put(1)]^1.$$

Let $h_1 \equiv h_0 \frown [P{\rightarrow}B.put(2)]^2$. A possible execution for the $B.put(2)$ process follows.

bootstrap method $B.put(2)$
with class *Buffer*, **caller** P, **label** 2, **history** h_0,
attributes *value*, *full* := 1, **true**

$\xrightarrow{\text{S1}'\beta}$

$\langle B : Buffer \mid$ Pr: **await** $\neg full$; $value := x$; $full :=$ **true**; **return** ϵ,
Att: $[\mathcal{H} \mapsto h_1][\textbf{self} \mapsto B][value \mapsto 1][full \mapsto \textbf{true}]$,
LVar: $[\textbf{caller} \mapsto P][\textbf{label} \mapsto 2][x \mapsto 2]\rangle$

$\xrightarrow{\text{S14}'}$

$\langle B : Buffer \mid$ Pr: **await** $\neg full$; $value := x$; $full :=$ **true**; **return** ϵ,
Att: $[\mathcal{H} \mapsto h_1 \frown [B.\textbf{release}] \frown [C{\leftrightarrow}B.get(;1)]^1]$
$[\textbf{self} \mapsto B][value \mapsto 1][full \mapsto \textbf{false}]\rangle$

$\xrightarrow{\text{S8}}$

$\langle B : Buffer \mid$ Pr: $value := x$; $full :=$ **true**; **return** ϵ,
Att: $[\mathcal{H} \mapsto h_1 \frown [B.\textbf{release}] \frown [C{\leftrightarrow}B.get(;1)]^1]$
$[\textbf{self} \mapsto B][value \mapsto 1][full \mapsto \textbf{false}]\rangle$

$\xrightarrow{\text{S5, S2}}$

$$
\begin{array}{l}
\quad \big\langle B : \mathit{Buffer} \;\big|\; \text{Pr: } \mathit{full} := \mathbf{true};\ \mathbf{return}\ \epsilon, \\
\qquad\qquad\quad \text{Att: } [\mathcal{H} \mapsto h_1 \frown [B.\mathbf{release}] \frown [C \leftrightarrow B.\mathit{get}(;1)]^1] \\
\qquad\qquad\qquad\quad [\mathbf{self} \mapsto B][\mathit{value} \mapsto 2][\mathit{full} \mapsto \mathbf{false}]\big\rangle \\
\xrightarrow{\text{S5, S2}} \\
\quad \big\langle B : \mathit{Buffer} \;\big|\; \text{Pr: } \mathbf{return}\ \epsilon, \\
\qquad\qquad\quad \text{Att: } [\mathcal{H} \mapsto h_1 \frown [B.\mathbf{release}] \frown [C \leftrightarrow B.\mathit{get}(;1)]^1] \\
\qquad\qquad\qquad\quad [\mathbf{self} \mapsto B][\mathit{value} \mapsto 2][\mathit{full} \mapsto \mathbf{true}]\big\rangle \\
\xrightarrow{\text{S20}'} \\
\quad \big\langle B : \mathit{Buffer} \;\big|\; \text{Pr: } \epsilon, \\
\qquad\qquad\quad \text{Att: } [\mathcal{H} \mapsto h_1 \frown [B.\mathbf{release}] \frown [C \leftrightarrow B.\mathit{get}(;1)]^1] \frown [P \leftarrow B.\mathit{put}(2;)]^2] \\
\qquad\qquad\qquad\quad [\mathbf{self} \mapsto B][\mathit{value} \mapsto 2][\mathit{full} \mapsto \mathbf{true}]\big\rangle.
\end{array}
$$

Using Rewrite Rule S1′β, we instantiate a *Buffer* object set up to execute $\mathit{put}(2)$. Because the statement **await** $\neg\mathit{full}$ is not enabled (*full* is true), we suspend and reactivate the process using S14′. While the process was suspended, object C called *get* to empty the buffer. Without any knowledge of how *get* is implemented, the guarantee ψ forces *full* to be false immediately after *get* has returned. From there, we use S8 to execute the **await** statement, we apply S5 and S2 twice in a row to execute the assignments, and we finish with S20′ to handle the **return** statement. □

The approach presented here is adapted from Dovland et al. [DJO05], who devised an encoding of the Creol language, following de Boer and Pierik [dBP04]. This encoding serves essentially the same role as the open system semantics presented here. With our approach, the closed system and the open system operational semantics are very similar: Half of the rewrite rules are common to both semantics, and for the other half there is an almost one-to-one relationship between the rules of the two semantics. This makes it easier to detect inconsistencies between them, which would translate into unsoundness or incompleteness of the proof method.

4.5 Implementing the Semantics in Maude

The Maude specifications of the closed system and open system operational semantics presented in Section 4.3 and Section 4.4 are given in Appendix B. In the context of this thesis, these specifications fulfill two main purposes: First, they extend this chapter by providing a complete and unambiguous definition of Creol's semantics. Second, by defining basic sorts such as `Exp` and `Stmt`, they also form the basis of the assertion analyzer described in Chapter 6.

In this section, we will briefly review how to use these specifications to execute Creol programs, then we will discuss the main implementation decisions, comparing it with the implementation of other Creol interpreters.

To execute a Creol program with either of the interpreters, we use Maude's built-in `rew` or `frew` command on a term that represents the program, after having loaded the appropriate `CREOL-INTERPRETER-FOR-`x`-SYSTEMS` module ($x \in \{$`CLOSED`, `OPEN`$\}$). The term is expressed in a Maude-compatible format, with quoted identifiers, extra space around tokens, and square brackets instead of parentheses. For example, here is the *Producer* class declaration from Section 4.1 expressed in this format:

```
class 'Producer ['buf : 'WritableBuffer]
begin
   op 'run is
      var 'i : int ;
      'i := 1 ;
      while true do
         'buf . 'put['i ;] ;
         'i := 'i plus 1
      od
end
```

To execute the program using the closed semantics interpreter, we must supply a `bootstrap system` command. To execute the program using the open system interpreter, we must supply a `bootstrap object` or `bootstrap method` command.

For the open system interpreter, some of the rewrite rules presented in Section 4.4 are not directly executable in Maude and are declared with the `nonexec` attribute. This is because we model the nondeterministic behavior of the environment by introducing variables on the right-hand side of rewrite rules, which isn't supported by Maude's built-in execution strategies. To work around this, the Maude implementation provides alternative versions of the `nonexec` rewrite rules that take user-supplied "random" data provided using `random data` commands; see Sections A.1.4 and A.4.5 for details.

The Maude specification of the closed system interpreter follows as nearly as possible the conventions of this chapter and differs significantly from other versions of the interpreter in use at the University of Oslo. Here is a summary of its main distinctive features:

- The input to the interpreter is very close to standard Creol syntax. Since the interpreter handles the conversion of Creol syntax to a system configuration with < *i* `: Interface |` ... > and < *c* `: Class |` ... > terms, there is no need for a separate conversion tool (such as the one developed by Fjeld [Fje05]).
- The Maude sorts used to implement the interpreter have an intuitive, consistent syntax. For example, commas are used systematically to separate items in lists, and the empty list is represented by the term `epsilon`.
- The syntax of statements and expressions closely follows the Creol syntax. In particular, redundant parentheses are avoided by specifying operator precedences in Maude. To enhance readability, the infix operators on `bool` and `int` are left alone, instead of being converted to a generic prefix notation [Fje05].
- The Maude `Bool` and `Int` sorts can be used directly in Creol expressions and statements without any wrapper. This is achieved by making `Value` a supersort of `Bool` and `Int`, and allows us to write `'x := 5` instead of `'x := int(5)`. The drawback is that predefined `Bool` and `Int` operators like `and`, `or`, `not`, `+`, and $*$ cannot be used; instead, we must write `&&`, `||`, `!`, `plus`, and `times`.
- Creol parentheses are represented by square brackets to distinguish them from Maude parentheses. This makes it possible to spell out the semantics of Creol parentheses, and lets us implement the n-ary nondeterministic merge operator in a satisfactory manner in Maude, as discussed in Section 4.3.

- The interpreter uses a syntax inspired by Full Maude to avoid specifying unused fields in rewrite rules. This is achieved by representing the fields in an `< o : c | ... >` term by a multiset and by using an `ETC` variable to match the unspecified fields. With this approach, a rewrite rule with the left-hand side

  ```
  < O : C | Pr: abort ; S, ETC >
  ```

  will match the term

  ```
  < 'Main # 0 : 'Main | Pr: abort, LVar: ['x |-> 1],
                        Att: [self |-> 'Main # 0],
                        PrQ: emptyMset, MsgQ: emptyMset,
                        LabCnt: 2 >
  ```

- The interpreter can be used together with Maude's built-in metaprogramming facilities. Maude's `upTerm` and `downTerm` partial functions convert a term to and from the metalevel. This is necessary to define custom execution strategies, which are often needed for testing or verifying Creol programs. To avoid any clashes with the `META-LEVEL` module, the interpreter uses as little of the standard `prelude.maude` file as possible. In particular, it avoids the `Qid` sort and instead hooks directly into Maude to define its own quoted identifier sort (`QuotedId`).

- The interpreter uses the global configuration pattern suggested by Ölveczky [Ölv07] to initialize objects and bind methods. Other versions of the Creol interpreter use messages and equations instead [JO07]. The latter approach works with Maude's built-in strategies, which apply equations before rewrite rules, but it violates the coherence requirement mentioned in section 5.3 of the Maude 2.3 manual [CDEL+07].

- Because the interpreter's main purpose is to define the semantics of Creol, the emphasis has been on the simplicity, readability, and correctness of the interpreter, not on its efficiency. In particular, there is no garbage collection to eliminate superfluous reply messages; on the other hand, the interpreter correctly resolves qualified and unqualified attributes in the presence of single or multiple inheritance.

Some of these ideas are expected to find their way into the official Creol interpreter in use at the University of Oslo.

> Fifteen years ago computer programming was so badly understood that hardly anyone ever *thought* about proving programs correct; we just fiddled with a program until it appeared to work.
>
> — Donald E. Knuth (1974)

Chapter 5

A Compositional Proof System for Creol

This chapter presents a proof system for reasoning about Creol programs originally developed by Dovland, Johnsen, and Owe [DJO05, DJO08]. The proof system is based on Hoare logic [Hoa69, Apt81, Apt84], which lets us reason directly on a program's text using axioms and proof rules. It also incorporates later developments that make it possible to analyze open distributed systems, such as the assume–guarantee paradigm [Jon81, MC81] and the concept of a communication history [Dah77].

Hoare logic was originally designed to verify procedure input–output relationships: establishing, for example, that a given *gcd* procedure computes the greatest common divisor of two integers, that a *bsearch* procedure performs a binary search, or that *quicksort* really sorts the items of an array. Unfortunately, such proofs are tedious to write, and unless they are verified by a computer, they are likely to contain more bugs than the program of interest [Knu02]. Incidentally, most programs do have bugs, and therefore are impossible to prove correct with respect to their specification; this lead Dijkstra and others to promote a programming style where the program and its proof are developed together [Dij75, Kal90].

With the rise of concurrent programming in the late 1970s and the 1980s, Hoare logic found a new field of application. Concurrent programs are notoriously difficult to study at the semantic level, because of their inherent nondeterminism. Indeed, the history of concurrent programming research is paved with published algorithms that are flawed and for which there exist very convincing semiformal arguments known as "behavioral proofs."[1] Using Hoare logic, we can verify that a certain property (data consistency, mutual exclusion, etc.) holds throughout the execution of a program. Hoare logic is also useful to verify that certain undesirable

[1] After presenting the first concurrent garbage collection algorithm, Dijkstra, Lamport, and their colleagues concluded: "It has been surprisingly hard to find the published solution and justification. It was only too easy to design what looked—sometimes even for weeks and to many people—like a perfectly valid solution, until the effort to prove it to be correct revealed a (sometimes deep) bug" [DLMSS78]. Lamport remarked in retrospect: "The lesson I learned from this is that behavioral proofs are unreliable and one should always use state-based reasoning for concurrent algorithms—that is, reasoning based on invariance" [Lam08].

situations cannot occur. In sequential programs, we want to avoid run-time errors (**abort**, method call on **null**, division by zero) and infinite loops. In a concurrent setting, we must also beware of *deadlocks*, which occur when several processes are blocked waiting for each other to proceed.

Section 5.1 introduces Hoare logic and covers basic Creol statements such as assignment, sequential composition, **if**, and **while**. Section 5.2 presents Hoare axioms and proof rules for the more advanced Creol statements that involve the communication history. Section 5.3 reviews the main proof strategies that can be used for Hoare logic. Section 5.4 recasts the Hoare axioms and proof rules to a backward-constructive style based on weakest liberal preconditions (WLPs). Section 5.5 explains how to use WLPs to verify the guarantee supplied in the **guar** clause of a Creol class and of its interfaces. Section 5.6 shows how to combine assume–guarantee specifications together to verify a complex system. Finally, Section 5.7 summarizes the contributions of this thesis to the proof system.

5.1 Local Reasoning with Hoare Logic

Hoare-style reasoning at the statement level constitutes the cornerstone of the proof system presented in this chapter. The fundamental question which Hoare logic tries to answer is, "If a statement starts executing in a certain state, what are the possible states at termination?" The statement may be a single statement or a list of statements. To express the answer to this question concisely, we use a *partial correctness formula* of the form

$$\{P\}\, S\, \{Q\},$$

with $P, Q \in \mathit{Assn}$ and $S \in \mathit{Stmt}$. Informally, the formula has the following meaning:

> *If the precondition P holds before S is executed and the execution terminates normally, then the postcondition Q holds at termination.*

Notice that the formula doesn't affirm that the statement S will not abort or loop forever; it only states that if S does terminate normally, then Q will hold at that point. In this context, it is useful to see an assertion P as a set of states σ such that the assertion holds in that state. Assuming a and b are the only two declared variables and are of type **int**, the assertion $a = 2$ describes the set $\{[a \mapsto 2][b \mapsto n] \mid n \in \mathbb{Z}\}$, and similarly $a = 2 \wedge b = 4$ describes the singleton $\{[a \mapsto 2][b \mapsto 4]\}$. Following the practice of logicians, we will write $\sigma \models P$ rather than $\sigma \in P$ to express that P holds in the state σ.

To express the stronger requirement of termination, we must resort to a *total correctness formula*

$$\{P\}\, S\, \{Q\}_{\mathrm{tot}},$$

which can be interpreted as follows:

> *If the precondition P holds before S is executed, then the execution terminates normally and the postcondition Q holds at termination.*

Total correctness is normally what we aim at when writing a program. Indeed, using the weaker criterion of partial correctness, the programs $x := 1/0$, **abort**, and **while true do skip od** are all valid sort algorithms, even though they never produce useful results.

In practice, total correctness is significantly more difficult to prove than partial correctness. To establish the total correctness of a program, we usually start by proving the program partially correct, then we show the absence of run-time errors, infinite loops, and deadlocks. In this thesis, we will focus on partial correctness and use the unqualified word "correctness" in that sense. For a thorough review of total correctness of concurrent programs, refer to Apt and Olderog [AO97] or de Roever et al. [dRdB$^+$01].

Here are a few examples of valid correctness formulas and their interpretation:

i. $\{\mathbf{true}\}\ b := 4\ \{b = 4\}$

   If we start in an arbitrary state and execute $b := 4$, the condition $b = 4$ holds in the final state.

ii. $\{a = 2\}\ b := 2 * a\ \{a = 2 \wedge b = 4\}$

   If we start in a state where $a = 2$ holds and execute $b := 2 * a$, the condition $a = 2 \wedge b = 4$ holds in the final state.

iii. $\{b \geq 0\}\ b := b + 1\ \{b \geq 1\}$

   If we start in a state where $b \geq 0$ holds and execute $b := b + 1$, the condition $b \geq 1$ holds in the final state.

iv. $\{\mathbf{false}\}\ \mathbf{skip}\ \{b = 100\}$

   If we start in an impossible state and execute **skip**, we may assume anything about the final state, including that $b = 100$ holds.

v. $\{\mathbf{true}\}\ \mathbf{while}\ i \neq 0\ \mathbf{do}\ i := i - 1\ \mathbf{od}\ \{i = 0\}$

   If we start in an arbitrary state and execute **while** $i \neq 0$ **do** $i := i - 1$ **od**, the condition $i = 0$ will hold in the final state.

Formula iv may seem counterintuitive, but since no state satisfies the precondition, the formula effectively asserts nothing at all. In a way, the formula is valid for the same reason that the assertion **false** $\Rightarrow b = 100$ holds for any value of b.

Formula v illustrates the difference between partial and total correctness. Clearly, if $i < 0$ holds before entering the loop, the loop will never terminate. However, the partial correctness formula doesn't express that; it only asserts that if the loop terminates, $i = 0$ will hold. In contrast, in a total correctness setting, the formula $\{\mathbf{true}\}\ \mathbf{while}\ i \neq 0\ \mathbf{do}\ i := i - 1\ \mathbf{od}\ \{i = 0\}_{\text{tot}}$ isn't valid, because it would imply that the loop always terminates, which clearly isn't the case. To make the total correctness formula valid, we must strengthen the precondition:

$$\{i \geq 0\}\ \mathbf{while}\ i \neq 0\ \mathbf{do}\ i := i - 1\ \mathbf{od}\ \{i = 0\}_{\text{tot}}.$$

It would have been equally correct, but needlessly restrictive, to specify $i \geq 10$, $i = 100$, or **false** as the precondition.

Paradoxically, while partial correctness cannot express termination, it lets us express nontermination. For example, the formula

$$\{i < 0\}\ \textbf{while}\ i \neq 0\ \textbf{do}\ i := i - 1\ \textbf{od}\ \{\textbf{false}\}$$

states that if the **while** loop starts in a state where $i < 0$, either it will end in an impossible state, or it won't terminate normally. Since we can exclude the possibility of reaching an impossible state, we conclude that the loop will run forever.

Our intuition tells us that formulas i to v are valid, but how can we justify it formally? One answer is: using the operational semantics of Section 4.4. More precisely, we must consider an arbitrary object $\langle o : c \mid \ldots \rangle$ in a state that satisfies P, simulate all possible executions of the statement S, and verify that Q holds in the final state for all of these.

Let us try to prove the partial correctness formula $\{a = 2\}\ b := 2 * a\ \{a = 2 \wedge b = 4\}$ using the operational semantics. For simplicity, we will assume that a and b are local variables; a similar argument could be made for attributes. Starting in a state where $a = 2$, we need to consider only one execution:

$$\begin{aligned} &\xrightarrow{\text{S5}} \langle o : c \mid \text{Pr: } b := 2 * a;\ S,\ \text{LVar: } \beta[a \mapsto 2] \rangle \\ &\quad\ \langle o : c \mid \text{Pr: } S,\ \text{LVar: } \beta[a \mapsto 2][b \mapsto 4] \rangle. \end{aligned}$$

Clearly, the postcondition $a = 2 \wedge b = 4$ holds in the final configuration. In general, we would need to consider Rewrite Rule S25′ (Parallel Activity), but we can ignore it here since it has no impact on a and b.

To make this argument more satisfactory, we must give a formal interpretation of correctness formulas based on the operational semantics. A partial correctness formula $\{P\}\ S\ \{Q\}$ is valid if and only if $\alpha'\beta' \models Q$ for all executions of the form

$$\begin{aligned} &\xrightarrow{*} \langle o : c \mid \text{Pr: } S;\ S',\ \text{LVar: } \beta,\ \text{Att: } \alpha \rangle \\ &\quad\ \langle o : c \mid \text{Pr: } S',\ \text{LVar: } \beta',\ \text{Att: } \alpha' \rangle, \end{aligned}$$

where $\alpha\beta \models P$. The conditions P and Q may refer to any local variable and object attribute, but not to $\mathcal{H}$. (In the next section, we will see how to handle $\mathcal{H}$.)

We have proved $\{a = 2\}\ b := 2 * a\ \{a = 2 \wedge b = 4\}$, but at what cost? Imagine verifying an entire Creol program by reasoning in terms of the language's operational semantics. This would require pages upon pages of executions accompanied by semiformal justifications.

This is where Hoare logic comes into play. Hoare logic is a framework for deriving valid correctness formulas in a mechanical way, using a set of axioms and proof rules. It allows us to reason directly on the program's syntax, without concerning ourselves with the operational semantics. The approach is mechanical in the sense that the applicability of an axiom or a proof rule can easily be checked using a computer or manually.

The goal of Hoare logic is to derive valid formulas. If a Hoare logic derives only valid formulas, we say that the logic is *sound*; and if it can be used to derive all valid formulas, we say that it is *complete*. Soundness is nonnegotiable; an unsound Hoare logic is comparable to a calculator that reports $1 + 1 = 3$ and should not be

used. Completeness is desirable, because without it some correct programs cannot be proved correct using Hoare logic.

Like most object-oriented programming languages, Creol comprises an imperative subset consisting of statements that produce side effects or affect the control flow. In addition, Creol offers a few statements that are relevant only in a concurrent or distributed context, such as **await**, $\Box$, and $|||$. In this section, we will review the Hoare axioms and proof rules that are shared by most imperative languages; in Section 5.2, we will consider the axioms and proof rules that apply specifically to Creol. Except when noted otherwise, the axioms and proof rules presented here are believed to be sound and complete.[1]

Axiom P1 (Null Statement)

$$\{Q\}\ \mathbf{skip}\ \{Q\}$$

If $\alpha\beta$ is a possible state before executing **skip**, then $\alpha\beta$ is a possible state after executing **skip**, and vice versa. Strictly speaking, Axiom P1 is not a single axiom but rather an axiom schema that allows us to derive an infinity of axioms, including $\{\mathbf{false}\}\ \mathbf{skip}\ \{\mathbf{false}\}$, $\{\mathbf{true}\}\ \mathbf{skip}\ \{\mathbf{true}\}$, and $\{x = 5\}\ \mathbf{skip}\ \{x = 5\}$.

Axiom P2 (Abnormal Termination)

$$\{\mathbf{true}\}\ \mathbf{abort}\ \{\mathbf{false}\}$$

If we start in an arbitrary state and execute an **abort** statement, the statement won't terminate normally.

Axiom P3 (Assignment)

$$\{Q^{\bar{z}}_{\bar{e}}\}\ \bar{z} := \bar{e}\ \{Q\}$$

Axiom P3 works backward: It asks for a postcondition Q that we want to hold after the assignment, and gives us back a precondition $P \equiv Q^{\bar{z}}_{\bar{e}}$. The notation $Q^{\bar{z}}_{\bar{e}}$ denotes the assertion Q where all free occurrences of $\bar{z}$ have been replaced by $\bar{e}$. (If a free variable in $\bar{e}$ would become bound as a result of the substitution, the bound variables in φ are renamed to avoid clashing.) For example, if

$$Q \equiv a \leq b; \qquad \bar{z} \equiv a, b; \qquad \bar{e} \equiv 2, 4;$$

we obtain $Q^{\bar{z}}_{\bar{e}} \equiv (a \leq b)^{a,b}_{2,4} \equiv 2 \leq 4$.

Axiom P3 may seem counterintuitive at first glance, but it correctly captures the semantics of the assignment statement, as illustrated by the following examples:

i. $\{0 = 0\}\ x := 0\ \{x = 0\}$

ii. $\{0 = 0 \wedge y = 5\}\ x := 0\ \{x = 0 \wedge y = 5\}$

iii. $\{x + 1 \geq 5\}\ x := x + 1\ \{x \geq 5\}$

[1] It is sobering to read that in the experience of Willem-Paul de Roever, "*every* alleged proof method for concurrency, which reached his desk and had not been proven complete, turned out to be incomplete, and every such proof method, which had not been proven sound, turned out to be unsound" [dRdB+01].

iv. $\{y = 2 \wedge x = 4\}\, x, y := y, x\, \{x = 2 \wedge y = 4\}$.

Using elementary number theory, we can simplify the preconditions; for example, $0 = 0$ is equivalent to **true** and $x + 1 \geq 5$ is equivalent to $x \geq 4$.

Because Creol objects cannot access each other's attributes directly, Axiom P3 is sound even in the presence of reference aliasing [dBP04]. On the other hand, the axiom schema will fail if we mix qualified and unqualified accesses to attributes, because the substitution operator $Q^{\bar{z}}_{\bar{e}}$ cannot know whether $x@c$ and x are the same variable. To avoid aliasing issues as well as potential name clashes with local variables, we will assume that the code systematically uses the qualified syntax $x@c$ to refer to attributes. In practice, this can easily be achieved by preprocessing the Creol program.

Axiom P3 implicitly assumes that the expressions $\bar{e}$ have no side effects and are totally defined. What about division by zero? An expedient adopted by Apt and Olderog [AO97] is to make division a total function by requiring that $x/0 = 0$ (a decision that was criticized by Lamport and Paulson [LP99]). In contrast, Dahl [Dah92] provides a more thorough treatment of partial functions, at the cost of more complex Hoare axioms and proof rules, and Owe addresses the issue in the underlying predicate logic [Owe93]. For this thesis, Apt and Olderog's expedient of letting $x/0 = 0$ is acceptable, since proving the absence of run-time errors is beyond the scope of partial correctness.

There is one more twist to the assignment axiom. Suppose we want to study the statement $x := x + 1$ in isolation. We would like to state that it increments x by 1, but how can we express this as a $\{P\}\, S\, \{Q\}$ correctness formula? Attempts like $\{x + 1 = x + 1\}\, x := x + 1\, \{x = x\}$ and $\{x = x\}\, x := x + 1\, \{x = x + 1\}$ lead to tautologies or contradictions. The solution is to introduce a logical variable for reasoning purposes. Such variables must not appear in the program itself. Using this technique, we would characterize the effect of $x := x + 1$ by introducing x_0 to freeze the initial state of x:

$$\{x = x_0\}\, x := x + 1\, \{x = x_0 + 1\}.$$

Once again, we have simplified the precondition using elementary number theory. In a formal system such as Hoare logic, altering formulas like this is not permitted unless we can justify it using formal rules. However, our intuition tells us that if $\{P\}\, S\, \{Q\}$ is valid, then $\{P'\}\, S\, \{Q'\}$ is also valid for $P' \Leftrightarrow P$ and $Q' \Leftrightarrow Q$. Proof Rule P6, presented shortly, will formalize a more general result.

Proof Rule P4 (Parenthesized Statement)

$$\frac{\{P\}\, S\, \{Q\}}{\{P\}\, (S)\, \{Q\}}$$

In a formal proof system, a proof rule lets us derive new formulas, called theorems, from existing axioms and theorems. Using Proof Rule P4, if $\{P\}\, S\, \{Q\}$ is provable using Hoare logic, then we can also derive $\{P\}\, (S)\, \{Q\}$.

Example 5.1. To prove $\{\textbf{true}\}\, (b := 4)\, \{b = 4\}$, we first prove $\{\textbf{true}\}\, b := 4\, \{b = 4\}$ using Axiom P3; then we invoke Proof Rule P4. The proof can be represented by a proof tree:

$$\dfrac{\dfrac{}{\{\mathbf{true}\}\, b := 4\, \{b = 4\}}\,\text{P3}}{\{\mathbf{true}\}\, (b := 4)\, \{b = 4\}}\,\text{P4}$$

The formula to prove appears at the bottom of the tree. □

Proof Rule P5 (Sequential Composition)

$$\dfrac{\{P\}\, S_1\, \{R\} \quad \{R\}\, S_2\, \{Q\}}{\{P\}\, S_1;\, S_2\, \{Q\}}$$

The proof rule for sequential composition allows us to combine existing proofs about S_1 and S_2 to build a proof about S_1; S_2. Intuitively, the rule reads as follows: If executing S_1 from a state that satisfies P leaves the program in a state that satisfies R, and executing S_2 from a state that satisfies R leaves the program in a state that satisfies Q, we may conclude that executing S_1; S_2 in a state where P holds will lead to a state where Q holds.

Example 5.2. The following proof tree shows that $\{a = 2\}\, b := a;\, c := b\, \{c = 2\}$ is a theorem:

$$\dfrac{\dfrac{}{\{a = 2\}\, b := a\, \{b = 2\}}\,\text{P3} \quad \dfrac{}{\{b = 2\}\, c := b\, \{c = 2\}}\,\text{P3}}{\{a = 2\}\, b := a;\, c := b\, \{c = 2\}}\,\text{P5}$$

□

Correctness formulas for general statement lists S_1; S_2; …; S_n can be derived using $n - 1$ applications of Proof Rule P5. Since sequential composition is associative, we are free to regard S_1; S_2; S_3 as either $(S_1;\, S_2)$; S_3 or S_1; $(S_2;\, S_3)$. Here is a sketch of one possible proof of $\{P\}\, S_1; \ldots; S_n\, \{Q\}$:

$$\dfrac{\{P\}\, S_1\, \{R\} \quad \dfrac{\{R\}\, S_2\, \{R'\} \quad \dfrac{\{R'\}\, S_3\, \{R''\} \quad \dfrac{\text{etc.}}{\{R''\}\, S_4; \ldots; S_n\, \{Q\}}\,\text{P5}}{\{R'\}\, S_3; \ldots; S_n\, \{Q\}}\,\text{P5}}{\{R\}\, S_2; \ldots; S_n\, \{Q\}}\,\text{P5}}{\{P\}\, S_1; \ldots; S_n\, \{Q\}}\,\text{P5}$$

To complete the proof, we would derive $\{P\}\, S_1\, \{R\}$, $\{R\}\, S_2\, \{R'\}$, …, using axioms and proof rules.

Proof Rule P6 (Consequence)

$$\dfrac{P \Rightarrow P' \quad \{P'\}\, S\, \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\, S\, \{Q\}}$$

The rule of consequence lets us strengthen a precondition, weaken a postcondition, or both. We often need this rule to compose statements together when their pre- or postconditions don't match exactly. The first and third hypotheses are first-order logic assertions that must be proved using the sequent calculus LK of Section 2.2 or any other proof system for first-order logic.

Example 5.3. The following tree shows how to derive $\{x \geq 10\}\, y := x\, \{y \geq 0\}$ from the axiom $\{x \geq 5\}\, y := x\, \{y \geq 5\}$:

$$\dfrac{x \geq 10 \Rightarrow x \geq 5 \quad \dfrac{}{\{x \geq 5\}\, y := x\, \{y \geq 5\}}\,\text{P3} \quad y \geq 5 \Rightarrow y \geq 0}{\{x \geq 10\}\, y := x\, \{y \geq 0\}}\,\text{P6}$$

To complete the proof, the implications $x \geq 10 \Rightarrow x \geq 5$ and $y \geq 5 \Rightarrow y \geq 0$ should be proved in first-order logic, with appropriate antecedents that specify the algebraic properties of integers. □

Example 5.4. The following proof tree shows how to derive $\{\mathbf{true}\}\, b := 4;\, b := b + 1\, \{b = 5\}$:

$$\dfrac{\dfrac{\dfrac{}{\{4 = 4\}\, b := 4\, \{b = 4\}}\,\text{P3}}{\{\mathbf{true}\}\, b := 4\, \{b = 4\}}\,\text{P6} \qquad \dfrac{\dfrac{}{\{b + 1 = 5\}\, b := b + 1\, \{b = 5\}}\,\text{P3}}{\{b = 4\}\, b := b + 1\, \{b = 5\}}\,\text{P6}}{\{\mathbf{true}\}\, b := 4;\, b := b + 1\, \{b = 5\}}\,\text{P5}$$

For brevity, we omitted the $P \Rightarrow P'$ and $Q' \Rightarrow Q$ premises when invoking the rule of consequence. To make the proof entirely formal, we would need to prove the following assertions:

$$\mathbf{true} \Rightarrow 4 = 4, \qquad b = 4 \Rightarrow b + 1 = 5,$$
$$b = 4 \Rightarrow b = 4, \qquad b = 5 \Rightarrow b = 5.$$ □

Notice that in the last example, we did not actually strengthen the preconditions or weaken the postconditions. Rather, we used the rule of consequence to replace assertions with logically equivalent ones.

Proof Rule P7 (Inline Assertion)

$$\dfrac{Q \Rightarrow P}{\{Q\}\, \mathbf{prove}\, P\, \{Q\}}$$

The **prove** statement lets the programmer specify an assertion P that must be true at a specific point during the program's execution. Since the statement does nothing, the precondition and the postcondition are identical. To ensure that programs containing wrong assertions cannot be proved correct, we require that the precondition implies P.

Proof Rule P8 (If Statement)

$$\dfrac{\{P_1\}\, S_1\, \{Q\} \quad \{P_2\}\, S_2\, \{Q\}}{\{\mathbf{if}\ B\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2\ \mathbf{fi}\}\ \mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}\, \{Q\}}$$

Proof Rule P8 for **if** statements may be interpreted as follows: If B is true, S_1's precondition must hold before executing S_1; otherwise, S_2's precondition must hold before executing S_2. By combining these two cases, we obtain the precondition **if** B **then** P_1 **else** P_2 **fi**, which may also be written $(B \wedge P_1) \vee (\neg B \wedge P_2)$ or even $(B \Rightarrow P_1) \wedge (\neg B \Rightarrow P_2)$.

Notice how the Boolean expression B escapes the programming language and enters the assertion language. This works because our assertion language *Assn* includes *BExp*, the set of Boolean expressions.

Example 5.5. To derive the formula $\{\mathbf{true}\}\ \mathbf{if}\ x < 0\ \mathbf{then}\ x := -x\ \mathbf{fi}\ \{x \geq 0\}$, we must first expand the statement to its canonical form as per Definition T3 of Section 4.3. Then we proceed as follows:

$$\dfrac{\dfrac{\dfrac{}{\{-x \geq 0\}\, x := -x\, \{x \geq 0\}}\,\text{P3} \quad \dfrac{}{\{x \geq 0\}\ \mathbf{skip}\ \{x \geq 0\}}\,\text{P1}}{\{\mathbf{if}\ x < 0\ \mathbf{then}\ -x \geq 0\ \mathbf{else}\ x \geq 0\ \mathbf{fi}\}\ \mathbf{if}\ \ldots\ \mathbf{fi}\ \{x \geq 0\}}\,\text{P8}}{\{\mathbf{true}\}\ \mathbf{if}\ x < 0\ \mathbf{then}\ x := -x\ \mathbf{else\ skip\ fi}\ \{x \geq 0\}}\,\text{P6}$$

Again, we omitted the $P \Rightarrow P'$ and $Q' \Rightarrow Q$ premises when invoking the rule of consequence. To make the proof entirely formal, we would also prove the following assertions:

$$\mathbf{true} \Rightarrow \mathbf{if}\ x < 0\ \mathbf{then}\ -x \geq 0\ \mathbf{else}\ x \geq 0\ \mathbf{fi},$$
$$x \geq 0 \Rightarrow x \geq 0.$$

□

Proof Rule P9 (While Loop)

$$\dfrac{\{I \wedge B\}\, S\, \{I\}}{\{I\}\ \mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}\ \{I \wedge \neg B\}}$$

When using the **while** loop rule, we must supply an invariant I that holds before entering the loop and after every iteration of the loop. We may assume that B holds just before S is executed, and that it doesn't hold when the loop terminates.

Example 5.6. Consider the following program fragment, which computes the sum of the natural numbers up to n:

```
i := 0;
s := 0;
while i < n do
    i := i + 1;
    s := s + i
od
```

Suppose we want to prove that $s = n * (n+1)/2$ holds when the loop terminates, under the assumption that $n \geq 0$. We can use $s = S_i \wedge i \leq n$ as the loop invariant, with $S_i \equiv i * (i+1)/2$. This would give rise to the following proof tree, in which I denotes the loop invariant:

$$\dfrac{\dfrac{\dfrac{\{I \wedge i < n\}\, i := i+1\, \{s = S_{i-1} \wedge i \leq n\} \quad \{s = S_{i-1} \wedge i \leq n\}\, s := s + i\, \{I\}}{\{I \wedge i < n\}\, i := i+1;\, s := s+i\, \{I\}}\,\text{P5}}{\{I\}\ \mathbf{while}\ \ldots\ \mathbf{od}\ \{I \wedge i \not< n\}}\,\text{P9}}{\{n \geq 0 \wedge i = 0 \wedge s = 0\}\ \mathbf{while}\ \ldots\ \mathbf{od}\ \{s = n * (n+1)/2\}}\,\text{P6}$$

The two hypotheses at the top of the tree can be proved using the assignment axiom and the rule of consequence. □

So far, we have used proof trees to show that a formula is a theorem. In practice, we usually prefer *proof outlines*, where the program text is systematically annotated with assertions. Here is the proof outline for the **while** loop of Example 5.6:

$\{n \geq 0 \wedge i = 0 \wedge s = 0\}$
$\{I\}$
while $i < n$ **do**
  $\{I \wedge i < n\}$
  $i := i + 1;$
  $\{s = S_{i-1} \wedge i \leq n\}$
  $s := s + i$
  $\{I\}$
od
$\{I \wedge i \not< n\}$
$\{s = n * (n+1)/2\}$

In the proof outline, each statement is bracketed by its precondition and its post-condition. Consecutive assertions $\{P\}\{P'\}$ before a statement or $\{Q'\}\{Q\}$ after a statement make applications of the consequence rule explicit; the associated proof obligation is $P \Rightarrow P'$ or $Q' \Rightarrow Q$.

From a proof tree it is easy to produce a proof outline, and vice versa. Proof outlines are less explicit, because we don't label them with the axioms and proof rules that were used to derive them, and because we often omit some assertions; but since they follow the structure of the original program, they also tend to be both more concise and more readable. In fact, using **prove** and **inv**, the programmer can already provide the backbone of a proof outline:

prove $n \geq 0 \wedge i = 0 \wedge s = 0;$
inv $s = i * (i+1)/2 \wedge i \leq n$
while $i < n$ **do**
  $i := i + 1;$
  prove $s = (i-1) * i/2 \wedge i \leq n;$
  $s := s + i$
od;
prove $s = n * (n+1)/2$

Proof Rule P10 (While Loop with Invariant Clause)

$$\frac{I' \Rightarrow I \quad \{I' \wedge B\}\, S\, \{I'\}}{\{I'\}\ \textbf{inv}\ I\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{I' \wedge \neg B\}}$$

If the programmer has specified an invariant I using the **inv** clause, we can choose $I' \equiv I$ as the invariant, or if I is too weak, we can strengthen it by providing an invariant I' that implies I.

5.2 Hoare Axioms and Proof Rules for the Creol-Specific Statements

The Hoare axioms and proof rules presented in Section 5.1 cover Creol's sequential subset and are shared by most imperative programming languages. We will now look at the statements that are specific to Creol: object creation, asynchronous invoc-

ation, asynchronous reply, conditional wait, nondeterministic choice, and nondeterministic merge. Since they involve unbounded nondeterminism, it will be beneficial to begin by studying this kind of nondeterminism in its simplest form.

For the sake of illustration, we will consider the language SEQ [DJO05] that consists of Creol's sequential subset extended with the random assignment statement

$$\bar{z} := \bar{e} \textbf{ for some } \bar{x} \textbf{ such that } P$$

This statement is similar to the standard assignment statement $\bar{z} := \bar{e}$, except that the expressions $\bar{e}$ may also refer to logical variables $\bar{x}$ that take random values respecting the assertion P. (If no such values exist, the statement does not terminate normally.) For example,

$$i, j := +x, -y \textbf{ for some } x, y \textbf{ such that } x > 0 \wedge y > 0$$

assigns a random positive value to i and a random negative value to j. Axiom P11 captures the semantics of the random assignment statement:

Axiom P11 (Random Assignment)

$$\{\forall \bar{x}.\ P \Rightarrow Q^{\bar{z}}_{\bar{e}}\}\ \bar{z} := \bar{e} \textbf{ for some } \bar{x} \textbf{ such that } P\ \{Q\} \qquad \text{for fresh variables } \bar{x}$$

Intuitively, the postcondition Q must hold after the random assignment if $Q^{\bar{z}}_{\bar{e}}$ held before for any values $\bar{x}$ that satisfy P. The side condition ensures that no occurrences of $\bar{x}$ in P or Q are accidentally bound by the universal quantifier.

The preconditions obtained using Axiom P11 can often be simplified drastically, as illustrated by the examples below:

i. $\{\overbrace{\forall x.\ \textbf{true} \Rightarrow j = 5}^{j=5}\}\ i := x \textbf{ for some } x \textbf{ such that } \textbf{true}\ \{j = 5\}$

ii. $\{\overbrace{\forall x.\ x = 2 \Rightarrow x = 2}^{\textbf{true}}\}\ i := x \textbf{ for some } x \textbf{ such that } x = 2\ \{i = 2\}$

iii. $\{\overbrace{\forall x, y.\ x > y \Rightarrow x > y}^{\textbf{true}}\}\ i, j := x, y \textbf{ for some } x, y \textbf{ such that } x > y\ \{i > j\}$

iv. $\{\overbrace{\forall x.\ x > j \Rightarrow x > 5}^{j \geq 5}\}\ i := x \textbf{ for some } x \textbf{ such that } x > j\ \{i > 5\}$

v. $\{\overbrace{\forall y.\ y > j \Rightarrow y > 5}^{j \geq 5}\}\ j := y \textbf{ for some } y \textbf{ such that } y > j\ \{j > 5\}.$

Formula v shows that the axiom schema is sound when the condition P refers to the variables $\bar{z}$ that appear on the left-hand side of the assignment: "If $j \geq 5$ and we assign to j a value that is greater than its current value, then $j > 5$ holds afterward."

In the operational semantics of Section 4.4, Rewrite Rule S25′ (Parallel Activity) is an important source of nondeterminism, because it can be invoked at any point to extend the history variable $\mathcal{H}$ with an event originating from another object. The axioms and proof rules presented so far don't take this into account; for example, Axiom P1 lets us derive

$$\{\mathcal{H} \textbf{ ew } [\textbf{self} \rightarrow \mathit{buf}.\textbf{new } \mathit{Buffer}]\}\ \textbf{skip}\ \{\mathcal{H} \textbf{ ew } [\textbf{self} \rightarrow \mathit{buf}.\textbf{new } \mathit{Buffer}]\},$$

even though $\mathcal{H}$ might have been extended while **skip** was executing. If we take into account the nondeterministic extension of the history, a **skip** statement is (from a local point of view) effectively equivalent to

$$\mathcal{H} := h \textbf{ for some } h \textbf{ such that } \mathit{interleave}(\mathcal{H}, h)$$

where $\mathit{interleave}(h, h')$ is defined below.

Definition Q1 (Parallel Activity Interleaving Assertion)

$$\begin{aligned}\mathit{interleave}(h, h') \triangleq\ & \mathit{lwf}(h', \textbf{self}) \\ & \wedge\ h \preceq h' \\ & \wedge\ h/(\mathit{out}_{\textbf{self}} \cup \mathit{ctl}_{\textbf{self}}) = h'/(\mathit{out}_{\textbf{self}} \cup \mathit{ctl}_{\textbf{self}}) \\ & \wedge\ (\mathcal{A}_c)^{\mathcal{H}}_{h'}\end{aligned}$$

The $\mathit{interleave}(h, h')$ assertion is primarily a syntactic reformulation of the predicate $\mathit{interleave}(\varphi, o, \alpha, h)$ from Definition T25 in Section 4.4. The symbol $\mathcal{A}_c$ denotes the class assumption. The last conjunct expresses that the assumption should hold after the history extension.

Using Axiom P11, we would derive the following axiom schema for **skip**:

$$\{\forall h.\ \mathit{interleave}(\mathcal{H}, h) \Rightarrow Q^{\mathcal{H}}_h\}\ \textbf{skip}\ \{Q\} \qquad \text{for any fresh variable } h.$$

The same could be done for the other statements considered in the previous section, but it would seriously burden the Hoare logic. Fortunately, there is an alternative. Dovland et al. [DJO05] pointed out that it is sufficient to consider parallel activity only when analyzing statements that access the history variable $\mathcal{H}$. A statement such as **skip** does not access the history, so it is immaterial whether the environment has sent a message to **self** immediately before **skip** has executed or after.

To account for parallel activity, we therefore propose a new criterion for the validity of a formula: A partial correctness formula $\{P\}\ S\ \{Q\}$ is valid if and only if $\alpha'\beta' \models Q$ for all executions of the form

$$\begin{aligned}\xrightarrow{*}\ & \langle o : c \mid \text{Pr: } S; S',\ \text{LVar: } \beta,\ \text{Att: } \alpha\rangle \\ & \langle o : c \mid \text{Pr: } S',\ \text{LVar: } \beta',\ \text{Att: } \alpha'\rangle,\end{aligned}$$

where $\alpha\beta \models P$ and Rewrite Rule S25′ is only applied immediately before Rewrite Rules S8–S11, S13′, S14′, S16′, S17′, S22, or S23′—namely, the rules that involve the history variable $\mathcal{H}$ or the MsgQ field. The conditions P and Q may refer to any local variable and object attribute, *including* $\mathcal{H}$. Using this new criterion, the **skip** axiom schema and the other axiom schemas and proof rules from Section 5.1 can be left unchanged.

We are now ready to study the axiom schemas that apply specifically to Creol. Most of these follow the general pattern

$$\begin{aligned}&\{\forall x, h.\ (\mathit{interleave}(\mathcal{H}, h) \wedge \langle \text{further constraints on } h \text{ and } x\rangle) \Rightarrow Q^{z,\mathcal{H}}_{x,h^\frown\langle \text{event}\rangle}\} \\ &\langle \text{Creol statement}\rangle \\ &\{Q\},\end{aligned}$$

which corresponds to the random assignment

$$z, \mathcal{H} := x, h \frown \langle \text{event} \rangle \textbf{ for some } x, h$$
$$\textbf{such that } \mathit{interleave}(\mathcal{H}, h) \wedge \langle \text{further constraints on } h \text{ and } x \rangle$$

The variable z represents a local variable or an attribute that is modified by the Creol statement. The history variable $\mathcal{H}$ is extended in a nondeterministic fashion to account for parallel activity; then it is extended by one more event that records the execution of the Creol statement.

The axiom schema for object creation closely follows this pattern:

Axiom P12 (Object Creation)

$$\Big\{\forall o, h.\ \big(\mathit{interleave}(\mathcal{H}, h) \wedge o \notin \mathit{objectIds}(h) \wedge \mathit{parent}(o) = \textbf{self}\big) \Rightarrow Q^{z,\mathcal{H}}_{o,h \frown [\textbf{self} \to o.\textbf{new}\ c(\bar{e})]}\Big\}$$
$$z := \textbf{new}\ c(\bar{e})$$
$$\{Q\}$$
$$\text{for fresh variables } o, h$$

Rewrite Rule S13′ of Section 4.4 models $z :=$ **new** $c(\bar{e})$ by assigning a fresh object identity o to z and extending the history with the event $[\textbf{self} \to o.\textbf{new}\ c(\bar{e})]$. In addition, since creating an object accesses the history, we must also consider Rewrite Rule S25′ and extend the history beforehand to reflect parallel activity. Using SEQ, we can encode object creation as follows:

$$z, \mathcal{H} := o, h \frown [\textbf{self} \to o.\textbf{new}\ c(\bar{e})] \textbf{ for some } o, h$$
$$\textbf{such that } \mathit{interleave}(\mathcal{H}, h) \wedge o \notin \mathit{objectIds}(h) \wedge \mathit{parent}(o) = \textbf{self}$$

Axiom P12 can be derived by applying Axiom P11 on the SEQ statement.

The new object's identifier is made unique by requiring that it hasn't occurred in the local history h and by specifying that $\mathit{parent}(o) = \textbf{self}$. The latter requirement will allow us to compose objects together in Section 5.6 without the risk of clashes in object identities.

Example 5.7. Axiom P12 is difficult to relate to because it encodes the effective postcondition into the precondition. This becomes especially visible when we use Axiom P12 to prove correctness formulas with **true** as the precondition. Consider the formula

$$\{\textbf{true}\}$$
$$\mathit{buf} := \textbf{new}\ \mathit{Buffer};$$
$$\{\exists h_0.\ \mathcal{H} = h_0 \frown [\textbf{self} \to \mathit{buf}.\textbf{new}\ \mathit{Buffer}] \wedge \mathit{buf} \notin \mathit{objectIds}(h_0)$$
$$\wedge\ \mathit{parent}(\mathit{buf}) = \textbf{self}\}.$$

Starting with the formula's postcondition, a direct application of Axiom P12 produces the following precondition:

$$\forall o, h.\ \big(\mathit{interleave}(\mathcal{H}, h) \wedge o \notin \mathit{objectIds}(h) \wedge \mathit{parent}(o) = \textbf{self}\big) \Rightarrow$$
$$\exists h_0.\ h \frown [\textbf{self} \to o.\textbf{new}\ \mathit{Buffer}] = h_0 \frown [\textbf{self} \to o.\textbf{new}\ \mathit{Buffer}]$$
$$\wedge\ o \notin \mathit{objectIds}(h_0) \wedge \mathit{parent}(o) = \textbf{self}.$$

This predicate simplifies to **true**. [*Hint:* Try substituting h for h_0 in the existential subformula.] □

Axiom P13 (Asynchronous Invocation)

$\{\forall k, h.\ (\mathit{interleave}(\mathcal{H}, h) \wedge [\mathbf{self}{\rightarrow}*]^k \textbf{ not in } h \wedge k \geq 0) \Rightarrow Q^{l,\mathcal{H}}_{k,h^\frown[\mathbf{self}\rightarrow O.m(\bar{e})]^k}\}$
$l!O.m(\bar{e})$
$\{Q\}$
for fresh variables k, h

In Section 4.4, we modeled the asynchronous method call $l!O.m(\bar{e})$ by an assignment to l and an extension of the history $\mathcal{H}$ with the event $[\mathbf{self}{\rightarrow}O.m(\bar{e})]^k$. Using SEQ, we can express this as follows:

$$l, \mathcal{H} := k, h^\frown[\mathbf{self}{\rightarrow}O.m(\bar{e})]^k \textbf{ for some } k, h \\ \textbf{such that } \mathit{interleave}(\mathcal{H}, h) \wedge [\mathbf{self}{\rightarrow}*]^k \textbf{ not in } h$$

We assign a fresh sequence number k to the label l. In addition, we extend the history nondeterministically to model the activity of the environment and append an invocation event to the history. Axiom P13 is derived directly from this statement.

Axiom P14 (Local Asynchronous Invocation)

$\{\forall k, h.\ (\mathit{interleave}(\mathcal{H}, h) \wedge [\mathbf{self}{\rightarrow}*]^k \textbf{ not in } h \wedge k \geq 0) \Rightarrow Q^{l,\mathcal{H}}_{k,h^\frown[\mathbf{self}\rightarrow\mathbf{self}.m@c(\bar{e})]^k}\}$
$l!m@c(\bar{e})$
$\{Q\}$
for fresh variables k, h

Local method calls are axiomatized in the same way.

Example 5.8. When invoking Axiom P13 or P14, we can obtain a much simpler precondition by filtering out the parallel activity and omitting sequence numbers in the postcondition. To illustrate this, we will show how to derive the following correctness formula:

$\{\mathcal{H}/out_{\mathbf{self}} \textbf{ ew } [\mathbf{self}{\rightarrow}buf.put(i-1)]\}$
$l!buf.put(i)$
$\{\mathcal{H}/out_{\mathbf{self}} \textbf{ ew } [\mathbf{self}{\rightarrow}buf.put(i-1)]^\frown[\mathbf{self}{\rightarrow}buf.put(i)]\}.$

First, from the desired postcondition, Axiom P13 produces the precondition

$$\forall k, h.\ (\mathit{interleave}(\mathcal{H}, h) \wedge [\mathbf{self}{\rightarrow}*]^k \textbf{ not in } h \wedge k \geq 0) \Rightarrow \\ (h^\frown[\mathbf{self}{\rightarrow}buf.put(i)])/out_{\mathbf{self}} \textbf{ ew } [\mathbf{self}{\rightarrow}buf.put(i-1)]^\frown \\ [\mathbf{self}{\rightarrow}buf.put(i)].$$

Since k does not occur on the right-hand side of the implication, we can simplify the precondition to

$$\forall h.\ \mathit{interleave}(\mathcal{H}, h) \Rightarrow \\ (h^\frown[\mathbf{self}{\rightarrow}buf.put(i)])/out_{\mathbf{self}} \textbf{ ew } [\mathbf{self}{\rightarrow}buf.put(i-1)]^\frown \\ [\mathbf{self}{\rightarrow}buf.put(i)].$$

The key observation is that $\mathit{interleave}(\mathcal{H}, h)$ requires the new history h to be an extension of $\mathcal{H}$ where all the new events originate from other objects. Therefore, $h/out_{\mathbf{self}} = \mathcal{H}/out_{\mathbf{self}}$. This allows us to substitute $\mathcal{H}$ for h in the right-hand side:

$$\forall h.\ \mathit{interleave}(\mathcal{H}, h) \Rightarrow \\ \big(\mathcal{H} \frown [\mathbf{self} \rightarrow \mathit{buf.put}(i)]\big)/\mathit{out}_{\mathbf{self}}\ \mathbf{ew}\ [\mathbf{self} \rightarrow \mathit{buf.put}(i-1)] \frown [\mathbf{self} \rightarrow \mathit{buf.put}(i)].$$

Since h no longer occurs on the right-hand side, and assuming the class makes no assumption about its environment, the precondition can be simplified further to

$$\big(\mathcal{H} \frown [\mathbf{self} \rightarrow \mathit{buf.put}(i)]\big)/\mathit{out}_{\mathbf{self}}\ \mathbf{ew}\ [\mathbf{self} \rightarrow \mathit{buf.put}(i-1)] \frown [\mathbf{self} \rightarrow \mathit{buf.put}(i)],$$

which is equivalent to

$$\mathcal{H}/\mathit{out}_{\mathbf{self}}\ \mathbf{ew}\ [\mathbf{self} \rightarrow \mathit{buf.put}(i-1)].$$

This last assertion is the desired precondition. □

Axiom P15 (Asynchronous Reply)

$$\begin{array}{l}
\big\{\mathbf{if}\ \mathit{pending}(\mathcal{H}, \mathbf{self}, \mathbf{self}, l)\ \mathbf{then} \\
\quad (\mathcal{G}_c)^{\mathcal{H}}_{\mathcal{H} \frown [\mathbf{self.reenter}]^l} \\
\quad \wedge\ \forall \bar{a}, h.\ \mathit{reenter}(\mathcal{H}, h, \bar{a}, l) \Rightarrow Q^{\bar{\mathcal{A}}, \mathcal{H}, l, z_1, \ldots, z_n}_{\bar{a}, h, -1, \mathit{returnVal}_1(h, \mathbf{self}, l), \ldots, \mathit{returnVal}_n(h, \mathbf{self}, l)} \\
\ \mathbf{else} \\
\quad \forall h.\ \big(\mathit{interleave}(\mathcal{H}, h) \wedge [\mathbf{self} \leftarrow *]^l\ \mathbf{in}\ h\big) \Rightarrow \\
\qquad Q^{\mathcal{H}, l, z_1, \ldots, z_n}_{h, -1, \mathit{returnVal}_1(h, \mathbf{self}, l), \ldots, \mathit{returnVal}_n(h, \mathbf{self}, l)} \\
\ \mathbf{fi}\big\} \\
l?(\bar{z}) \\
\{Q\}
\end{array}$$

for fresh variables $\bar{a}, h$

The operational semantics of the asynchronous reply statement $l?(\bar{z})$ is given by Rewrite Rules S22 (Asynchronous Reply) and S23′ (Local Reentry and Continuation). The *pending* predicate is from Definition T24 in Section 4.4; the *reenter* predicate and the $\mathit{returnVal}_i$ function will be defined shortly. Using SEQ, we can encode the statement's semantics as follows:

$$\begin{array}{l}
\mathbf{if}\ \mathit{pending}(\mathcal{H}, \mathbf{self}, \mathbf{self}, l)\ \mathbf{then} \\
\quad \mathbf{prove}\ (\mathcal{G}_c)^{\mathcal{H}}_{\mathcal{H} \frown [\mathbf{self.reenter}]^l}; \\
\quad \bar{\mathcal{A}}, \mathcal{H} := \bar{a}, h\ \mathbf{for\ some}\ \bar{a}, h\ \mathbf{such\ that}\ \mathit{reenter}(\mathcal{H}, h, \bar{a}, l) \\
\mathbf{else} \\
\quad \mathcal{H} := h\ \mathbf{for\ some}\ h\ \mathbf{such\ that}\ \mathit{interleave}(\mathcal{H}, h) \wedge [\mathbf{self} \leftarrow *]^l\ \mathbf{in}\ h \\
\mathbf{fi}; \\
l, z_1, \ldots, z_n := -1, \mathit{returnVal}_1(\mathcal{H}, \mathbf{self}, l), \ldots, \mathit{returnVal}_n(\mathcal{H}, \mathbf{self}, l)
\end{array}$$

If the call associated with l is a pending call to **self**, we release the processor and reacquire it in a state where the history $\mathcal{H}$ has become h and the object's writable attributes, denoted by $\bar{\mathcal{A}}$, have the values $\bar{a}$. (We don't need to extend the history with *interleave* before accessing $\mathcal{H}$ in the **if** condition because the history extension has no effect on the truth of the condition.) The **prove** statement gives rise to the $(\mathcal{G}_c)^{\mathcal{H}}_{\mathcal{H} \frown [\mathbf{self.reenter}]^l}$ term in the precondition of Axiom P15, ensuring that the guarantee holds when the processor is released.

If l refers to a remote call, or to a local call that has already been serviced, we extend the history with events originating from other objects, including a reply event $[\mathbf{self} \leftarrow *]^l$.

In all cases, we assign -1 to l to prevent it from being used again, and we assign the return values from the method call to $\bar{z}$. The assignment to l can be omitted if we can determine through static program analysis that l is a dead variable [NNH99].

Definition Q2 (Local Reentry Assertion)

$$\begin{aligned} \mathit{reenter}(h, h', \bar{a}, l) \triangleq\ & \mathit{lwf}(h', \mathbf{self}) \\ & \wedge\ h \frown [\mathbf{self}.\mathbf{reenter}]^{l} \preceq h' \\ & \wedge\ h' \ \mathbf{ew}\ [\mathbf{self} \leftarrow \mathbf{self}.*]^{l} \\ & \wedge\ [\mathbf{caller} \leftarrow \mathbf{self}]^{\mathbf{label}}\ \mathbf{not\ in}\ h' \\ & \wedge\ (\mathcal{A}_c \wedge \mathcal{G}_c)^{\bar{A}, \mathcal{H}}_{\bar{a}, h'} \end{aligned}$$

The $\mathit{reenter}(h, h', \bar{a}, l)$ assertion is a syntactic reformulation of the $\mathit{reenter}(\varphi, o, o', k, k', \bar{z}, \alpha, \alpha')$ predicate from Definition T24 in Section 4.4, except that we now expect the class assumption $\mathcal{A}_c$ and the class guarantee $\mathcal{G}_c$ to hold after the processor is released. (In Section 4.4, we took the more liberal approach of letting the program continue when the guarantee is broken.)

We assume that the free variables occurring in the assumption and in the guarantee do not clash with the method's local variables and parameters. An easy way to achieve this is to use a distinct syntax for logical variables.

Definition Q3 (Call Return Value)

$$\begin{aligned} \mathit{returnVal}_i(h \frown [o \leftarrow o'.m@c(\bar{v}; \bar{w})]^{k}, o, k) &\triangleq w_i \\ \mathit{returnVal}_i(h \frown v, o, k) &\triangleq \mathit{returnVal}_i(h, o, k) \qquad \text{[otherwise]} \end{aligned}$$

The $\mathit{returnVal}_i(h, o, k)$ auxiliary (partial) function extracts the return values for the method call identified by the pair (o, k) from the history.

We will now look at the **await** statement. In the open system semantics, **await** g was handled by Rewrite Rules S8 (Guard Crossing), S14′ (Process Suspension and Reactivation), and S25′ (Parallel Activity). Here, we proceed by case on the guard:

i. **await** $B_1 \,\&\, \cdots \,\&\, B_n$

   The guard is a simple or complex Boolean guard.

ii. **await** $\left[B_1 \,\&\, \cdots \,\&\, B_n \,\&\right]\ l_1? \,\&\, \cdots \,\&\, l_p?$

   The guard is a simple or complex reply guard, with an optional Boolean component. Since & is commutative, we can assume without loss of generality that the Boolean guards precede the reply guards.

iii. **await wait** $\left[\&\, B_1 \,\&\, \cdots \,\&\, B_n\right] \left[\&\, l_1? \,\&\, \cdots \,\&\, l_p?\right]$

   The guard is a simple **wait** guard or a complex guard with at least one **wait** component. By idempotence, we may assume that **wait** occurs only once.[1]

We will start with case i, when the guard g is a simple or complex Boolean guard of the form $B_1 \,\&\, \cdots \,\&\, B_n$.

[1] One way to avoid some of the complications caused by the **wait** guard would be to replace it with a separate **release** statement with the same semantics. It is expected that future definitions of the Creol language will adopt this convention.

Axiom P16 (Conditional Wait with Boolean Guards)

$$\begin{array}{l}\{\textbf{if } \bigwedge_{i=1}^{n} B_i \textbf{ then } \forall h.\ \mathit{interleave}(\mathcal{H}, h) \Rightarrow Q_h^{\mathcal{H}} \\ \qquad\qquad \textbf{else } (\mathcal{G}_c)^{\mathcal{H}}_{\mathcal{H}^\frown[\textbf{self.release}]} \\ \qquad\qquad\quad \wedge \forall \bar{a}, h.\ \left(\mathit{release}(\mathcal{H}, h, \bar{a}) \wedge \bigwedge_{i=1}^{n} (B_i)^{\bar{\mathcal{A}}}_{\bar{a}}\right) \Rightarrow Q^{\bar{\mathcal{A}}, \mathcal{H}}_{\bar{a}, h} \textbf{ fi}\} \\ \textbf{await } B_1 \,\&\, \cdots \,\&\, B_n \\ \{Q\} \\ \text{for fresh variables } \bar{a}, h\end{array}$$

Axiom P16 is derived from the following SEQ code:

$$\begin{array}{l}\textbf{if } \bigwedge_{i=1}^{n} B_i \textbf{ then} \\ \quad \mathcal{H} := h \textbf{ for some } h \textbf{ such that } \mathit{interleave}(\mathcal{H}, h) \\ \textbf{else} \\ \quad \textbf{prove } (\mathcal{G}_c)^{\mathcal{H}}_{\mathcal{H}^\frown[\textbf{self.release}]}; \\ \quad \bar{\mathcal{A}}, \mathcal{H} := \bar{a}, h \textbf{ for some } \bar{a}, h \textbf{ such that } \mathit{release}(\mathcal{H}, h, \bar{a}) \wedge \bigwedge_{i=1}^{n} (B_i)^{\bar{\mathcal{A}}}_{\bar{a}} \\ \textbf{fi}\end{array}$$

If $\bigwedge_{i=1}^{n} B_i$ is true, the **await** statement is skipped. We still need to extend the history because by our definition of validity, Rewrite Rule S25′ (Parallel Activity) can be invoked immediately before Rewrite Rule S8 (Guard Crossing). If $\bigwedge_{i=1}^{n} B_i$ is false, we release the processor—and require the guarantee to hold. As with Rewrite Rule S14′, the history $\mathcal{H}$ and the writable attributes $\bar{\mathcal{A}}$ are modified nondeterministically.

Definition Q4 (Processor Release Assertion)

$$\begin{array}{rcl}\mathit{release}(h, h', \bar{a}) & \triangleq & \mathit{lwf}(h', \textbf{self}) \\ & & \wedge\ h^\frown[\textbf{self.release}] \preceq h' \\ & & \wedge\ \mathit{mayAcquireProcessor}(h', \textbf{self}, \textbf{caller}, \textbf{label}) \\ & & \wedge\ [\textbf{caller}{\leftarrow}\textbf{self}]^{\textbf{label}} \textbf{ not in } h' \\ & & \wedge\ \big((h^\frown[\textbf{self.release}])/(\mathit{out}_{\textbf{self}} \cup \mathit{ctl}_{\textbf{self}}) = \\ & & \qquad h'/(\mathit{out}_{\textbf{self}} \cup \mathit{ctl}_{\textbf{self}}) \Rightarrow \\ & & \qquad \bar{\mathcal{A}} = \bar{a}\big) \\ & & \wedge\ \left(\mathcal{A}_c \wedge \mathcal{G}_c\right)^{\bar{\mathcal{A}}, \mathcal{H}}_{\bar{a}, h'}\end{array}$$

The $\mathit{release}(h, h', \bar{a})$ assertion is primarily a syntactic reformulation of the $\mathit{release}(\varphi, o, o', k, z, \alpha, \alpha')$ predicate from Definition T25. The *mayAcquireProcessor* predicate is from Definition T21 in Section 4.4.

Axiom P17 (Conditional Wait with Reply Guards)

$$\begin{array}{l}\{\forall h.\ \mathit{interleave}(\mathcal{H}, h) \Rightarrow \\ \qquad \textbf{if } \bigwedge_{i=1}^{n} B_i \wedge \bigwedge_{j=1}^{p} [\textbf{self}{\leftarrow}*]^{l_j} \textbf{ in } h \textbf{ then} \\ \qquad\quad Q_h^{\mathcal{H}} \\ \qquad \textbf{else} \\ \qquad\quad (\mathcal{G}_c)^{\mathcal{H}}_{h^\frown[\textbf{self.release}]} \\ \qquad\quad \wedge \forall \bar{a}, h'.\ \left(\mathit{release}(h, h', \bar{a}) \wedge \bigwedge_{i=1}^{n} (B_i)^{\bar{\mathcal{A}}}_{\bar{a}} \wedge \bigwedge_{j=1}^{p} [\textbf{self}{\leftarrow}*]^{l_j} \textbf{ in } h'\right) \Rightarrow Q^{\bar{\mathcal{A}}, \mathcal{H}}_{\bar{a}, h'} \\ \qquad \textbf{fi}\} \\ \textbf{await } l_1? \,\&\, \cdots \,\&\, l_p? \left[\,\&\, B_1 \,\&\, \cdots \,\&\, B_n\right] \\ \{Q\} \\ \text{for fresh variables } \bar{a}, h, h'\end{array}$$

Axiom P17 corresponds to the following SEQ code:

$\mathcal{H} := h$ **for some** h **such that** $\mathit{interleave}(\mathcal{H}, h)$;
if $\neg\big(\bigwedge_{i=1}^{n} B_i \wedge \bigwedge_{j=1}^{p} [\mathbf{self}{\leftarrow}*]^{l_j}$ **in** $\mathcal{H}\big)$ **then**
  prove $(\mathcal{G}_c)^{\mathcal{H}}_{\mathcal{H}^\frown[\mathbf{self.release}]}$;
  $\bar{\mathcal{A}}, \mathcal{H} := \bar{a}, h'$ **for some** $\bar{a}, h'$
    such that $\mathit{release}(\mathcal{H}, h', \bar{a}) \wedge \bigwedge_{i=1}^{n} (B_i)^{\bar{\mathcal{A}}}_{\bar{a}} \wedge \bigwedge_{j=1}^{p} [\mathbf{self}{\leftarrow}*]^{l_j}$ **in** h'
fi

We start by extending the history nondeterministically to reflect parallel activity. If one of the Boolean conditions B_i is false, we release the processor and reacquire it in a state in which the Boolean conditions are true and the replies have arrived. As before, we require that the guarantee holds when the processor is released.

Axiom P18 (Unconditional Wait)

$\big\{(\mathcal{G}_c)^{\mathcal{H}}_{\mathcal{H}^\frown[\mathbf{self.release}]}$
$\wedge\ \forall \bar{a}, h.\ \big(\mathit{release}(\mathcal{H}, h, \bar{a}) \wedge \bigwedge_{i=1}^{n} (B_i)^{\bar{\mathcal{A}}}_{\bar{a}} \wedge \bigwedge_{j=1}^{p} [\mathbf{self}{\leftarrow}*]^{l_j}$ **in** $h\big) \Rightarrow Q^{\bar{\mathcal{A}}, \mathcal{H}}_{\bar{a}, h}\big\}$
await wait $\big[\&\, l_1? \,\&\, \cdots \,\&\, l_p?\big]\big[\&\, B_1 \,\&\, \cdots \,\&\, B_n\big]$
$\{Q\}$
for fresh variables $\bar{a}, h$

An **await** statement that contains a **wait** guard unconditionally releases the processor and reacquires it at some point when the Boolean conditions B_i are true and the replies associated with the labels l_j have arrived. Axiom P18 is derived from

prove $(\mathcal{G}_c)^{\mathcal{H}}_{\mathcal{H}^\frown[\mathbf{self.release}]}$;
$\bar{\mathcal{A}}, \mathcal{H} := \bar{a}, h$ **for some** $\bar{a}, h$
  such that $\mathit{release}(\mathcal{H}, h, \bar{a}) \wedge \bigwedge_{i=1}^{n} (B_i)^{\bar{\mathcal{A}}}_{\bar{a}} \wedge \bigwedge_{j=1}^{p} [\mathbf{self}{\leftarrow}*]^{l_j}$ **in** h

We have now covered all the simple Creol statements that alter the history variable $\mathcal{H}$. The last two Creol statements to review are the nondeterministic choice statement $S_1 \,\square\, S_2$ and the nondeterministic merge statement $|||_{i=1}^{n} S_i$.

Proof Rule P19 (Nondeterministic Choice)

$$\frac{\{P_1\}\, S_1\, \{Q\} \quad \{P_2\}\, S_2\, \{Q\} \quad \{P_1'\}\, S_1^\star\, \{Q\} \quad \{P_2'\}\, S_2^\star\, \{Q\}}{\begin{array}{l}
\big\{\forall h.\ \mathit{interleave}(\mathcal{H}, h) \Rightarrow \\
\quad \textbf{if } \mathit{ready}(S_1 \,\square\, S_2, \bar{\mathcal{A}}, h) \textbf{ then} \\
\quad\quad \mathit{pickReadyBranch}(S_1, S_2, P_1, P_2, \bar{\mathcal{A}}, h) \\
\quad \textbf{else if } \mathit{enabled}(S_1 \,\square\, S_2, \bar{\mathcal{A}}, h) \textbf{ then} \\
\quad\quad \forall h'.\ \big(\mathit{interleave}(h, h') \wedge \mathit{ready}(S_1 \,\square\, S_2, \bar{\mathcal{A}}, h')\big) \Rightarrow \\
\quad\quad\quad \mathit{pickReadyBranch}(S_1, S_2, P_1, P_2, \bar{\mathcal{A}}, h') \\
\quad \textbf{else} \\
\quad\quad (\mathcal{G}_c)^{\mathcal{H}}_{h^\frown[\mathbf{self.release}]} \\
\quad\quad \wedge\ \forall \bar{a}, h'.\ \big(\mathit{release}(h, h', \bar{a}) \wedge \mathit{ready}(S_1^\star \,\square\, S_2^\star, \bar{a}, h')\big) \Rightarrow \\
\quad\quad\quad\quad \mathit{pickReadyBranch}(S_1^\star, S_2^\star, P_1', P_2', \bar{a}, h') \\
\quad \textbf{fi fi}\big\} \\
S_1 \,\square\, S_2 \\
\{Q\}
\end{array}}$$

where $S_i^\star \equiv \mathit{clearWait}(S_i)$ and $\bar{a}, h, h'$ are fresh variables

The Hoare logic rule for $S_1 \,\square\, S_2$ corresponds to Rewrite Rules S9 (Nondeterministic Choice), S14′ (Process Suspension and Reactivation), and S25′ (Parallel Activity) in the operational semantics. The *clearWait* function was introduced by Definition T9 in Section 4.3. The precondition of $S_1 \,\square\, S_2$ encodes the following algorithm:

1. Extend the history nondeterministically with events from other objects.
2. If $S_1 \,\square\, S_2$ is ready, choose a ready branch.
3. If $S_1 \,\square\, S_2$ is enabled but not ready, block until it becomes ready and choose a ready branch.
4. If $S_1 \,\square\, S_2$ is disabled, release the processor and reacquire it in a state where one of the branches is ready.

In pseudocode, this gives

$\mathcal{H} := h$ **for some** h **such that** $\mathit{interleave}(\mathcal{H}, h)$;
if $\mathit{ready}(S_1 \,\square\, S_2, \bar{\mathcal{A}}, \mathcal{H})$ **then**
  $\langle S_1$ or S_2, whichever is ready$\rangle$
else if $\mathit{enabled}(S_1 \,\square\, S_2, \bar{\mathcal{A}}, \mathcal{H})$ **then**
  $\mathcal{H} := h'$ **for some** h' **such that** $\mathit{interleave}(\mathcal{H}, h') \wedge \mathit{ready}(S_1 \,\square\, S_2, \bar{\mathcal{A}}, h')$;
  $\langle S_1$ or S_2, whichever is ready$\rangle$
else
  prove $(\mathcal{G}_c)^{\mathcal{H}}_{\mathcal{H}^\frown[\mathbf{self.release}]}$;
  $\bar{\mathcal{A}}, \mathcal{H} := \bar{a}, h'$ **for some** $\bar{a}, h'$
    such that $\mathit{release}(\mathcal{H}, h', \bar{a}) \wedge \mathit{ready}(S_1^\star \,\square\, S_2^\star, \bar{a}, h')$;
  $\langle S_1^\star$ or $S_2^\star$, whichever is ready$\rangle$
fi fi

The *pickReadyBranch*, *ready*, and *enabled* assertions, and the auxiliary *satisfied* assertion, are defined below.

Definition Q5 (Ready Branch Choice Assertion)

$$\begin{aligned}\mathit{pickReadyBranch}(S_1, S_2, P_1, P_2, \bar{a}, h) \triangleq\ & \big(\mathit{ready}(S_1, \bar{a}, h) \Rightarrow (P_1)^{\bar{\mathcal{A}}, \mathcal{H}}_{\bar{a}, h}\big) \\ & \wedge \big(\mathit{ready}(S_2, \bar{a}, h) \Rightarrow (P_2)^{\bar{\mathcal{A}}, \mathcal{H}}_{\bar{a}, h}\big)\end{aligned}$$

Definition Q6 (Statement Readiness Assertion)

$$\begin{aligned}
\mathit{ready}(l?(\bar{z}); S, \bar{a}, h) &\triangleq \mathit{satisfied}(l?, \bar{a}, h) \\
\mathit{ready}(\mathbf{await}\ g; S, \bar{a}, h) &\triangleq \mathit{satisfied}(g, \bar{a}, h) \\
\mathit{ready}((S_1 \,\square\, S_2); S, \bar{a}, h) &\triangleq \mathit{ready}(S_1, \bar{a}, h) \vee \mathit{ready}(S_2, \bar{a}, h) \\
\mathit{ready}((|||_{i=1}^n S_i); S, \bar{a}, h) &\triangleq \textstyle\bigvee_{i=1}^n \mathit{ready}(S_i, \bar{a}, h) \\
\mathit{ready}((S); S', \bar{a}, h) &\triangleq \mathit{ready}(S, \bar{a}, h) \\
\mathit{ready}(S, \bar{a}, h) &\triangleq \mathbf{true} \qquad \text{[otherwise]}
\end{aligned}$$

Definition Q7 (Statement Enabledness Assertion)

$$\begin{aligned}
\mathit{enabled}(\mathbf{await}\ g; S, \bar{a}, h) &\triangleq \mathit{satisfied}(g, \bar{a}, h) \\
\mathit{enabled}((S_1 \,\square\, S_2); S, \bar{a}, h) &\triangleq \mathit{enabled}(S_1, \bar{a}, h) \vee \mathit{enabled}(S_2, \bar{a}, h) \\
\mathit{enabled}((|||_{i=1}^n S_i); S, \bar{a}, h) &\triangleq \textstyle\bigvee_{i=1}^n \mathit{enabled}(S_i, \bar{a}, h) \\
\mathit{enabled}((S); S', \bar{a}, h) &\triangleq \mathit{enabled}(S, \bar{a}, h) \\
\mathit{enabled}(S, \bar{a}, h) &\triangleq \mathbf{true} \qquad \text{[otherwise]}
\end{aligned}$$

Definition Q8 (Guard Satisfaction Assertion)

$$
\begin{array}{ll}
\mathit{satisfied}(B, \bar{a}, h) & \triangleq B^{\bar{\mathcal{A}},\mathcal{H}}_{\bar{a},h} \\
\mathit{satisfied}(l?, \bar{a}, h) & \triangleq [\mathbf{self} \leftarrow *]^{l}\ \mathbf{in}\ h \\
\mathit{satisfied}(\mathbf{wait}, \bar{a}, h) & \triangleq \mathbf{false} \\
\mathit{satisfied}(g_1\ \&\ g_2, \bar{a}, h) & \triangleq \mathit{satisfied}(g_1, \bar{a}, h) \wedge \mathit{satisfied}(g_2, \bar{a}, h)
\end{array}
$$

Proof Rule P19 can be contrasted with the following proof rule:

Proof Rule P20 (Nondeterministic Choice, Incomplete)

$$\frac{\{P_1\}\, S_1\, \{Q\} \quad \{P_2\}\, S_2\, \{Q\}}{\{P_1 \wedge P_2\}\, S_1 \,\square\, S_2\, \{Q\}}$$

This new rule is sound but not complete. It would be complete if $S_1 \,\square\, S_2$ chose which branch to execute completely at random, but in Section 4.3 we saw that it prefers branches that are ready or enabled over branches that are not.

Example 5.9. The following statement will make the contrast between Proof Rules P19 and P20 more apparent:

$$\mathbf{await}\ x = 1 \,\square\, x := 2$$

If $x \neq 1$ holds before the statement is executed, only the second branch can be chosen, because the first branch is not ready (the guard $x = 1$ is false). Hence,

$$\{x \neq 1\}\ \mathbf{await}\ x = 1 \,\square\, x := 2\, \{x = 2\}$$

is valid. Can this be proved with Hoare logic? Using Proof Rule P20, the best we can achieve is a correctness formula with **false** $\wedge$ **true** as the precondition:

$$\frac{\{\mathbf{false}\}\ \mathbf{await}\ x = 1\, \{x = 2\} \quad \{\mathbf{true}\}\ x := 2\, \{x = 2\}}{\{\mathbf{false} \wedge \mathbf{true}\}\ \mathbf{await}\ x = 1 \,\square\, x := 2\, \{x = 2\}}\ \text{P20}$$

The problem here is that in terms of Proof Rule P20, either branch can be executed, whereas the operational semantics always chooses the second branch if $x \neq 1$.

With its reliance on *ready* and *enabled*, Proof Rule P19 appears more promising:

$$\frac{\begin{array}{l}\{\mathbf{false}\}\ \mathbf{await}\ x = 1\, \{x = 2\} \quad \{\mathbf{true}\}\ x := 2\, \{x = 2\} \\ \{\mathbf{false}\}\ \mathbf{await}\ x = 1\, \{x = 2\} \quad \{\mathbf{true}\}\ x := 2\, \{x = 2\}\end{array}}{\begin{array}{l}\big\{\forall h.\ \mathit{interleave}(\mathcal{H}, h) \Rightarrow \\ \quad \mathbf{if}\ \mathit{ready}(\mathbf{await}\ x = 1 \,\square\, x := 2, \bar{\mathcal{A}}, h)\ \mathbf{then} \\ \quad\quad \mathit{pickReadyBranch}(\mathbf{await}\ x = 1, x := 2, \mathbf{false}, \mathbf{true}, \bar{\mathcal{A}}, h) \\ \quad \mathbf{else\ if}\ \mathit{enabled}(\mathbf{await}\ x = 1 \,\square\, x := 2, \bar{\mathcal{A}}, h)\ \mathbf{then} \\ \quad\quad \forall h'.\ \big(\mathit{interleave}(h, h') \wedge \mathit{ready}(\mathbf{await}\ x = 1 \,\square\, x := 2, \bar{\mathcal{A}}, h')\big) \Rightarrow \\ \quad\quad\quad \mathit{pickReadyBranch}(\mathbf{await}\ x = 1, x := 2, \mathbf{false}, \mathbf{true}, \bar{\mathcal{A}}, h') \\ \quad \mathbf{else} \\ \quad\quad (\mathcal{G}_c)^{\mathcal{H}}_{h^\frown[\mathbf{self.release}]} \\ \quad\quad \wedge\ \forall \bar{a}, h'.\ \big(\mathit{release}(h, h', \bar{a}) \wedge \mathit{ready}(\mathbf{await}\ x = 1 \,\square\, x := 2, \bar{a}, h')\big) \Rightarrow \\ \quad\quad\quad\quad \mathit{pickReadyBranch}(\mathbf{await}\ x = 1, x := 2, \mathbf{false}, \mathbf{true}, \bar{a}, h') \\ \quad \mathbf{fi\ fi}\big\} \\ \mathbf{await}\ x = 1 \,\square\, x := 2 \\ \{x = 2\}\end{array}}\ \text{P19}$$

The precondition in the conclusion is fairly complex, but it can be simplified a lot. Because parallel activity does not influence the readiness or enabledness of either branch of the $\Box$ statement, we can drop "$\forall h.\ \mathit{interleave}(\mathcal{H}, h) \Rightarrow$" and substitute $\mathcal{H}$ for h. And since the second branch of the $\Box$ statement is always ready, the statement as a whole is ready and the first branch of the precondition's **if** expression is taken. The precondition simplifies to

$$\mathit{pickReadyBranch}(\textbf{await}\ x = 1, x := 2, \textbf{false}, \textbf{true}, \bar{\mathcal{A}}, \mathcal{H}).$$

By the definition of *pickReadyBranch*, this expands to

$$\big(\mathit{ready}(\textbf{await}\ x = 1, \bar{\mathcal{A}}, \mathcal{H}) \Rightarrow \textbf{false}\big) \wedge \big(\mathit{ready}(x := 2, \bar{\mathcal{A}}, \mathcal{H}) \Rightarrow \textbf{true}\big),$$

which in turn expands to

$$(x = 1 \Rightarrow \textbf{false}) \wedge (\textbf{true} \Rightarrow \textbf{true}).$$

This assertion is equivalent to $x \neq 1$. Thus, Proof Rule P19 can be used to derive the correctness formula $\{x \neq 1\}\ \textbf{await}\ x = 1 \Box x := 2\ \{x = 2\}$. $\Box$

Proof Rule P21 (Nondeterministic Merge, Await-Free)

$$\frac{\{P\}\ \Box_{i=1}^{n}(S_i; |||_{j=1,j\neq i}^{n} S_j)\ \{Q\}}{\{P\}\ |||_{i=1}^{n} S_i\ \{Q\}} \quad \text{if } \mathit{awaitFree}(|||_{i=1}^{n} S_i)$$

Nondeterministic merge is difficult to axiomatize because an **await** statement occurring in branch S_i may give control to another branch S_j if the **await** guard is not satisfied, instead of releasing the processor.

To avoid these difficulties, Proof Rule P21 requires that the S_i branches contain no **await** statements. We can then treat $|||_{i=1}^{n} S_i$ as a nondeterministic permutation of the S_i branches, using the $\Box$ statement. For $n = 2$, we have

$$\frac{\{P\}\ S_1; S_2 \Box S_2; S_1\ \{Q\}}{\{P\}\ S_1 ||| S_2\ \{Q\}}\ \text{P21}$$

and for $n = 3$, we have

$$\frac{\{P\}\ S_1; (S_2 ||| S_3) \Box S_2; (S_1 ||| S_3) \Box S_3; (S_1 ||| S_2)\ \{Q\}}{\{P\}\ S_1 ||| S_2 ||| S_3\ \{Q\}}\ \text{P21}$$

Notice that the n case is defined in terms of the $n - 1$ case. The $\mathit{awaitFree}(S)$ predicate is defined below.

Definition Q9 (Await-Freedom)

$$\begin{aligned}
&\mathit{awaitFree}(\textbf{await}\ g) && \triangleq \textbf{false} \\
&\mathit{awaitFree}(\textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}) && \triangleq \mathit{awaitFree}(S_1) \wedge \mathit{awaitFree}(S_2) \\
&\mathit{awaitFree}(\big[\textbf{inv}\ I\big]\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}) && \triangleq \mathit{awaitFree}(S) \\
&\mathit{awaitFree}(S_1; S_2) && \triangleq \mathit{awaitFree}(S_1) \wedge \mathit{awaitFree}(S_2) \\
&\mathit{awaitFree}(S_1 \Box S_2) && \triangleq \mathit{awaitFree}(S_1) \wedge \mathit{awaitFree}(S_2) \\
&\mathit{awaitFree}(|||_{i=1}^{n} S_i) && \triangleq \textstyle\bigwedge_{i=1}^{n} \mathit{awaitFree}(S_i) \\
&\mathit{awaitFree}((S)) && \triangleq \mathit{awaitFree}(S) \\
&\mathit{awaitFree}(S) && \triangleq \textbf{true} \qquad \text{[otherwise]}
\end{aligned}$$

To fully capture the behavior of the $|||_{i=1}^{n} S_i$ statement in the presence of **await** statements, we could for example provide a proof rule that performs an interference-freedom test on the S_i branches to the proof system, as is customary for shared-variable concurrency [AO97, And00, dRdB+01]; however, this is very complex and will not be attempted here. With this one exception, we have now reviewed all the proof rules and axioms necessary to reason about Creol statements.

5.3 Proof Strategies for Hoare Logic

The aim of Hoare logic is to prove correctness formulas $\{P\}\, S\, \{Q\}$. By the postulated soundness of the Hoare logic, the proved formulas are then valid with respect to Creol's operational semantics. But given an arbitrarily complex method body S, how do we proceed to reason about it? Clearly, we must break it down, but how?

When constructing proofs, we usually rely on a proof strategy. The strategy tells us which axioms or proof rules to invoke, and in which order. Proof strategies are especially important for automation. In his program verification textbook [Dah92], Dahl identified four main proof construction strategies.

Conceptually, the simplest strategy is *forward construction* (which Dahl calls "right construction"). Using this strategy, to prove the formula

$$\{P\}\, S_1; \ldots; S_n\, \{Q\},$$

we start with P and work forward by proving the formulas

$$\{P\}\, S_1\, \{Q_1\}, \qquad \{Q_1\}\, S_2\, \{Q_2\}, \qquad \ldots, \qquad \{Q_{n-1}\}\, S_n\, \{Q_n\}.$$

Using $n-1$ applications of Proof Rule P5 (Sequential Composition), we obtain the theorem $\{P\}\, S_1; \ldots; S_n\, \{Q_n\}$. We conclude by proving that Q_n implies Q, invoking Proof Rule P6 (Consequence).

Forward construction requires axiom schemas and proof rules that let us specify an arbitrary precondition P and that define the postcondition Q in terms of P. For example, here are forward-constructive versions of the assignment axiom and of the **if** statement rule:

Axiom P22 (Assignment, Forward Constructive)

$$\{P\}\, \bar{z} := \bar{e}\, \{\exists \bar{x}.\ P^{\bar{z}}_{\bar{x}} \wedge \bar{z} = \bar{e}^{\bar{z}}_{\bar{x}}\}$$

Proof Rule P23 (If Statement, Forward Constructive)

$$\frac{\{P \wedge B\}\, S_1\, \{Q_1\} \qquad \{P \wedge \neg B\}\, S_2\, \{Q_2\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{Q_1 \vee Q_2\}}$$

A forward-constructive axiom schema or proof rule for a statement S is *right maximal* if for any precondition P, it allows the derivation of $\{P\}\, S\, \{Q\}$, where Q is the strongest predicate Q' such that $\{P\}\, S\, \{Q'\}$ is valid. If the axiom schemas for all the statements are right maximal and the proof rules allow us to derive right-maximal theorems from right-maximal axioms, then every valid correctness formula is provable using forward construction, assuming that we have an oracle to assess the validity of first-order logic assertions [Apt81].

A similar strategy is *backward construction* (or "left construction"). Axiom P3 (Assignment) is an example of a backward-constructive axiom, because its precondition $P \equiv Q^{\bar{x}}_{\bar{e}}$ is defined in terms of its postcondition Q. Another example is the following axiom schema for the **prove** statement:

Axiom P24 (Inline Assertion, Backward Constructive)

$$\{P \wedge Q\}\ \textbf{prove}\ P\ \{Q\}$$

To compute a backward-constructive proof for

$$\{P\}\ S_1; \ldots; S_n\ \{Q\},$$

we start with Q and work backward using the theorems

$$\{P_n\}\ S_n\ \{Q\}, \qquad \{P_{n-1}\}\ S_{n-1}\ \{P_n\}, \qquad \ldots, \qquad \{P_1\}\ S_1\ \{P_2\},$$

and we conclude by proving that P implies P_1. A theorem is *left maximal* if the precondition P is the weakest predicate P' such that $\{P'\}\ S\ \{Q\}$ is valid.

The third proof strategy identified by Dahl is *top-down construction*. This strategy is used to reason about compound statements, and requires proof rules whose conclusion involves both an arbitrary precondition P and an arbitrary postcondition Q. A top-down rule for the **if** statement is given below:

Proof Rule P25 (If Statement, Top-Down)

$$\frac{\{P \wedge B\}\ S_1\ \{Q\} \quad \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{Q\}}$$

Top-down proofs start with an arbitrary correctness formula $\{P\}\ S\ \{Q\}$ that must be proved and work "outside in" in the program S.

The fourth proof strategy is *bottom-up construction*. This strategy works "inside out" by letting us derive correctness formulas about compound statements from arbitrary formulas about their constituent statements. Compare the following bottom-up **if** statement rule with Proof Rule P25:

Proof Rule P26 (If Statement, Bottom-Up)

$$\frac{\{P_1\}\ S_1\ \{Q_1\} \quad \{P_2\}\ S_2\ \{Q_2\}}{\{\textbf{if}\ B\ \textbf{then}\ P_1\ \textbf{else}\ P_2\ \textbf{fi}\}\ \textbf{if}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{Q_1 \vee Q_2\}}$$

The table below lists the axiom schemas and proof rules presented in this chapter and specifies whether they are forward constructive (FC), backward constructive (BC), top-down (TD), and/or bottom-up (BU). Proof Rule P6 (Consequence) is not listed, as it does not fall into any of these categories.

Name of Axiom or Proof Rule	**FC**	**BC**	**TD**	**BU**
Parenthesized Statement	P4	P4	P4	P4
If Statement	P23	P8	P25	P26
While Loop	P9	—	—	—
While Loop with Invariant Clause	P10	—	—	—

Name of Axiom or Proof Rule	FC	BC	TD	BU
Null Statement	P1	P1	–	–
Inline Assertion	P7	P7/24	–	–
Assignment	P22	P3	–	–
Sequential Composition	P5	P5	–	–
Random Assignment	–	P11	–	–
Object Creation	–	P12	–	–
Asynchronous Invocation	–	P13	–	–
Local Asynchronous Invocation	–	P14	–	–
Asynchronous Reply	–	P15	–	–
Conditional Wait with Boolean Guards	–	P16	–	–
Conditional Wait with Reply Guards	–	P17	–	–
Unconditional Wait	–	P18	–	–
Nondeterministic Choice	–	P19/20	–	–
Nondeterministic Merge	–	P21	–	–
Abnormal Termination	–	–	–	P2

Forward and backward construction are more suitable for automation, because the theorems can be chained together mechanistically starting from a method body's pre- or postcondition. Backward construction is usually preferred, because it normally leads to simpler assertions; for example, the forward version of the assignment axiom is nowhere as simple as the backward version. For this reason, most of the axiom schemas and proof rules presented in Sections 5.1 and 5.2 are backward constructive, and among these all but Proof Rule P20 (Nondeterministic Choice, Incomplete) are left maximal.

5.4 Weakest Liberal Preconditions

The assertion analyzer described in Chapter 6 uses a variant of the backward construction strategy based on weakest liberal preconditions, a functional formalism introduced by Dijkstra [Dij75]. The *weakest liberal precondition* (WLP) of a statement S with respect to a postcondition Q is the less restrictive precondition P such that $\{P\}\, S\, \{Q\}$ is valid. The adjective "liberal" reminds us that we operate in a partial correctness setting; the analogous concept for total correctness is called the *weakest conservative precondition*, or simply *weakest precondition*.

While the WLP of a statement is primarily a semantic concept defined in terms of formula validity (another semantic concept), Definitions Q10–Q12 attempt to give it a syntactic form. Just as the Hoare logic is designed to be sound and complete with respect to the operational semantics, the $wlp(S, Q)$ function of Definitions Q10–Q12 is intended to be logically equivalent to the semantic WLPs for the class of Creol programs that specify valid assertions—that is, valid assume–guarantee specifications, inline assertions, and loop invariants.

The computation of $wlp(S, Q)$ mimics a backward-constructive proof built from left-maximal theorems. For example, to prove the correctness formula

$$\{P\}\, S_1; \ldots; S_n\, \{Q\},$$

we compute $P_n \equiv wlp(S_n, Q)$, $P_{n-1} \equiv wlp(S_{n-1}, P_n)$, ..., $P_1 \equiv wlp(S_1, P_2)$, and we conclude by proving that P implies P_1—that is, P is at least as strong as the weakest liberal precondition.

Definition Q10 (WLP for Most Creol Statements)

$$
\begin{array}{ll}
wlp(\textbf{skip}, Q) & \triangleq Q \\
wlp(\textbf{abort}, Q) & \triangleq \textbf{true} \\
wlp(\textbf{prove } P, Q) & \triangleq P \wedge Q \\
wlp(\bar{x} := \bar{e}, Q) & \triangleq Q^{\bar{x}}_{\bar{e}} \\
wlp((S), Q) & \triangleq wlp(S, Q) \\
wlp(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, Q) & \triangleq \textbf{if } B \textbf{ then } wlp(S_1, Q) \textbf{ else } wlp(S_2, Q) \textbf{ fi} \\
wlp(S_1; S_2, Q) & \triangleq wlp(S_1, wlp(S_2, Q)) \\
wlp(\bar{z} := \bar{e} \textbf{ for some } \bar{x} & \triangleq \text{precondition of Axiom P11} \\
\quad \textbf{such that } P, Q) & \\
wlp(z := \textbf{new } c(\bar{e}), Q) & \triangleq \text{precondition of Axiom P12} \\
wlp(l!O.m(\bar{e}), Q) & \triangleq \text{precondition of Axiom P13} \\
wlp(l!m@c(\bar{e}), Q) & \triangleq \text{precondition of Axiom P14} \\
wlp(l?(\bar{z}), Q) & \triangleq \text{precondition of Axiom P15} \\
wlp(\textbf{await } B_1 \,\&\, \cdots \,\&\, B_n, Q) & \triangleq \text{precondition of Axiom P16} \\
wlp(\textbf{await } l_1? \,\&\, \cdots \,\&\, l_p? & \triangleq \text{precondition of Axiom P17} \\
\quad \left[\& B_1 \,\&\, \cdots \,\&\, B_n\right], Q) & \\
wlp(\textbf{await wait } \left[\& l_1? \,\&\, \cdots \,\&\, l_p?\right] & \triangleq \text{precondition of Axiom P18} \\
\quad \left[\& B_1 \,\&\, \cdots \,\&\, B_n\right], Q) & \\
wlp(|||_{i=1}^n S_i, Q) & \triangleq wlp(\Box_{i=1}^n (S_i; |||_{j=1, j\neq i}^n S_j), Q) \\
 & \quad \text{if } awaitFree(|||_{i=1}^n S_i)
\end{array}
$$

Definition Q11 (WLP for Nondeterministic Choice)

$$
\begin{array}{l}
wlp(S_1 \Box S_2, Q) \\
\quad \triangleq \forall h.\, interleave(\mathcal{H}, h) \Rightarrow \\
\qquad \textbf{if } ready(S_1 \Box S_2, \bar{\mathcal{A}}, h) \textbf{ then} \\
\qquad\quad pickReadyBranch(S_1, S_2, wlp(S_1, Q), wlp(S_2, Q), \bar{\mathcal{A}}, h) \\
\qquad \textbf{else if } enabled(S_1 \Box S_2, \bar{\mathcal{A}}, h) \textbf{ then} \\
\qquad\quad \forall h'.\, \big(interleave(h, h') \wedge ready(S_1 \Box S_2, \bar{\mathcal{A}}, h')\big) \Rightarrow \\
\qquad\qquad pickReadyBranch(S_1, S_2, wlp(S_1, Q), wlp(S_2, Q), \bar{\mathcal{A}}, h') \\
\qquad \textbf{else} \\
\qquad\quad (\mathcal{G}_c)^{\mathcal{H}}_{h^\frown[\textbf{self.release}]} \\
\qquad\quad \wedge\, \forall \bar{a}, h'.\, \big(release(h, h', \bar{a}) \wedge ready(S_1^\star \Box S_2^\star, \bar{a}, h')\big) \Rightarrow \\
\qquad\qquad\quad pickReadyBranch(S_1^\star, S_2^\star, wlp(S_1^\star, Q), wlp(S_2^\star, Q), \bar{a}, h') \\
\qquad \textbf{fi fi}
\end{array}
$$

The WLP for the **while** loop poses a particular challenge. The operational semantics suggests the following definition:

$$
\begin{array}{l}
wlp(\textbf{while } B \textbf{ do } S \textbf{ od}) \triangleq \textbf{if } B \textbf{ then } wlp(S, wlp(\textbf{while } B \textbf{ do } S \textbf{ od}, Q)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } Q \textbf{ fi}.
\end{array}
$$

However, the recursion is not well-founded, since $wlp(\textbf{while } B \textbf{ do } S \textbf{ od}, Q)$ appears on both sides of the equation. Instead of resorting to fixed point theory to give a

meaning to the above equation, we can use the following trick, inspired by Dijkstra [Dij75]. Let $H_k(\textbf{while } B \textbf{ do } S \textbf{ od}, Q)$ denote the WLP for **while** B **do** S **od** when at most k iterations of the loop are performed. Thus:

$$\begin{aligned}
&H_0(\textbf{while } B \textbf{ do } S \textbf{ od}, Q) \triangleq Q\\
&H_1(\textbf{while } B \textbf{ do } S \textbf{ od}, Q) \triangleq wlp(\textbf{if } B \textbf{ then } S \textbf{ fi}, Q)\\
&H_2(\textbf{while } B \textbf{ do } S \textbf{ od}, Q) \triangleq wlp(\textbf{if } B \textbf{ then } S \textbf{ fi}, wlp(\textbf{if } B \textbf{ then } S \textbf{ fi}, Q)).
\end{aligned}$$

In general, for $k > 0$, we have

$$H_k(\textbf{while } B \textbf{ do } S \textbf{ od}, Q) \triangleq wlp(\textbf{if } B \textbf{ then } S \textbf{ fi}, H_{k-1}(\textbf{while } B \textbf{ do } S \textbf{ od}, Q)).$$

We do the same for **inv** I **while** B **do** S **od**, except that we require that I is satisfied at every iteration:

$$\begin{aligned}
&H_0(\textbf{inv } I \textbf{ while } B \textbf{ do } S \textbf{ od}, Q) \triangleq I \wedge Q\\
&H_1(\textbf{inv } I \textbf{ while } B \textbf{ do } S \textbf{ od}, Q) \triangleq I \wedge wlp(\textbf{if } B \textbf{ then } S \textbf{ fi}, I \wedge Q)\\
&H_2(\textbf{inv } I \textbf{ while } B \textbf{ do } S \textbf{ od}, Q) \triangleq I \wedge wlp(\textbf{if } B \textbf{ then } S \textbf{ fi},\\
&\qquad\qquad I \wedge wlp(\textbf{if } B \textbf{ then } S \textbf{ fi}, I \wedge Q))\\
&\quad\vdots\\
&H_k(\textbf{inv } I \textbf{ while } B \textbf{ do } S \textbf{ od}, Q) \triangleq I \wedge wlp(\textbf{if } B \textbf{ then } S \textbf{ fi},\\
&\qquad\qquad H_{k-1}(\textbf{inv } I \textbf{ while } B \textbf{ do } S \textbf{ od}, Q)).
\end{aligned}$$

We can then define the WLP for **while** loops in terms of H_k as follows:

Definition Q12 (WLP for While Loop)

$$wlp([\textbf{inv } I] \textbf{ while } B \textbf{ do } S \textbf{ od}, Q) \triangleq \exists k.\ H_k([\textbf{inv } I] \textbf{ while } B \textbf{ do } S \textbf{ od}, Q)$$

5.5 Verification of a Class's Assume–Guarantee Specification

In the previous sections, we reviewed the axioms and proof rules necessary to reason about statements in a method body. In this section, we will see how to apply these to verify the assume–guarantee specification provided in the **asum** and **guar** clauses of a Creol class c and of the interfaces it implements.

The basic idea is as follows: We have the code for class c and all its superclasses and superinterfaces. From the **asum** and **guar** clauses, we derive a class assumption $\mathcal{A}_c$ and a class guarantee $\mathcal{G}_c$. Using backward construction, we verify that the initialization code establishes the guarantee and that each method maintains it. Backward construction gives rise to verification conditions of the form $P \Rightarrow wlp(\langle \text{code} \rangle, Q)$, which must be proved in first-order logic. If we can prove all of the verification conditions, we have proved that the class's implementation respects the guarantee $\mathcal{G}_c$ with respect to the assumption $\mathcal{A}_c$ (assuming that the WLPs presented in Section 5.4 are sound).

The first step is to compute the assumption and the guarantee for the class of interest. The derived assume–guarantee specification is obtained from the **asum** and **guar** clauses of the class and of all the interfaces that it implements, projected onto

their respective alphabets. Let i be an interface that inherits $j_1, \ldots, j_n$. The alphabet $o{:}i$ of an object o typed by interface i is the set

$$\begin{aligned} o{:}i \triangleq\ & o{:}j_1 \cup \cdots \cup o{:}j_n \\ & \cup \{ [o' \rightarrow o.m(\bar{v})]^k, [o' \leftarrow o.m(\bar{v}; \bar{w})]^k \mid m \text{ is declared by } i \text{ and and } o' \neq o \} \\ & \cup \{ [o \rightarrow o'.m(\bar{v})]^k, [o \leftarrow o'.m(\bar{v}; \bar{w})]^k \mid m \text{ is declared or inherited} \\ & \qquad\qquad\qquad\qquad\qquad\qquad \text{by } i\text{'s cointerface and } o' \neq o \}. \end{aligned}$$

Let c be a class that implements some interfaces $j_1, \ldots, j_n$. Let φ_x stand for the **asum** clause of $x \in \{c, j_1, \ldots, j_n\}$, and similarly let ψ_x stand for the **guar** clause of x. The *derived class assumption* $\mathcal{A}_c$ and the *derived class guarantee* $\mathcal{G}_c$ for class c are defined as follows:

$$\begin{aligned} \mathcal{A}_c &\triangleq (\varphi_c)^{\mathcal{H}}_{\mathcal{H}/\mathbf{self}} \wedge (\varphi_{j_1})^{\mathcal{H}}_{\mathcal{H}/\mathbf{self}:j_1} \wedge \cdots \wedge (\varphi_{j_n})^{\mathcal{H}}_{\mathcal{H}/\mathbf{self}:j_n} \\ \mathcal{G}_c &\triangleq (\psi_c)^{\mathcal{H}}_{\mathcal{H}/\mathbf{self}} \wedge (\psi_{j_1})^{\mathcal{H}}_{\mathcal{H}/\mathbf{self}:j_1} \wedge \cdots \wedge (\psi_{j_n})^{\mathcal{H}}_{\mathcal{H}/\mathbf{self}:j_n}. \end{aligned}$$

Our goal is to verify the guarantee, taking the assumption for granted. The guarantee is expected to hold after initialization of the object, be maintained by all methods, and hold before and after all processor releases.

Let us begin with object initialization. The initialization code is spread across class c and all its ancestor classes $c_1, \ldots, c_n$, enumerated in postorder with respect to the inheritance tree. Let $\bar{\mathcal{A}}$ denote the list of writable attributes, including inherited attributes, and let $\bar{\mathcal{D}}$ denote the list of default values to assign to $\bar{\mathcal{A}}$. (For example, if $\mathcal{A}_i$ has type **bool**, then $\mathcal{D}_i \equiv$ **false**.) Furthermore, let $\bar{x}, \bar{x}_1, \ldots, \bar{x}_n$ denote the context parameters of $c, c_1, \ldots, c_n$, and let $\bar{e}_1, \ldots, \bar{e}_n$ denote the lists of arguments corresponding to $\bar{x}_1, \ldots, \bar{x}_n$. Finally, let $\langle$initializer$\rangle$ stand for the following code:

$\bar{\mathcal{A}} := \bar{\mathcal{D}}$;
$\bar{x}_n@c_n := \bar{e}_n$; ...; $\bar{x}_1@c_1 := \bar{e}_1$;
$init@c_1()$; ...; $init@c_n()$; $init@c()$;
$run()$

For simplicity, we assume that every class provides an *init* method and that c declares or inherits a *run* method. The general case would be handled by removing calls to nonexistent methods. To verify that the object initialization leaves the object in a state that satisfies the guarantee, we must prove the correctness formula

$\{\mathcal{A}_c \wedge \mathcal{H} = [parent(\mathbf{self}) \rightarrow \mathbf{self}.\mathbf{new}\ c(\bar{p})]\}$
$\langle$initializer$\rangle$;
$\mathcal{H} := \mathcal{H} \frown [\mathbf{self}.\mathbf{initialized}]$
$\{\mathcal{G}_c\}$,

where $\bar{p}$ is a list of fresh variables. Informally, this formula can be read as follows:

> *If we start in a state such that the object has just been created and the assumption $\mathcal{A}_c$ holds, and we execute the object initialization code, then the guarantee $\mathcal{G}_c$ should hold at termination.*

To prove the correctness formula, it is sufficient to prove the following implication, which builds on the *wlp* function:

$$\begin{aligned} & \big(\mathcal{A}_c \wedge \mathcal{H} = [parent(\mathbf{self}) \rightarrow \mathbf{self}.\mathbf{new}\ c(\bar{p})]\big) \Rightarrow \\ & wlp(\langle \text{initializer} \rangle\ ,\ (\mathcal{G}_c)^{\mathcal{H}}_{\mathcal{H} \frown [\mathbf{self}.\mathbf{initialized}]}). \end{aligned}$$

The implication is equivalent to

$$\big(\mathcal{A}_c \Rightarrow wlp(\langle \text{initializer} \rangle\,,\, (\mathcal{G}_c)^{\mathcal{H}}_{\mathcal{H}^\frown[\mathbf{self.initialized}]})\big)^{\mathcal{H}}_{[parent(\mathbf{self})\rightarrow \mathbf{self.new}\ c(\bar{p})]}.$$

Within the initialization code, the guarantee $\mathcal{G}_c$ must hold whenever the processor is released or a local reentry is performed. Since this requirement is already expressed in the WLPs for $l?(\bar{z})$, **await** g, $S_1 \,\square\, S_2$, and $|||_{i=1}^{n} S_i$, there are no separate verification conditions for these cases. If we can prove the above verification condition, we are assured that the guarantee will hold when the initialization code releases the processor.

Just as the initialization code must establish the guarantee, every method m declared by a class c' that is either c or an ancestor of c must preserve it. Consider the following method declaration:

op $m($**in** $x_1 : \tau_1, \ldots, x_n : \tau_n$ **out** $y_1 : \tau'_1, \ldots, y_p : \tau'_p)$ **is**
  var $v_1 : \tau''_1, \ldots, v_q : \tau''_q$;
  S

Let $\bar{\mathcal{D}}_y$ and $\bar{\mathcal{D}}_v$ denote the lists of default values of $\bar{y}$ and $\bar{v}$, respectively. Now, let $\langle \text{method} \rangle$ stand for the following code:

$\bar{y} := \bar{\mathcal{D}}_y$;
$\bar{v} := \bar{\mathcal{D}}_v$;
S

To verify that the method m maintains the guarantee, we must prove the correctness formula

$\{\mathcal{A}_c$
$\wedge\ \mathcal{G}_c$
$\wedge\ lwf(\mathcal{H}, \mathbf{self})$
$\wedge\ mayAcquireProcessor(\mathcal{H}, \mathbf{self}, \mathbf{caller}, \mathbf{label})$
$\wedge\ [\mathbf{caller}\rightarrow\mathbf{self}.m@c'(\bar{x})]^{\mathbf{label}}$ **in** $\mathcal{H}$
$\wedge\ [\mathbf{caller}\leftarrow\mathbf{self}]^{\mathbf{label}}$ **not in** $\mathcal{H}\}$
$\langle \text{method} \rangle$;
$\mathcal{H} := \mathcal{H}^\frown[\mathbf{caller}\leftarrow\mathbf{self}.m@c'(\bar{x};\bar{y})]^{\mathbf{label}}$
$\{\mathcal{G}_c\}$.

Both the precondition and the postcondition contain the guarantee $\mathcal{G}_c$. In addition, the precondition is strengthened by the assumption $\mathcal{A}_c$, and by various program-independent properties that must hold when a method starts executing. Using backward construction, we obtain the verification condition

$\big(\mathcal{A}_c$
$\wedge\ \mathcal{G}_c$
$\wedge\ lwf(\mathcal{H}, \mathbf{self})$
$\wedge\ mayAcquireProcessor(\mathcal{H}, \mathbf{self}, \mathbf{caller}, \mathbf{label})$
$\wedge\ [\mathbf{caller}\rightarrow\mathbf{self}.m@c'(\bar{x})]^{\mathbf{label}}$ **in** $\mathcal{H}$
$\wedge\ [\mathbf{caller}\leftarrow\mathbf{self}]^{\mathbf{label}}$ **not in** $\mathcal{H}\big) \Rightarrow$
$wlp\big(\langle \text{method} \rangle\,,\, (\mathcal{G}_c)^{\mathcal{H}}_{\mathcal{H}^\frown[\mathbf{caller}\leftarrow\mathbf{self}.m@c'(\bar{x};\bar{y})]^{\mathbf{label}}}\big)$.

5.6 Compositional Reasoning

In the previous section, we saw how to verify a class guarantee by identifying and proving verification conditions for the class's initializer and methods. The next and final step consists of deducing global properties about the entire system from the guarantees of the classes that compose it. This presents three main difficulties:

1. A system consists of objects, not classes, yet all we can verify so far are class guarantees.
2. As a result of Creol's asynchronous communication model, which allows method overtaking, the local histories of different objects in the system may disagree on common events.
3. Classes can make assumptions about the environment, and these assumptions should be verified when objects are composed together to form larger systems.

We will review two solutions to these problems.

The first solution is based on Dovland et al. [DJO05, DJO08]. In their papers, they use a single class invariant $\mathcal{J}_c$ that serves both as an assumption and as a guarantee. From the class invariant $\mathcal{J}_c$, they derive the following *composable object invariant* $\mathcal{J}_{o:c(\bar{e})}$ for an instance o of class c constructed with $\bar{e}$ as the arguments corresponding to the context parameters $\bar{x}$:

$$\mathcal{J}_{o:c(\bar{e})} \triangleq \exists \bar{a}.\ (\mathcal{J}_c)^{\mathcal{H}}_{o,\bar{e},\bar{a}}.$$

To avoid name clashes with attributes of other objects when the invariant is composed with other invariants, the attributes $\bar{a}$ are hidden behind an existential quantifier. The history variable $\mathcal{H}$ may occur free in $\mathcal{J}_{o:c(\bar{e})}$. The set of possible histories for o is $\{h \mid h \preceq h' \wedge (\mathcal{J}_{o:c(\bar{e})})^{\mathcal{H}}_{h'}\}$—that is, the prefix closure of the set of histories that satisfy the invariant.

Asynchronous communication is handled by requiring that the class invariant $\mathcal{J}_c$ respects the following *asynchronous input property*:

$$\forall h, h_{\text{in}}.\ (wf(h) \wedge h \in (\mathcal{H} \,|||\, h_{\text{in}}) \wedge h_{\text{in}}/(out_o \cup ctl_o) = \epsilon \wedge \mathcal{J}_c) \Rightarrow (\mathcal{J}_c)^{\mathcal{H}}_{h}.$$

Informally, this formula means the following:

> *If $\mathcal{J}_c$ holds for $\mathcal{H}$, then $\mathcal{J}_c$ must also hold if we merge additional input events h_{in} into the history and the result is well-formed.*

The formula is justified as follows [DJO08]:

> In the asynchronous setting, an object may independently decide to send a message and, due to overtaking, messages may arrive in a different order than sent. The invariant of an object should therefore restrict messages seen by the object, but allow the existence of additional input not processed yet.

If two objects o_1 and o_2 execute in parallel in the same system, their *combined invariant* $\mathcal{J}_{o_1:c_1(\bar{e}_1)\|o_2:c_2(\bar{e}_2)}$ is the conjunction of the object invariants for o_1 and o_2:

$$\mathcal{J}_{o_1:c_1(\bar{e}_1)\|o_2:c_2(\bar{e}_2)} \triangleq \left(\mathcal{J}_{o_1:c_1(\bar{e}_1)}\right)^{\mathcal{H}}_{\mathcal{H}/o_1} \wedge \left(\mathcal{J}_{o_2:c_2(\bar{e}_2)}\right)^{\mathcal{H}}_{\mathcal{H}/o_2}.$$

This can be taken further to compute a *global invariant* $\mathcal{J}_{\text{sys}}$ for the system:

$$\mathcal{J}_{\text{sys}} \triangleq \bigwedge_{o:c(\bar{e}) \textbf{ in } \mathcal{H}} \left(\mathcal{J}_{o:c(\bar{e})}\right)^{\mathcal{H}}_{\mathcal{H}/o}.$$

(We write $o{:}c(\bar{e})$ **in** $\mathcal{H}$ as an abbreviation for $[o'{\rightarrow}o.\textbf{new}\ c(\bar{e})]$ **in** $\mathcal{H}$.) The conjunction operator ranges over all the objects in the system, which is a finite number at any point during the system's execution. Global properties G about the system can be verified by proving $\mathcal{J}_{\text{sys}} \Rightarrow G$. Since the global invariant is defined directly in terms of the class invariants, the proof system is compositional.

With the approach of Dovland et al., class invariants are combined together using little more than the logical conjunction operator. On the other hand, the asynchronous input property prevents classes from making useful assumptions about the environment; for example, a class invariant may not state that a call to *write* can occur only after a call to *open*.

The second approach we will review is significantly more complex but also more powerful. To account for the unordered asynchronous communication taking place in Creol, we will distinguish between the moment when a message is sent and the moment when it becomes visible to the receiver. This leads us to replace the ambiguous $[o{\rightarrow}o'.m@c(\bar{v})]^k$ and $[o{\leftarrow}o'.m@c(\bar{v};\bar{w})]^k$ events with the following:

$[o{\rightharpoonup}o'.m@c(\bar{v})]^k$	asynchronous invocation emission
$[o{\rightharpoondown}o'.m@c(\bar{v})]^k$	asynchronous invocation reception
$[o{\leftharpoonup}o'.m@c(\bar{v};\bar{w})]^k$	asynchronous reply emission
$[o{\leftharpoondown}o'.m@c(\bar{v};\bar{w})]^k$	asynchronous reply reception

To see how this helps us, consider the following example. A producer P asynchronously calls $put(3)$ on a buffer B, then $put(4)$, without waiting for the replies. Using old-style $\rightarrow$ and $\leftarrow$ events, this would be recorded in P's local history as

$$[P{\rightarrow}B.put(3)]^3 \frown [P{\rightarrow}B.put(4)]^4.$$

Now, suppose that the invocation messages arrive out of order. We would then have the following sequence in B's local history:

$$[P{\rightarrow}B.put(4)]^4 \frown [P{\rightarrow}B.put(3)]^3.$$

Clearly, there is no global history $\mathcal{H}$ that can agree with both P's and B's local history, since they disagree on the order of these two shared events. Dovland et al. solved this problem by preventing the receiver from contradicting the sender, and called this the asynchronous input property. But if we distinguish between sending and receiving a message, there is no contradiction anymore; P's local history contains the sequence

$$[P{\rightharpoonup}B.put(3)]^3 \frown [P{\rightharpoonup}B.put(4)]^4,$$

B's local history contains

$$[P{\rightharpoondown}B.put(4)]^4 \frown [P{\rightharpoondown}B.put(3)]^3,$$

and the global history could then have the sequence

$$[P{\rightharpoonup}B.put(3)]^3 \frown [P{\rightharpoonup}B.put(4)]^4 \frown [P{\rightarrow}B.put(4)]^4 \frown [P{\rightarrow}B.put(3)]^3$$

as a subsequence.

In this new setting, we define a *composable object assumption* $\mathcal{A}_{o:c(\bar{e})}$ and a *composable object guarantee* $\mathcal{G}_{o:c(\bar{e})}$ as follows:

$$\begin{array}{lcl}\mathcal{A}_{o:c(\bar{e})} & \triangleq & \exists \bar{a}.\ (\mathcal{A}_c)^{\mathbf{self},\bar{x},\bar{\mathcal{A}}}_{o,\bar{e},\bar{a}} \\ \mathcal{G}_{o:c(\bar{e})} & \triangleq & \exists \bar{a}.\ (\mathcal{G}_c)^{\mathbf{self},\bar{x},\bar{\mathcal{A}}}_{o,\bar{e},\bar{a}}.\end{array}$$

Here, we assume that the class assumption $\mathcal{A}_c$ and the class guarantee $\mathcal{G}_c$ use the new-style $\rightharpoonup$, $\rightarrow$, $\leftharpoondown$, and $\leftarrow$ events instead of $\rightarrow$ and $\leftarrow$. For two objects o_1 and o_2 executing in parallel, the *combined guarantee* is

$$\mathcal{G}_{o_1:c_1(\bar{e}_1)\|o_2:c_2(\bar{e}_2)} \triangleq \mathcal{G}_{o_1:c_1(\bar{e}_1)} \wedge \mathcal{G}_{o_2:c_2(\bar{e}_2)} \wedge gwf(\mathcal{H}),$$

where *gwf* is an adaptation of the well-formedness predicate of Definition T19 to our new setting, as defined by the following equations:

$$\begin{array}{lcl}
gwf(\epsilon) & \triangleq & \mathbf{true} \\
gwf(h \frown [o{\rightarrow}o'.\mathbf{new}\ c(\bar{v})]) & \triangleq & gwf(h) \wedge o' \notin objectIds(h) \wedge parent(o') = o \\
gwf(h \frown [o{\rightharpoonup}o'.m@c(\bar{v})]^k) & \triangleq & gwf(h) \wedge [o{\rightharpoonup}o'.m@c(\bar{v})]^k\ \mathbf{not\ in}\ h \\
gwf(h \frown [o{\rightarrow}o'.m@c(\bar{v})]^k) & \triangleq & gwf(h) \wedge [o{\rightharpoonup}o'.m@c(\bar{v})]^k\ \mathbf{in}\ h \\
 & & \wedge\ [o{\rightarrow}o'.m@c(\bar{v})]^k\ \mathbf{not\ in}\ h \\
gwf(h \frown [o{\leftharpoondown}o'.m@c(\bar{v};\bar{w})]^k) & \triangleq & gwf(h) \wedge [o{\rightarrow}o'.m@c(\bar{v})]^k\ \mathbf{in}\ h \\
 & & \wedge\ [o{\leftharpoondown}o'.m@c(\bar{v};\bar{w})]^k\ \mathbf{not\ in}\ h \\
gwf(h \frown [o{\leftarrow}o'.m@c(\bar{v};\bar{w})]^k) & \triangleq & gwf(h) \wedge [o{\leftharpoondown}o'.m@c(\bar{v};\bar{w})]^k\ \mathbf{in}\ h \\
 & & \wedge\ [o{\leftarrow}o'.m@c(\bar{v};\bar{w})]^k\ \mathbf{not\ in}\ h \\
gwf(h \frown [o.\mathbf{initialized}]) & \triangleq & gwf(h) \wedge [o.\mathbf{initialized}]^k\ \mathbf{not\ in}\ h \\
gwf(h \frown [o.\mathbf{release}]) & \triangleq & gwf(h) \\
gwf(h \frown [o.\mathbf{reenter}]^k) & \triangleq & gwf(h).
\end{array}$$

The *combined assumption* is then

$$\mathcal{A}_{o_1:c_1(\bar{e}_1)\|o_2:c_2(\bar{e}_2)} \triangleq \mathcal{G}_{o_1:c_1(\bar{e}_1)\|o_2:c_2(\bar{e}_2)} \Rightarrow (\mathcal{A}_{o_1:c_1(\bar{e}_1)} \wedge \mathcal{A}_{o_2:c_2(\bar{e}_2)}).$$

The assumption is weakened by the combined guarantee. For the entire system, the *global guarantee* is

$$\mathcal{G}_{\text{sys}} \triangleq \bigwedge_{o:c(\bar{e})\ \mathbf{in}\ \mathcal{H}} (\mathcal{G}_{o:c(\bar{e})})^{\mathcal{H}}_{\mathcal{H}/o} \ \wedge\ gwf(\mathcal{H})$$

and the *global assumption* is

$$\mathcal{A}_{\text{sys}} \triangleq \mathcal{G}_{\text{sys}} \Rightarrow \bigwedge_{o:c(\bar{e})\ \mathbf{in}\ \mathcal{H}} (\mathcal{A}_{o:c(\bar{e})})^{\mathcal{H}}_{\mathcal{H}/o}.$$

Since the system has no environment that can fulfill its assumption, we require that $\mathcal{A}_{\text{sys}} \Leftrightarrow \mathbf{true}$. If this is the case, then $\mathcal{G}_{\text{sys}}$ is valid for the system, and we can verify global properties P about the system by proving $\mathcal{G}_{\text{sys}} \Rightarrow P$.

5.7 Contributions

The proof system presented in this chapter is firmly rooted in the work of Dovland, Johnsen, and Owe. In their 2005 paper [DJO05], they introduced the first proof system for Creol, based on WLPs and an encoding. Their 2008 paper [DJO08] presents a higher-level proof system that abstracts away Creol labels and that requires the programmer to specify pre- and postconditions for individual methods, in addition to the class's semantic specification. In both papers, the focus is on simplicity of reasoning.

In this thesis, we followed a somewhat different approach. The primary focus is on soundness and completeness with respect to the language's operational semantics. The main differences between the proof system we presented here and the proof system developed by Dovland et al. in their 2005 paper are listed below:

- Instead of relying on a single invariant that simultaneously serves as assumption and guarantee, we expect the programmer to supply the assumption and the guarantee separately. This makes it possible to restrict the possible parallel activity using the assumption. (A single invariant cannot achieve that, because parallel activity may occur at any point in the program, where the guarantee might not hold.)
- As a step toward completeness, we record three additional types of events ($[o.\textbf{initialized}]$, $[o.\textbf{release}]$, and $[o.\textbf{reenter}]^k$) in the history, and for invocation and reply events, we record the sequence numbers associated with the method calls. This additional information allows us to precisely capture the behavior of Creol's idiosyncratic **await** statement and local reentry feature, among other things. The WLPs and the verification conditions associated with a class have been adapted to deal with this information.
- We provide sound and complete axioms for **await** $l?$ and $l?(\bar{z})$ with respect to later versions of the Creol semantics. In the 2005 paper, the **await** $l?$ statement always releases the processor, even if the reply associated with l has already arrived, and the $l?(\bar{z})$ statement blocks if l is a pending local call.
- We give a syntactically simpler (but logically equivalent) WLP for the general case of **await** g.
- The axiomatization of the nondeterministic statements $S_1 \,\square\, S_2$ and $|||_{i=1}^{n} S_i$ is original to this thesis. Dovland et al. did not consider them in their papers.
- The treatment of compositional reasoning in an asychronous setting by distinguishing emission from reception is an old idea [dRdB$^+$01], but due to the complexity it introduces it has not been applied to Creol previously.

To understand a program you must become both the machine and the program.

— Alan J. Perlis (1982)

Chapter 6

An Assertion Analyzer Based on the Proof System

The Creol language lets the programmer embed four types of assertions directly in the program text: assumptions (**asum**), guarantees (**guar**), inline assertions (**prove**), and loop invariants (**inv**). The assertion analyzer is a tool that takes the complete source code of a class as input, including superclasses and superinterfaces, and attempts to verify the assertions that appear in it. The output is a report that indicates whether the verification was successful.

In this chapter, we will review the tool's design and implementation. Section 6.1 provides an overview of the tool's capabilities. Section 6.2 presents its internal architecture. Section 6.3 explains how Maude is used to parse Creol programs. Section 6.4 explains how Creol statements and assertions are represented as Maude terms. Section 6.5 describes how the tool generates the verification report. Section 6.6 explains how the tool computes the weakest liberal precondition (WLP) of Creol statements. And finally, Section 6.7 explains how the tool normalizes, simplifies, untypes, and pretty-prints assertions.

6.1 The Assertion Analyzer at a Glance

We will start by having a look at the input and output of the assertion analyzer, without going into details. (Appendix A provides a comprehensive user's guide.) When invoking the assertion analyzer, we must supply the following input:

- The source code of a Creol class and of all its superclasses and superinterfaces.
- The declarations of the necessary custom data types and functions.
- An optional set of assertion simplification rules.
- A **verify class** command that specifies the class to verify and the simplification rules to use.

Using the approach presented in Section 5.5, the tool computes the verification conditions for the class's initialization code and methods (including inherited ones).

As a by-product, the tool also verifies the inline assertions and the loop invariants found in the method bodies. The output is a report of the following form:

Verification of class *c*

Initialization code:
❑ Establishes the guarantee iff $\hat{Q}_0$ holds

Method m_1 of c_1:
❑ Maintains the guarantee iff $\hat{Q}_1$ holds

⋮

Method m_n of c_n:
❑ Maintains the guarantee iff $\hat{Q}_n$ holds

The assertions $\hat{Q}_0, \hat{Q}_1, \ldots, \hat{Q}_n$ are proof obligations that we must carry out (by hand or using a mechanical theorem prover) to verify that the code respects the class's assume–guarantee specification. The hat on $\hat{Q}_i$ is there to remind us that the assertions are pretty-printed, as explained in Section 6.7.

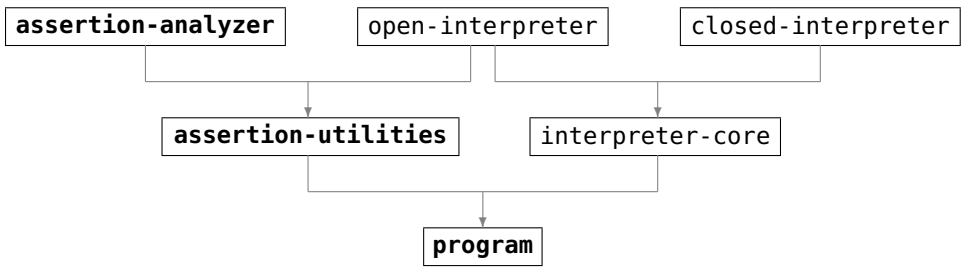
Assuming the presence of a set of assertions $\{P_1, \ldots, P_k\}$ expressing the defining properties of the non-logical symbols from our assertion language (such as $+$, **in**, and **ew**), we must prove for each $\hat{Q}_i$ that the sequent $P_1, \ldots, P_k \vdash \hat{Q}_i$ is valid in the sense of Section 2.2. If the sequent is not valid, we can conclude that the corresponding initialization code or method body violates the class guarantee.

Sometimes, the tool can automatically determine whether the sequent is valid, in which case the judgment will be "☑ Establishes the guarantee", "❑ Fails to establish the guarantee", "☑ Maintains the guarantee", or "❑ Breaks the guarantee".

Complications arises for methods whose body contains an inline assertion, a **while** loop, or a nondeterministic merge statement. For these, the tool can produce the judgment "❑ Maintains the guarantee if $\hat{Q}_i$ holds" (with "if" instead of "iff") or give up entirely and state "❑ Don't know". We will see why this happens and what it means in Section 6.6.

6.2 Architecture of the Assertion Analyzer

The assertion analyzer shares a large amount of its source code with the interpreters described in Section 4.5. The source code for these tools is spread across six source files, whose contents are listed in Appendix B. The dependency graph below shows how these files relate to each other:



An arrow pointing from *x* to *y* indicates that the file `creol-x.maude` relies on modules declared in `creol-y.maude`. The three files shown in bold are those that are necessary to run the assertion analyzer.

- The file `creol-program.maude` defines the concrete Creol language syntax and converts Creol programs to system configurations. It also provides several auxiliary functions for manipulating system configurations.
- The file `creol-assertion-utilities.maude` declares auxiliary functions to extract the assume–guarantee specification of a class and to perform basic logical simplifications.
- The file `creol-assertion-analyzer.maude` contains the modules that are specific to the assertion analyzer. These modules provide functions for computing WLPs and verification conditions, generating a verification report with proof obligations, and massaging the assertions.

6.3 Parsing Creol Programs

The Creol classes and interfaces provided to the assertion analyzer are specified in a Maude-compatible syntax that closely follows the abstract syntax used throughout this thesis. The example below shows the correspondence between the two syntaxes:

interface *Counter*
begin
with any:
  op *inc*
end

```
interface 'Counter
begin
with any :
  op 'inc
end
```

class *SimpleCounter*
  implements *Counter*
begin
  var n : **int**
with any:
  op *inc* **is**
    $n := n + 1$
  guar $G(\mathcal{H})$
end

```
class 'SimpleCounter
  implements 'Counter
begin
  var 'n : int
with any :
  op 'inc is
    'n := 'n plus 1
  guar 'G[~H~]
end
```

The Creol syntax can be specified by a context-free grammar [Fje05, HMU06]. A *context-free grammar* $\mathcal{G} = (N, \Sigma, P, S)$ consists of a finite set of *terminal symbols* N, a finite set of *nonterminal symbols* Σ that is disjoint from N, a finite set of *production rules* P of the form $A ::= \alpha$, where A is a nonterminal and α is a string over terminals and nonterminals, and a distinguished *start symbol* $S \in \Sigma$. The empty string is written ϵ. The *language* $L(\mathcal{G})$ associated with the grammar $\mathcal{G}$ is the set of strings over Σ that can be generated from S by using the production rules from P as left-to-right rewrite rules. Two grammars $\mathcal{G}$ and $\mathcal{G}'$ are *equivalent* if $L(\mathcal{G}) = L(\mathcal{G}')$.

Although Maude does not understand context-free grammars, we can use sort hierarchies, mixfix operators, and attributes such as `assoc` and `prec` to mimic a grammar. Consider the following incomplete fragment of a grammar for Creol class declarations, with angle brackets (⟨⟩) enclosing nonterminal symbols:

⟨*class*⟩ ::= **class** ⟨*id-with-params*⟩ ⟨*class-head-clauses-opt*⟩
begin ⟨*var-clause-opt*⟩ ⟨*class-mtds-opt*⟩
⟨*asum-clause-opt*⟩ ⟨*guar-clause-opt*⟩ **end**

⟨*id-with-params*⟩ ::= ⟨*id*⟩
⟨*id-with-params*⟩ ::= ⟨*id*⟩ (⟨*typed-id-list*⟩)

⟨*class-head-clauses-opt*⟩ ::= ϵ
⟨*class-head-clauses-opt*⟩ ::= ⟨*class-head-clause*⟩ ⟨*class-head-clauses-opt*⟩

⟨*class-head-clause*⟩ ::= **implements** ⟨*super-list*⟩
⟨*class-head-clause*⟩ ::= **contracts** ⟨*super-list*⟩
⟨*class-head-clause*⟩ ::= **inherits** ⟨*super-list*⟩

⟨*super-list*⟩ ::= ⟨*super*⟩
⟨*super-list*⟩ ::= ⟨*super*⟩ , ⟨*super-list*⟩

⟨*super*⟩ ::= ⟨*id-with-args*⟩

⟨*id-with-args*⟩ ::= ⟨*id*⟩
⟨*id-with-args*⟩ ::= ⟨*id*⟩ (⟨*exp-list*⟩)

⟨*var-clause-opt*⟩ ::= ϵ
⟨*var-clause-opt*⟩ ::= **var** ⟨*typed-id-list*⟩

⟨*class-mtds-opt*⟩ ::= ϵ
⟨*class-mtds-opt*⟩ ::= ⟨*mtd-decl-grp*⟩
⟨*class-mtds-opt*⟩ ::= ⟨*class-mtds-opt*⟩ **with** ⟨*type*⟩ : ⟨*mtd-decl-grp*⟩

⟨*asum-clause-opt*⟩ ::= ϵ
⟨*asum-clause-opt*⟩ ::= **asum** ⟨*assn*⟩

⟨*guar-clause-opt*⟩ ::= ϵ
⟨*guar-clause-opt*⟩ ::= **guar** ⟨*assn*⟩

Most of these production rules can be implemented in Maude by defining a mixfix operator, substituting sorts for nonterminals. For example, the declarations

```
op implements_ : SuperList -> ClassHeadClause [ctor] .

op _[_] : Id ExpList -> IdWithArgs [ctor] .
```

implement the production rules

⟨*class-head-clause*⟩ ::= **implements** ⟨*super-list*⟩
⟨*id-with-args*⟩ ::= ⟨*id*⟩ (⟨*exp-list*⟩)

For lists, a natural Maude representation relies on the `assoc` attribute:

```
op _,_ : SuperList SuperList -> SuperList [ctor assoc] .
```

In Maude, parentheses can be used to resolve parsing ambiguities and for applying functions (non-mixfix operators). To avoid confusion, we systematically use square brackets to represent Creol parentheses, even in contexts where Maude's parentheses would have worked equally well.

In Section 4.2, we specified the precedence of Creol's infix operators. For example, we mentioned that $\neg$ binds more strongly than $\wedge$, which in turn binds more strongly than $\vee$. Using the `prec` attribute, we can easily implement this in Maude:

```
op !_ : BExp -> BExp [ctor prec 5] .
op _&&_ : BExp BExp -> BExp [ctor assoc comm prec 13] .
op _||_ : BExp BExp -> BExp [ctor assoc comm prec 15] .
```

(Like in C and Java, the operators `!`, `&&`, and `||` represent $\neg$, $\wedge$, and $\vee$, respectively.) We also take this as an opportunity to supply other attributes that make sense, such as `assoc` and `comm` for `&&` and `||`.

For production rules like $\langle \textit{super} \rangle ::= \langle \textit{id-with-args} \rangle$, which are of the form $A ::= A'$, Maude doesn't let us write

```
*** WRONG
op _ : IdWithArgs -> Super [ctor] .
```

Instead, we must use a subsort declaration:

```
subsort IdWithArgs < Super .
```

Production rules with ϵ on their right-hand side (called *empty production rules* or *ϵ production rules*) also require special care. A straightforward conversion leaves us with the following invalid operator declaration:

```
*** WRONG
op : -> GuarClauseOpt [ctor] .
```

To work around this, we can insert a dummy `epsilon` token as follows:

```
op epsilon : -> GuarClauseOpt [ctor] .
```

However, this forces the user to type `epsilon` explicitly in the source code. A better option is to rewrite the grammar to avoid ϵ production rules before we convert it to Maude. For context-free grammars $\mathcal{G}$ that don't generate the empty string (that is, $\epsilon \notin L(\mathcal{G})$), this step is always possible. A result from formal language theory states that any such grammar $\mathcal{G}$ can be converted into an equivalent grammar $\mathcal{G}'$ where all the production rules are of the form $A ::= BC$ or $A ::= a$, with $A, B, C \in N$ and $a \in \Sigma$. The grammar $\mathcal{G}'$ is said to be in *Chomsky normal form* [HMU06].

To avoid ϵ rules, we can duplicate each production rule in which a nonterminal that can expand to ϵ occurs, and then simply omit the ϵ rule. For example, if we omit the ϵ rule for $\langle \textit{guar-clause-opt} \rangle$, and rename the symbol $\langle \textit{guar-clause} \rangle$, we obtain

$$
\begin{array}{lll}
\langle \textit{class} \rangle & ::= & \textbf{class}\ \langle \textit{id-with-params} \rangle\ \langle \textit{class-head-clauses-opt} \rangle \\
& & \textbf{begin}\ \langle \textit{var-clause-opt} \rangle\ \langle \textit{class-mtds-opt} \rangle \\
& & \langle \textit{asum-clause-opt} \rangle\ \langle \textit{guar-clause} \rangle\ \textbf{end} \\
\langle \textit{class} \rangle & ::= & \textbf{class}\ \langle \textit{id-with-params} \rangle\ \langle \textit{class-head-clauses-opt} \rangle \\
& & \textbf{begin}\ \langle \textit{var-clause-opt} \rangle\ \langle \textit{class-mtds-opt} \rangle \\
& & \langle \textit{asum-clause-opt} \rangle\ \textbf{end} \\
\langle \textit{guar-clause} \rangle & ::= & \textbf{guar}\ \langle \textit{assn} \rangle
\end{array}
$$

Next to the original ⟨*class*⟩ rule, we introduced a rule in which ⟨*guar-clause*⟩ is omitted. This process can be repeated to eliminate the ϵ rules for ⟨*class-head-clauses-opt*⟩, ⟨*var-clause-opt*⟩, ⟨*class-mtds-opt*⟩, and ⟨*asum-clause-opt*⟩, each time doubling the number of ⟨*class*⟩ rules. Through this process, the original rule for ⟨*class*⟩ gives rise to 31 new rules.

Fortunately, we can restructure the ⟨*class*⟩ rule to avoid this combinatorial explosion. By introducing a ⟨*tail*⟩ nonterminal, we can factor out the complexity associated with ⟨*asum-clause-opt*⟩ and ⟨*guar-clause-opt*⟩:

⟨*class*⟩ ::= **class** ⟨*id-with-params*⟩ ⟨*class-head-clauses-opt*⟩
begin ⟨*var-clause-opt*⟩ ⟨*class-mtds-opt*⟩ ⟨*tail*⟩
⟨*tail*⟩ ::= **asum** ⟨*assn*⟩ **guar** ⟨*assn*⟩ **end**
⟨*tail*⟩ ::= **asum** ⟨*assn*⟩ **end**
⟨*tail*⟩ ::= **guar** ⟨*assn*⟩ **end**
⟨*tail*⟩ ::= **end**

This is easy to represent using Maude operators:

```
op class__begin___ :
   Id ClassHeadClauses VarClause ClassMtds Tail -> Class [ctor] .

op asum_guar_end : Assn Assn -> Tail [ctor] .
op asum_end : Assn -> Tail .
op guar_end : Assn -> Tail .
op end : -> Tail .
```

We can go further and define equations to normalize the tail of a class declaration so that it always uses the operator `asum_guar_end`:

```
eq asum ASUM end = asum ASUM guar true end .
eq guar GUAR end = asum true guar GUAR end .
eq end = asum true guar true end .
```

When converting a grammar to Maude, most production rules give rise to an operator declaration. This sometimes leads to conflicts, as we will see with the following grammar fragment:

⟨*id-with-params*⟩ ::= ⟨*id*⟩ (⟨*typed-id-list*⟩)
⟨*id-with-args*⟩ ::= ⟨*id*⟩ (⟨*exp-list*⟩)

In Maude, we would represent the preceding rules as follows:

```
op _[_] : Id TypedIdList -> IdWithParams [ctor] .

op _[_] : Id ExpList -> IdWithArgs [ctor] .
```

Because the term `'x['y]` can be parsed both as an `IdWithParams` term and as an `IdWithArgs` term, Maude reports a parsing ambiguity—even in a context where only one interpretation is possible. The solution here is to make `IdWithParams` a subsort of `IdWithArgs`. In general, Maude parsing ambiguities assume various guises and must be handled on a case-by-case basis.

Another general problem is that some of the operators required to represent Creol programs clash with built-in operators. For example, Maude's `META-LEVEL` module

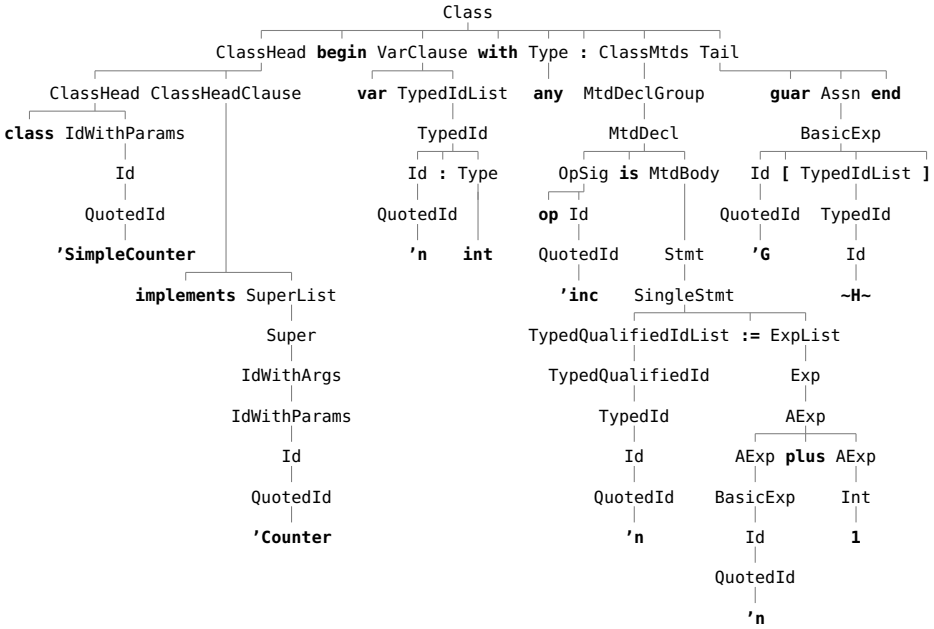
defines a `_,_` operator that collides with Creol's `_,_` operator. Many of these clashes are avoided by defining a `QuotedId` sort that hooks directly into Maude, distinct from the built-in `Qid` sort:

```
sort QuotedId .

op <QIds> : -> QuotedId [special (id-hook QuotedIdentifierSymbol)] .
```

(Thanks to the `special` attribute, the `<QIds>` operator stands for any quoted identifier constant, as explained in section 6.3.2 of the Maude 2.3 manual [CDEL+07].) Using `QuotedId`, we can ensure that the sorts used to represent Creol programs are not part of the same connected component in the subsort graph as the sorts used by `META-LEVEL` to metarepresent Maude terms, a sufficient condition for operator overloading in Maude.

Other clashes are resolved by renaming built-in operators when importing modules defined by Maude. (Maude provides a syntax for achieving this.) Finally, for common built-in operators, we can simply rename the Creol constructs. Thus, `_plus_` and `if_th_el_fi` are used for Creol instead of `_+_` and `if_then_else_fi`, which keep their standard meaning.

At the beginning of the section, we introduced the concrete Maude-compatible syntax with the declaration of a `'Counter` interface and a `'SimpleCounter` class. Here is a parse tree for the class declaration:



A significant drawback of the concrete syntax is that its structure owes more to Maude's limited parsing capabilities than to the actual structure of the program. To make Creol programs more malleable, `creol-program.maude` uses equations to convert them to system configuration terms. For example:

```
< 'Counter : Interface | Inh: epsilon, Param: epsilon,
                         Asum: true, Guar: true >
< 'SimpleCounter : Class | Impl: 'Counter, Ctrc: epsilon,
                           Inh: epsilon, Param: epsilon,
```

```
                    Att: 'n : int,
                    Mtd: < 'inc : Method | In: epsilon,
                                  Out: epsilon,
                                  LVar: epsilon,
                                  Code: 'n := 'n plus 1 >,
                    ObjCnt: 0, Asum: true, Guar: 'G[~H~] >
```

This conversion process affects the structure of interface and class declarations. The statements in the method bodies and the assertions in the assume–guarantee specification, whose Maude representation is not so convoluted, are left unchanged.

6.4 Representing Statements and Assertions

Statements and assertions play a crucial part in the assertion analyzer. In this section, we will study how they are represented in Maude. We will also review the more fundamental sorts used to represent identifiers, types, and expressions.

```
sort Id .
subsort QuotedId < Id .

op none : -> Id [ctor] .
op nu : -> Id [ctor] .
op self : -> Id [ctor] .
op caller : -> Id [ctor] .
op label : -> Id [ctor] .
op ~H~ : -> Id [ctor] .
op _$_ : Id Int -> Id [ctor prec 1] .
```

The `Id` sort represents identifiers in Creol programs, such as variable names and class names. Any quoted identifier can be used as a Creol identifier; in addition, the constants `none`, `nu` (ν), `self`, `caller`, `label`, and `~H~` ($\mathcal{H}$) are special identifiers used in Creol programs or internally in the Creol tools. The `_$_` constructor is used by the assertion analyzer to construct logical variables, such as `'h $ 1` (h_1).

```
sort Type .
subsort Id < Type .

op bool : -> Type [ctor] .
op int : -> Type [ctor] .
op any : -> Type [ctor] .
op event : -> Type [ctor] .
op history : -> Type [ctor] .
```

The `Type` sort specifies the Creol types `bool`, `int`, and `any` as well as the reasoning types `event` and `history`. In addition, any identifier can be used as a type.

```
sort TypedId .
subsort Id < TypedId .

op _:_ : Id Type -> TypedId [ctor prec 3 right id: none] .
```

The `TypedId` sort represents identifiers accompanied by a typing annotation. The constructor reflects the syntax of Creol variable declarations, allowing terms of the

form `x : t`, with `x` ∈ `Id` and `t` ∈ `Type`. The subsort declaration makes it possible to specify a plain identifier where a typed identifier is expected. Using the `right id` attribute, we specify the identity axiom `x = x : none`; this ensures that we can match any `TypedId` term with the pattern `X : T`, where `X` is a variable of sort `Id` and `T` is a variable of sort `Type`.

```
sort QualifiedId .
subsort Id < QualifiedId .

op _@_ : Id Id -> QualifiedId [ctor prec 1 right id: none] .
```

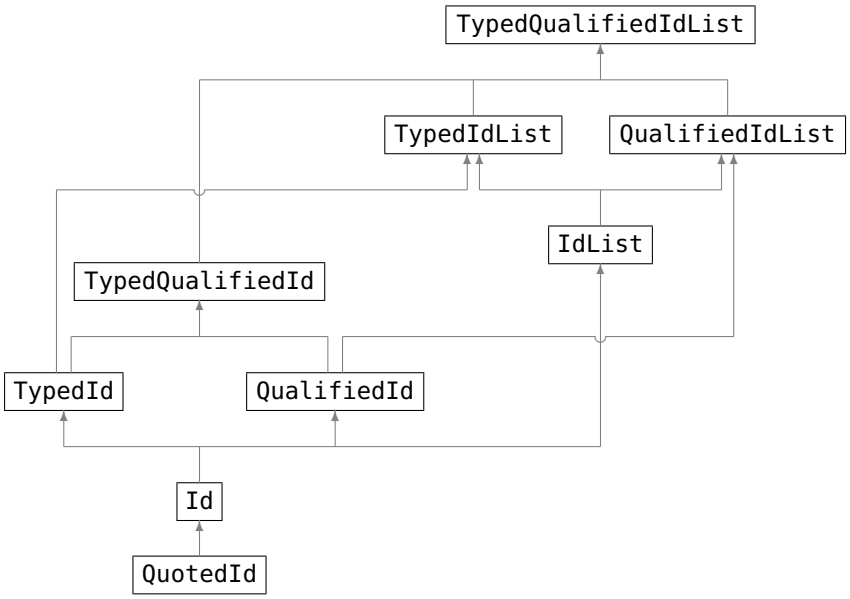
The `QualifiedId` sort represents identifiers of the form `x @ c`, with `x`, `c` ∈ `Id`. The subsort declaration ensures that plain identifiers are allowed as well. Using the `right id` attribute, we specify the identity axiom `x = x @ none`, so that we can match any `QualifiedId` term with the pattern `X @ C`, where `X` and `C` are of sort `Id`.

```
sort TypedQualifiedId .
subsort QualifiedId < TypedQualifiedId .
subsort TypedId < TypedQualifiedId .

op _:_ : QualifiedId Type -> TypedQualifiedId [ctor ditto] .
```

The `TypedQualifiedId` combines `TypedId` and `QualifiedId`, allowing terms of the form `x @ c : t`, where both `@ c` and `: t` can be omitted. The `ditto` attribute is a shorthand for the attributes that were specified in the earlier declaration of `_:_`, excluding `ctor`. We could also have written

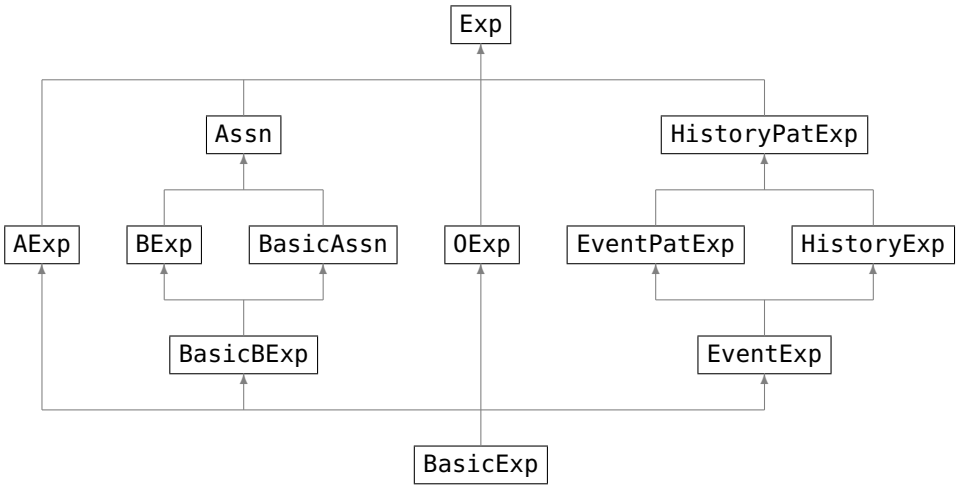
```
op _:_ : QualifiedId Type -> TypedId [ctor prec 3 right id: none] .
```

In addition, `creol-program.maude` declares the following list sorts: `IdList`, `TypedIdList`, `QualifiedIdList`, and `TypedQualifiedIdList`. The subsort graph below illustrates how the identifier sorts relate to each other and to the list sorts:

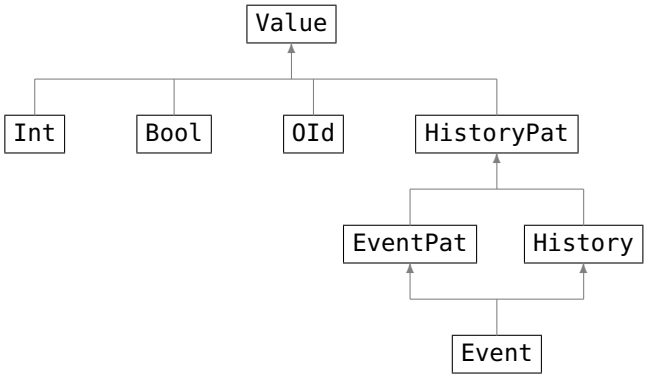


The expression sorts comprise arithmetic expressions (`AExp`), Boolean expressions (`BExp`), object expressions (`OExp`), assertions (`Assn`), event expressions (`EventExp`),

history expressions (`HistoryExp`), event pattern expressions (`EventPatExp`), and history pattern expressions (`HistoryPatExp`). The event- and history-related expressions may appear as subexpressions in assertions. The subsort graph below illustrates how the expression sorts relate to each other:



In parallel with the expression sorts, `creol-program.maude` declares a hierarchy of value sorts:



The `BasicExp` sort specifies the subsorts and operators that are common to all expression sorts. This is necessary to respect Maude's preregularity requirements on overloaded operators. The sort is declared as follows:

```
sort BasicExp .
subsort QualifiedId < BasicExp .
subsort IdWithArgs < BasicExp .

op if_th_el_fi : BExp BasicExp BasicExp -> BasicExp [ctor] .
op [_] : BasicExp -> BasicExp [ctor] .
```

Thus, a `BasicExp` can be a qualified identifier x `@` c, a function application f`[`$\bar{e}$`]`, a condition expression `if` B `th` e_1 `el` e_2 `fi`, or a parenthesized expression `[`e`]`.

The specific expression sorts provide additional constructors reflecting the abstract syntax of Creol. For example, arithmetic expressions are declared as follows:

```
sort AExp .
subsort BasicExp < AExp .
```

```
subsort Int < AExp .
subsort IntTypedId < AExp .

op plus_ : AExp -> AExp [ctor prec 3] .
op minus_ : AExp -> AExp [ctor prec 3] .
op _times_ : AExp AExp -> AExp
                              [ctor assoc comm prec 5 gather (E e)] .
op _div_ : AExp AExp -> AExp [ctor prec 5 gather (E e)] .
op _plus_ : AExp AExp -> AExp
                              [ctor assoc comm prec 7 gather (E e)] .
op _minus_ : AExp AExp -> AExp [ctor prec 7 gather (E e)] .
op if_th_el_fi : BExp AExp AExp -> AExp [ctor] .
op [_] : AExp -> AExp [ctor] .
```

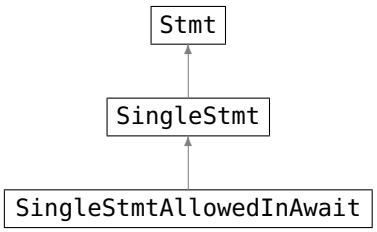
The `AExp` sort is declared with three subsorts: `BasicExp`, `Int`, and `IntTypedId`. The `Int` subsort enables us to use integer constants as arithmetic expressions, while `IntTypedId` provides typed qualified identifiers of the form *x* `@` *c* `:` `int`. The `IntTypedId` sort is declared using a membership axiom as follows:

```
sort IntTypedId .
subsort IntTypedId < TypedId .
subsort IntTypedId < TypedQualifiedId .

mb (Z : int) : IntTypedId .
```

The other expression sorts are declared in a similar fashion. In particular, the assertion sort `Assn` is declared as a supersort of `BExp` as follows:

```
sort Quantifier .

op forall : -> Quantifier [ctor] .
op exists : -> Quantifier [ctor] .

sort Assn .
subsort BasicAssn < Assn .
subsort BExp < Assn .

op !_ : Assn -> Assn [ctor ditto] .
op _&&_ : Assn Assn -> Assn [ctor ditto] .
op _||_ : Assn Assn -> Assn [ctor ditto] .
op _==>_ : Assn Assn -> Assn [ctor prec 17 gather (e E)] .
op _<==>_ : Assn Assn -> Assn [ctor comm prec 19 gather (e e)] .
op __._ : Quantifier TypedIdList Assn -> Assn [ctor prec 21] .
op if_th_el_fi : Assn Assn Assn -> Assn [ctor] .
op [_] : Assn -> Assn [ctor] .
```

Let us now look at Creol statements. Statements are represented by three sorts, as shown in the subsort graph below:



The `SingleStmtAllowedInAwait` sort is used for the statements that may appear as the last guard in an `await` statement, namely

L `?[`$\bar{z}$`],` *O* `.` *m*`[`$\bar{e}$ `;` $\bar{z}$`],` *m* `@` *c*`[`$\bar{e}$ `;` $\bar{z}$`].`

The `SingleStmt` sort adds all the other statements except sequential composition (S_1 `;` S_2), which is declared in `Stmt` only.

The concrete syntax of the canonical Creol statements is defined by the following operators:

```
op skip : -> SingleStmt [ctor] .
op abort : -> SingleStmt [ctor] .
op prove_ : Assn -> SingleStmt [ctor prec 23] .
op _:=_ : TypedQualifiedIdList ExpList -> SingleStmt
                                                    [ctor prec 23] .
op _:= new_ : TypedQualifiedId IdWithArgs -> SingleStmt
                                                    [ctor prec 23] .
op _!_._[_] : TypedId OExp Id ExpList -> SingleStmt [ctor prec 5] .
op _!_[_] : TypedId TypedQualifiedId ExpList -> SingleStmt
                                                     [ctor prec 5] .
op _?[_] : TypedId TypedQualifiedIdList -> SingleStmtAllowedInAwait
                                                     [ctor prec 5] .
op await_ : Guard -> SingleStmt [ctor prec 19] .
op if_th_el_fi : BExp Stmt Stmt -> SingleStmt [ctor] .
op while_do_od : BExp Stmt -> SingleStmt [ctor] .
op inv_while_do_od : Assn BExp Stmt -> SingleStmt [ctor] .
op _[]_ : Stmt Stmt -> SingleStmt
                      [ctor assoc comm prec 27 format (d s d s d)] .
op _|||_ : Stmt Stmt -> SingleStmt [ctor assoc comm prec 29] .
op [_] : Stmt -> SingleStmt [ctor] .

op emptyStmt : -> Stmt [ctor] .
op _;_ : Stmt Stmt -> Stmt [ctor assoc prec 25 id: emptyStmt] .
```

The n-ary operator S_1 `|||` $\cdots$ `|||` S_n is implemented using an associative and commutative binary operator `|||`. Because we use square brackets for Creol parentheses, we can distinguish between `[`S_1 `|||` S_2`]` `|||` S_3 and S_1 `|||` S_2 `|||` S_3, even though the binary operator `|||` is associative in Maude.

The empty statement `emptyStmt`, which is distinct from the null statement `skip`, is the identity element of the sequential composition operator (`;`). It should not appear in actual programs. The pattern `SS ; S`, where `SS` has sort `SingleStmt` and `S` has sort `Stmt`, can be used to match any non-empty statement.

In addition to Creol's canonical statements, `creol-program.maude` supports various synthetic statements, which can be seen as abbreviations:

```
op _.__ : OExp Id SyncCallArgs -> SingleStmtAllowedInAwait
                                                     [ctor prec 1] .
op __ : QualifiedId SyncCallArgs -> SingleStmtAllowedInAwait
op await_&&&_ : Guard SingleStmtAllowedInAwait -> SingleStmt
                                                    [ctor prec 19] .
op await_ : SingleStmtAllowedInAwait -> SingleStmt [ctor prec 19] .
op !_._[_] : OExp Id ExpList -> SingleStmt [ctor] .
```

```
op !_[_] : QualifiedId ExpList -> SingleStmt [ctor] .
op if_th_fi : BExp Stmt -> SingleStmt [ctor] .
```

The `SyncCallArgs` sort used in the signature of the synchronous call operators `_.__` and `__` is declared as follows:

```
sort SyncCallArgs .

op [_;_] : ExpList TypedQualifiedIdList -> SyncCallArgs [ctor] .
op [_;] : ExpList -> SyncCallArgs .
op [;_] : TypedQualifiedIdList -> SyncCallArgs .
op [;] : -> SyncCallArgs .
op [] : -> SyncCallArgs .
```

A synchronous call takes a list of input arguments and a list of output arguments, both of which are optional. Equations introduce dummy `epsilon` tokens so that we can match synchronous method call arguments with the pattern `[EL ; ZZL]`, where `EL` has sort `ExpList` and `ZZL` has sort `TypedQualifiedIdList`:

```
eq [EL ;] = [EL ; epsilon] .
eq [; ZZL] = [epsilon ; ZZL] .
eq [;] = [epsilon ; epsilon] .
eq [] = [epsilon ; epsilon] .
```

The Creol interpreters expands the synthetic statements into the canonical statements they abbreviate using equations. In contrast, the assertion analyzer treats them as first-class citizens, because for some of them it can produce "optimized" WLPs, as we will see in Section 6.6.

To handle empty argument lists gracefully, `creol-program.maude` specifies the following operators and equations:

```
op _!_._[] : TypedId OExp Id -> SingleStmt [prec 5] .
op _!_[] : TypedId QualifiedId -> SingleStmt [prec 5] .
op !_._[] : OExp Id -> SingleStmt .
op !_[] : QualifiedId -> SingleStmt .
op _?[] : TypedId -> SingleStmtAllowedInAwait [prec 5] .

eq LL ! OEXP . M[] = LL ! OEXP . M[epsilon] .
eq LL ! M @ C[] = LL ! M @ C[epsilon] .
eq ! OEXP . M[] = ! OEXP . M[epsilon] .
eq ! M @ C[] = ! M @ C[epsilon] .
eq LL ?[] = LL ?[epsilon] .
```

6.5 Producing the Verification Report

At the heart of the assertion analyzer lies the following rewrite rule, which in one step transforms the input program and its `verify class` command into a comprehensive verification report:

```
crl [start-verification] :
{
   verify class C with simplifications QID
   CONFIG
```

```
}
 =>
{
   Verification of class C
   classInitializationJudgment(C, AG, AAL, CONFIG, MOD)
   classMethodJudgments(C, AG, AAL, CONFIG, MOD)
}
if AG := classAGSpec(C, CONFIG)
   /\ AAL := classWritableAttributes(C, CONFIG)
   /\ MOD := upModule(QID, false) .
```

The `classInitializationJudgment` and `classMethodJudgments` functions expand to the judgments for the class to verify. We will review them shortly. The `classAGSpec` function extracts the assume–guarantee specification for the class, taking into account superclasses and superinterfaces, and the `classWritableAttributes` function returns the list of writable attributes for the class.

The verification report is a term of sort `Report`. The report is made more readable using Maude's `format` attribute, which lets us color tokens and insert whitespace between them. Here is the declaration of the `Report` constructor:

```
op Verification of class__ : Id JudgmentList -> Report
                      [ctor gather (e &) format (b b osb b bnn o)] .
```

A report consists of a banner followed by a list of judgments:

```
sort JudgmentList .
subsort Judgment < JudgmentList .

op Nothing to verify : -> JudgmentList [ctor format (b b b on)] .
op __ : JudgmentList JudgmentList -> JudgmentList
         [ctor assoc prec 9 id: Nothing to verify format (b bn o)] .
```

A judgment consists of a head and a body, separated by a colon:

```
sort Judgment .

op _:_ : JudgmentHead judgmentBody -> Judgment
                                [ctor prec 7 format (b b bnssss on)] .
```

The head of a judgment identifies the code that has been verified:

```
sort JudgmentHead .

op Initialization code : -> JudgmentHead [ctor format (b b o)] .
op Method_of_ : Id Id -> JudgmentHead
                                     [ctor prec 3 format (b b b b o)] .
```

The body of a judgment states the result of the verification:

```
sort JudgmentBody .

op Maintains the guarantee : -> JudgmentBody
                                               [ctor format (g g g o)] .
op Maintains the guarantee iff_holds :
   PrettyAssn -> JudgmentBody
                            [ctor format (y y y y nssssy nssssy o)] .
     ⋮
```

```
op Fails to establish the guarantee : -> JudgmentBody
                                       [ctor format (r r r r r o)] .
```

To increase the report's readability, the judgment bodies are colored according to a traffic light scheme: green for positive judgments, yellow for inconclusive judgments, and red for negative judgments. (In Section 6.1, the glyphs ☑, ❓, and ❑ carried this information.)

The judgment for the class's initialization code is generated by the `classInitializationJudgment` function and has the general form

```
Initialization code : Establishes the invariant iff Q̂ holds
```

where $\hat{Q}$ is a proof obligation that is derived from the verification condition given in Section 5.5 through the massaging process described in Section 6.7. Recall that the verification condition for a class's initialization code is

$$\big(\mathcal{A}_c \Rightarrow wlp(\langle \text{initializer} \rangle\,,\, (\mathcal{G}_c)^{\mathcal{H}}_{\mathcal{H}^\frown[\textbf{self.initialized}]})\big)^{\mathcal{H}}_{[parent(\textbf{self})\to\textbf{self.new}\ c(\bar{p})]},$$

where $\bar{p}$ is a list of fresh logical variables.

The `classInitializationJudgment` function is defined as follows:

```
ceq classInitializationJudgment(C, < ASUM, GUAR >, AAL, CONFIG,
                                MOD) =
    Initialization code :
        judgmentBody(S,
            (ASUM ==> wlp(S, Q, < ASUM, GUAR >, AAL))
                { ~H~ : history |->
                  ['parent[self : any] -> self : any . new C[PPL]] },
            MOD)
if Q := GUAR { ~H~ : history |-> ~H~ : history
                                 ^^ [self : any . initialized] }
   /\ PPL := freshLogicalVarList(classParams(C, CONFIG), Q)
   /\ S := (initializeVars(AAL) ; initialPr(C[PPL], CONFIG)) .
```

This definition relies on many auxiliary functions and operators, which we will review in turn.

The `wlp` function implements the *wlp* function presented in Section 5.4. In addition to a statement `S` and a postcondition `Q`, it takes an assume–guarantee specification `< ASUM, GUAR >` and the list of attributes `AAL` as arguments. We will study the implementation of `wlp` in the next section.

The substitution operator `(X) { z̄ |-> ē }` simultaneously replaces all free occurrences of the variables $\bar{z}$ with the corresponding expressions from $\bar{e}$ in X, which can be a statement or an expression. (If Maude had supported superscripts and subscripts, we could have written $X^{\bar{z}}_{\bar{e}}$ instead.) The substitution operator ensures that free variables in $\bar{e}$ don't get captured by a quantifier, by renaming bound variables in X if necessary. For example:

```
red (forall 'x $ 1 . 'y) { 'y |-> 'x $ 1 } .
*** result Assn: forall 'x $ 2 . 'x $ 1
```

The ^^ operator denotes history concatenation ($\frown$).

The `freshLogicalVarList` function generates fresh names for logical variables. It takes a list of variables $\bar{z}$ and a list of expressions $\bar{e}$ and returns a list of logical variables $\bar{x}$ that have the same types as the corresponding variables in $\bar{z}$ and that do not occur free in $\bar{e}$. For example:

```
red freshLogicalVarList(('x : int, 'y : bool), ('x $ 1, 'b)) .
*** result TypedIdList: 'x $ 2 : int, 'y $ 1 : bool
```

The `classParams` function returns the context parameters for the specified class.

The `initializeVars` function returns a list of assignment statements that initialize the specified variables to their default values. For example,

```
red initializeVars('x : int, 'y : bool, 'z : 'Counter) .
*** result Stmt: 'x : int := 0 ; 'y : bool := false ;
***              'z : 'Counter := null
```

Notice that the variables in the assignment statements carry typing annotations. The assertion analyzer works on code that is fully typed and fully qualified. Typing helps the assertion simplification process, while qualifying is necessary to avoid aliasing issues and potential name clashes, as noted in Section 5.1. This also explains why we wrote `~H~ : history` and `self : any` rather than just `~H~` and `self` in the `classInitializationJudgment` equation.

The `initialPr` function returns the fully typed and fully qualified code for the initial process, which consists of the initialization of the context parameters, the calls to `'init`, and the call to `'run`.

Finally, `judgmentBody` takes the verification condition, normalizes it, simplifies it (using the simplification module `MOD`), untypes it, pretty-prints it, and embeds it in a judgment body. The argument `S` influences the wording of the judgment body. If `S` contains a **while** loop or some other statement that weakens the proof system, the resulting judgment body says `if` instead of `iff`:

```
ceq judgmentBody(S, Q, MOD) =
    if wlpIsComplete(S) then
        Maintains the guarantee iff PRETTY holds
    else
        Maintains the guarantee if PRETTY holds
    fi
if PRETTY := massaged(Q, MOD) .
```

Equations are used to adapt the wording of the judgment bodies for the initialization code:

```
eq Initialization code : Maintains the guarantee =
   Initialization code : Establishes the guarantee .
     ⋮
eq Initialization code : Breaks the guarantee =
   Initialization code : Fails to establish the guarantee .
```

Equations are also used to reword judgment bodies that contain the trivial proof obligations `true` and `false`:

```
eq Maintains the guarantee iff true holds =
   Maintains the guarantee .
eq Maintains the guarantee iff false holds =
   Breaks the guarantee .
     ⋮
eq Establishes the guarantee if true holds =
   Establishes the guarantee .
eq Establishes the guarantee if false holds =
   Don't know .
```

We have seen how the assertion analyzer generates the judgment associated with a class's initialization code. The judgments relative to the methods declared in a class are generated in much the same way. The `classMethodJudgments` function, which orchestrates this, simply calls `methodJudgment` for every method provided by the class, producing judgments of the form

`Method` *m* `of` *c* `: Maintains the invariant iff` $\hat{Q}$ `holds`

The verification condition for a method is

$$
\begin{array}{l}
(\mathcal{A}_c \\
\quad \wedge\ \mathcal{G}_c \\
\quad \wedge\ \mathit{lwf}(\mathcal{H}, \mathbf{self}) \\
\quad \wedge\ \mathit{mayAcquireProcessor}(\mathcal{H}, \mathbf{self}, \mathbf{caller}, \mathbf{label}) \\
\quad \wedge\ [\mathbf{caller} \rightarrow \mathbf{self}.m@c'(\bar{x})]^{\mathbf{label}}\ \mathbf{in}\ \mathcal{H} \\
\quad \wedge\ [\mathbf{caller} \leftarrow \mathbf{self}]^{\mathbf{label}}\ \mathbf{not\ in}\ \mathcal{H}) \Rightarrow \\
\mathit{wlp}\big(\langle \mathrm{method} \rangle\,,\ (\mathcal{G}_c)^{\mathcal{H}}_{\mathcal{H} \frown [\mathbf{caller} \leftarrow \mathbf{self}.m@c'(\bar{x};\bar{y})]^{\mathbf{label}}}\big).
\end{array}
$$

Putting this together, we obtain the following definition for `methodJudgment`:

```
ceq methodJudgment(C, < ASUM, GUAR >, AAL, M, XXL, YYL, VVL, S,
                   CONFIG, MOD) =
    Method M of C :
        judgmentBody(S',
            [ASUM
             && GUAR
             && 'lwf[~H~ : history, self : any]
             && 'mayAcquireProcessor[~H~ : history, self : any,
                                     caller : any, label : int]
             && [label : int % caller : any -> self : any . M[XXL]]
                in ~H~ : history
             && ! [[label : int % caller : any <- self : any . *]
                   in ~H~ : history]] ==>
            wlp(S', (GUAR) { ~H~ : history |-> ~H~ : history
                                   ^^ [label : int % caller : any <-
                                       self : any . M[XXL ; YYL]] },
                < ASUM, GUAR >, AAL),
            MOD)
if S' := (initializeVars(YYL, VVL) ;
          qualifiedAndTyped(S, C, AAL, (XXL, YYL, VVL), CONFIG)) .
```

The most noteworthy feature of this definition is the `qualifiedAndTyped` call, which systematically types and qualifies the variables that appear in the method body `S`. The third argument lists the attributes, whereas the fourth argument lists the local variables (including the input and output parameters).

6.6 Computing the Weakest Liberal Preconditions

The verification conditions computed by the assertion analyzer rely on the `wlp` function. For most of the Creol statements, the `wlp` function implemented in Maude is almost the same as the *wlp* function from Definitions Q10–Q12 (Section 5.4), the difference being that the assume–guarantee specification `AG` and the list of attributes `AAL` are now explicit parameters of `wlp`. For example, here is the definition of `wlp` for some of the basic Creol statements:

```
eq wlp(skip, Q, AG, AAL) = Q .
eq wlp(abort, Q, AG, AAL) = true .
eq wlp(prove P, Q, AG, AAL) = P && Q .
eq wlp(ZZL := EL, Q, AG, AAL) = (Q) { ZZL |-> EL } .
eq wlp([S], Q, AG, AAL) = wlp(S, Q, AG, AAL) .
eq wlp(if B th S1 el S2 fi, Q, AG, AAL) =
   if B th wlp(S1, Q, AG, AAL) el wlp(S2, Q, AG, AAL) fi .
ceq wlp(S1 ; S2, Q, AG, AAL) = wlp(S1, wlp(S2, Q, AG, AAL), AG, AAL)
if S1 =/= emptyStmt and S2 =/= emptyStmt .
```

For the other Creol statements, the transition from the abstract WLP given by the *wlp* function to the concrete `wlp` is less straightforward. This is largely due to a difference in philosophy between the proof system and the assertion analyzer: Where the proof system tends to be minimalistic and reductionist, the assertion analyzer introduces some complications to obtain syntactically simpler (yet logically equivalent) preconditions. And pragmatism forces us to revisit the **while** loop, whose abstract WLP is unusable in practice.

In the rest of this section, we will look more specifically at the following statements:

$z := \mathbf{new}\ c(\bar{e})$, $\quad |||_{i=1}^{n} S_i$,
await g, $\quad S_1 \,\square\, S_2$,
await $g \,\&\, l?(\bar{z})$, $\quad \big[\mathbf{inv}\ I\big]\ \mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}$.
$m@c(\bar{e};\bar{z})$,

We start with object creation. In Chapter 5, we gave the following WLP for the statement $z := \mathbf{new}\ c(\bar{e})$:

$$\forall o, h.\ \big(\mathit{interleave}(\mathcal{H}, h) \wedge o \notin \mathit{objectIds}(h) \wedge \mathit{parent}(o) = \mathbf{self}\big) \Rightarrow Q^{z,\mathcal{H}}_{o,h^\frown[\mathbf{self}\to o.\mathbf{new}\ c(\bar{e})]}\,.$$

In contrast, the assertion analyzer uses the equation

```
ceq wlp(ZZ := new C[EL], Q, AG, AAL) =
    interleaved(
        forall OO .
            ['isFreshObjectId[OO, ~H~ : history]
             && 'parent[OO] eq self : any] ==>
            (Q) { ZZ |-> OO }
                { ~H~ : history |-> ~H~ : history
                                ^^ [self : any -> OO . new C[EL]] },
        AG)
if OO := freshLogicalVar('o : any, Q) .
```

where `interleaved` is defined as follows:

```
ceq interleaved(Q, AG) =
    forall HH .
        interleave(~H~ : history, HH, AG) ==>
        (Q) { ~H~ : history |-> HH }
if HH := freshLogicalVar('h : history, Q) .
```

The `interleaved` function transforms `Q` into `forall HH . [interleave(~H~, HH, AG) ==> Q`$^{\sim H\sim}_{HH}$`]`. It occurs in the `wlp` of most Creol-specific statements. Its purpose is to make the source code of the assertion analyzer simpler to read. Besides this, the main difference between the abstract WLP and the concrete `wlp` is the use of a function `'isFreshObjectId[OO, ~H~]` instead of `! [OO in 'objectIds[~H~]]`, to avoid overusing operators.

The `interleave` function (with no `d`) is defined as follows:

```
eq interleave(HEXP, HEXP', < ASUM, GUAR >) =
   'lwf[HEXP', self : any]
   && HEXP pr HEXP'
   && 'agreeOnOutAndCtl[HEXP, HEXP', self : any]
   && ASUM { ~H~ : history |-> HEXP' } .
```

Compared with Definition Q1, the only difference is that we write `'agreeOnOut-AndCtl[HEXP, HEXP', self : any]` instead of `HEXP / (out[self : any] | ctl[self : any]) eq HEXP' / (out[self : any] | ctl[self : any])`, to make the resulting assertion more manageable.

The next statement we consider is conditional wait. In Section 5.2, we distinguished three cases:

i. **await** $B_1 \,\&\, \cdots \,\&\, B_n$

ii. **await** $l_1? \,\&\, \cdots \,\&\, l_p? \left[\&\, B_1 \,\&\, \cdots \,\&\, B_n\right]$

iii. **await wait** $\left[\&\, l_1? \,\&\, \cdots \,\&\, l_p?\right] \left[\&\, B_1 \,\&\, \cdots \,\&\, B_n\right]$

The assertion analyzer handles all three cases in one equation:

```
ceq wlp(await G, Q, < ASUM, GUAR >, AAL) =
    if cleared(G) == G then
        if P :: BExp then
            *** case i
            if P th
                Q
            el
                GUAR { ~H~ : history |->
                       ~H~ : history ^^ [self : any . release] }
                && [forall AAL', HH .
                        release(~H~ : history, HH, AAL',
                                < ASUM, GUAR >, AAL) ==>
                        (P ==> Q) { AAL |-> AAL' }
                                  { ~H~ : history |-> HH }]
            fi
        else
            *** case ii
            forall HH .
                [interleave(~H~ : history, HH, < ASUM, GUAR >) ==>
```

```
                if (P) { ~H~ : history |-> HH } th
                    (Q) { ~H~ : history |-> HH }
                el
                    GUAR { ~H~ : history |->
                           HH ^^ [self : any . release] }
                    && [forall AAL', HH' .
                            release(HH, HH', AAL', < ASUM, GUAR >,
                                    AAL) ==>
                            (P ==> Q) { AAL |-> AAL' }
                                      { ~H~ : history |-> HH' }]
                fi]
            fi
        else
            *** case iii
            GUAR { ~H~ : history |->
                   ~H~ : history ^^ [self : any . release] }
            && [forall AAL', HH .
                    release(~H~ : history, HH, AAL', < ASUM, GUAR >,
                            AAL) ==>
                    (P ==> Q) { AAL |-> AAL' }
                              { ~H~ : history |-> HH }]
        fi
    if P := satisfied(cleared(G), AAL, ~H~, AAL)
       /\ HH := freshLogicalVar('h : history, Q)
       /\ HH' := freshLogicalVar('h : history, (HH, Q))
       /\ AAL' := freshLogicalVarList(AAL, (HH, HH', Q)) .
```

The variable `P` is bound to an assertion corresponding to the guard `G`. It is computed by the `satisfied` function. For example:

```
red satisfied('x eq 'y, AAL, ~H~ : history, AAL) .
*** result BasicBExp: 'x eq 'y

red satisfied('l ?, AAL, ~H~ : history, AAL) .
*** result BasicAssn: ['l % self : any <- * . *] in ~H~ : history
```

In the `wlp` definition, we first check if there is a **wait** guard. If `cleared(G) == G`, there is no such guard and the choice is between cases i and ii; otherwise, we have case iii. To distinguish cases i and ii, we use the `:: BExp` operator on the assertion `P` to check whether the assertion is a Boolean expression or a general assertion.

The `release` function's implementation follows Definition Q4:

```
eq release(HEXP, HEXP', AAL', < ASUM, GUAR >, AAL) =
   'lwf[HEXP', self : any]
   && HEXP ^^ [self : any . release] pr HEXP'
   && 'mayAcquireProcessor[HEXP', self : any, caller : any,
                           label : int]
   && ! [[label : int % caller : any <- self : any . *] in HEXP']
   && [[HEXP ^^ [self : any . release]] /
       (out[self : any] | ctl[self : any]) eq
       HEXP' / (out[self : any] | ctl[self : any]) ==>
       AAL all eq AAL']
   && (ASUM && GUAR) { AAL |-> AAL' }
                     { ~H~ : history |-> HEXP' } .
```

In the proof system presented in Chapter 5, we assumed that synthetic statements were expanded to the canonical statements they abbreviated. Thus, we gave no proof rule for **if** B **then** S **fi**, which can be treated as **if** B **then** S **else skip fi**. However, for some of the synthetic statements, we can provide syntactically simpler preconditions than we would obtain by expanding them first.

Consider the statement **await** g & $l?(\bar{z})$, which abbreviates **await** g & $l?$; $l?(\bar{z})$. If $\bar{z} \equiv z_1, \ldots, z_n$, the WLP for $l?(\bar{z})$ is

$$
\begin{array}{l}
\textbf{if } \mathit{pending}(\mathcal{H}, \textbf{self}, \textbf{self}, l) \textbf{ then} \\
\quad (\mathcal{G}_c)^{\mathcal{H}}_{\mathcal{H}^\frown[\textbf{self.reenter}]^l} \\
\quad \wedge\ \forall \bar{a}, h.\ \big(\mathit{reenter}(\mathcal{H}, h, \bar{a}, l) \Rightarrow Q^{\bar{\mathcal{A}},\mathcal{H},l,z_1,\ldots,z_n}_{\bar{a},h,-1,\mathit{returnVal}_1(h,\textbf{self},l),\ldots,\mathit{returnVal}_n(h,\textbf{self},l)}\big) \\
\textbf{else} \\
\quad \forall h.\ \big(\mathit{interleave}(\mathcal{H}, h) \wedge [\textbf{self} \leftarrow *]^l \textbf{ in } h\big) \Rightarrow \\
\qquad Q^{\mathcal{H},l,z_1,\ldots,z_n}_{h,-1,\mathit{returnVal}_1(h,\textbf{self},l),\ldots,\mathit{returnVal}_n(h,\textbf{self},l)} \\
\textbf{fi}
\end{array}
$$

but immediately after **await** g & $l?$ we can assume that $[\textbf{self} \leftarrow *]^l$ **in** $\mathcal{H}$ will hold, and thus the above precondition can be replaced by

$$Q^{l,z_1,\ldots,z_n}_{-1,\mathit{returnVal}_1(h,\textbf{self},l),\ldots,\mathit{returnVal}_n(h,\textbf{self},l)},$$

which corresponds to the SEQ statement

$$l, z_1, \ldots, z_n := -1, \mathit{returnVal}_1(h, \textbf{self}, l), \ldots, \mathit{returnVal}_n(h, \textbf{self}, l)$$

Based on this, we can define an optimized WLP for **await** g & $l?(\bar{z})$ as follows:

$$
\begin{array}{l}
\mathit{wlp}(\textbf{await } g \,\&\, l?(\bar{z}),\, Q) \triangleq \mathit{wlp}\big(\textbf{await } g \,\&\, l?, \\
\qquad\qquad Q^{l,z_1,\ldots,z_n}_{-1,\mathit{returnVal}_1(h,\textbf{self},l),\ldots,\mathit{returnVal}_n(h,\textbf{self},l)}\big).
\end{array}
$$

This leads to the following concrete `wlp`:

```
eq wlp(await G &&& LL ?[ZZL], Q, AG, AAL) =
   wlp(await G &&& LL ?,
       (Q) { LL |-> -1 }
           { ZZL |-> returnVals(~H~ : history, self : any, LL,
                                length(ZZL)) },
       AG, AAL) .
```

We can perform a similar optimization for synchronous method calls. In Chapter 4, we defined $m@c(\bar{e}; \bar{z})$ as an abbreviation for $\nu!m@c(\bar{e})$; $\nu?(\bar{z})$, where ν is a special label that cannot occur in the program text. Using backward construction, we find the following WLP for $\nu!m@c(\bar{e})$; $\nu?(\bar{z})$:

$$
\begin{array}{l}
\forall k, h.\ \big(\mathit{interleave}(\mathcal{H}, h) \wedge \big[\textbf{self} \rightarrow *\big]^k \textbf{ not in } h \wedge k \geq 0\big) \Rightarrow \\
\quad \big(\textbf{if } \mathit{pending}(\mathcal{H}, \textbf{self}, \textbf{self}, k) \textbf{ then} \\
\qquad (\mathcal{G}_c)^{\mathcal{H}}_{h^\frown[\textbf{self.reenter}]^k} \\
\qquad \wedge\ \forall \bar{a}, h'.\ \mathit{reenter}(\mathcal{H}, h', \bar{\mathcal{A}}, \bar{a}, k) \Rightarrow \\
\qquad\qquad Q^{\bar{\mathcal{A}},\mathcal{H},\nu,z_1,\ldots,z_n}_{\bar{a},h',-1,\mathit{returnVal}_1(h',\textbf{self},k),\ldots,\mathit{returnVal}_n(h',\textbf{self},k)} \\
\quad \textbf{else} \\
\qquad \forall h'.\ \big(\mathit{interleave}(\mathcal{H}, h') \wedge [\textbf{self} \leftarrow *]^k \textbf{ in } h'\big) \Rightarrow \\
\qquad\qquad Q^{\mathcal{H},\nu,z_1,\ldots,z_n}_{h',-1,\mathit{returnVal}_1(h',\textbf{self},k),\ldots,\mathit{returnVal}_n(h',\textbf{self},k)} \\
\quad \textbf{fi}\big)^{\mathcal{H}}_{h^\frown[\textbf{self} \rightarrow \textbf{self}.m@c(\bar{e})]^k}.
\end{array}
$$

By the definition of *pending*, the **then** branch of the conditional expression will always be taken. (If we call $m@c$ asynchronously on **self**, we can expect the call to remain pending at least until we release the processor.) And since the guarantee should be insensitive to additional events originating from the environment and ν shouldn't occur in Q, we can use the optimized WLP

$$\begin{aligned}
&wlp(m@c(\bar{e};\bar{z}),\, Q)\\
&\quad \triangleq\ \forall k.\ \big[\mathbf{self}{\rightarrow}*\big]^k \textbf{ not in } \mathcal{H} \wedge k \geq 0 \Rightarrow\\
&\qquad \big((\mathcal{G}_c)^{\mathcal{H}}_{\mathcal{H}^\frown[\mathbf{self}.\mathbf{reenter}]^k}\\
&\qquad\ \wedge \forall \bar{a}, h.\ reenter(\mathcal{H}, h, \bar{\mathcal{A}}, \bar{a}, k) \Rightarrow\\
&\qquad\qquad Q^{\bar{\mathcal{A}},\mathcal{H},z_1,\ldots,z_n}_{\bar{a},h,returnVal_1(h,\mathbf{self},k),\ldots,returnVal_n(h,\mathbf{self},k)}\big)^{\mathcal{H}}_{\mathcal{H}^\frown[\mathbf{self}{\rightarrow}\mathbf{self}.m@c(\bar{e})]^k},
\end{aligned}$$

In Maude, this gives

```
ceq wlp(M @ C[EL ; ZZL], Q, < ASUM, GUAR >, AAL) =
    forall KK .
        'isFreshSequenceNum[KK, self : any, ~H~ : history] ==>
        (GUAR { ~H~ : history |->
                ~H~ : history ^^ [KK % self : any . reenter] }
         && [forall AAL', HH .
                 reenter(~H~ : history, HH, AAL', KK,
                         < ASUM, GUAR >, AAL) ==>
                 (Q) { AAL |-> AAL' }
                     { ~H~ : history |-> HH }
                     { ZZL |-> returnVals(HH, self : any, KK,
                                          length(ZZL)) }])
        { ~H~ : history |-> ~H~ : history
                        ^^ [KK % self : any -> self : any . M[EL]] }
  if KK := freshLogicalVar('k : int, Q)
     /\ HH := freshLogicalVar('h : history, Q)
     /\ AAL' := freshLogicalVarList(AAL, (KK, HH, Q)) .
```

The next statement we will look at is nondeterministic merge. In Chapter 5, we defined its WLP as follows:

$$wlp(|||_{i=1}^n S_i,\, Q) \triangleq wlp(\Box_{i=1}^n (S_i; |||_{j=1,j\neq i}^n S_j),\, Q) \qquad \text{if } awaitFree(|||_{i=1}^n S_i).$$

From this, we deduce

$$\begin{aligned}
&wlp(S_1 ||| S_2 ||| S_3,\, Q)\\
&\quad = wlp(S_1; (S_2 ||| S_3) \Box S_2; (S_1 ||| S_3) \Box S_3; (S_1 ||| S_2),\, Q)\\
&\quad = wlp(S_1; (S_2; S_3 \Box S_2; S_3) \Box S_2; (S_1; S_3 \Box S_3; S_1) \Box S_3; (S_1; S_2 \Box S_2; S_1),\, Q).
\end{aligned}$$

As an easy speed optimization, the assertion analyzer avoids this intermediate step and converts `S1 ||| ··· ||| Sn` directly into a `|||`-free statement. This is done by the `perms` function. For example:

```
red perms(S1 ||| S2 ||| S3) .
*** result SingleStmt: S1 ; (S2 ; S3 [] S3 ; S2)
***                    [] S2 ; (S1 ; S3 [] S3 ; S1)
***                    [] S3 ; (S1 ; S2 [] S2 ; S1)
```

Using `perms` and an auxiliary `awaitFree` predicate modeled after Definition Q9, we can define the concrete `wlp` as follows:

```
ceq wlp(S1..SK ||| SK+1..SN, Q, AG, AAL) =
    wlp(perms(S1..SK ||| SK+1..SN), Q, AG, AAL)
if awaitFree(S1..SK ||| SK+1..SN) .
```

The variable names `S1..SK` and `SK+1..SN` convey the idea that each of these may represent several branches of a general n-ary `|||` statement. The `perms` function is implemented below:

```
eq perms(S) = perms(S, S) .

ceq perms(S1, S1) = S1
if simpleBranch(S1) .
ceq perms(S1, S1 ||| S') = S1 ; perms(S')
if simpleBranch(S1) .
ceq perms(S1 ||| S', S1 ||| S2..SN) =
    S1 ; perms(S2..SN, S2..SN) [] perms(S', S1 ||| S2..SN)
if simpleBranch(S1) .
```

In general, a merge statement with n branches gives rise to no less than $n! - 1$ choice statements. To make matters worse, the WLP for the choice statement is prohibitively complex:

$$
\begin{array}{l}
\forall h.\ \mathit{interleave}(\mathcal{H}, h) \Rightarrow \\
\quad \textbf{if } \mathit{ready}(S_1 \,\square\, S_2, \bar{\mathcal{A}}, h) \textbf{ then} \\
\qquad \mathit{pickReadyBranch}(S_1, S_2, \mathit{wlp}(S_1,\, Q), \mathit{wlp}(S_2,\, Q), \bar{\mathcal{A}}, h) \\
\quad \textbf{else if } \mathit{enabled}(S_1 \,\square\, S_2, \bar{\mathcal{A}}, h) \textbf{ then} \\
\qquad \forall h'.\ \big(\mathit{interleave}(h, h') \wedge \mathit{ready}(S_1 \,\square\, S_2, \bar{\mathcal{A}}, h')\big) \Rightarrow \\
\qquad\quad \mathit{pickReadyBranch}(S_1, S_2, \mathit{wlp}(S_1,\, Q), \mathit{wlp}(S_2,\, Q), \bar{\mathcal{A}}, h') \\
\quad \textbf{else} \\
\qquad (\mathcal{G}_c)^{\mathcal{H}}_{h^\frown[\textbf{self.release}]} \\
\qquad \wedge\ \forall \bar{a}, h'.\ \big(\mathit{release}(h, h', \bar{a}) \wedge \mathit{ready}(S_1^\star \,\square\, S_2^\star, \bar{a}, h')\big) \Rightarrow \\
\qquad\quad \mathit{pickReadyBranch}(S_1^\star, S_2^\star, \mathit{wlp}(S_1^\star,\, Q), \mathit{wlp}(S_2^\star,\, Q), \bar{a}, h') \\
\quad \textbf{fi fi}
\end{array}
$$

Fortunately, in the frequent case where the statements S_1 and S_2 are always ready, the precondition simplifies to $\mathit{wlp}(S_1, Q) \wedge \mathit{wlp}(S_2, Q)$. The assertion analyzer exploits this in its `wlp` implementation:

```
ceq wlp(S1 [] S2, Q, < ASUM, GUAR >, AAL) =
    if logicallySimplified(ready(S1, AAL, ~H~ : history, AAL)
                           && ready(S2, AAL, ~H~ : history, AAL))
           == true then
        wlp(S1, Q, < ASUM, GUAR >, AAL)
        && wlp(S2, Q, < ASUM, GUAR >, AAL)
    else
        interleaved(
            if ready(S1 [] S2, AAL, ~H~ : history, AAL) th
                pickReadyBranch(S1, S2,
                                wlp(S1, Q, < ASUM, GUAR >, AAL),
                                wlp(S2, Q, < ASUM, GUAR >, AAL),
                                AAL, ~H~ : history, AAL)
            el if enabled(S1 [] S2, AAL, ~H~ : history, AAL) th
                interleaved(
                    ready(S1 [] S2, AAL, ~H~ : history, AAL) ==>
```

```
                    pickReadyBranch(S1, S2,
                                    wlp(S1, Q, < ASUM, GUAR >, AAL),
                                    wlp(S2, Q, < ASUM, GUAR >, AAL),
                                    AAL, ~H~ : history, AAL),
                    < ASUM, GUAR >)
            el
                GUAR { ~H~ : history |->
                       ~H~ : history ^^ [self : any . release] }
                && [forall AAL', HH .
                         [release(~H~ : history, HH, AAL',
                                  < ASUM, GUAR >, AAL)
                          && ready(S1* [] S2*, AAL', HH, AAL)] ==>
                         pickReadyBranch(S1*, S2*,
                                    wlp(S1*, Q, < ASUM, GUAR >, AAL),
                                    wlp(S2*, Q, < ASUM, GUAR >, AAL),
                                    AAL', HH, AAL)]
            fi fi,
            < ASUM, GUAR >)
     fi
if HH := freshLogicalVar('h : history, Q)
   /\ AAL' := freshLogicalVarList(AAL, (HH, Q))
   /\ S1* := clearWait(S1)
   /\ S2* := clearWait(S2) .
```

The `ready` function returns an assertion that specifies whether a given statement is ready. If for both `S1` and `S2` the assertion can be simplified to `true`, both statements are always ready and we use the optimized WLP.

The last statement we will look at is the **while** loop. In Section 5.4, we gave a WLP based on Dijkstra [Dij75], but unfortunately it is generally impossible to compute [IS97]. Instead, the assertion analyzer relies on the loop invariant provided by the programmer. If no loop invariant is provided, we return `false` as the precondition:

```
eq wlp(while B do S od, Q, AG, AAL) = false .
```

As a result, the assertion analyzer will not be able to complete the proof and will produce the judgment `Don't know`.

```
eq wlp(inv I while B do S od, Q, AG, AAL) =
   I {{ [I && B] ==> wlp(S, I, AG, AAL) }}
     {{ [I && ! B] ==> Q }} .
```

If a loop invariant is provided, we return it as the precondition, because a valid loop invariant must hold before entering the loop. In addition, we generate two side conditions, enclosed in double braces (`{{ }}`). The first condition ensures that each iteration maintains the invariant, whereas the second condition ensures that the invariant, together with the negated loop condition, is strong enough to imply the **while** loop's postcondition. This is comparable to Proof Rule P9 from Section 5.1:

$$\frac{\{I \wedge B\}\, S\, \{I\}}{\{I\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{I \wedge \neg B\}}$$

We enclose the side conditions in double braces instead of simply using the conjunction operator, because they are not part of the precondition and should not be altered when computing the WLP of statements that occur before the **while** loop.

Since we need to carry them along so that they become part of the verification condition, we put them in "bubbles", which are simply passed unchanged by the `wlp` function. Equations ensures that they are moved to the top level of a complex assertion:

```
eq ! (PHI {{ PHI' }}) = (! PHI) {{ PHI' }} .
eq (PHI1 {{ PHI' }}) && PHI2 = (PHI1 && PHI2) {{ PHI' }} .
     ⋮
eq forall XXL . (PHI {{ PHI' }}) =
   (forall XXL . PHI) {{ PHI' }} .
eq exists XXL . (PHI {{ PHI' }}) =
   (exists XXL . PHI) {{ PHI' }} .
```

Side conditions can be combined:

```
eq PHI {{ PHI' {{ PHI'' }} }} = PHI {{ PHI' && PHI'' }} .
eq PHI {{ PHI' }} {{ PHI'' }} = PHI {{ PHI' && PHI'' }} .
```

Importantly, substitutions do not affect side conditions:

```
eq (PHI {{ PHI' }}) RHO = ((PHI) RHO) {{ PHI' }} .
```

The advantage of this approach is that side conditions are propagated implicitly by the backward proof construction process, without any extra work. However, it does require some care: For every `wlp`, the postcondition `Q` must appear somewhere in the precondition; otherwise the side conditions attached to it will be lost. (A notable exception is the equation

```
eq wlp(while B do S od, Q, AG, AAL) = false .
```

for which there is no point in keeping the side conditions.)

6.7 Massaging the Verification Conditions

The verification conditions that are shown to the user pass through the `massaged` function, which transforms them syntactically to make them more readable. To see why this is necessary, consider the following **if** statement:

if $x < 5$ **then**
  await $x \geq 0$
fi

Intuitively, control flow can take three paths, depending on the initial value of x:

1. If $x \geq 5$, the **if** statement is bypassed.
2. If $0 \leq x < 5$, the **then** branch is taken but the **await** statement does nothing.
3. If $x < 0$, the **then** branch is taken and the **await** statement releases the processor and reacquires it in a state where $x \geq 0$.

Let us assume that the code is located in the implementation of class c, in which x is declared with type **int**. For simplicity, we will also assume that the class has no assume–guarantee specification. In these circumstances, what is the WLP of the **if** statement for the postcondition $k \leq x$? The command

```
red wlp(if 'x @ 'c : int lt 5 th await 'x @ 'c : int ge 0 fi,
        'k : int le 'x @ 'c : int, < true, true >, 'x @ 'c : int) .
```

gives the answer

```
result Assn: if 'x @ 'c : int lt 5 th if 'x @ 'c : int ge 0 th 'k :
    int le 'x @ 'c : int el true && forall 'x $ 1 : int, 'h $ 1 :
    history . [true && true && ! [[label : int % caller : any <- self
    : any . *] in 'h $ 1 : history] && 'lwf['h $ 1 : history, self :
    any] && 'mayAcquireProcessor['h $ 1 : history, self : any,
    caller : any, label : int] && ~H~ : history ^^ [self : any .
    release] pr 'h $ 1 : history ==> 'x $ 1 : int ge 0 ==> 'k : int
    le 'x $ 1 : int] fi el 'k : int le 'x @ 'c : int fi
```

This output hardly qualifies as readable. There is no apparent structure, and some parts can obviously be simplified (for example, `true && true`). The systematic qualification and typing for the variables also diminish readability. For larger programs, the output would have been completely opaque.

Contrast this with the output we get when we use the `massaged` function to normalize, simplify, untype, and pretty-print the assertion:

```
Forall 'x $ 1 : int, 'h $ 1 : history .
       5 le 'x @ 'c
  ===>
       'k le 'x @ 'c
 And
       'x @ 'c lt 5
    /\ 0 le 'x @ 'c
  ===>
       'k le 'x @ 'c
 And
       ! ([label % caller <- self . *] in 'h $ 1)
    /\ 'lwf['h $ 1, self]
    /\ 'mayAcquireProcessor['h $ 1, self, caller, label]
    /\ 'x @ 'c lt 0
    /\ 0 le 'x $ 1
    /\ ~H~ ^^ [self . release] pr 'h $ 1
  ===>
       'k le 'x $ 1
```

The pretty-printer introduces low-precedence logical operators that are displayed in bold in the margin. The indentation indicates their relative precedences: `/\` ($\wedge$) binds more strongly than `===>` ($\Longrightarrow$), which binds more strongly than **`And`** ($\bigwedge$), which binds more strongly than **`Forall`** ($\forall$).

The massaged assertion reflects the structure of the original program. The three implications joined by **`And`** correspond to the three cases we identified informally:

1. If $x \geq 5$, we must have $k \leq x$ initially if we want $k \leq x$ to hold afterward.
2. If $0 \leq x < 5$, we must have $k \leq x$ initially if we want $k \leq x$ to hold afterward.

3. If $x < 0$, then $k \leq x_1$ must hold for all $x_1 \in \mathbb{Z}$ such that $0 \leq x_1$ before executing the statement if we want $k \leq x$ to hold afterward. In other words, we need $k \leq 0$. (For this example, we can ignore the conjuncts that involve $\mathcal{H}$ and h_1.)

From the above argument, it would seem that setting $k := 0$ initially is sufficient to ensure that the postcondition holds. This is easy to confirm:

```
red massaged(wlp('k := 0 ; if 'x @ 'c : int lt 5 th
                              await 'x @ 'c : int ge 0 fi,
                 'k : int le 'x @ 'c : int, < true, true >,
                 'x @ 'c : int)) .
*** result Bool: true
```

Here, the `massaged` function simplified the assertion

```
if 'x @ 'c : int lt 5 th if 'x @ 'c : int ge 0 th 0 le 'x @ 'c : int
el true && forall 'x $ 1 : int, 'h $ 1 : history . [true && true &&
! [[label : int % caller : any <- self : any . *] in 'h $ 1 :
history] && 'lwf['h $ 1 : history, self : any] &&
'mayAcquireProcessor['h $ 1 : history, self : any, caller : any,
label : int] && ~H~ : history ^^ [self : any . release] pr 'h $ 1 :
history ==> 'x $ 1 : int ge 0 ==> 0 le 'x $ 1 : int] fi el 0 le 'x @
'c : int fi
```

returned by `wlp` to `true`.

The purpose of the preceding example was to illustrate the critical importance of assertion massaging for the assertion analyzer. We are now going to look at how `massaged` is implemented. Here is the defining equation:

```
eq massaged(PHI, MOD) =
   pretty(untyped(
       simplifiedAndNormalized(normalized(PHI), MOD, 5))) .
```

Assertion massaging involves the following steps:

1. The assertion is *normalized*: Quantifiers are moved outward, conditional expressions are moved outward and rewritten using implications, negations are moved inward, and nested implications are "uncurried" using the rewrite rule $P \Rightarrow Q \Rightarrow R \longrightarrow (P \wedge Q) \Rightarrow R$.

2. The assertion is *simplified* using rewrite rules that replace terms or subformulas with logically equivalent terms or subformulas. The tool's predefined simplification rules can be extended or replaced by a set of user-defined rules.

3. The assertion is *untyped*, meaning that typed identifiers of the form $z : \tau$ are replaced with z.

4. The assertion is *pretty-printed*.

Steps 1 and 2 are repeated to ensure that the assertion is fully normalized and simplified. This is done by `simplifiedAndNormalized` as follows:

```
eq simplifiedAndNormalized(PHI, MOD, 0) = PHI .
ceq simplifiedAndNormalized(PHI, MOD, N) =
    if PHI == PHI' then
        PHI'
```

```
    else
        simplifiedAndNormalized(PHI', MOD, N - 1)
    fi
if PHI' := normalized(simplified(PHI, MOD)) [otherwise] .
```

Normalization itself is implemented using two distinct passes:

```
eq normalized(PHI) = normalized2(normalized1(PHI)) .
```

Each normalization pass is implemented by its own set of rewrite rules, which are executed on demand at the metalevel:

```
eq normalized1(PHI) =
   rewritten(PHI, upModule('CREOL-NORMALIZATION-RULES-1, false)) .

eq normalized2(PHI) =
   rewritten(PHI, upModule('CREOL-NORMALIZATION-RULES-2, false)) .
```

The `rewritten` auxiliary function is defined as follows:

```
eq rewritten(PHI, MOD) =
   downTerm(getTerm(metaRewrite(MOD, upTerm(PHI), unbounded)),
            PHI) .
```

The first normalization pass eliminates square brackets (`[]`) and bi-implications (`<==>`), moves negations inward, moves quantifiers outward, and moves conditional expressions outward. For example, here is a selection of the rewrite rules defined in the `CREOL-NORMALIZATION-RULES-1` module:

```
rl [PHI] => PHI .
rl (PHI1 <==> PHI2) => (PHI1 ==> PHI2) && (PHI2 ==> PHI1) .
rl ! (PHI1 && PHI2) => (! PHI1 || ! PHI2) .
rl ! QUANT XXL . PHI => opposite(QUANT) XXL . (! PHI) .
crl PHI1 && (QUANT XX0, XXL . PHI2) =>
    QUANT XX0 . PHI1 && (QUANT XXL . PHI2)
if not XX0 occurs free in PHI1 .
crl PHI1 && (QUANT XX0, XXL . PHI2) =>
    QUANT YY0 . PHI1 && (QUANT XXL . (PHI2) { XX0 |-> YY0 })
if XX0 occurs free in PHI1
   /\ YY0 := freshLogicalVar(XX0, (XXL, PHI1, PHI2)) .
rl PHI1 && (if PHI2 th PHI3 el PHI4 fi) =>
   if PHI2 th PHI1 && PHI3 el PHI1 && PHI4 fi .
```

The second normalization pass expands conditional expressions to conjunctions of implications, uncurries nested implications, and moves `||` outward:

```
rl if PHI1 th PHI2 el PHI3 fi =>
   ((PHI1 ==> PHI2) && (! PHI1 ==> PHI3)) .
rl (PHI1 ==> (PHI2 ==> PHI3)) => ((PHI1 && PHI2) ==> PHI3) .
rl (PHI1 ==> (PHI2 && (PHI3 ==> PHI4))) =>
   ((PHI1 ==> PHI2) && ((PHI1 && PHI3) ==> PHI4)) .
rl (PHI1 || PHI2) && PHI3 => (PHI1 && PHI3) || (PHI2 && PHI3) .
rl ((PHI1 || PHI2) ==> PHI3) => (PHI1 ==> PHI3) && (PHI2 ==> PHI3) .
```

The simplification is performed in a similar manner, except that this time we take a user-specified module as argument:

```
eq simplified(PHI, MOD) = rewritten(PHI, MOD) .
```

The predefined simplification rules are located in the two modules `CREOL-LOGICAL-SIMPLIFICATION-RULES` and `CREOL-SIMPLIFICATION-RULES`. The first module defines rewrite rules that exploit properties of the logical operators. For example:

```
rl PHI && PHI => PHI .
rl true && PHI => PHI .
rl false && PHI => false .
rl PHI && (! PHI) => false .
rl PHI || PHI => PHI .
rl true || PHI => true .
rl false || PHI => PHI .
rl PHI || (! PHI) => true .
rl (true ==> PHI) => PHI .
rl (false ==> PHI) => true .
rl (PHI ==> true) => true .
rl (PHI ==> PHI) => true .
crl QUANT XXL, XX, XXL' . PHI => QUANT XXL, XXL' . PHI
if not XX occurs free in PHI .
```

The second module imports the logical simplification rules and extends these with simplifications that involve non-logical symbols. Some of the rules compute the value of expressions involving constants; for example:

```
rl N1 times N2 => N1 * N2 .
rl N1 eq N2 => N1 == N2 .
rl #[emptyHistory] => 0 .
```

Other rules eliminate certain operators that can be seen as abbreviations:

```
rl E1 ne E2 => ! (E1 eq E2) .
rl A1 gt A2 => A2 lt A1 .
rl A1 ge A2 => A2 le A1 .
```

Some rules exploit properties of the non-logical symbols to simplify assertions:

```
rl A times 1 => A .
rl 0 eq A1 minus A2 => A1 eq A2 .
rl A1 lt A2 && A2 lt A1 => false .
crl (HEXP ^^ EEXP) / EPEXP => HEXP / EPEXP
if EEXP cannot match EPEXP .
```

A few rules try to expand variables to their definition:

```
crl ((XX eq E && PHI1) ==> PHI2) =>
    ((XX eq E && (PHI1) { XX |-> E }) ==> (PHI2) { XX |-> E })
if XX occurs free in (PHI1, PHI2)
   and not XX occurs free in E
   and not (E :: TypedQualifiedId) .
crl ((XX eq YY && PHI1) ==> PHI2) =>
     (XX eq YY && (PHI1) { XX |-> YY }) ==> (PHI2) { XX |-> YY }
if XX occurs free in (PHI1, PHI2)
   and YY occurs free in (PHI1, PHI2) .
```

In addition to the predefined rules, the user can supply custom simplification rules. These are typically used to expand a custom function application. For example, the following two rules simplify assertions containing the factorial function:

```
rl 'fact[N] => if N > 1 then N * 'fact[N - 1] else 1 fi .
rl 'fact[A] eq 1 => A le 1 .
```

The third step of the massaging process consists of removing the typing annotations tied to the variables that occur in the assertion. This is done by `untyped`:

```
eq untyped(PHI) = downTerm(metaUntyped(upTerm(PHI)), PHI) .
```

Instead of defining `untyped` recursively on all the types of assertions and terms that may occur in assertions, type removal is implemented at the metalevel, where it can be done uniformly with very little code:

```
eq metaUntyped(CONST) = CONST .
eq metaUntyped(VAR) = VAR .
eq metaUntyped('_:_[TERM1, TERM2]) = TERM1 .
eq metaUntyped('__._[TERM1, TERM2, TERM3]) =
   '__._[TERM1, TERM2, metaUntyped(TERM3)] .
eq metaUntyped(QID[TERMLIST]) =
   QID[metaUntyped(TERMLIST)] [otherwise] .
ceq metaUntyped((TERM0, TERMLIST)) =
    metaUntyped(TERM0), metaUntyped(TERMLIST)
if TERMLIST =/= empty .
```

The third equation removes the type in the metaterm for *z* `:` *t*. The fourth equation introduces an exception for `forall` and `exists`, so that newly introduced bound variables keep their type.

In addition to `untyped`, the assertion analyzer also provides an `unqualified` function that removes `@` *c* qualifiers from the output. By having `massaged` use it, we could further condense the output. However, this facility is disabled by default, because for some programs the qualification conveys essential information.

Finally, pretty-printing is implemented by a `PrettyAssn` sort that extends `Assn` with a set of low-precedence operators formatted with the `format` attribute:

```
sort PrettyAssn .
subsort Assn < PrettyAssn .

op _And:_ : PrettyAssn PrettyAssn -> PrettyAssn
 [ctor assoc id: true prec 35 format (y !ynssss oynsssssssssss y)] .
op Forall_._ : TypedIdList PrettyAssn -> PrettyAssn
                   [ctor prec 33 format (!y oy y nsssssssssss y)] .
op Exists_._ : TypedIdList PrettyAssn -> PrettyAssn
                   [ctor prec 33 format (!y oy y nsssssssssss y)] .
op _And_ : PrettyAssn PrettyAssn -> PrettyAssn
    [ctor assoc comm prec 31 format (y !ynssssss oynsssssssssss y)] .
op _===>_ : PrettyAssn PrettyAssn -> PrettyAssn
 [ctor prec 29 gather (e E) format (y !ynsssssss oynsssssssssss y)] .
op _\/_ : PrettyAssn PrettyAssn -> PrettyAssn
  [ctor assoc comm prec 27 format (y !ynssssssss oynsssssssssss y)] .
op _/\_ : PrettyAssn PrettyAssn -> PrettyAssn
             [ctor assoc comm prec 25 format (y !ynssssssss oy y)] .
```

The conversion from a standard `Assn` to a `PrettyAssn` is done using recursive descent by the `pretty` function and its auxiliary functions:

```
op pretty : Assn -> PrettyAssn .
op prettyQuantifier : Assn -> PrettyAssn .
op prettyMiddleAnd : Assn -> PrettyAssn .
op prettyImplies : Assn -> PrettyAssn .
op prettyOr : Assn -> PrettyAssn .
op prettyInnerAnd : Assn -> PrettyAssn .

eq pretty(PHI {{ PHI' }}) =
   pretty(PHI) And: prettyQuantifier(PHI') .
eq pretty(PHI) = prettyQuantifier(PHI) [otherwise] .
    ⋮
eq prettyInnerAnd(PHI1 && PHI2) =
   prettyInnerAnd(PHI1) /\ prettyInnerAnd(PHI2) .
eq prettyInnerAnd(PHI) = PHI [otherwise] .
```

The assertion massaging process relies on Maude's support for rewrite rules and metaprogramming to achieve a lot with very little code. The approach used is fairly ad hoc but has proved surprisingly successful on several small examples, including those presented in the next chapter.

The programmer who tries using toy-language rules to reason about real Pascal programs is in for a rude surprise.

— Leslie Lamport (1993)

Chapter 7

Case Studies

We will now use the assertion analyzer tool presented in Chapter 6 to verify four Creol classes: an Internet bank account (Section 7.1), a read–write lock (Section 7.2), an iterative factorial implementation (Section 7.3), and a recursive factorial implementation (Section 7.4). Despite their simplicity, the examples involve most Creol constructs, notably **if** statements, **while** loops, **await** statements, method calls, and **prove** statements. Examples of any complexity that rely on the Creol dialect described in Chapter 4 (excluding the general case of $|||_{i=1}^{n} S_i$) could be treated in the same way. The complete Maude code for the examples is given in Appendix C.

7.1 An Internet Bank Account

Consider a *NetBankAccount* class that models a simplistic Internet bank account. In a real-world scenario, the user logs into the Internet bank, makes some deposits and payments, and logs out. The transactions are normally performed at night, and if there is not enough money in the account, the payments are delayed. In Creol, this can be modeled by asynchronous method calls:

$account$:= **new** *NetBankAccount*;
!$account.deposit(500)$;
!$account.payBill(875)$;
!$account.deposit(500)$

Because of method overtaking, the bank could receive the deposit and payment requests in any order. Furthermore, to prevent the user from going overdrawn, the bank should first process the two $500 deposits, then pay the bill. The *deposit* and *payBill* methods are declared by the *BankAccount* interface:

interface *BankAccount*
begin
with any:
  op *deposit*(**in** *amount* : **int**)
  op *payBill*(**in** *amount* : **int**)
  asum *noNegatives*($\mathcal{H}$)
end

The *noNegatives* predicate used in the interface's **asum** clause is defined recursively on histories by inspection of invocation events:

$$\begin{array}{lll} \mathit{noNegatives}(\epsilon) & \triangleq & \mathbf{true} \\ \mathit{noNegatives}(h \frown [o{\rightarrow}\mathbf{self}.\mathit{deposit}(n)]^k) & \triangleq & n \geq 0 \wedge \mathit{noNegatives}(h) \\ \mathit{noNegatives}(h \frown [o{\rightarrow}\mathbf{self}.\mathit{payBill}(n)]^k) & \triangleq & n \geq 0 \wedge \mathit{noNegatives}(h) \\ \mathit{noNegatives}(h \frown v) & \triangleq & \mathit{noNegatives}(h). \qquad \text{[otherwise]} \end{array}$$

The **asum** clause expresses the assumption that *deposit* and *payBill* must be called with nonnegative amounts.[1] It is the callers' responsibility to ensure that this requirement is met. The *NetBankAccount* class, which implements the *BankAccount* interface, is declared as follows:

```
class NetBankAccount
    implements BankAccount
begin
    var balance : int
with any:
    op deposit(in amount : int) is
        balance := balance + amount
    op payBill(in amount : int) is
        await balance ≥ amount;
        balance := balance − amount
    guar balance ≥ 0 ∧ balance = sum(ℋ)
end
```

The *NetBackAccount* class achieves synchronization using **await**, in addition to relying on Creol's implicit mutual exclusion for processes in the same object. The class's **guar** clause specifies a guarantee that should hold initially and whenever the processor is released. Intuitively, *NetBankAccount* guarantees that the balance will always be nonnegative and equal to the difference between the deposits and the payments performed so far.

Notice that the *deposit* method does not check that *amount* is nonnegative. As a result, the balance could become negative, breaking the class's guarantee. However, this would violate the the *BankAccount* interface's assumption, which forbids passing negative values to *deposit*. When verifying a class, we may take the assumption for granted.

The *sum* function used in the **guar** clause is defined as follows:

$$\begin{array}{lll} \mathit{sum}(\epsilon) & \triangleq & 0 \\ \mathit{sum}(h \frown [o{\leftarrow}\mathbf{self}.\mathit{deposit}(n)]^k) & \triangleq & \mathit{sum}(h) + n \\ \mathit{sum}(h \frown [o{\leftarrow}\mathbf{self}.\mathit{payBill}(n)]^k) & \triangleq & \mathit{sum}(h) - n \\ \mathit{sum}(h \frown v) & \triangleq & \mathit{sum}(h). \qquad \text{[otherwise]} \end{array}$$

If we disable the predefined non-logical simplification rules and run the assertion analyzer on the code for *NetBankAccount*, we get the report below.

[1] Our minimalistic dialect of Creol provides only the integer type **int**. If we extended the language with a natural number type **nat**, we could use it in the *deposit* and *payBill* method signatures and omit the **asum** clause.

Verification of class *NetBankAccount*

Initialization code:

☐ Establishes the guarantee iff

$$\begin{array}{l} \quad \mathit{noNegatives}([\mathit{parent}(\mathbf{self}) \rightarrow \mathbf{self}.\mathbf{new}\ \mathit{NetBankAccount}()]) \\ \Longrightarrow \\ \quad 0 = \mathit{sum}([\mathit{parent}(\mathbf{self}) \rightarrow \mathbf{self}.\mathbf{new}\ \mathit{NetBankAccount}()] \frown \\ \qquad [\mathbf{self}.\mathbf{initialized}]) \\ \wedge\ 0 \geq 0 \end{array}$$

holds

Method *deposit* of *NetBankAccount*:

☐ Maintains the guarantee iff

$$\begin{array}{l} \quad \neg([\mathbf{caller} \leftarrow \mathbf{self}.*]^{\mathbf{label}}\ \mathbf{in}\ \mathcal{H}) \\ \wedge\ \mathit{lwf}(\mathcal{H}, \mathbf{self}) \\ \wedge\ \mathit{mayAcquireProcessor}(\mathcal{H}, \mathbf{self}, \mathbf{caller}, \mathbf{label}) \\ \wedge\ \mathit{noNegatives}(\mathcal{H}) \\ \wedge\ \mathit{balance} = \mathit{sum}(\mathcal{H}) \\ \wedge\ \mathit{balance} \geq 0 \\ \wedge\ [\mathbf{caller} \rightarrow \mathbf{self}.\mathit{deposit}(\mathit{amount})]^{\mathbf{label}}\ \mathbf{in}\ \mathcal{H} \\ \Longrightarrow \\ \quad \mathit{sum}(\mathcal{H} \frown [\mathbf{caller} \leftarrow \mathbf{self}.\mathit{deposit}(\mathit{amount};)]) = \mathit{amount} + \mathit{balance} \\ \wedge\ \mathit{amount} + \mathit{balance} \geq 0 \end{array}$$

holds

Method *payBill* of *NetBankAccount*:

☐ Maintains the guarantee iff

$\forall \mathit{balance}_1 : \mathbf{int},\ h_1 : \mathbf{history}.$

$$\begin{array}{l} \quad \neg(\mathit{balance} \geq \mathit{amount}) \\ \wedge\ \neg([\mathbf{caller} \leftarrow \mathbf{self}.*]^{\mathbf{label}}\ \mathbf{in}\ \mathcal{H}) \\ \wedge\ \mathit{lwf}(\mathcal{H}, \mathbf{self}) \\ \wedge\ \mathit{mayAcquireProcessor}(\mathcal{H}, \mathbf{self}, \mathbf{caller}, \mathbf{label}) \\ \wedge\ \mathit{noNegatives}(\mathcal{H}) \\ \wedge\ \mathit{balance} = \mathit{sum}(\mathcal{H}) \\ \wedge\ \mathit{balance} \geq 0 \\ \wedge\ [\mathbf{caller} \rightarrow \mathbf{self}.\mathit{payBill}(\mathit{amount})]^{\mathbf{label}}\ \mathbf{in}\ \mathcal{H} \\ \Longrightarrow \\ \quad \mathit{balance} = \mathit{sum}(\mathcal{H} \frown [\mathbf{self}.\mathbf{release}]) \\ \wedge \\ \quad \neg([\mathbf{caller} \leftarrow \mathbf{self}.*]^{\mathbf{label}}\ \mathbf{in}\ \mathcal{H}) \\ \wedge\ \mathit{lwf}(\mathcal{H}, \mathbf{self}) \\ \wedge\ \mathit{mayAcquireProcessor}(\mathcal{H}, \mathbf{self}, \mathbf{caller}, \mathbf{label}) \\ \wedge\ \mathit{noNegatives}(\mathcal{H}) \\ \wedge\ \mathit{balance} = \mathit{sum}(\mathcal{H}) \\ \wedge\ \mathit{balance} \geq 0 \\ \wedge\ \mathit{balance} \geq \mathit{amount} \\ \wedge\ [\mathbf{caller} \rightarrow \mathbf{self}.\mathit{payBill}(\mathit{amount})]^{\mathbf{label}}\ \mathbf{in}\ \mathcal{H} \\ \Longrightarrow \\ \quad \mathit{sum}(\mathcal{H} \frown [\mathbf{caller} \leftarrow \mathbf{self}.\mathit{payBill}(\mathit{amount};)]^{\mathbf{label}}) = \mathit{balance} - \mathit{amount} \\ \wedge\ \mathit{balance} - \mathit{amount} \geq 0 \\ \wedge \end{array}$$

$$
\begin{array}{ll}
 & \neg(\mathit{balance} \geq \mathit{amount}) \\
\wedge & \neg([\mathbf{caller}{\leftarrow}\mathbf{self}.*]^{\mathbf{label}}\ \mathbf{in}\ \mathcal{H}) \\
\wedge & \neg([\mathbf{caller}{\leftarrow}\mathbf{self}.*]^{\mathbf{label}}\ \mathbf{in}\ h_1) \\
\wedge & \mathit{lwf}(\mathcal{H}, \mathbf{self}) \\
\wedge & \mathit{lwf}(h_1, \mathbf{self}) \\
\wedge & \mathit{mayAcquireProcessor}(\mathcal{H}, \mathbf{self}, \mathbf{caller}, \mathbf{label}) \\
\wedge & \mathit{mayAcquireProcessor}(h_1, \mathbf{self}, \mathbf{caller}, \mathbf{label}) \\
\wedge & \mathit{noNegatives}(\mathcal{H}) \\
\wedge & \mathit{noNegatives}(h_1) \\
\wedge & \mathit{balance}_1 = \mathit{sum}(h_1) \\
\wedge & \mathit{balance} = \mathit{sum}(\mathcal{H}) \\
\wedge & \mathit{balance}_1 \geq 0 \\
\wedge & \mathit{balance}_1 \geq \mathit{amount} \\
\wedge & \mathit{balance} \geq 0 \\
\wedge & (\mathcal{H} \frown [\mathbf{self}.\mathbf{release}])/(\mathit{out}_{\mathbf{self}} \cup \mathit{ctl}_{\mathbf{self}}) = h_1/(\mathit{out}_{\mathbf{self}} \cup \mathit{ctl}_{\mathbf{self}}) \Rightarrow \\
 & \quad \mathit{balance}_1 = \mathit{balance} \\
\wedge & [\mathbf{caller}{\rightarrow}\mathbf{self}.\mathit{payBill}(\mathit{amount})]^{\mathbf{label}}\ \mathbf{in}\ \mathcal{H} \\
\wedge & \mathcal{H} \frown [\mathbf{self}.\mathbf{release}] \preceq h_1 \\
\Longrightarrow & \\
 & \mathit{sum}(h_1 \frown [\mathbf{caller}{\leftarrow}\mathbf{self}.\mathit{payBill}(\mathit{amount};)]^{\mathbf{label}}) = \mathit{balance}_1 - \mathit{amount} \\
\wedge & \mathit{balance}_1 - \mathit{amount} \geq 0
\end{array}
$$

holds

Notice in particular that the applications of the auxiliary functions *sum* and *noNegatives* are not expanded. To be of any use to the assertion analyzer, the defining equations for the auxiliary functions must be converted into simplification rules. The process is fairly straightforward; for example, here are the rules for *sum*:

$$
\begin{array}{l}
\mathit{sum}(\epsilon) \longrightarrow 0 \\
\mathit{sum}(H \frown [O{\leftarrow}\mathbf{self}{:}\mathbf{any}.\mathit{deposit}(A)]^{A'}) \longrightarrow \mathit{sum}(H) + A \\
\mathit{sum}(H \frown [O{\leftarrow}\mathbf{self}{:}\mathbf{any}.\mathit{payBill}(A)]^{A'}) \longrightarrow \mathit{sum}(H) - A \\
\mathit{sum}(H \frown Y) \longrightarrow \mathit{sum}(H) \qquad \textbf{if } Y \textbf{ cannot match reply}.
\end{array}
$$

In the above, $H \in \mathit{HistoryExp}$, $O \in \mathit{OExp}$, $A, A' \in \mathit{AExp}$, and $Y \in \mathit{EventExp}$. The last rule is invoked when it can be determined that the event expression Y is an object creation event, a method invocation event, or an internal control event—all of which fall into the "otherwise" case of the *sum* definition. Since the simplification rules are applied before the type annotations are removed, the program and logical variables that occur in the rules (in this case, **self**) must carry a type.

If we do the same for *noNegatives* and run the assertion analyzer with the predefined simplifications enabled, we obtain the following report:

Verification of class *NetBankAccount*

Initialization code:
☑ Establishes the guarantee

Method *deposit* of *NetBankAccount*:
❑ Maintains the guarantee iff

$$
\begin{array}{ll}
 & \neg([\mathbf{caller}{\leftarrow}\mathbf{self}.*]^{\mathbf{label}}\ \mathbf{in}\ \mathcal{H}) \\
\wedge & \mathit{lwf}(\mathcal{H}, \mathbf{self})
\end{array}
$$

$$
\begin{array}{l}
\quad \wedge\ \mathit{mayAcquireProcessor}(\mathcal{H}, \mathbf{self}, \mathbf{caller}, \mathbf{label}) \\
\quad \wedge\ \mathit{noNegatives}(\mathcal{H}) \\
\quad \wedge\ 0 \leq \mathit{sum}(\mathcal{H}) \\
\quad \wedge\ [\mathbf{caller} \rightarrow \mathbf{self}.\mathit{deposit}(\mathit{amount})]^{\mathbf{label}}\ \mathbf{in}\ \mathcal{H} \\
\Longrightarrow \\
\qquad 0 \leq \mathit{amount} + \mathit{sum}(\mathcal{H}) \\
\text{holds}
\end{array}
$$

Method *payBill* of *NetBankAccount*:
☑ Maintains the guarantee

The tool automatically discharges the verification conditions associated with the initialization code (an implicit $\mathit{balance} := 0$ statement) and the *payBill* method. On the other hand, it doesn't manage to simplify the *deposit* method's verification condition to **true**, so we must proceed by hand.

The verification condition is of the form $(P_1 \wedge \cdots \wedge P_6) \Rightarrow Q$, where $Q \equiv 0 \leq \mathit{amount} + \mathit{sum}(\mathcal{H})$. Among the premises, we find $0 \leq \mathit{sum}(\mathcal{H})$. The key to complete the proof is to observe that *amount* must be nonnegative (by the $\mathit{noNegatives}(\mathcal{H})$ premise), and then the conclusion Q is clearly true if the premises are true.

We can formalize this into the following simplification rule, generalizing a bit:

$$
\begin{array}{l}
(A_1 \leq A_2 \wedge \mathit{noNegatives}(H) \wedge [O \rightarrow \mathbf{self}.\mathit{deposit}(A_3)]^k\ \mathbf{in}\ H \wedge \varphi) \Rightarrow \\
A_1 \leq A_2 + A_3 \\
\quad \longrightarrow \\
\mathbf{true}.
\end{array}
$$

The variable φ stands for the additional conjuncts, which do not influence the truth value of the formula on the left-hand side of the rewrite rule. If we run the assertion analyzer after adding this simplification rule, the rule will replace the entire implication $(P_1 \wedge \cdots \wedge P_6) \Rightarrow Q$ with **true**, and the report will now state that *deposit* maintains the guarantee.

This completes the verification of the *NetBankAccount* class using the assertion analyzer. To increase our confidence in the result, we could use a theorem prover to verify that the simplification rules are sound with respect to the equational definitions of the auxiliary functions *noNegatives* and *sum*.

7.2 Readers–Writers Synchronization

We will now study an implementation of a read–write lock, a synchronization tool for protecting shared resources that can be accessed for reading and writing [And00]. When used correctly, read–write locks enforce the following policy:

1. Multiple processes may read the shared data simultaneously.
2. A process is allowed to modify the shared data only when no other processes are accessing the data in any way.

Reader and writer processes synchronize their accesses to the data using an instance of a class that implements the *RWLock* interface. For example, here is the

pseudocode of a *Reader* and a *Writer* class that use a read–write lock to synchronize accesses to an unspecified shared resource:

```
class Reader (lock : RWLock)
begin
   op run is
      while true do
         lock.beginRead();
         ⟨read data⟩;
         !lock.endRead();
         ⟨other processing⟩
      od
end
```

```
class Writer (lock : RWLock)
begin
   op run is
      while true do
         lock.beginWrite();
         ⟨write data⟩;
         !lock.endWrite();
         ⟨other processing⟩
      od
end
```

The calls to *beginRead* and *beginWrite* block until the lock is granted to the process, at which point the process may access the shared data; once it is finished, it releases the lock by calling *endRead* or *endWrite*. The *RWLock* interface declaration follows:

```
interface RWLock
begin
with any:
   op beginRead
   op endRead
   op beginWrite
   op endWrite
   asum #(H/[∗→self.beginRead()]) ≥ #(H/[∗→self.endRead()])
        ∧ #(H/[∗→self.beginWrite()]) ≥ #(H/[∗→self.endWrite()])
   guar numWriters(H) = 0 ∨ (numWriters(H) = 1 ∧ numReaders(H) = 0)
end
```

The **asum** clause expresses the requirement that at any time the number of incoming calls to *beginRead* should be greater than or equal to *endRead*, and similarly for *beginWrite* and *endWrite*. (Recall that the function $\#(h)$ returns the length of h.) The **guar** clause states that implementations of the interface realize the locking policy described earlier. The *numReaders* and *numWriters* functions return the number of active readers and writers:

$$
\begin{array}{lll}
numReaders(\epsilon) & \triangleq 0 & \\
numReaders(h \frown [o{\leftarrow}\textbf{self}.beginRead()]^k) & \triangleq numReaders(h) + 1 & \\
numReaders(h \frown [o{\leftarrow}\textbf{self}.endRead()]^k) & \triangleq numReaders(h) - 1 & \\
numReaders(h \frown v) & \triangleq numReaders(h) & \text{[otherwise]}
\end{array}
$$

$$
\begin{array}{lll}
numWriters(\epsilon) & \triangleq 0 & \\
numWriters(h \frown [o{\leftarrow}\textbf{self}.beginWrite()]^k) & \triangleq numWriters(h) + 1 & \\
numWriters(h \frown [o{\leftarrow}\textbf{self}.endWrite()]^k) & \triangleq numWriters(h) - 1 & \\
numWriters(h \frown v) & \triangleq numWriters(h). & \text{[otherwise]}
\end{array}
$$

The *WriterFriendlyRWLock* class that we will verify in this section is an implementation of the "writers' preference" solution to the readers–writers problem. It is declared as follows:

class *WriterFriendlyRWLock*
  implements *RWLock*
begin
  var nr : **int**, nw : **int**, dw : **int**
with any:
  op *beginRead* **is**
    await $nw = 0 \wedge dw = 0$;
    $nr := nr + 1$
  op *endRead* **is**
    prove $nr > 0$;
    $nr := nr - 1$
  op *beginWrite* **is**
    $dw := dw + 1$;
    await $nr = 0 \wedge nw = 0$;
    $dw := dw - 1$;
    $nw := nw + 1$
  op *endWrite* **is**
    prove $nw > 0$;
    $nw := nw - 1$
  guar $nr = numReaders(\mathcal{H}) \wedge nw = numWriters(\mathcal{H})$
end

The class declares three attributes: nr (the number of active readers), nw (the number of active writers), and dw (the number of delayed writers). Readers are allowed to proceed only when no writers are active or waiting to become active—hence the label "writers' preference". The class guarantee relates the value of the attributes nr and nw to the history $\mathcal{H}$; this will simplify the proofs later on. Notice that the *endRead* and *endWrite* methods start with a **prove** statement.

If we convert the defining equations for *numReaders* and *numWriters* into simplification rules and run the assertion analyzer, we obtain the following report:

Verification of class *WriterFriendlyRWLock*

Initialization code:
  ☑ Establishes the guarantee

Method *beginRead* of *WriterFriendlyRWLock*:
  ☑ Maintains the guarantee

Method *endRead* of *WriterFriendlyRWLock*:
  ❓ Maintains the guarantee if

$$\begin{aligned}
&\neg([\mathbf{caller} \leftarrow \mathbf{self}.*]^{\mathbf{label}} \mathbf{\ in\ } \mathcal{H}) \\
\wedge\ & lwf(\mathcal{H}, \mathbf{self}) \\
\wedge\ & mayAcquireProcessor(\mathcal{H}, \mathbf{self}, \mathbf{caller}, \mathbf{label}) \\
\wedge\ & 0 = numWriters(\mathcal{H}) \\
\wedge\ & \#(\mathcal{H}/[* \rightarrow \mathbf{self}.endRead()]) \leq \#(\mathcal{H}/[* \rightarrow \mathbf{self}.beginRead()]) \\
\wedge\ & \#(\mathcal{H}/[* \rightarrow \mathbf{self}.endWrite()]) \leq \#(\mathcal{H}/[* \rightarrow \mathbf{self}.beginWrite()]) \\
\wedge\ & [\mathbf{caller} \rightarrow \mathbf{self}.endRead()]^{\mathbf{label}} \mathbf{\ in\ } \mathcal{H}
\end{aligned}$$

$$
\begin{array}{l}
\quad \Longrightarrow \\
\qquad 0 < \mathit{numReaders}(\mathcal{H}) \\
\wedge \\
\qquad \neg([\mathbf{caller} \leftarrow \mathbf{self}.*]^{\mathbf{label}} \mathbf{in}\ \mathcal{H}) \\
\quad \wedge\ \mathit{lwf}(\mathcal{H}, \mathbf{self}) \\
\quad \wedge\ \mathit{mayAcquireProcessor}(\mathcal{H}, \mathbf{self}, \mathbf{caller}, \mathbf{label}) \\
\quad \wedge\ 0 = \mathit{numReaders}(\mathcal{H}) \\
\quad \wedge\ 1 = \mathit{numWriters}(\mathcal{H}) \\
\quad \wedge\ \#(\mathcal{H}/[* \rightarrow \mathbf{self}.\mathit{endRead}()]) \leq \#(\mathcal{H}/[* \rightarrow \mathbf{self}.\mathit{beginRead}()]) \\
\quad \wedge\ \#(\mathcal{H}/[* \rightarrow \mathbf{self}.\mathit{endWrite}()]) \leq \#(\mathcal{H}/[* \rightarrow \mathbf{self}.\mathit{beginWrite}()]) \\
\quad \wedge\ [\mathbf{caller} \rightarrow \mathbf{self}.\mathit{endRead}()]^{\mathbf{label}} \mathbf{in}\ \mathcal{H} \\
\quad \Longrightarrow \\
\qquad \mathbf{false} \\
\text{holds}
\end{array}
$$

Method *beginWrite* of *WriterFriendlyRWLock*:

☑ Maintains the guarantee

Method *endWrite* of *WriterFriendlyRWLock*:

❓ Maintains the guarantee if

$$
\begin{array}{l}
\qquad \neg([\mathbf{caller} \leftarrow \mathbf{self}.*]^{\mathbf{label}} \mathbf{in}\ \mathcal{H}) \\
\quad \wedge\ \mathit{lwf}(\mathcal{H}, \mathbf{self}) \\
\quad \wedge\ \mathit{mayAcquireProcessor}(\mathcal{H}, \mathbf{self}, \mathbf{caller}, \mathbf{label}) \\
\quad \wedge\ 0 = \mathit{numWriters}(\mathcal{H}) \\
\quad \wedge\ \#(\mathcal{H}/[* \rightarrow \mathbf{self}.\mathit{endRead}()]) \leq \#(\mathcal{H}/[* \rightarrow \mathbf{self}.\mathit{beginRead}()]) \\
\quad \wedge\ \#(\mathcal{H}/[* \rightarrow \mathbf{self}.\mathit{endWrite}()]) \leq \#(\mathcal{H}/[* \rightarrow \mathbf{self}.\mathit{beginWrite}()]) \\
\quad \wedge\ [\mathbf{caller} \rightarrow \mathbf{self}.\mathit{endWrite}()]^{\mathbf{label}} \mathbf{in}\ \mathcal{H} \\
\quad \Longrightarrow \\
\qquad \mathbf{false} \\
\text{holds}
\end{array}
$$

Let us first consider *endWrite*. The proof obligation is of the form $(P_1 \wedge \cdots \wedge P_7) \Rightarrow$ **false**, or equivalently $\neg P_1 \vee \cdots \vee \neg P_7$, with

$$
\begin{array}{rcl}
P_1 & \equiv & \neg([\mathbf{caller} \leftarrow \mathbf{self}.*]^{\mathbf{label}} \mathbf{in}\ \mathcal{H}) \\
P_2 & \equiv & \mathit{lwf}(\mathcal{H}, \mathbf{self}) \\
P_3 & \equiv & \mathit{mayAcquireProcessor}(\mathcal{H}, \mathbf{self}, \mathbf{caller}, \mathbf{label}) \\
P_4 & \equiv & 0 = \mathit{numWriters}(\mathcal{H}) \\
P_5 & \equiv & \#(\mathcal{H}/[* \rightarrow \mathbf{self}.\mathit{endRead}()]) \leq \#(\mathcal{H}/[* \rightarrow \mathbf{self}.\mathit{beginRead}()]) \\
P_6 & \equiv & \#(\mathcal{H}/[* \rightarrow \mathbf{self}.\mathit{endWrite}()]) \leq \#(\mathcal{H}/[* \rightarrow \mathbf{self}.\mathit{beginWrite}()]) \\
P_7 & \equiv & [\mathbf{caller} \rightarrow \mathbf{self}.\mathit{endWrite}()]^{\mathbf{label}} \mathbf{in}\ \mathcal{H}.
\end{array}
$$

To complete the proof, we must show that the premises contradict each other. The premises P_1 and P_7 state that there is a pending call to *endWrite*. If the call hasn't been processed yet, and *endWrite* hasn't been called more times than *beginWrite* (P_6), and the history is well-formed (P_2), then $\mathit{numWriters}(\mathcal{H})$ should be at least 1 (by definition). This contradicts P_4. Since it is impossible to satisfy all the premises simultaneously, we can formulate the following simplification rule:

$$
\begin{array}{l}
[O \rightarrow \mathbf{self}{:}\mathbf{any}.\mathit{endWrite}()]^A \mathbf{in}\ H \\
\wedge\ \neg([O \leftarrow \mathbf{self}{:}\mathbf{any}.*]^A \mathbf{in}\ H)
\end{array}
$$

$$
\begin{array}{l}
\quad \wedge\ \mathit{numWriters}(H) = 0 \\
\quad \wedge\ \mathit{lwf}(H, \mathbf{self}{:}\mathbf{any}) \\
\quad \wedge\ \#(H/[* \rightarrow \mathbf{self}{:}\mathbf{any}.\mathit{endWrite}()]) \leq \#(H/[* \rightarrow \mathbf{self}{:}\mathbf{any}.\mathit{beginWrite}()]) \\
\quad \longrightarrow \\
\quad \mathbf{false}.
\end{array}
$$

This rule is sufficient to complete the proof for *endWrite*.

The proof for *endRead* is slightly more challenging. The proof obligation is of the form $(P \Rightarrow Q) \wedge (P' \Rightarrow \mathbf{false})$, where P and P' are conjunctions. The second implication can be eliminated using a similar rule to the one shown above:

$$
\begin{array}{l}
\quad [O \rightarrow \mathbf{self}{:}\mathbf{any}.\mathit{endRead}()]^A \ \mathbf{in}\ H \\
\quad \wedge\ \neg([O \leftarrow \mathbf{self}{:}\mathbf{any}.*]^A \ \mathbf{in}\ H) \\
\quad \wedge\ \mathit{numReaders}(H) = 0 \\
\quad \wedge\ \mathit{lwf}(H, \mathbf{self}{:}\mathbf{any}) \\
\quad \wedge\ \#(H/[* \rightarrow \mathbf{self}{:}\mathbf{any}.\mathit{endRead}()]) \leq \#(H/[* \rightarrow \mathbf{self}{:}\mathbf{any}.\mathit{beginRead}()]) \\
\quad \longrightarrow \\
\quad \mathbf{false}.
\end{array}
$$

If we now run the assertion analyzer, we get the following output for *endRead*:

Method *endRead* of *WriterFriendlyRWLock*:

❑ Maintains the guarantee if

$$
\begin{array}{l}
\quad \neg([\mathbf{caller} \leftarrow \mathbf{self}.*]^{\mathbf{label}} \ \mathbf{in}\ \mathcal{H}) \\
\quad \wedge\ \mathit{lwf}(\mathcal{H}, \mathbf{self}) \\
\quad \wedge\ \mathit{mayAcquireProcessor}(\mathcal{H}, \mathbf{self}, \mathbf{caller}, \mathbf{label}) \\
\quad \wedge\ 0 = \mathit{numWriters}(\mathcal{H}) \\
\quad \wedge\ \#(\mathcal{H}/[* \rightarrow \mathbf{self}.\mathit{endRead}()]) \leq \#(\mathcal{H}/[* \rightarrow \mathbf{self}.\mathit{beginRead}()]) \\
\quad \wedge\ \#(\mathcal{H}/[* \rightarrow \mathbf{self}.\mathit{endWrite}()]) \leq \#(\mathcal{H}/[* \rightarrow \mathbf{self}.\mathit{beginWrite}()]) \\
\quad \wedge\ [\mathbf{caller} \rightarrow \mathbf{self}.\mathit{endRead}()]^{\mathbf{label}} \ \mathbf{in}\ \mathcal{H} \\
\Longrightarrow \\
\quad 0 < \mathit{numReaders}(\mathcal{H})
\end{array}
$$

holds

Intuitively, if there is a pending call to *endRead*, and *endRead* hasn't been called more times than *beginRead*, and $\mathcal{H}$ is well-formed, then $\mathit{numReaders}(\mathcal{H})$ should be greater than 0. This enables us to formulate the following simplification rule and complete the proof:

$$
\begin{array}{l}
\big([O \rightarrow \mathbf{self}{:}\mathbf{any}.\mathit{endRead}()]^A \ \mathbf{in}\ H \\
\quad \wedge\ \neg([O \leftarrow \mathbf{self}{:}\mathbf{any}.*]^A \ \mathbf{in}\ H) \\
\quad \wedge\ \mathit{lwf}(H, \mathbf{self}{:}\mathbf{any}) \\
\quad \wedge\ \#(H/[* \rightarrow \mathbf{self}{:}\mathbf{any}.\mathit{endRead}()]) \leq \#(H/[* \rightarrow \mathbf{self}{:}\mathbf{any}.\mathit{beginRead}()]) \\
\quad \wedge\ \varphi\big) \Rightarrow \\
0 < \mathit{numReaders}(H) \\
\quad \longrightarrow \\
\quad \mathbf{true}.
\end{array}
$$

Incidentally, we now know not only that the *WriterFriendlyRWLock* class implements the locking policy mandated by the *RWLock* interface, but also that the assertions in the **prove** statements of *endRead* and *endWrite* will hold as long as the interface assumption holds.

7.3 An Iterative Factorial Program

Creol offers both iteration and recursion for performing repeated actions. In this section, we will verify a class called *IterativeFactorial* that uses a **while** loop to compute the factorial of an integer. In the next section, we will study a recursive implementation of the class. Both classes support the following *Factorial* interface:

interface *Factorial*
begin
with any:
  op *compute*(**in** x : **int** **out** y : **int**)
  guar $G(\mathcal{H})$
end

The *Factorial* interface makes no assumptions about the environment. The guarantee $G(\mathcal{H})$ states that the results of the *compute* method are mathematically sound:

$$\begin{array}{lll} G(\epsilon) & \triangleq & \textbf{true} \\ G(h \frown [o \leftarrow \textbf{self}.compute(x;y)]^k) & \triangleq & G(h) \wedge y = x! \\ G(h \frown v) & \triangleq & G(h). \qquad \text{[otherwise]} \end{array}$$

The factorial of $n \in \mathbb{Z}$, denoted $n!$, is defined by a pair of conditional equations:

$$\begin{array}{lll} n! & \triangleq & n * (n-1)! \qquad \text{if } n > 1 \\ n! & \triangleq & 1 \qquad\qquad\quad \text{if } n \leq 1. \end{array}$$

The *IterativeFactorial* class is declared as follows:

class *IterativeFactorial*
  implements *Factorial*
begin
with any:
  op *compute*(**in** x : **int** **out** y : **int**) **is**
    var i : **int**;
    $i := 1$;
    $y := 1$;
    inv $G(\mathcal{H}) \wedge i \geq 1 \wedge (i \leq x \vee (i = 1 \wedge x < 1)) \wedge y = i!$
    while $i < x$ **do**
      $i := i + 1$;
      $y := y * i$
    od
end

The most noteworthy feature of the *compute* method's implementation is the loop invariant, supplied in the **inv** clause. To enable verification, we must supply a sufficiently strong invariant. The invariant and the negated loop condition taken together must imply $G(\mathcal{H}) \wedge y = x!$, which must hold at the end of the loop so that $G(\mathcal{H} \frown [\textbf{caller} \leftarrow \textbf{self}.compute(x;y)]^{\textbf{label}})$ holds immediately after the method has returned. Naturally, to qualify as a loop invariant, the assertion specified in the **inv** clause must also hold before entering the loop and after each iteration.

Before we run the assertion analyzer on the *IterativeFactorial* class, we can already define the following simplification rule to reduce $n!$ to its value when n is known:

$$n! \longrightarrow \textbf{if } n > 1 \textbf{ then } n * (n-1)! \textbf{ else } 1 \textbf{ fi}.$$

Given the above invariant and simplification rule, the assertion analyzer produces the following output:

Verification of class *IterativeFactorial*

Initialization code:
☑ Establishes the guarantee

Method *compute* of *IterativeFactorial*:
❓ Maintains the guarantee if

$$\begin{array}{l} \quad G(\mathcal{H}) \\ \wedge\ i < x \\ \wedge\ 1 \leq i \\ \Longrightarrow \\ \quad (i+1)! = i! * (i+1) \end{array}$$

$$\bigwedge$$

$$\begin{array}{l} \quad G(\mathcal{H}) \\ \wedge\ x < 1 \\ \Longrightarrow \\ \quad 1 = x! \end{array}$$

holds

The proof obligation consists of two conjuncts. Observing that $(n+1)! = n! * (n+1)$ for $n \geq 1$, we can eliminate the first conjunct with the rule

$$((1 \leq A \wedge \varphi) \Rightarrow (A+1)! = A! * (A+1)) \longrightarrow \textbf{true}.$$

To eliminate the second conjunct, we can use the rule

$$A! = 1 \longrightarrow A \leq 1.$$

This completes the proof.

7.4 A Recursive Factorial Program

With an appropriate loop invariant, verifying *IterativeFactorial* turned out to be easy. Let us now consider the recursive version of the factorial program:

```
class RecursiveFactorial
    implements Factorial
begin
with any:
    op compute(in x : int out y : int) is
        if x ≤ 1 then
            y := 1
        else
            compute(x − 1; y);
```

$y := y * x$
fi
end

If we run the assertion analyzer on *RecursiveFactorial* together with the simplification rules specified in the previous section, we get the following report:

Verification of class *RecursiveFactorial*

Initialization code:
☑ Establishes the guarantee

Method *compute* of *RecursiveFactorial*:
❓ Maintains the guarantee iff
$\forall k_1 : \mathbf{int}, h_1 : \mathbf{history}.$
$\neg([\mathbf{caller} \leftarrow \mathbf{self}.*]^{\mathbf{label}} \mathbf{in}\ h_1)$
$\wedge\ G(\mathcal{H})$
$\wedge\ G(h_1)$
$\wedge\ isFreshSequenceNum(k_1, \mathbf{self}, \mathcal{H})$
$\wedge\ lwf(h_1, \mathbf{self})$
$\wedge\ mayAcquireProcessor(\mathcal{H}, \mathbf{self}, \mathbf{caller}, \mathbf{label})$
$\wedge\ 1 < x$
$\wedge\ [\mathbf{caller} \rightarrow \mathbf{self}.compute(x)]^{\mathbf{label}} \mathbf{in}\ \mathcal{H}$
$\wedge\ h_1\ \mathbf{ew}\ [\mathbf{self} \leftarrow \mathbf{self}.*]^{k_1}$
$\wedge\ \mathcal{H} \frown [\mathbf{self} \rightarrow \mathbf{self}.compute(x-1)]^{k_1} \frown [\mathbf{self}.reenter]^{k_1} \preceq h_1$
$\Longrightarrow$
$x! = x * returnVal_1(h_1, \mathbf{self}, k_1)$
holds

We must show that the conclusion $x! = x * returnVal_1(h_1, \mathbf{self}, k_1)$ follows from the premises. (Recall that the function $returnVal_i(h, o, k)$ extracts the ith return value for the method call identified by the pair (o, k) from the history h.) From the seventh premise, we know that $x > 1$. By the definition of $n!$, the conclusion will be true if and only if $returnVal_1(h_1, \mathbf{self}, k_1) = (x-1)!$ is true. In other words, we must show that the return value of the call identified by $(\mathbf{self}, k_1)$ is $(x-1)!$ according to h_1.

The last premise tells us that the call $(\mathbf{self}, k_1)$ had $x - 1$ as input argument according to h_1. The next-to-last premise tells us that the call has returned. Since the interface guarantee holds for h_1 (by the third premise), and h_1 is well-formed (by the fifth premise), h_1 contains the reply event $[\mathbf{self} \leftarrow \mathbf{self}.compute(x-1; y)]^{k_1}$, with $y = (x-1)!$, and we get $returnVal_1(h_1, \mathbf{self}, k_1) = (x-1)!$ by the definition of *returnVal*. Quod erat demonstrandum.

As we did for the other examples, we can capture this reasoning in a simplification rule and run the assertion analyzer again:

$(G(H')$
$\wedge\ lwf(H', \mathbf{self}{:}\mathbf{any})$
$\wedge\ H \frown [\mathbf{self}{:}\mathbf{any} \rightarrow \mathbf{self}{:}\mathbf{any}.compute(A-1)]^{A'} \frown [\mathbf{self}{:}\mathbf{any}.\mathbf{reenter}]^{A'} \preceq H'$
$\wedge\ \varphi) \Rightarrow$
$A! = A * returnVal_1(H', \mathbf{self}{:}\mathbf{any}, A')$
$\longrightarrow$
true.

It is instructive to compare the verification process for the *RecursiveFactorial* class with that of *IterativeFactorial*. For the iterative version, we must supply a loop invariant that completely captures the behavior of the loop. In contrast, for the recursive version, there is no extra specification work involved: When performing the recursive call $x - 1$, we can directly use the guarantee $G(\mathcal{H})$ to deduce that the return value of the recursive call is $(x-1)!$, like in a proof by induction.

With both implementations, we must keep in mind that we have only proved that the *compute* method will return a correct result if it terminates. We have not proved that the method will actually terminate. In fact, we could use the assertion analyzer to prove the following class correct in the sense of partial correctness:

class *NonterminatingFactorial*
  implements *Factorial*
begin
with any:
  op *compute*(**in** x : **int** **out** y : **int**) **is**
    compute$(x; y)$
end

In the proof, we get a circular argument: *compute*$(x; y)$ gives $y = x!$ if *compute*$(x; y)$ gives $y = x!$. Nevertheless, as we saw in Section 5.1, partial correctness is strong enough to express that a program will never terminate. For the preceding class, we can assert the nontermination of *compute* through the following class guarantee:

guar $[* \leftarrow \textbf{self}.compute(*)]$ **not in** $\mathcal{H}$

Informally, the guarantee states that calls to *compute* will never return. Using the assertion analyzer, we can easily verify that this guarantee holds.

7.5 Summary

We considered four examples in this chapter, which between them cover most Creol constructs. The table below summarizes the verification of the examples.

Example	Number of VCs	Number of Def. Rules	Number of Proved VCs	Number of Add. Rules
Bank Account	3	8	2	1
Read-Write Lock	5	12	3	3
Iterative Factorial	2	4	1	2
Recursive Factorial	2	4	1	1

The second column gives the number of verification conditions for each example, which is always $1 + \langle \text{method count} \rangle$. The third column gives the number of simplification rules that were specified based on the defining equations of the custom functions. The fourth column gives the number of verification conditions that were discharged automatically by the tool, using the built-in and custom simplification rules. Finally, the fifth column gives the number of additional simplification rules required to discharge the remaining proof obligations.

These are general philosophical and moral principles, and I hold them to be self-evident—which is just as well, because all the actual evidence is against them.

— C. A. R. Hoare (1985)

Chapter 8

Conclusion

The Creol language supports object-orientation in a high-level and intuitive way by means of concurrent objects with processor release points and asynchronous methods calls. The language lets programmers supply assume–guarantee specifications, which define the provided and required interaction and semantic behavior of classes and interfaces.

In this thesis, we presented Creol's syntax, provided two variants of the operational semantics, presented a proof system to verify assume–guarantee specifications, implemented the proof system in Maude, and used the resulting Maude program, the assertion analyzer, to verify four Creol classes. We can now summarize the results and propose directions for future work.

8.1 Results

In Section 1.2 of the introduction, we posed three questions that we wished to answer in the thesis. The first question was:

> *How can we adapt the existing proof system to fully account for the more challenging aspects of Creol's formal semantics, such as object reentry and the nondeterministic statements?*

This question is concerned with the soundness and completeness (left-maximality) of the individual axioms and proof rules with respect to the operational semantics, and the completeness of the set of rules. Section 5.7 lists the steps that we took to answer it. In particular:

- We introduced three types of events in the history, corresponding to the end of an object's initialization, the release of the processor, and object reentry.
- For invocation and reply events, we added the sequence numbers of the method calls.
- We provided proof rules for Creol's nondeterministic choice statement and a special case of the nondeterministic merge statement, both of which are specific to Creol and were not covered by previous work.

- We provided left-maximal axioms for the conditional wait with reply guard statement and for the reply statement.
- We eliminated the asynchronous input property requirement on the class invariant by separating the assumption and guarantee parts of the invariant.

The second question we set out to answer was:

How suited are Maude and rewriting logic to implementing Hoare logic?

The main challenges were to parse Creol programs, to represent Creol statements and assertions, to compute the weakest liberal precondition (WLP) of Creol statements, and to normalize, simplify, and pretty-print assertions, using Maude as the implementation language.

For a term-rewriting system, Maude performed remarkably well at parsing Creol. Maude's sort hierarchies, mixfix operators, and attributes were powerful enough to represent Creol statements and assertions in an intuitive manner, except in a few cases where Maude's syntactic idiosyncrasies must be accommodated. Although Maude offers no direct substitute for the empty production rules of a context-free grammar, we could easily transform Creol's grammar to avoid them. For a few Creol constructs, trial and error was needed to avoid parsing conflicts in Maude and pass the "preregularity" checks. A context-free grammar front-end to Maude would have been a most welcome feature.

The equational subset of rewriting logic made it straightforward to implement the WLP formalism. Thanks to their functional nature, the abstract WLP definitions could be implemented directly in Maude. Using pattern matching and conditional equations, it was also easy to identify special cases and produce optimized WLPs for these.

While most of the assertion analyzer relies on equations, rewrite rules applied at the metalevel proved extremely useful for normalizing and simplifying the assertions generated by the tool. Furthermore, Maude's support for term formatting made it possible to pretty-print the verification conditions generated by the assertion analyzer, a near-necessity if humans are to understand the tool's output.

Restrictions on the concrete tokens that can be used in Maude to generate verification conditions prevented us from generating assertions in a format compatible with off-the-shelf theorem provers. However, this remains a minor issue that can easily be worked around using conversion programs. Such programs can easily be written using a lexical analyzer generator or a scripting language that supports regular expressions.

As expected, the conciseness, clarity, and expressiveness of Maude's syntax manifested themselves in the implementation of the assertion analyzer. Because the `wlp` function implemented in Maude is virtually identical to the mathematical WLP given in Chapter 5, it is likely to be error-free. The prospect of code sharing between the various Creol tools, which was another reason for choosing Maude as the implementation language, materialized to the extent that about 60 percent of the assertion analyzer's code is also used by the interpreters. And since Maude is highly optimized, the assertion analyzer is fairly fast, requiring a few seconds to

verify the classes presented in Chapter 7 on a standard PC running Linux. On the whole, Maude was a very good choice for implementing the tool.

Let us now turn to the third question:

> *To what extent do Creol's reference operational semantics and proof system enable program verification in practice?*

The assertion analyzer was expected to expose the strengths and weaknesses of Creol's reference semantics in the context of program verification. Most Creol statements, including assignment, have simple preconditions, but a few advanced constructs were more problematic.

In particular, the nondeterministic choice statement, which had never been axiomatized before, has a very complex precondition in the general case. Fortunately, by inspecting its branches statically, we could drastically optimize the WLP. Similar syntactic optimizations were possible for other statements, including some of the synthetic statements. All of this contributed to making the verification conditions produced by the assertion analyzer more manageable.

With appropriate simplifications and optimizations, it seems that we don't need to sacrifice completeness to achieve simplicity. With a suitable theorem prover and appropriate lemmas regarding the built-in functions, verification of Creol programs appears within reach. The remaining uncertainties concern the parts of the proof system that haven't been implemented, namely the general case of the nondeterministic merge statement and the composition of assume–guarantee specifications to prove a complete system correct.

Beyond answering the three questions posed in Section 1.2, the thesis lead to the development of an intermediate semantics that bridges the gap between Creol's traditional "closed system" operational semantics and the proof system. The resulting open system operational semantics defines the behavior of a single method execution seen in isolation, using a communication history to abstract away the environment. Previously, this role was assumed by SEQ, the augmented Creol subset from which the Hoare logic was initially derived, following de Boer and Pierik [dBP04]. The open system semantics helps prevent inconsistencies between the reference closed system semantics and the Hoare logic, which would translate into unsoundness or incompleteness of the proof method.

The table below summarizes the main characteristics of the four semantics presented in this thesis, along three axes:

Semantic	**State-Based or Predicate-Based?**	**Operational or Syntax-Driven?**	**Global or Local?**
Closed System	State-Based	Operational	Global
Open System	State-Based	Operational	Local
SEQ Encoding	State-Based	Syntax-Driven	Local
Hoare Logic	Predicate-Based	Syntax-Driven	Local

The first axis is "state-based versus predicate-based". A state-based semantics describes how a state is transformed into a new state when executing a statement,

whereas a predicate-based semantics describes how an assertion is transformed into another assertion by a statement.

The second axis is "operational versus syntax-driven". An operational semantics defines rules that are invoked following the control flow of the program of interest, whereas a syntax-driven semantics works directly on the program's source text. The operational/syntax-driven opposition coincides with the traditional semantic/syntactic divide.

The third axis is "global versus local". A global semantics considers a complete system in which all the components are known, whereas a local semantics focuses on one process and abstracts away the other processes and objects by using a communication history.

The table suggests the following general four-step approach to the development of a Hoare logic from a traditional closed system semantics, using the open system semantics and the SEQ encoding as stepping stones:

1. Specify an open system semantics that abstracts away the environment using a history, reusing the parts of the closed system semantics that cover the language's sequential subset.
2. Develop a Hoare logic for the language's sequential subset.
3. Reformulate the open semantics as an encoding in terms of the language's sequential subset augmented with random assignment.
4. Mechanically derive a Hoare logic from this encoding.

Because the open system semantics is expressed in rewriting logic, it is straightforward to detect inconsistencies with the reference closed system semantics by comparing the rewrite rules—a sketch of the proof can be found in Blanchette and Owe [BO08]. From the new semantics, we can easily derive a history-based Hoare logic that is sound and complete by construction. This approach can be adapted to other object-oriented languages where communication is done by messages passing, method interaction, or both.

8.2 Future Work

The work on the proof system left some questions only partially answered and suggests many directions for future research. An obvious gap is that we gave no rule that handles the general case of the nondeterministic merge statement. One option would be to eliminate the statement from Creol, arguing that its noncompositional semantics violates the spirit of the language—but this would reduce the language's expressive power [Cho05]. Failing that, we could impose syntactic restrictions on its use to ease its axiomatization. A third option would be to develop a proof rule that performs an interference-freedom test on the statement's branches.

The other main limitation of the proof system is that it is concerned with partial correctness only and thus cannot be used to establish the absence of run-time errors, infinite loops, and deadlocks. To catch these errors, we would need to formulate total correctness axioms and proof rules for the Creol-specific statements.

In addition, many improvements suggest themselves for the assertion analyzer tool built on top of the proof system. We implemented the verification of a single class, but didn't try to automate the process of verifying global properties about an entire system from the assume–guarantee specifications of the classes that compose it. Other possible improvements include explicitly supporting mythical statements (statements that have no effect on the program's behavior but that are used for reasoning purposes [Dah92]), and allowing the user to supply equations for defining auxiliary functions from which the assertion analyzer would automatically generate simplification rules, using Maude's support for metaprogramming.

Because virtual method calls might bind differently in a subclass than in a base class, verifying a subclass currently involves (re)verifying the methods inherited from the superclass. Dovland et al. recently proposed a calculus that enables proof reuse [DJOS08]. Implementing this calculus, either in the assertion analyzer or as a separate tool, would be of great practical interest.

A program that verifies another program is itself a prime candidate for formal verification. For the assertion analyzer, this would involve proving that the Creol WLPs are sound with respect to the open system operational semantics, that the open system operational semantics is a safe approximation of the closed system operational semantics, that the built-in simplification rules are logically correct, and that the rest of the Maude code is correct. The functional nature of the Maude code means that we can perform the proofs by induction—for example, using Maude's Inductive Theorem Prover (ITP). The other proofs could be performed using an arbitrary theorem prover, following the lines of de Roever et al. [dRdB$^+$01].

To make the assertion analyzer more useful in practice, it would be desirable to integrate the tool with ITP [SM07] or Bjarne Holen's Maude-based automated theorem prover [Hol05] to discharge proof obligations automatically. Alternatively, it could be interesting to try implementing the proof system in an interactive theorem prover such as Isabelle/HOL [NPW02] or PVS [CORSS95] or in a program analysis framework such as KeY [BHS07] or ASF+SDF [dH03], and compare the result with the Maude implementation we developed here.

A more modest endeavor would be to develop libraries of simplification rules for specific application areas, corresponding to the proof rules or lemmas used by interactive theorem provers. For simple assume–guarantee specifications, where verification conditions are manageable and a limited number of lemmas are needed, the assertion analyzer could be of practical use. However, until it is tried on larger programs, it is not clear to what extent the Creol proof system is usable in practice. Creol's simple semantics and compositional proof system inspire hope, but much still remains to be done.

Appendix A

User's Guide to the Assertion Analyzer and the Interpreters

This appendix explains how to use the Creol tools that were developed in this thesis and whose source code is included in Appendix B.

- The *assertion analyzer* verifies that the implementation of a Creol class respects the class's assume–guarantee specification.
- The *interpreter for closed systems* executes a self-contained Creol program representing a complete distributed system.
- The *interpreter for open systems* executes a single Creol process considered in isolation.

The appendix is organized as follows: Section A.1 presents a short example Creol program and shows how to use the three tools on it. Sections A.2 and A.3 give the concrete syntax of Creol programs and assertions. Section A.4 gives the syntax of the bootstrapping commands needed for the tools. Section A.5 explains how to supply custom simplification rules to the assertion analyzer. Section A.6 shows how to define custom data types and functions for use with the tools. Finally, Section A.7 lists the known bugs and limitations of the tools.

A.1 Getting Started

Before you can run the Creol tools, you first need to install Maude 2 on your computer. Maude is free (open source) software; executables can be downloaded from `http://maude.cs.uiuc.edu/` (for Linux, Mac OS X, and FreeBSD) and `http://moment.dsic.upv.es/mfw/` (for Windows). The Creol tools have been tested with Maude 2.3, but other 2.*x* versions are likely to work just as well.

In addition, you must install the following files together in a directory:

```
creol-program.maude
creol-assertion-utilities.maude
creol-assertion-analyzer.maude
```

```
creol-interpreter-core.maude
creol-closed-interpreter.maude
creol-open-interpreter.maude
creol-tools.maude
```

The first six files contain the Maude modules that implement the tools. The seventh file is provided for convenience; it simply loads the other six. The contents of these files are listed in Appendix B.

A.1.1 Specifying the Creol Program

Once the software is installed, the first step to use any of the Creol tools is to specify the Creol program of interest as a Maude term. The Maude-compatible Creol syntax is provided by a functional module called `CREOL-PROGRAM` defined in `creol-program.maude`. The concrete syntax mimics the abstract Creol syntax introduced in Section 4.2.

Consider the following Creol program in which two processes synchronize using a binary semaphore (a "mutex"), specified using the abstract syntax:

```
interface Mutex
begin
with any:
    op lock
    op unlock
end

class SimpleMutex
    implements Mutex
begin
    var locked : bool
with any:
    op lock is
        await ¬locked;
        locked := true
    op unlock is
        locked := false
    guar locked ⇔ H/out_self ew
                 [∗←self.lock(∗)]
end
```

```
class Process (mutex : Mutex)
begin
    var n : int
    op run is
        while true do
            mutex.lock();
            n := n + 1;
            mutex.unlock()
        od
end

class Main
begin
    op run is
        var mutex : Mutex,
            p1 : any, p2 : any;
        mutex := new SimpleMutex;
        p1 := new Process(mutex);
        p2 := new Process(mutex)
end
```

For our convenience, we normally define the program in a Maude file of its own. Here is the content of `mutex-program.maude`, which specifies the mutex program:

```
load creol-program.maude .

fmod MUTEX-PROGRAM is
  including CREOL-PROGRAM .

  op prog : -> Config .
```

```
  eq prog =
     interface 'Mutex
     begin
     with any :
        op 'lock
        op 'unlock
     end

     class 'SimpleMutex
        implements 'Mutex
     begin
        var 'locked : bool

     with any :
        op 'lock is
           await ! 'locked ;
           'locked := true

        op 'unlock is
           'locked := false

        guar 'locked <==> ~H~ / out[self] ew [* <- self . 'lock[*]]
     end

     class 'Process ['mutex : 'Mutex]
     begin
        var 'n : int

        op 'run is
           while true do
              'mutex . 'lock[] ;
              'n := 'n plus 1 ;
              'mutex . 'unlock[]
           od
     end

     class 'Main
     begin
        op 'run is
           var 'mutex : 'Mutex, 'p1 : any, 'p2 : any ;
           'mutex := new 'SimpleMutex ;
           'p1 := new 'Process['mutex] ;
           'p2 := new 'Process['mutex]
     end
     .
endfm
```

The `MUTEX-PROGRAM` module defines a constant `prog` of sort `Config` that expands to a Maude term representing the mutex program. The rest of the file is simply boilerplate for Maude. To check that Maude can parse the program, load the module in Maude and type the following command:

```
red prog .
```

For this specific example, the result should have sort `Config` and consist of exactly one < *i* : `Interface` | ... > term and three < *c* : `Class` | ... > terms, corresponding to the interface and class declarations found in the program.

Before we see how to invoke the individual tools, a couple of warnings are in order. First, the tools do not perform any static program checking beyond the basic parsability requirements enforced by Maude. To catch type errors, undeclared variables, and other similar errors, a separate tool, such as the one developed by Fjeld [Fje05], must be used. Second, because Maude gives terms under evaluation the benefit of the doubt, many syntax errors, such as typing `not` *B* instead of `!` *B* or `if` *B* `then` e_1 `else` e_2 `fi` instead of `if` *B* `th` e_1 `el` e_2 `fi`, will often slip through and lead to an error term of kind `[Config]`. The easiest way to diagnose such errors is to enable Maude's term coloring feature using the command

```
set print color on .
```

When term coloring is enabled, the standard syntax highlighting is replaced by a color code that identifies error terms. Symbols at the root of the error are displayed in red or magenta; other affected symbols are rendered in blue or cyan. To avoid error terms in the first place, refer to Sections A.2 and A.3 whenever you are uncertain of the concrete syntax of a Creol construct.

A.1.2 Running the Assertion Analyzer

The Creol assertion analyzer is a tool that attempts to verify whether the class's implementation respects its assume–guarantee specification. The tool is available through a system module called `CREOL-ASSERTION-ANALYZER` found in the file `creol-assertion-analyzer.maude`, which requires `creol-assertion-utilities.maude` to be already loaded. The implementation of the tool is described in Chapter 6.

To run the assertion analyzer on the mutex program defined in Section A.1.1, define a system module that includes both the mutex program module and the assertion analyzer modules as follows:

```
load mutex-program.maude .
load creol-assertion-utilities.maude .
load creol-assertion-analyzer.maude .

mod MUTEX-VERIFICATION is
  including CREOL-ASSERTION-ANALYZER .
  including MUTEX-PROGRAM .

  op init : -> GlobalConfig .

  eq init =
     {
        prog
        verify class 'SimpleMutex
     } .
endm
```

The new module defines an `init` constant that can be used to verify the `'SimpleMutex` class. The `init` constant, of sort `GlobalConfig`, is defined to expand to the

Creol program and a `verify class` command that specifies which class should be verified. (Section A.4.1 gives the complete syntax of `verify class`.) To run the assertion analyzer, use the `rew` command on `init`:

```
rew init .
```

This produces the following output:

```
result GlobalConfig: {
Verification of class 'SimpleMutex

Initialization code :
    Establishes the guarantee

Method 'lock of 'SimpleMutex :
    Maintains the guarantee

Method 'unlock of 'SimpleMutex :
    Maintains the guarantee
}
```

The report contains one judgment for the class's implicit initialization code and one judgment per method (including inherited methods). The initialization code corresponds to the code executed to set up an object, including the call to `'run` (if that method is provided).

For the methods, the judgments are of the following forms:

i. `Maintains the guarantee iff` $\hat{Q}$ `holds`

   The assertion $\hat{Q}$ is a proof obligation that we must carry out to verify that the method maintains the guarantee.

ii. `Maintains the guarantee`

   The verification succeeded; the method's body always maintains the guarantee. This is the same as form i with $\hat{Q} \equiv$ `true`.

iii. `Breaks the guarantee`

   The verification failed; the method's body sometimes breaks the guarantee. This is the same as form i with $\hat{Q}$ taken to be an invalid assertion (for example, $\hat{Q} \equiv$ `false`).

iv. `Maintains the guarantee if` $\hat{Q}$ `holds`

   This is a weaker version of form i, with `if` instead of `iff`. This judgment can occur only for methods that contain `prove`, `while`, or `|||` statements, for which proof system is incomplete. The invalidity of $\hat{Q}$ means that at least one of the following is true:

   a. The method contains a `prove` statement with an invalid assertion.

   b. The method contains a `while` loop with an invalid or too weak invariant.

   c. The method contains a `|||` statement with a nested `await` statement.

   d. The method body sometimes breaks the guarantee.

v. `Don't know`

   This is the same as form iv with $\hat{Q}$ taken to be an invalid assertion (for example, $\hat{Q} \equiv$ `false`). This judgment can occur only for methods that contain `prove`, `while`, or `|||` statements, for the reasons given above.

In judgments of forms i and iv, the formula $\hat{Q} \in$ `PrettyAssn` is a pretty-printed assertion expressed using the syntax defined in Section A.3.2. To increase the output's readabibility, the judgments are colored according to a traffic light scheme: *green* for positive judgments (form ii), *yellow* for inconclusive judgments (forms i, iv, and v), and *red* for negative judgments (form iii).

For the initialization code, the wording of the judgments is adapted slightly, but the meaning is essentially the same:

i. `Establishes the guarantee iff` $\hat{Q}$ `holds`

ii. `Establishes the guarantee`

iii. `Fails to establish the guarantee`

iv. `Establishes the guarantee if` $\hat{Q}$ `holds`

v. `Don't know`

As the class guarantee, the assertion analyzer and the open system interpreter use the conjunction of the `guar` clauses specified in the class of interest and in its superinterfaces, without projecting them onto their respective alphabets. A similar computation is made for the class assumption. In addition, the tool takes for granted that the assumption is insensitive to additional output, and that the guarantee is insensitive to additional input that respects the assumption. In practice, the above constraints can easily be met by respecting the following syntactic rules:

1. Define the assumption by case on input events that belong to the interface or class's alphabet, and let all other events fall through.

2. Define the guarantee by case on output or internal control events that belong to the interface or class's alphabet, and let all other events fall through.

For the mutex program, the assertion analyzer was able to verify the class guarantee with no intervention. For more complex examples, such as those studied in Chapter 7 and specified in Appendix C, the output would have contained proof obligations $\hat{Q}$ that must be carried out by hand or through simplification rules. Section A.5 explains how to specify custom simplification rules.

A.1.3 Running the Interpreter for Closed Systems

The Creol interpreter for closed systems is a tool that executes a self-contained Creol program. It lets us simulate a closed distributed system, in which all the classes are known in advance. This enables us to test a program before we subject it to formal verification using the assertion analyzer. The tool is provided by a system module called `CREOL-INTERPRETER-FOR-CLOSED-SYSTEMS` defined in the file `creol-closed-interpreter.maude`, which relies on `creol-interpreter-core.maude`.

The interpreter is based on the reference operational semantics of Section 4.3; its implementation is briefly described in Section 4.5.

The mutex program introduced in Section A.1.1 is an example of a closed system, since we have the code for all the classes involved in the system (`'SimpleMutex`, `'Process`, and `'Main`). To execute the program using the interpreter for closed systems, first define a system module that includes both the mutex program module and the closed system interpreter module:

```
load mutex-program.maude .
load creol-interpreter-core.maude .
load creol-closed-interpreter.maude .

mod MUTEX-CLOSED-SYSTEM is
  including CREOL-INTERPRETER-FOR-CLOSED-SYSTEMS .
  including MUTEX-PROGRAM .

  op init : -> GlobalConfig .

  eq init =
     {
        prog
        bootstrap system 'Main
     } .
endm
```

The new module defines an `init` constant that can be used to launch the system. The `init` constant, of sort `GlobalConfig`, is defined to expand to the Creol program and a `bootstrap system` command that specifies the class to instantiate at startup. (Section A.4.2 gives the complete syntax of `bootstrap system`.)

To execute the Creol program, you can use Maude's built-in evaluation strategies. Each rewrite step in Maude corresponds to one step in the operational semantics of Creol. For example, the following command will display the system's configuration after executing 500 rewrite steps:

```
frew [500] init .
```

The output is a global system configuration consisting of `< i : Interface | ... >`, `< c : Class | ... >`, and `< o : c | ... >` terms, as well as `Invoke` and `Reply` messages. Since Creol is highly nondeterministic, an execution given by `rew` or `frew` is generally only one among many possible. To perform a systematic search of the states reachable from the initial state, use Maude's `search` command.

A.1.4 Running the Interpreter for Open Systems

The Creol interpreter for open systems is a tool that lets us simulate the execution of a specific process, abstracting the other processes and objects in the environment. The tool is provided by a system module called `CREOL-INTERPRETER-FOR-OPEN-SYSTEMS` defined in the file `creol-open-interpreter.maude`, which requires both `creol-assertion-utilities.maude` and `creol-interpreter-core.maude` to be loaded. The interpreter is based on the alternative operational semantics given in Section 4.4; its implementation is briefly described in Section 4.5.

To execute processes based on the mutex program presented in Section A.1.1 using the interpreter for open systems, start by defining a system module that includes both the mutex program module and the open system interpreter module:

```
load mutex-program.maude .
load creol-assertion-utilities.maude .
load creol-interpreter-core.maude .
load creol-open-interpreter.maude .

mod MUTEX-OPEN-SYSTEM is
  including CREOL-INTERPRETER-FOR-OPEN-SYSTEMS .
  including MUTEX-PROGRAM .

  op init : -> GlobalConfig .

  eq init =
     {
        prog
        bootstrap object root # 0 := new 'SimpleMutex
                  with parent root
     } .
endm
```

The new module defines an `init` constant that can be used to execute the `'SimpleMutex` initialization code. The `init` constant, of sort `GlobalConfig`, is defined to expand to the Creol program and a `bootstrap object` command that specifies which object's initialization code should be executed. To execute the process, you can use the `rew` or `frew` command. For example:

```
frew init .
```

In this case, the output would contain the following term:

```
< root # 0 : 'SimpleMutex | Pr: emptyStmt, LVar: emptyState,
                             Att: ['locked @ 'SimpleMutex |-> false]
                                  [self |-> root # 0]
                                  [~H~ |-> [root -> root # 0
                                      . new 'SimpleMutex[epsilon]] ^^
                                           [root # 0 . initialized]],
                             MsgQ: emptyMSet, Asum: true,
                             Guar: locked @ 'SimpleMutex : bool <==>
                                   ~H~ : history / out[self : any] ew
                                   [* <- self : any . 'lock[*]]
                             ROAtt: self : 'SimpleMutex >
```

Or, using the notation of Section 4.4:

$$\begin{aligned}\langle\, &\mathbf{root}\#0 : \mathit{SimpleMutex} \mid \text{Pr: } \epsilon,\ \text{LVar: } \emptyset,\\ &\text{Att: } [\mathit{locked@SimpleMutex} \mapsto \mathbf{false}]\\ &\quad [\mathbf{self} \mapsto \mathbf{root}\#0]\\ &\quad [\mathcal{H} \mapsto [\mathbf{root} \rightarrow \mathbf{root}\#0.\mathbf{new}\ \mathit{SimpleMutex}()]\ ^\frown\\ &\qquad [\mathbf{root}\#0.\mathbf{initialized}]],\\ &\text{MsgQ: } \emptyset,\ \text{Asum: } \mathbf{true},\\ &\text{Guar: } \mathit{locked@SimpleMutex} : \mathbf{bool} \Leftrightarrow\\ &\quad \mathcal{H}{:}\mathbf{history}/\mathit{out}_{\mathbf{self}}\ \mathbf{ew}\ [* \leftarrow \mathbf{self}{:}\mathbf{any}.\mathit{lock}()],\\ &\text{ROAtt: } \mathbf{self} : \mathit{SimpleMutex}\,\rangle\end{aligned}$$

This term corresponds to one possible state of the object after executing its initialization code. By replacing the `bootstrap object` command by a `bootstrap method` command, you can follow a process that starts as a result of an asynchronous method invocation.

For example, the following command launches a process associated with a call to the `'lock` method of `'SimpleMutex`, at a point when the `'SimpleMutex` object's `'locked` attribute is true:

```
bootstrap method root # 0 . 'lock[epsilon]
          with class 'SimpleMutex,
               caller root # 1,
               label 1,
               history [root -> root # 0
                        . new 'SimpleMutex[epsilon]] ^^
                       [root # 0 . initialized] ^^
                       [1 % root # 1 -> root # 0 . 'lock[epsilon]],
               attributes 'locked @ 'SimpleMutex := true
```

Some of the rewrite rules that implement the interpreter for open systems rely on the presence of user-supplied "random" data in the form of `random data` commands. Thus, to complete the execution of the process initiated by the preceding `bootstrap method` command, we would need to supply a random attribute state in which `'locked` is false and the history variable `~H~` ($\mathcal{H}$) is extended:

```
random data [self |-> root # 0]
            ['locked @ 'SimpleMutex |-> false]
            [~H~ |-> [root -> root # 0
                      . new 'SimpleMutex[epsilon]] ^^
                     [root # 0 . initialized] ^^
                     [1 % root # 2 -> root # 0 . 'lock[epsilon]] ^^
                     [1 % root # 2 <- root # 0
                      . 'lock[epsilon ; epsilon]] ^^
                     [1 % root # 1 -> root # 0 . 'lock[epsilon]] ^^
                     [root # 0 . release] ^^
                     [2 % root # 2 -> root # 0
                      . 'unlock[epsilon]] ^^
                     [2 % root # 2 <- root # 0
                      . 'unlock[epsilon ; epsilon]]]
```

In Section 4.4, we modeled the nondeterministic behavior of the environment by introducing variables on the right-hand side of rewrite rules, which isn't supported by Maude's built-in execution strategies. To work around this, the Maude implementation provides altered versions of the rewrite rules that take user-supplied random data (histories and states) provided along with the Creol program.

Section A.4 gives the complete syntax of the `bootstrap object`, `bootstrap method`, and `random data` commands. If the instantiated class's assumption or guarantee involves a custom function, that function must be defined using the syntax described in Section A.6.

In its current state, the interpreter for open systems is mostly a curiosity. An improvement that would make it more useful would be to generate random data automatically, alleviating the need for `random data` commands.

A.2 Programming Language Syntax

This section describes the concrete syntax used to represent Creol programs in Maude. This syntax follows the conventions of Section 4.2 as closely as possible. For an informal description of the semantics, see Section 4.2. For a formal operational semantic, see Section 4.3.

In this section and the following two sections, tall square brackets ($\big[\,\big]$) denote optional clauses; they should not be confused with monospace square brackets (`[]`), which are part of the concrete Creol syntax.

A.2.1 Basic Syntactic Entities

The `Bool` sort is defined in the Maude prelude and has two constructor terms: `true` and `false`. The `Int` sort is also defined in the Maude prelude; it consists of terms of the form $\big[\texttt{-}\big]d_1 \ldots d_k$, where $d_i \in \{\texttt{0}, \texttt{1}, \ldots, \texttt{9}\}$.

The `Id` sort provides quoted identifiers such as `'x` and `'hungryCat` as well as the special identifiers `self`, `caller`, `label`, and `~H~` ($\mathcal{H}$). The `Id` sort also contains terms of the form x `$` k, with $x \in$ `Id` and $k \in$ `Int`; such terms are reserved by the assertion analyzer, which uses them for bound variables in the output.

The `Type` sort corresponds to Creol types. Any `Id` term can be used as a type. In addition, `bool`, `int`, and `any` are built-in types that may occur in Creol programs, whereas `event` and `history` can be used in assertions.

The `TypedId` sort reflects the syntax of Creol variable declarations. Terms of this sort have the form x or x `:` t, with $x \in$ `Id` and $t \in$ `Type`. For example, `'full : bool` is a term of sort `TypedId`. The `QualifiedId` sort provides terms of the form x or x `@` c, where $x \in$ `Id` is the name of a Creol attribute or method and $c \in$ `Id` is the name of a Creol class.

For a given sort X, the sort X`List` consists of comma-separated lists of terms from X. The empty list is represented by the constant `epsilon`. In many contexts, `epsilon` can be omitted; this is indicated by a "$_?$" subscript attached to a metavariable of sort X`List` (for example, $\bar{e}_?$ with $\bar{e} \in$ `ExpList`). When referencing a sort defined in this user's guide, we will often indicate the section in which it is defined, like this: $\texttt{Id}_{\text{A.2.1}}$. For X`List` sorts, we point to the section where X is defined.

A.2.2 Interface Declarations

Creol interfaces are declared using the following syntax:

```
interface ı [[X̄]]
  [inherits j̄]
begin
with k :
   op m₁[[[in X̄₁] [out Ȳ₁]]]
      ⋮
```

$$
\begin{aligned}
\imath, m_i &\in \texttt{Id}_{\text{A.2.1}} \\
k &\in \texttt{Type}_{\text{A.2.1}} \\
\bar{X}, \bar{X}_i, \bar{Y}_i &\in \texttt{TypedIdList}_{\text{A.2.1}} \\
\bar{\jmath} &\in \texttt{SuperList}_{\text{A.2.2}} \\
P, Q &\in \texttt{Assn}_{\text{A.3.1}}
\end{aligned}
$$

```
   op m_n[[in X̄_n] [out Ȳ_n]]
  [asum P]
  [guar Q]
end
```

The `Super` sort provides terms of the form x or $x[\bar{e}]$, with $x \in \texttt{Id}$ and $\bar{e} \in \texttt{ExpList}$. The `Exp` sort is a supersort for $\texttt{AExp}_{\text{A.2.5}}$, $\texttt{BExp}_{\text{A.2.6}}$, and $\texttt{OExp}_{\text{A.2.7}}$.

A.2.3 Class Declarations

Creol classes are declared using the following syntax:

```
class c [[X̄]]
  [implements ī]
  [contracts j̄]
  [inherits q̄]
begin
  [var W̄]
  [G_0]
with k_1 :
   G_1
   ⋮
with k_m :
   G_m
  [asum P]
  [guar Q]
end
```

$$
\begin{aligned}
c &\in \texttt{Id}_{\text{A.2.1}} \\
k_i &\in \texttt{Type}_{\text{A.2.1}} \\
\bar{W}, \bar{X} &\in \texttt{TypedIdList}_{\text{A.2.1}} \\
\bar{\imath}, \bar{\jmath}, \bar{q} &\in \texttt{SuperList}_{\text{A.2.2}} \\
G_i &\in \texttt{MtdDeclGroup}_{\text{A.2.3}} \\
P, Q &\in \texttt{Assn}_{\text{A.3.1}}
\end{aligned}
$$

The `MtdDeclGroup` sort consists of sequences of terms of the form

```
op m[[[in X̄] [out Ȳ]]] is
  [var V̄ ;]
   S
```

$$
\begin{aligned}
m &\in \texttt{Id}_{\text{A.2.1}} \\
\bar{V}, \bar{X}, \bar{Y} &\in \texttt{TypedIdList}_{\text{A.2.1}} \\
S &\in \texttt{Stmt}_{\text{A.2.4}}
\end{aligned}
$$

A.2.4 Statements

The `Stmt` sort of Creol statements consists of terms with these syntaxes:

```
skip
abort
prove P
z̄ := ē
z := new c[[ē]]
[L] ! O . m[ē_?]
[L] ! m [@ c][ē_?]
L ?[z̄_?]
O . m[[ē_? ; z̄_?]]
m[[ē_? ; z̄_?]]
```

$$
\begin{aligned}
c, L, m &\in \texttt{Id}_{\text{A.2.1}} \\
z &\in \texttt{QualifiedId}_{\text{A.2.1}} \\
\bar{z} &\in \texttt{QualifiedIdList}_{\text{A.2.1}} \\
\bar{e} &\in \texttt{ExpList}_{\text{A.2.2}} \\
S, S_i &\in \texttt{Stmt} \\
B &\in \texttt{BExp}_{\text{A.2.6}} \\
O &\in \texttt{OExp}_{\text{A.2.7}} \\
g &\in \texttt{Guard}_{\text{A.2.8}} \\
I, P &\in \texttt{Assn}_{\text{A.3.1}}
\end{aligned}
$$

```
m @ c[[ē? ; z̄?]]
await g
await [g &&&] L ?[z̄?]
await [g &&&] O . m[[ē? ; z̄?]]
await [g &&&] m [@ c][[ē? ; z̄?]]
if B th S1 [el S2] fi
[inv I] while B do S od
S1 ; S2
S1 [] S2                (□)
S1 ||| ··· ||| Sn       (|||)
[S]
```

Section A.2.9 summarizes the semantic pecularities of the `await` and `|||` statements compared with the Creol interpreter in use at the University of Oslo.

A.2.5 Arithmetic Expressions

The sort `AExp` of arithmetic expressions consists of terms with these syntaxes:

```
n
z [: int]
f[ē]
#[H]
plus A                  (+)
minus A                 (−)
A1 times A2             (∗)
A1 div A2               (/)
A1 plus A2              (+)
A1 minus A2             (−)
if B th A1 el A2 fi
[A]
```

$$
\begin{array}{rl}
n & \in \texttt{Int}_{\text{A.2.1}} \\
f & \in \texttt{Id}_{\text{A.2.1}} \\
z & \in \texttt{QualifiedId}_{\text{A.2.1}} \\
\bar{e} & \in \texttt{ExpList}_{\text{A.2.2}} \\
A, A_1, A_2 & \in \texttt{AExp} \\
B & \in \texttt{BExp}_{\text{A.2.6}} \\
H & \in \texttt{HistoryExp}_{\text{A.3.5}}
\end{array}
$$

A.2.6 Boolean Expressions

The sort `BExp` of Boolean expressions consists of terms with these syntaxes:

```
b
z [: bool]
f[ē]
A1 eq A2                (=)
A1 ne A2                (≠)
A1 lt A2                (<)
A1 gt A2                (>)
A1 le A2                (≤)
A1 ge A2                (≥)
O1 eq O2                (=)
O1 ne O2                (≠)
! B                     (¬)
B1 && B2                (∧)
```

$$
\begin{array}{rl}
b & \in \texttt{Bool}_{\text{A.2.1}} \\
f & \in \texttt{Id}_{\text{A.2.1}} \\
z & \in \texttt{QualifiedId}_{\text{A.2.1}} \\
\bar{e} & \in \texttt{ExpList}_{\text{A.2.2}} \\
A_1, A_2 & \in \texttt{AExp}_{\text{A.2.5}} \\
B, B_1, B_2 & \in \texttt{BExp} \\
O_1, O_2 & \in \texttt{OExp}_{\text{A.2.7}}
\end{array}
$$

```
B1 || B2                (∨)
if B th B1 el B2 fi
[B]
```

A.2.7 Object Expressions

The sort `OExp` of object expressions consists of terms with these syntaxes:

```
null
z [: any]
z : j
f[ē]
if B th O1 el O2 fi
[O]
```

$f, j \in \texttt{Id}_{\text{A.2.1}}$
$z \in \texttt{QualifiedId}_{\text{A.2.1}}$
$\bar{e} \in \texttt{ExpList}_{\text{A.2.2}}$
$B \in \texttt{BExp}_{\text{A.2.6}}$
$O, O_1, O_2 \in \texttt{OExp}$

A.2.8 Guards

The sort `Guard` of `await` statement guards consists of terms with these syntaxes:

```
B
L ?
wait
g1 &&& g2
```

$L \in \texttt{Id}_{\text{A.2.1}}$
$B \in \texttt{BExp}_{\text{A.2.6}}$
$g_1, g_2 \in \texttt{Guard}$

A.2.9 Semantic Peculiarities

The Creol dialect supported by the tools broadly corresponds to the language understood by the interpreter in use at the University of Oslo. There are, however, three subtle differences that must be kept in mind. The first difference concerns `await` and is illustrated by this program:

```
class 'Main
begin
   var 'x : int

   op 'run is
      'l1 ! 'doNothing[] ;
      await 'l1 ? ;
      'l2 ! 'incrementX[] ;
      [await 'l1 ? &&& 'x ne 0 [] 'x := 2] ;
      await 'l2 ?

   op 'doNothing is
      skip

   op 'incrementX is
      'x := 'x plus 1
end

bootstrap system 'Main
```

In this thesis, we have taken the view that complex guards are evaluated atomically. As a result, when control reaches the `[]` statement in `'run`, the left branch is not ready (since `'x ne 0` is false) and the right branch is chosen. The attribute `'x` equals 3 at the end of the program's execution. In contrast, the standard Creol interpreter treats

```
[await 'l1 ? &&& 'x ne 0 [] 'x := 2]
```

the same as

```
[await 'l1 ? ; await 'x ne 0 [] 'x := 2]
```

with the consequence that the left branch is always ready (since `'l1 ?` is true). Either branch can be chosen, and `'x` equals 1 or 3 at the end.

The second difference concerns `|||` statements. As pointed out in Section 4.3, we adopted a commutative definition for `|||`, instead of the "first branch's preference" definition advocated by Husby [Hus05].

The third difference concerns nested `|||` statements. In this thesis, we consider the complex statement `[`S_1 `|||` S_2`]` `|||` S_3 in a strict compositional sense ("merge S_1 and S_2, then merge the result with S_3") and distinguish it from the three-way merge S_1 `|||` S_2 `|||` S_3. Thus, from a state in which `'x` equals 0, the following statement will always lead to a state in which `'x` equals 3:

```
['x := 1 ||| await 'x ne 0 ; 'x := 2]
||| await 'x ne 0 ; 'x := 3
```

In contrast, the standard Creol interpreter would treat the above statement as

```
'x := 1
||| await 'x ne 0 ; 'x := 2
||| await 'x ne 0 ; 'x := 3
```

and `'x` would equal either 2 or 3 at the end.

A.3 Assertion Language Syntax

This section describes the syntax of first-order assertions. These assertions may appear in a Creol interface or class's assume–guarantee specification, as inline assertions in the body of a method, or as loop invariants. In addition, assertions figure in some of the judgments produced by the assertion analyzer.

A.3.1 Plain Assertions

The sort `Assn` of assertions admits terms with these syntaxes:

B		$\bar{X} \in$	`TypedIdList`$_{\text{A.2.1}}$
H_1 `eq` H_2	$(=)$	$B \in$	`BExp`$_{\text{A.2.6}}$
H_1 `ne` H_2	$(\neq)$	$P, P_1, P_2 \in$	`Assn`
Y^* `in` H		$Y^* \in$	`EventPatExp`$_{\text{A.3.4}}$
H `bw` H^*		$H, H_1, H_2 \in$	`HistoryExp`$_{\text{A.3.5}}$
H `ew` H^*		$H^* \in$	`HistoryPatExp`$_{\text{A.3.6}}$

H^* `pr` H	($\preceq$)
`!` P	($\neg$)
P_1 `&&` P_2	($\wedge$)
P_1 `||` P_2	($\vee$)
P_1 `==>` P_2	($\Rightarrow$)
P_1 `<==>` P_2	($\Leftrightarrow$)
`forall` $\bar{X}$ `.` P	($\forall$)
`exists` $\bar{X}$ `.` P	($\exists$)
`if` P `th` P_1 `el` P_2 `fi`	
`[`P`]`	

Assertions may refer to program variables directly as x or x `@` c. They may also refer to the mythical variable `~H~` ($\mathcal{H}$). There is no dedicated syntax for user-defined logical variables; any variable name that does not occur in the program's text can be used as a logical variable. Names of the form x `$` k are reserved by the assertion analyzer for its own logical variables.

Occurrences of variables in assertions can be typed (z `:` t) or untyped (z). As a first step in their processing, the assertion analyzer and the interpreter for open systems supply the missing typing annotations for the program variables and the special variables `self`, `caller`, `label`, and `~H~` ($\mathcal{H}$). Thus, in the assertions that occur in a program's text, it is sufficient to provide types for logical variables.

A.3.2 Pretty-Printed Assertions

The sort `PrettyAssn` of pretty-printed assertions admits terms with the following syntaxes, listed in decreasing order of precedence:

P		$P \in \texttt{Assn}_{\text{A.3.1}}$
$\hat{P}_1$ `/\` $\hat{P}_2$	($\wedge$)	$\hat{P}, \hat{P}_1, \hat{P}_2 \in$ `PrettyAssn`
$\hat{P}_1$ `\/` $\hat{P}_2$	($\vee$)	
$\hat{P}_1$ **`===>`** $\hat{P}_2$	($\Longrightarrow$)	
$\hat{P}_1$ **`And`** $\hat{P}_2$	($\bigwedge$)	
`Forall` $\bar{X}$ `.` $\hat{P}$	($\forall$)	
`Exists` $\bar{X}$ `.` $\hat{P}$	($\exists$)	
$\hat{P}_1$ **`And:`** $\hat{P}_2$	($\bigwedge$)	

A.3.3 Event Expressions

The sort `EventExp` of event expressions consists of terms with these syntaxes:

x [`: event`]	$c, m, x \in \texttt{Id}_{\text{A.2.1}}$
`[`O `->` O' `. new` c`[`$\bar{e}$`]]`	$\bar{e}, \bar{e}' \in \texttt{ExpList}_{\text{A.2.2}}$
`[`A `%` O `->` O' `.` m [`@` c]`[`$\bar{e}$`]]`	$A \in \texttt{AExp}_{\text{A.2.5}}$
`[`A `%` O `<-` O' `.` m [`@` c]`[`$\bar{e}$ `;` $\bar{e}'$`]]`	$O, O' \in \texttt{OExp}_{\text{A.2.7}}$
`[`O `. initialized]`	
`[`O `. release]`	
`[`A `%` O `. reenter]`	

See Section 4.4 for a description of these event types.

A.3.4 Event Pattern Expressions

The sort `EventPatExp` of event pattern expressions consists of terms with the following syntaxes:

`Y`	$c, m \in \texttt{Id}_{\text{A.2.1}}$
`new`	$A \in \texttt{AExp}_{\text{A.2.5}}$
`invoke`	$O, O' \in \texttt{OExp}_{\text{A.2.7}}$
`reply`	$Y \in \texttt{EventExp}_{\text{A.3.3}}$
`initialized`	$Y^*, Y_1^*, Y_2^* \in \texttt{EventPatExp}$
`release`	
`reenter`	
`control`	
`[O -> *]`	
`[O <- *]`	
`[* -> O]`	
`[* <- O]`	
`[* -> O . new c[*]]`	
`[* -> O . m [@ c][*]]`	
`[* <- O . m [@ c][*]]`	
`[A % O -> O' . *]`	
`[A % O -> * . *]`	
`[A % O <- O' . *]`	
`[A % O <- * . *]`	
`[O . reenter]`	
`in[O]`	
`out[O]`	
`ctl[O]`	
`O`	
`~ Y*`	$(\blacksquare^{\mathrm{C}})$
Y_1^* `&` Y_2^*	$(\cap)$
Y_1^* `|` Y_2^*	$(\cup)$

Event patterns correspond to sets of events:

$$
\begin{aligned}
\texttt{new} &\triangleq \{\texttt{[O -> O' . new c[}\bar{e}\texttt{]]} \mid O, O', c, \bar{e}\} \\
\texttt{invoke} &\triangleq \{\texttt{[k \% O -> O' . m [@ c][}\bar{e}\texttt{]]} \mid k, O, O', m, c, \bar{e}\} \\
\texttt{reply} &\triangleq \{\texttt{[k \% O <- O' . m [@ c][}\bar{e}\texttt{ ; }\bar{e}'\texttt{]]} \mid k, O, O', m, c, \bar{e}, \bar{e}'\} \\
\texttt{initialized} &\triangleq \{\texttt{[O . initialized]} \mid O\} \\
\texttt{release} &\triangleq \{\texttt{[O . release]} \mid O\} \\
\texttt{reenter} &\triangleq \{\texttt{[k \% O . reenter]} \mid k, O\} \\
\texttt{control} &\triangleq \texttt{initialized} \cup \texttt{release} \cup \texttt{reenter} \\
\texttt{[O -> *]} &\triangleq \{\texttt{[O -> O' . new c[}\bar{e}\texttt{]]} \mid O', c, \bar{e}\} \\
&\quad \cup \{\texttt{[k \% O -> O' . m [@ c][}\bar{e}\texttt{]]} \mid k, O', m, c, \bar{e}\} \\
\texttt{[O <- *]} &\triangleq \{\texttt{[k \% O <- O' . m [@ c][}\bar{e}\texttt{ ; }\bar{e}'\texttt{]]} \mid k, O', m, c, \bar{e}, \bar{e}'\} \\
\texttt{[* -> O]} &\triangleq \{\texttt{[O' -> O . new c[}\bar{e}\texttt{]]} \mid O', c, \bar{e}\} \\
&\quad \cup \{\texttt{[k \% O' -> O . m [@ c][}\bar{e}\texttt{]]} \mid k, O', m, c, \bar{e}\} \\
\texttt{[* <- O]} &\triangleq \{\texttt{[k \% O' <- O . m [@ c][}\bar{e}\texttt{ ; }\bar{e}'\texttt{]]} \mid k, O', m, c, \bar{e}, \bar{e}'\}
\end{aligned}
$$

$$
\begin{array}{lll}
\texttt{[* -> } O \texttt{ . new } c\texttt{[*]]} & \triangleq & \{\texttt{[}O' \texttt{ -> } O \texttt{ . new } c\texttt{[}\bar{e}\texttt{]]} \mid O', \bar{e}\} \\
\texttt{[* -> } O \texttt{ . } m \texttt{ [@ } c\texttt{][*]]} & \triangleq & \{\texttt{[}k \texttt{ \% } O' \texttt{ -> } O \texttt{ . } m \texttt{ [@ } c\texttt{][}\bar{e}\texttt{]]} \mid k, O', \bar{e}\} \\
\texttt{[* <- } O \texttt{ . } m \texttt{ [@ } c\texttt{][*]]} & \triangleq & \{\texttt{[}k \texttt{ \% } O' \texttt{ <- } O \texttt{ . } m \texttt{ [@ } c\texttt{][}\bar{e} \texttt{ ; } \bar{e}'\texttt{]]} \mid k, O', \bar{e}, \bar{e}'\} \\
\texttt{[}k \texttt{ \% } O \texttt{ -> } O'\texttt{ . *]} & \triangleq & \{\texttt{[}k \texttt{ \% } O \texttt{ -> } O'\texttt{ . } m \texttt{ [@ } c\texttt{][}\bar{e}\texttt{]]} \mid m, c, \bar{e}\} \\
\texttt{[}k \texttt{ \% } O \texttt{ -> * . *]} & \triangleq & \{\texttt{[}k \texttt{ \% } O \texttt{ -> } O'\texttt{ . } m \texttt{ [@ } c\texttt{][}\bar{e}\texttt{]]} \mid O', m, c, \bar{e}\} \\
\texttt{[}k \texttt{ \% } O \texttt{ <- } O'\texttt{ . *]} & \triangleq & \{\texttt{[}k \texttt{ \% } O \texttt{ <- } O'\texttt{ . } m \texttt{ [@ } c\texttt{][}\bar{e} \texttt{ ; } \bar{e}'\texttt{]]} \mid m, c, \bar{e}, \bar{e}'\} \\
\texttt{[}k \texttt{ \% } O \texttt{ <- * . *]} & \triangleq & \{\texttt{[}k \texttt{ \% } O \texttt{ <- } O'\texttt{ . } m \texttt{ [@ } c\texttt{][}\bar{e} \texttt{ ; } \bar{e}'\texttt{]]} \mid O', m, c, \bar{e}, \bar{e}'\} \\
\texttt{[}O \texttt{ . reenter]} & \triangleq & \{\texttt{[}k \texttt{ \% } O \texttt{ . reenter]} \mid k\} \\
\texttt{in[}O\texttt{]} & \triangleq & \texttt{[* -> }O\texttt{]} \cup \texttt{[}O \texttt{ <- *]} \\
\texttt{out[}O\texttt{]} & \triangleq & \texttt{[}O \texttt{ -> *]} \cup \texttt{[* <- }O\texttt{]} \\
\texttt{ctl[}O\texttt{]} & \triangleq & \texttt{[}O \texttt{ . initialize]} \cup \texttt{[}O \texttt{ . release]} \\
 & & \quad \cup \texttt{[}O \texttt{ . reenter]} \\
O & \triangleq & \texttt{in[}O\texttt{]} \cup \texttt{out[}O\texttt{]} \cup \texttt{ctl[}O\texttt{]} \\
\sim v^* & \triangleq & v^{\mathrm{C}} \\
v_1^* \texttt{ \& } v_2^* & \triangleq & v_1^* \cap v_2^* \\
v_1^* \texttt{ | } v_2^* & \triangleq & v_1^* \cup v_2^*
\end{array}
$$

A.3.5 History Expressions

The sort `HistoryExp` of history expressions consists of terms with these syntaxes:

`emptyHistory`	(ϵ)	$x \in \texttt{Id}_{\text{A.2.1}}$
x `[: history]`		$H, H_1, H_2 \in \texttt{HistoryExp}$
Y		$Y \in \texttt{EventExp}_{\text{A.3.3}}$
H_1 `^^` H_2	($\frown$)	$Y^* \in \texttt{EventPatExp}_{\text{A.3.4}}$
H `/` Y^*	($/$)	

See Section 4.4 for the definition of the history operators.

A.3.6 History Pattern Expressions

The sort `HistoryPatExp` of history pattern expressions consists of terms with these syntaxes:

H		$H \in \texttt{HistoryExp}_{\text{A.3.5}}$
Y^*		$Y^* \in \texttt{EventPatExp}_{\text{A.3.4}}$
H_1^* `^^` H_2^*	($\frown$)	$H_1^*, H_2^* \in \texttt{HistoryPatExp}$

A.3.7 Built-in Functions

The following functions may appear in the proof obligations generated by the assertion analyzer:

`'lwf[`H`, `O`]`	$k \in \texttt{Int}_{\text{A.2.1}}$
`'mayAcquireProcessor[`H`, `O`, `O'`, `A`]`	$A \in \texttt{AExp}_{\text{A.2.5}}$
`'agreeOnOutAndCtl[`H`, `H'`, `O`]`	$O, O' \in \texttt{OExp}_{\text{A.2.7}}$
`'isFreshObjectId[`O`, `H`]`	$H, H' \in \texttt{HistoryExp}_{\text{A.3.5}}$

```
'isFreshSequenceNum[A, O, H]
'parent[O]
'returnVal $ k[H, O, A]
```

The `'lwf[H, O]` predicate represents the local history well-formedness predicate of Definition T20 in Section 4.4.

The `'mayAcquireProcessor[H, O, O', A]` predicate represents the processor acquisition predicate of Definition T21 in Section 4.4.

The `'agreeOnOutAndCtl[H, H', O]` predicate stands for `H / (out[O] | ctl[O]) eq H' / (out[O] | ctl[O])`.

The `'isFreshObjectId[O, H]` predicate stands for `! [O in 'objectIds[H]]`.

The `'isFreshSequenceNum[A, O, H]` predicate stands for `A ge 0 && ! [[A % O -> * . *] in H]`.

The `'parent[O]` function returns the parent of an object. Using first-order logic, we can for example deduce from `'parent[O] ne 'parent[O']` that `O ne O'`.

The `'returnVal $ k[H, O, A]` function represents the kth return value of the function call uniquely identified by the pair (O, A) according to the history H. The function is undefined if no such call has returned or if the invoked method has less than k output parameters.

A.4 Tool-Specific Commands

Each of the Creol tools has its own syntax for launching the system. The assertion analyzer requires a `verify class` command. The interpreter for closed systems requires a `bootstrap system` command. The interpreter for open systems requires a `bootstrap object` or a `bootstrap method` command, usually accompanied by `random data` commands.

A.4.1 Verify Class Command

To verify a class using the assertion analyzer, you must supply a `verify class` command along with the program:

```
verify class c                          | c ∈ Id_A.2.1
      [with simplifications q]          | q ∈ Qid
```

This command tells the assertion analyzer which class should be verified, and which simplification rules should be used to simplify the assertions produced by the tool. The simplification rules are supplied as a quoted Maude module name (for example, `'MY-SIMPLIFICATION-RULES`).

If the `with simplifications` clause is omitted, the built-in simplification rules (defined in `CREOL-SIMPLIFICATION-RULES`) are used. Section A.5 explains how to define custom simplification rules.

A.4.2 Bootstrap System Command

To run a Creol program using the interpreter for closed system, you must provide a `bootstrap system` command that specifies the name of the class to instantiate to launch the system, with optional arguments corresponding to the class's context parameters:

`bootstrap system` $c\,[[\bar{e}]]$	$c \in \texttt{Id}_{\text{A.2.1}}$ $\bar{e} \in \texttt{ExpList}_{\text{A.2.2}}$

When executing the program, the first rewrite step replaces the `bootstrap system` command with an object term that represents an instance of class c. After that, the expressions $\bar{e}$ are assigned to the context parameters specified in the class declaration, and the class's `'init` and `'run` methods are invoked.

A.4.3 Bootstrap Object Command

To launch a Creol object's initial process using the interpreter for open systems, you must use a `bootstrap object` command that specifies the new object's identity, its class, and the parent object's identity:

`bootstrap object` o `:= new` c `with parent` o'	$c \in \texttt{Id}_{\text{A.2.1}}$ $o, o' \in \texttt{OId}_{\text{A.4.4}}$

The new identity o must be of the form o `'#` n, with $n \in$ `Int`.

A.4.4 Bootstrap Method Command

To launch a Creol process that starts as a result of an asynchronous method invocation, you must use a `bootstrap method` command:

`bootstrap method` o `.` m $[$`@` $c'$$]$$[\bar{v}]$
`with class` c`,`
`caller` o'`,`
`label` k`,`
`history` h
$[$`, attributes` $\bar{z}$ `:=` $\bar{w}$$]$
	$k \in \texttt{Int}_{\text{A.2.1}}$
	$c, c', m \in \texttt{Id}_{\text{A.2.1}}$
	$\bar{z} \in \texttt{QualifiedIdList}_{\text{A.2.1}}$
	$o, o' \in \texttt{OId}_{\text{A.4.4}}$
	$\bar{v}, \bar{w}' \in \texttt{ValueList}_{\text{A.4.4}}$
	$h \in \texttt{History}_{\text{A.4.4}}$

The `bootstrap method` command lets you specify the called object's identity (o), the called method (m or m `@` c'), the arguments passed to the method ($\bar{v}$), the called object's class (c), the caller object's identity (o'), the sequence number associated with the call (k), the local history for the called object at the moment when the method starts executing (h), and the values ($\bar{w}$) of the object's attributes ($\bar{z}$).

The `History` sort is a subsort of $\texttt{HistoryExp}_{\text{A.3.5}}$ that consists exclusively of constants. The `Value` sort is a supersort of `Int`, `Bool`, `OId`, and `History`. Finally, the `OId` sort consists of `null`, `root`, and terms of the form x `#` n_1 `...` `#` n_p, where $x \in \texttt{Id} \cup \{\texttt{null}, \texttt{root}\}$, $n_i \in$ `Int`, and $p \geq 1$.

A.4.5 Random Data Commands

The `random data` commands that may accompany a `bootstrap object` or `bootstrap method` command have the following syntax:

`random data` h	$h \in$ `History`$_{\text{A.4.4}}$
`random data` α	$\alpha \in$ `State`$_{\text{A.4.5}}$

The `State` sort consists of terms with these syntaxes:

`emptyState`	($\emptyset$)	
`[`z `|->` v`]`		$z \in$ `QualifiedId`$_{\text{A.2.1}}$, $v \in$ `Value`$_{\text{A.4.4}}$
α_1 α_2		$\alpha_1, \alpha_2 \in$ `State`

A.5 Simplification Rules

The computation of weakest liberal preconditions performed by the assertion analyzer often leads to complex proof obligations. To make its output more readable, the assertion analyzer performs syntactic simplifications on the proof obligations before it displays them. The tool's built-in simplification rules can be extended or replaced by the user.

A.5.1 Built-in Simplification Rules

Before we see how to write custom simplification rules, let us consider the built-in simplification rules. These rules are defined in two system modules. The first module, `CREOL-LOGICAL-SIMPLIFICATION-RULES` (located in `creol-assertion-utilities.maude`), restricts itself to logical simplifications, that is, simplifications that involve only the logical operators (`!`, `&&`, `||`, etc.). For example, here are three rules defined by that module:

```
rl PHI && PHI => PHI .
rl (true ==> PHI) => PHI .
rl ((PHI1 && PHI2) ==> PHI1) => true .
```

The second simplification module, `CREOL-SIMPLIFICATION-RULES` (located in `creol-assertion-analyzer.maude`), imports the logical simplifications and extends these with simplifications that involve non-logical symbols. For example:

```
rl N1 plus N2 => N1 + N2 .
rl A1 ge A2 => A2 le A1 .
rl ((A1 eq A2 && PHI) ==> A1 le A2) => true .
```

These simplifications are complemented by logical normalizations, which move quantifiers outward, replace `P ==> Q ==> R` with `(P && Q) ==> R`, and so on. These normalizations are specified in the modules `CREOL-NORMALIZATION-RULES-1` and `-2` (located in `creol-assertion-analyzer.maude`).

A.5.2 Custom Simplification Rules

When we verify a class guarantee using the assertion analyzer, we can define custom simplification rules that are tailored for the class of interest. To illustrate this, we will consider the `'SimpleMutex` class of Section A.1.1, but with a `guar` clause that involves the `'isLocked[`*h*`, `*o*`]` custom predicate.

```
class 'SimpleMutex
   implements 'Mutex
begin
   var 'locked : bool

with any :
   op 'lock is
      await ! 'locked ;
      'locked := true

   op 'unlock is
      'locked := false

   guar 'locked <==> 'isLocked[~H~, self]
end
```

Informally, `'isLocked[`*h*`, `*o*`]` returns `true` if and only if the last method call that returned according to the history *h* was `'lock`.

If we run the assertion analyzer on `'SimpleMutex` using only the built-in simplification rules, we obtain various proof obligations containing terms such as these:

```
'isLocked[['parent[self] -> self . new 'SimpleMutex[epsilon]]
          ^^ [self . initialized], self]
'isLocked[~H~ ^^ [label % caller <- self
                  . 'lock[epsilon ; epsilon]], self]
'isLocked['h $ 1 ^^ [label % caller <- self
                     . 'lock[epsilon ; epsilon]], self]
'isLocked[~H~ ^^ [label % caller <- self
                  . 'unlock[epsilon ; epsilon]], self]
```

Our goal would be to reduce these to `false`, `true`, `true`, and `false`, respectively, based on the informal definition of `'isLocked[`*h*`, `*o*`]`. To achieve this, we can define the following simplification rules:

```
mod MUTEX-SIMPLIFICATION-RULES is
  including CREOL-SIMPLIFICATION-RULES .

  var A : AExp .
  var EEXP : EventExp .
  var HEXP : HistoryExp .
  vars OEXP OEXP' : OExp .

  rl 'isLocked[emptyHistory, OEXP] => false .
  rl 'isLocked[HEXP ^^ [A % OEXP' <- OEXP .
                        'lock[epsilon ; epsilon]],
              OEXP] =>
     true .
  rl 'isLocked[HEXP ^^ [A % OEXP' <- OEXP .
```

```
                        'unlock[epsilon ; epsilon]],
               OEXP] =>
     false .
  crl 'isLocked[HEXP ^^ EEXP, OEXP] => 'isLocked[HEXP, OEXP]
  if EEXP cannot match reply .
endm
```

At the beginning of the module, we include `CREOL-SIMPLIFICATION-RULES`, which defines the built-in simplification rules. To tell the assertion analyzer about our simplification rules, we must specify the name of the simplification module in the `verify class` command:

```
verify class 'SimpleMutex
       with simplifications 'MUTEX-SIMPLIFICATION-RULES
```

When writing simplification rules, we must keep the following points in mind:

- *Simplification rules operate at the syntactic level, not at the semantic level.*

  Consider the `'isLocked` predicate presented above. Although the predicate is fundamentally a function from `History` $\times$ `OId` to `Bool`, syntactically it takes a `HistoryExp` and an `OExp` and returns a `BExp`.

  The passage from values to expressions requires some care, as illustrated by the following example. Let `'fact[`n`]` denote the factorial of $n \in$ `Int`. The simplification rule

  ```
  var N : Int .

  rl 'fact[N] => if N > 1 then N * 'fact[N - 1] else 1 fi .
  ```

  will successfully simplify `'fact[`n`]` to its value ($n!$) for any integer n. If we naively lift the above rule to expressions, we obtain

  ```
  var A : AExp .

  *** WRONG
  rl 'fact[A] => if A gt 1 th A times 'fact[A minus 1] el 1 fi .
  ```

  This new rule is wrong because leads to infinite expansions such as this:

  ```
      'fact['x]
  ⟶
      if 'x gt 1 th 'x times 'fact['x minus 1] el 1 fi
  ⟶
      if 'x gt 1 th 'x times if 'x minus 1 gt 1 th ('x minus 1)
      times 'fact['x minus 1 minus 1] el 1 fi el 1 fi
  ⟶
      ...
  ```

- *We must use the* `otherwise` *attribute with care.*

  Because syntactic-level equality (`eq`) does not coincide with semantic-level equality (`==`), the `otherwise` attribute will often lead to unsuspected result. For example, the simplification rule

```
var EEXP : EventExp .
var HEXP : HistoryExp .
var OEXP : OExp .

*** WRONG
rl 'isLocked[HEXP ^^ EEXP, OEXP] => 'isLocked[HEXP, OEXP]
                                                [otherwise] .
```

would allow the simplification

```
    'isLocked['h : history ^^ 'e : event, self : any]
⟶
    'isLocked['h : history, self : any].
```

Syntactically, `'e : event` doesn't match any of the alternatives; semantically, it can match *any* of the alternatives.

As a substitute for `otherwise`, we can use the `Y cannot match Y*` operator, which takes an event expression `Y` and an event pattern expression `Y*` and returns `true` only if it can determine that the expression cannot match the pattern. Thus, for `'isLocked`, we can use the following rule to eliminate trailing events that cannot affect the value of `'isLocked`:

```
crl 'isLocked[HEXP ^^ EEXP, OEXP] => 'isLocked[HEXP, OEXP]
if EEXP cannot match reply .
```

- *The program and logical variables that occur in simplification rules must carry typing annotations.*

  To prevent spurious simplifications, the assertion analyzer applies simplification rules on fully typed assertions, even though it displays the resulting proof obligations without types. This is why we needed `self : any` and not just `self` in the `'isLocked` simplification rules above.

- *Program and free logical variables should not be introduced on the right-hand side of a simplification rule.*

  The assertion analyzer performs various simplifications and substitutions assuming that program and logical variables that don't occur syntactically in a subexpression don't influence it. Simplification rules that introduce these variables in their right-hand side can lead to unsound proof obligations. For example, the following simplification rule is wrong:

  ```
  *** WRONG
  rl 'G => 'isLocked[~H~ : history, self : any] .
  ```

  Exceptionally, it is safe to refer to the special variables `self`, `caller`, and `label`, since their values stay the same for the lifetime of a process.

- *Some of the logical and non-logical operators have associativity, commutativity, or identity attributes in their Maude declarations.*

  When writing simplification rules, it helps to know which operators are declared with which attributes. For these, look for the operator declarations in `creol-program.maude`.

For the 'SimpleMutex example, we used custom simplification rules to expand a custom function application to its definition. Another common use of custom simplifications is to simplify formulas involving built-in non-logical symbols or functions. Chapter 7 has several examples of simplification rules, many of which involve built-in symbols such as ^^, pr, and 'lwf.

A.6 Custom Data Types and Functions

The Creol tools support only two built-in data types: bool and int. This proved sufficient for the examples presented in this thesis, but many other Creol programs require more powerful data types. While arbitrarily complex data structures can be represented using Creol objects, it is better style to use Creol's functional sublanguage to define data types and operations on them.

Example 4.1 of Section 4.2 featured a 2D point data type to store a pair of integer coordinates, allowing us to write code like this:

```
var 'p1 : 'point2D, 'p2 : 'point2D, 'width : int, 'height : int ;
'p1 := 'point2D[320, 200] ;
'p2 := 'point2D[1024, 768] ;
'width := 'getX['p2] minus 'getX['p1] ;
'height := 'getY['p2] minus 'getY['p1]
```

Custom functions, including constructors, are applied using the syntax $f[\bar{e}]$, with $f \in$ Id and $\bar{e} \in$ ExpList.

When the Creol interpreters need to evaluate an expression that involves function applications, it evaluates the arguments and adds @ to the left of the function name. Then, user-supplied equations are applied to expand non-constructor function applications, until only constructors are left. For example, the following functional module implements the 'point2D data type:

```
fmod POINT2D-EQUATIONS is
  including CREOL-VALUE .

  vars X X1 X2 : Int .
  vars Y Y1 Y2 : Int .

  eq defaultValue('point2D) = @ 'point2D[0, 0] .

  eq @ 'getX[@ 'point2D[X, Y]] = X .
  eq @ 'getY[@ 'point2D[X, Y]] = Y .
  eq @ 'add[@ 'point2D[X1, Y1], @ 'point2D[X2, Y2]] =
     @ 'point2D[X1 + X2, Y1 + Y2] .
endfm
```

The first equation defines a default value for the type. This is necessary so that variables of type 'point2D are correctly initialized. The other equations define the functions that operate on 'point2D values. To use the 'point2D in a Creol program, we must simply include the POINT2D-EQUATIONS module in the program's module.

Custom functions can be used to implement operations on built-in data types just as well as they can be used on custom data types. A common use case occurs when

using the interpreter for open systems; custom functions occurring in the class's assume–guarantee specification must be defined using equations. For example:

```
fmod MUTEX-EQUATIONS is
  including CREOL-HISTORY .

  var EV : Event .
  var H : History .
  var K : Int .
  vars O O' : OId .

  eq @ 'isLocked[emptyHistory, O] = false .
  eq @ 'isLocked[H ^^ [K % O' <- O . 'lock[epsilon ; epsilon]], O] =
     true .
  eq @ 'isLocked[H ^^
                   [K % O' <- O . 'unlock[epsilon ; epsilon]], O] =
     false .
  eq @ 'isLocked[H ^^ EV, O] = @ 'isLocked[H, O] [otherwise] .
endfm
```

In Sections 7.3 and 7.4, we saw two implementations of the factorial function, one using a `while` loop and the other using a recursive method call. A simpler and more Creolesque (but also less instructive) solution would have been to define a `'fact` function using Creol's functional sublanguage, as follows:

```
fmod FACTORIAL-EQUATIONS is
  including CREOL-VALUE .

  var N : Int .

  eq @ 'fact[N] = if N >= 1 then N * @ 'fact[N - 1] else 1 fi .
endfm
```

A.7 Known Bugs and Limitations

Here are the known bugs and limitations in the tools at the time of writing:

- Method overloading is not supported by any of the tools.
- In the assertion analyzer, the special variables `self` and `caller` are typed as `any` instead of their actual type.
- The interpreter for open systems fails on `forall` (∀) or `exists` (∃) quantifiers in assume–guarantee specifications.
- The assertion massaging process can be slow for large programs.
- The assertion analyzer records invocation and reply events without qualification (that is, *m* instead of *m* `@` *c*).
- The tools assume that the input programs are syntactically correct and well-typed.

Comments and bug reports concerning the tools or their documentation should be directed to `jasmincb@ifi.uio.no` or `olaf@ifi.uio.no`.

Appendix B

Specifications of the Assertion Analyzer and the Interpreters

This appendix presents the complete Maude specification of the Creol tools that were developed in this thesis. The code is put in the public domain.

B.1 Creol Program Syntax

```
***(
    creol-program.maude

    This file implements the Creol language's syntax described in
    Sections 4.2, A.2, and A.3 of Verification of Assertions in Creol
    Programs.
)

fmod CREOL-PRELUDE is
  *** import what we need from the prelude and rename operators that
  *** clash with Creol
  including (INT + META-LEVEL) *
      (op ~_ to maude~_,
       op _&_ to _maude&_,
       op _|_ to _maude|_,
       op _:=_ to _maude:=_,
       op _/_ to _maude/_,
       op _xor_ : Nat Nat -> Nat to _xor2_,
       op __ : NatList NatList -> NatList to _‘,‘,_,
       sort Type to MaudeType).
endfm

fmod CREOL-LIST is
  including CREOL-PRELUDE .

  *** Base for comma-separated lists
  sort EmptyList .

  op epsilon : -> EmptyList [ctor] .
```

```
  op _,_ : EmptyList EmptyList -> EmptyList
                      [ctor assoc prec 21 id: epsilon format (d d s d)] .

  *** Supersort for list elements
  sort ListElem .

  *** Supersort for comma-separated lists
  sort List .
  subsort EmptyList < List .
  subsort ListElem < List .

  op _,_ : List List -> List [ctor ditto] .

  op _without_ : List List -> List [prec 9] .
  op length : List -> Nat .

  var ELEM : ListElem .
  vars LIST LIST' : List .

  eq epsilon without LIST = epsilon .
  eq ELEM without (LIST, ELEM, LIST') = epsilon .
  ceq (ELEM, LIST) without LIST' =
      (ELEM without LIST'), (LIST without LIST')
  if LIST =/= epsilon .
  eq ELEM without LIST = ELEM [otherwise] .

  eq length(epsilon) = 0 .
  eq length(ELEM, LIST) = 1 + length(LIST) .
endfm

fmod CREOL-MULTISET is
  including CREOL-PRELUDE .

  *** Base for ++-separated multisets
  sort EmptyMSet .

  op emptyMSet : -> EmptyMSet [ctor] .
  op _++_ : EmptyMSet EmptyMSet -> EmptyMSet
                                  [ctor assoc comm prec 7 id: emptyMSet] .

  *** Supersort for multiset elements
  sort MSetElem .

  *** Supersort for sets
  sort MSet .
  subsort EmptyMSet < MSet .
  subsort MSetElem < MSet .

  op _++_ : MSet MSet -> MSet [ctor ditto] .

  op _\_ : MSet MSet -> MSet [prec 9] .
  op _in mset_ : MSetElem MSet -> Bool [prec 11] .

  vars ELEM ELEM' : MSetElem .
  vars MSET MSET' : MSet .
```

```
  eq emptyMSet \ MSET' = emptyMSet .
  eq (ELEM ++ MSET) \ MSET' =
     (MSET \ MSET') ++ if ELEM in mset MSET' then emptyMSet
                                             else ELEM fi .

  eq ELEM in mset emptyMSet = false .
  eq ELEM in mset (ELEM' ++ MSET) =
     (ELEM == ELEM') or (ELEM in mset MSET) .
endfm

fmod CREOL-IDENTIFIER is
  including CREOL-LIST .
  including CREOL-MULTISET .

  *** Plain quoted identifier (used instead of Qid to avoid clashes with
  *** META-LEVEL)
  sort QuotedId .

  op <QIds> : -> QuotedId [special (id-hook QuotedIdentifierSymbol)] .

  *** Creol identifier
  sort Id .
  subsort QuotedId < Id .

  *** special identifiers
  op none : -> Id [ctor] .
  op nu : -> Id [ctor] .            *** fresh label
  op self : -> Id [ctor] .
  op caller : -> Id [ctor] .
  op label : -> Id [ctor] .
  op ~H~ : -> Id [ctor] .           *** mythical history

  *** generated logical variables
  op _$_ : Id Int -> Id [ctor prec 1] .

  *** auxiliary function
  op baseId : Id -> Id .

  *** Comma-separated list of identifiers
  sort IdList .
  subsort Id < IdList .
  subsort Id < ListElem .
  subsort IdList < List .
  subsort EmptyList < IdList .

  op _,_ : IdList IdList -> IdList [ctor ditto] .

  *** Multiset of identifiers
  sort IdMSet .
  subsort Id < IdMSet .
  subsort Id < MSetElem .
  subsort IdMSet < MSet .
  subsort EmptyMSet < IdMSet .
```

```
  op _++_ : IdMSet IdMSet -> IdMSet [ctor ditto] .

  var N : Int .
  var X : Id .

  eq baseId(X $ N) = baseId(X) .
  eq baseId(X) = X [otherwise] .
endfm

fmod CREOL-QUALIFIED-IDENTIFIER is
  including CREOL-IDENTIFIER .

  *** Identifier that may be qualified with a class name (e.g., 'x @ 'C)
  *** or not (e.g., 'x, 'x @ none)
  sort QualifiedId .
  subsort Id < QualifiedId .

  op _@_ : Id Id -> QualifiedId [ctor prec 1 right id: none] .

  *** Comma-separated list of qualified identifiers
  sort QualifiedIdList .
  subsort QualifiedId < QualifiedIdList .
  subsort QualifiedId < ListElem .
  subsort QualifiedIdList < List .
  subsort IdList < QualifiedIdList .

  op _,_ : QualifiedIdList QualifiedIdList -> QualifiedIdList
                                                    [ctor ditto] .
endfm

fmod CREOL-TYPE is
  including CREOL-IDENTIFIER .

  *** Creol data type
  sort Type .
  subsort Id < Type .

  *** built-in program data types
  op bool : -> Type [ctor] .
  op int : -> Type [ctor] .
  op any : -> Type [ctor] .

  *** built-in reasoning data types
  op event : -> Type [ctor] .
  op history : -> Type [ctor] .
endfm

fmod CREOL-TYPED-IDENTIFIER is
  including CREOL-TYPE .

  *** Typed identifier
  sort TypedId .
  subsort Id < TypedId .

  op _:_ : Id Type -> TypedId [ctor prec 3 right id: none] .
```

```
  *** Comma-separated list of typed identifiers
  sort TypedIdList .
  subsort TypedId < TypedIdList .
  subsort TypedId < ListElem .
  subsort TypedIdList < List .
  subsort IdList < TypedIdList .

  op _,_ : TypedIdList TypedIdList -> TypedIdList [ctor ditto] .

  *** auxiliary function
  op asIdList : TypedIdList -> IdList .

  *** Identifier followed by typed parameters
  *** e.g., 'C['x : bool, 'y : int, 'z : 'I]
  sort IdWithParams .
  subsort Id < IdWithParams .

  op _[_] : Id TypedIdList -> IdWithParams
                                        [ctor prec 3 right id: epsilon] .

  var T0 : Type .
  var X0 : Id .
  var XXL : TypedIdList .

  eq asIdList(epsilon) = epsilon .
  eq asIdList(X0 : T0, XXL) = X0, asIdList(XXL) .
endfm

fmod CREOL-TYPED-QUALIFIED-IDENTIFIER is
  including CREOL-QUALIFIED-IDENTIFIER .
  including CREOL-TYPED-IDENTIFIER .

  *** Typed qualified identifier
  sort TypedQualifiedId .
  subsort QualifiedId < TypedQualifiedId .
  subsort TypedId < TypedQualifiedId .

  op _:_ : QualifiedId Type -> TypedQualifiedId [ctor ditto] .

  *** Comma-separated list of typed qualified identifiers
  sort TypedQualifiedIdList .
  subsort TypedQualifiedId < TypedQualifiedIdList .
  subsort TypedQualifiedId < ListElem .
  subsort TypedQualifiedIdList < List .
  subsort TypedIdList < TypedQualifiedIdList .
  subsort QualifiedIdList < TypedQualifiedIdList .

  op _,_ :
     TypedQualifiedIdList TypedQualifiedIdList -> TypedQualifiedIdList
                                                        [ctor ditto] .

  *** auxiliary operator
  op _@@_ : TypedIdList Id -> TypedQualifiedIdList [prec 23] .
  op _@@_ : IdList Id -> QualifiedIdList [ditto] .
```

```
  *** Boolean-typed qualified identifier
  sort BoolTypedId .
  subsort BoolTypedId < TypedId .
  subsort BoolTypedId < TypedQualifiedId .

  mb (Z : bool) : BoolTypedId .

  *** Integer-typed qualified identifier
  sort IntTypedId .
  subsort IntTypedId < TypedId .
  subsort IntTypedId < TypedQualifiedId .

  mb (Z : int) : IntTypedId .

  *** Unknown-interface-typed qualified identifier
  sort AnyTypedId .
  subsort AnyTypedId < TypedId .
  subsort AnyTypedId < TypedQualifiedId .

  mb (Z : any) : AnyTypedId .
  mb (Z : QUOTEDID) : AnyTypedId .

  *** Event-typed qualified identifier
  sort EventTypedId .
  subsort EventTypedId < TypedId .
  subsort EventTypedId < TypedQualifiedId .

  mb (Z : event) : EventTypedId .

  *** History-typed qualified identifier
  sort HistoryTypedId .
  subsort HistoryTypedId < TypedId .
  subsort HistoryTypedId < TypedQualifiedId .

  mb (Z : history) : HistoryTypedId .

  var C : Id .
  var QUOTEDID : QuotedId .
  var T0 : Type .
  var X0 : Id .
  var XXL : TypedIdList .
  var Z : QualifiedId .

  eq epsilon @@ C = epsilon .
  eq X0 : T0, XXL @@ C = X0 @ C : T0, (XXL @@ C) .
endfm

fmod CREOL-OBJECT-IDENTITY is
  including CREOL-IDENTIFIER .

  *** Object identity
  sort OId .

  *** Base part of an object identity
```

```
  sort OIdBase .
  subsort Id < OIdBase .
  subsort OId < OIdBase .

  *** main constructor
  op _#_ : OIdBase Int -> OId [ctor prec 1] .

  *** special object identifiers
  op null : -> OId [ctor] .
  op root : -> OId [ctor] .

  *** auxiliary operators
  op parent : OId -> OIdBase .
  op seq : OId -> Int .

  *** Multiset of object identifiers
  sort OIdMSet .
  subsort OId < OIdMSet .
  subsort OId < MSetElem .
  subsort OIdMSet < MSet .
  subsort EmptyMSet < OIdMSet .

  op _++_ : OIdMSet OIdMSet -> OIdMSet [ctor ditto] .

  var N : Int .
  var OID : OId .
  var OIDBASE : OIdBase .

  eq parent(OIDBASE # N) = OIDBASE .
  eq parent(OID) = null [otherwise] .

  eq seq(OIDBASE # N) = N .
  eq seq(OID) = 0 [otherwise] .
endfm

fmod CREOL-VALUE is
  including CREOL-OBJECT-IDENTITY .
  including CREOL-TYPED-QUALIFIED-IDENTIFIER .

  *** Arithmetic (integer), Boolean, or object value
  sort Value .
  subsort Bool < Value .
  subsort Int < Value .
  subsort OId < Value .

  *** value of custom type (represented as a term, e.g., @ 's[5])
  op @_[_] : Id ValueList -> Value [ctor] .

  *** Comma-separated list of values
  sort ValueList .
  subsort Value < ValueList .
  subsort Value < ListElem .
  subsort ValueList < List .
  subsort TypedQualifiedIdList < ValueList . *** for preregularity
```

```
  op _,_ : ValueList ValueList -> ValueList [ctor ditto] .

  op defaultValue : Type -> Value .
  op typeOf : Value -> Type .

  var BOOL : Bool .
  var F : Id .
  var N : Nat .
  var O : OId .
  var VL : ValueList .
  var X : Id .

  eq defaultValue(bool) = false .
  eq defaultValue(int) = 0 .
  eq defaultValue(any) = null .
  eq defaultValue(X) = null [otherwise] .

  eq typeOf(BOOL) = bool .
  eq typeOf(N) = int .
  eq typeOf(O) = any .
  eq typeOf(@ F[VL]) = none .       *** unknown
endfm

fmod CREOL-EXPRESSION is
  including CREOL-TYPE .
  including CREOL-VALUE .
  including CREOL-TYPED-QUALIFIED-IDENTIFIER .

  *** Base for expressions
  sort BasicExp .
  subsort QualifiedId < BasicExp .
  subsort IdWithArgs < BasicExp .

  op if_th_el_fi : BExp BasicExp BasicExp -> BasicExp [ctor] .
  op [_] : BasicExp -> BasicExp [ctor] .

  *** Arithmetic (integer) expression
  sort AExp .
  subsort BasicExp < AExp .
  subsort Int < AExp .
  subsort IntTypedId < AExp .

  op plus_ : AExp -> AExp [ctor prec 3] .
  op minus_ : AExp -> AExp [ctor prec 3] .
  op _times_ : AExp AExp -> AExp [ctor assoc comm prec 5 gather (E e)] .
  op _div_ : AExp AExp -> AExp [ctor prec 5 gather (E e)] .
  op _plus_ : AExp AExp -> AExp [ctor assoc comm prec 7 gather (E e)] .
  op _minus_ : AExp AExp -> AExp [ctor prec 7 gather (E e)] .
  op if_th_el_fi : BExp AExp AExp -> AExp [ctor] .
  op [_] : AExp -> AExp [ctor] .

  *** Basic Boolean expression
  sort BasicBExp .
  subsort BasicExp < BasicBExp .
  subsort Bool < BasicBExp .
```

```
subsort BoolTypedId < BasicBExp .

*** arithmetic comparison operators
op _eq_ : AExp AExp -> BasicBExp [ctor comm prec 11] .
op _ne_ : AExp AExp -> BasicBExp [ctor comm prec 11] .
op _lt_ : AExp AExp -> BasicBExp [ctor prec 11] .
op _gt_ : AExp AExp -> BasicBExp [ctor prec 11] .
op _le_ : AExp AExp -> BasicBExp [ctor prec 11] .
op _ge_ : AExp AExp -> BasicBExp [ctor prec 11] .

*** Boolean comparison operators (undocumented, but useful)
op _eq_ : BExp BExp -> BasicBExp [ctor ditto] .
op _ne_ : BExp BExp -> BasicBExp [ctor ditto] .

*** object comparison operators
op _eq_ : OExp OExp -> BasicBExp [ctor ditto] .
op _ne_ : OExp OExp -> BasicBExp [ctor ditto] .

*** Boolean expression
sort BExp .
subsort BasicBExp < BExp .

op !_ : BExp -> BExp [ctor prec 5] .
op _&&_ : BExp BExp -> BExp [ctor assoc comm prec 13] .
op _||_ : BExp BExp -> BExp [ctor assoc comm prec 15] .
op if_th_el_fi : BExp BExp BExp -> BExp [ctor] .
op [_] : BExp -> BExp [ctor] .

*** Logical quantifier
sort Quantifier .

op forall : -> Quantifier [ctor] .
op exists : -> Quantifier [ctor] .

*** auxiliary function
op opposite : Quantifier -> Quantifier .

*** Basic first-order logic assertion
sort BasicAssn .
subsort BasicBExp < BasicAssn .

*** Complex first-order logic assertion
sort Assn .
subsort BasicAssn < Assn .
subsort BExp < Assn .

op !_ : Assn -> Assn [ctor ditto] .
op _&&_ : Assn Assn -> Assn [ctor ditto] .
op _||_ : Assn Assn -> Assn [ctor ditto] .
op _==>_ : Assn Assn -> Assn [ctor prec 17 gather (e E)] .
op _<==>_ : Assn Assn -> Assn [ctor comm prec 19 gather (e e)] .
op __._ : Quantifier TypedIdList Assn -> Assn
                                        [ctor prec 21 gather (e & E)] .
op if_th_el_fi : Assn Assn Assn -> Assn [ctor] .
op [_] : Assn -> Assn [ctor] .
```

```
*** fresh identifier generators
op freshLogicalVar : TypedQualifiedId ExpList -> TypedId .
op freshLogicalVarHelper : TypedQualifiedId ExpList Int -> TypedId .
op freshLogicalVarList : TypedQualifiedIdList ExpList -> TypedIdList .

*** Object expression
sort OExp .
subsort BasicExp < OExp .
subsort AnyTypedId < OExp .

op if_th_el_fi : BExp OExp OExp -> OExp [ctor] .
op [_] : OExp -> OExp [ctor] .

*** used internally to assign object identifiers to variables
subsort OId < OExp .

*** Expression of any type (arithmetic, Boolean, object, assertion)
sort Exp .
subsort AExp < Exp .
subsort Assn < Exp .
subsort OExp < Exp .
subsort Value < Exp .

*** Comma-separated list of expressions
sort ExpList .
subsort Exp < ExpList .
subsort Exp < ListElem .
subsort ExpList < List .
subsort ValueList < ExpList .

op _,_ : ExpList ExpList -> ExpList [ctor ditto] .

*** auxiliary operator and functions
op _occurs free in_ : TypedQualifiedId ExpList -> Bool [prec 23] .
op qualified : ExpList TypedIdList Id -> ExpList .
op qualifiedAndTyped : ExpList TypedIdList Id -> ExpList .

*** Variable substitution
sort Subst .

op emptySubst : -> Subst [ctor] .
op {_|->_} : TypedQualifiedIdList ExpList -> Subst [ctor] .

op typedBuiltInVarSubst : -> Subst .

*** application of a substitution on a list of expressions
op __ : ExpList Subst -> ExpList [prec 1] .
op __ : BasicExp Subst -> BasicExp [ditto] .
op __ : AExp Subst -> AExp [ditto] .
op __ : BExp Subst -> BExp [ditto] .
op __ : Assn Subst -> Assn [ditto] .
op __ : OExp Subst -> OExp [ditto] .

*** other operators
```

```
op _without_ : Subst TypedQualifiedId -> Subst [prec 5] .
op targets : Subst -> ExpList .

*** Identifier followed by arguments
*** e.g., 'C[true, 4, caller]
sort IdWithArgs .
subsort IdWithParams < IdWithArgs .

op _[_] : Id ExpList -> IdWithArgs [ctor ditto] .

vars A A1 A2 : AExp .
vars C C0 : Id .
vars E E0 E1 E2 : Exp .
vars EL EL' : ExpList .
var F : Id .
var N : Int .
var O : OId .
vars PHI PHI1 PHI2 : Assn .
var QUANT : Quantifier .
var RHO : Subst .
vars T T0 : Type .
var VL : ValueList .
vars X X0 : Id .
var XX0 : TypedId .
vars XXL XXL' : TypedIdList .
var YY0 : TypedId .
vars Z Z0 : QualifiedId .
var ZZ : TypedQualifiedId .
vars ZZL ZZL' : TypedQualifiedIdList .

eq opposite(forall) = exists .
eq opposite(exists) = forall .

eq QUANT XXL . QUANT XXL' . PHI = QUANT XXL, (XXL' without XXL) . PHI .
eq QUANT epsilon . PHI = PHI .
eq {epsilon |-> epsilon} = emptySubst .

eq freshLogicalVar(ZZ, EL) = freshLogicalVarHelper(ZZ, EL, 1) .

eq freshLogicalVarHelper(X @ C : T, EL, N) =
   if not baseId(X) $ N occurs free in EL then
       baseId(X) $ N : T
   else
       freshLogicalVarHelper(X @ C : T, EL, (N + 1))
   fi .

eq freshLogicalVarList(epsilon, EL) = epsilon .
ceq freshLogicalVarList((X0 @ C0 : T0, ZZL), EL) =
    YY0, freshLogicalVarList(ZZL, (YY0, EL))
if YY0 := freshLogicalVar((X0 @ C0 : T0), EL) .

eq ZZ occurs free in epsilon = false .
eq ZZ occurs free in E0, EL =
   (E0 {ZZ |-> none} =/= E0) or ZZ occurs free in EL .
```

```
eq qualified(EL, XXL, C) = (EL) { XXL |-> XXL @@ C } .

eq qualifiedAndTyped(EL, XXL, C) =
   (qualified(EL, XXL, C)) typedBuiltInVarSubst .

eq typedBuiltInVarSubst =
   { nu, self, caller, label, ~H~ |->
     nu : int, self : any, caller : any, label : int, ~H~ : history } .

eq (ZZ) emptySubst = ZZ .
eq (Z : T) {ZZL, Z0 : T0 |-> EL, E0} =
   if Z0 == Z then E0 else (Z : T) {ZZL |-> EL} fi .

eq (if PHI th E1 el E2 fi) RHO =
   if (PHI) RHO th (E1) RHO el (E2) RHO fi .
eq ([E]) RHO = [(E) RHO] .

eq (F[EL]) RHO = F[(EL) RHO] .
eq (@ F[VL]) RHO = F[VL] .

eq (N) RHO = N .
eq (plus A) RHO = plus (A) RHO .
eq (minus A) RHO = minus (A) RHO .
eq (A1 times A2) RHO = (A1) RHO times (A2) RHO .
eq (A1 div A2) RHO = (A1) RHO div (A2) RHO .
eq (A1 plus A2) RHO = (A1) RHO plus (A2) RHO .
eq (A1 minus A2) RHO = (A1) RHO minus (A2) RHO .

eq (true) RHO = true .
eq (false) RHO = false .
eq (E1 eq E2) RHO = (E1) RHO eq (E2) RHO .
eq (E1 ne E2) RHO = (E1) RHO ne (E2) RHO .
eq (E1 lt E2) RHO = (E1) RHO lt (E2) RHO .
eq (E1 gt E2) RHO = (E1) RHO gt (E2) RHO .
eq (E1 le E2) RHO = (E1) RHO le (E2) RHO .
eq (E1 ge E2) RHO = (E1) RHO ge (E2) RHO .
eq (! PHI) RHO = ! (PHI) RHO .
eq (PHI1 && PHI2) RHO = (PHI1) RHO && (PHI2) RHO .
eq (PHI1 || PHI2) RHO = (PHI1) RHO || (PHI2) RHO .
eq (PHI1 ==> PHI2) RHO = (PHI1) RHO ==> (PHI2) RHO .
eq (PHI1 <==> PHI2) RHO = (PHI1) RHO <==> (PHI2) RHO .

*** makes sure that free variables introduced by the substitution RHO
*** don't get bound accidentally, by renaming XX0 if necessary
ceq (QUANT XX0, XXL . PHI) RHO =
    QUANT XX0 . (QUANT XXL . PHI) (RHO without XX0)
if not XX0 occurs free in targets(RHO) .
ceq (QUANT XX0, XXL . PHI) RHO =
    (QUANT YY0 . (QUANT XXL . (PHI) { XX0 |-> YY0 })) (RHO without XX0)
if YY0 := freshLogicalVar(XX0, (XXL, PHI, targets(RHO))) [otherwise] .

eq (O) RHO = O .

eq (epsilon) RHO = epsilon .
ceq (E0, EL) RHO = (E0) RHO, (EL) RHO if EL =/= epsilon .
```

```
  eq emptySubst without ZZ = emptySubst .
  ceq { ZZL, Z, ZZL' |-> EL, E, EL' } without Z : T =
      { ZZL, ZZL' |-> EL, EL' } without Z : T
  if length(ZZL) == length(EL) .
  eq { ZZL |-> EL } without Z : T = { ZZL |-> EL } [otherwise] .

  eq targets(emptySubst) = epsilon .
  eq targets({ ZZL |-> EL }) = EL .
endfm

fmod CREOL-HISTORY is
  including CREOL-VALUE .

  *** Event in the history
  sort Event .

  op [_->_. new_[_]] : OId OId Id ValueList -> Event [ctor] .
  op [_%_->_._[_]] : Int OId OId QualifiedId ValueList -> Event [ctor] .
  op [_%_<-_._[_;_]] :
     Int OId OId QualifiedId ValueList ValueList -> Event [ctor] .
  op [_. initialized] : OId -> Event [ctor] .
  op [_. release] : OId -> Event [ctor] .
  op [_%_. reenter] : Int OId -> Event [ctor] .

  *** Event patterns
  sort EventPat .
  subsort Event < EventPat .
  subsort OId < EventPat .

  op new : -> EventPat [ctor] .
  op invoke : -> EventPat [ctor] .
  op reply : -> EventPat [ctor] .
  op initialized : -> EventPat [ctor] .
  op release : -> EventPat [ctor] .
  op reenter : -> EventPat [ctor] .
  op control : -> EventPat [ctor] .
  op [_-> *] : OId -> EventPat [ctor] .
  op [_<- *] : OId -> EventPat [ctor] .
  op [* ->_] : OId -> EventPat [ctor] .
  op [* <-_] : OId -> EventPat [ctor] .
  op [* ->_. new_[*]] : OId Id -> EventPat [ctor] .
  op [* -> _._[*]] : OId QualifiedId -> EventPat [ctor] .
  op [* <- _._[*]] : OId QualifiedId -> EventPat [ctor] .
  op [_%_->_. *] : Int OId OId -> EventPat [ctor] .
  op [_%_-> * . *] : Int OId -> EventPat [ctor] .
  op [_%_<-_. *] : Int OId OId -> EventPat [ctor] .
  op [_%_<- * . *] : Int OId -> EventPat [ctor] .
  op [_. reenter] : OId -> EventPat [ctor] .
  op in[_] : OId -> EventPat [ctor] .
  op out[_] : OId -> EventPat [ctor] .
  op ctl[_] : OId -> EventPat [ctor] .
  op ~_ : EventPat -> EventPat [ctor prec 3] .
  op _&_ : EventPat EventPat -> EventPat [ctor assoc comm prec 5] .
  op _|_ : EventPat EventPat -> EventPat [ctor assoc comm prec 7] .
```

```
*** auxiliary operator
op _matches_ : Event EventPat -> Bool [prec 11] .

*** Communication history (i.e., ^^-separated sequence of events)
sort History .
subsort Event < History .

op emptyHistory : -> History [ctor] .
op _^^_ : History History -> History
                                [ctor assoc prec 5 id: emptyHistory] .

*** auxiliary operators (projection, sequence length, membership)
op _/_ : History EventPat -> History [prec 9] .
op #[_] : History -> Int .
op _in_ : EventPat History -> Bool [prec 11] .

*** History pattern
sort HistoryPat .
subsort History < HistoryPat .
subsort EventPat < HistoryPat .
subsort HistoryPat < Value .

op _^^_ : HistoryPat HistoryPat -> HistoryPat [ctor ditto] .

*** auxiliary operators (begins with, ends with, prefix)
op _bw_ : History HistoryPat -> Bool [prec 11] .
op _ew_ : History HistoryPat -> Bool [prec 11] .
op _pr_ : HistoryPat History -> Bool [prec 11] .

var C : Id .
var EV : Event .
vars EVPAT EVPAT1 EVPAT2 : EventPat .
vars H H' : History .
var HPAT : HistoryPat .
var K : Int .
var M : Id .
vars O O' : OId .
var VL : ValueList .
var WL : ValueList .

eq defaultValue(history) = emptyHistory .

eq typeOf(H) = history .

eq EV matches EV = true .
eq EV matches O = EV matches (in[O] | out[O] | ctl[O]) .

eq [O -> O' . new C[VL]] matches new = true .
eq [K % O -> O' . M @ C[VL]] matches invoke = true .
eq [K % O <- O' . M @ C[VL ; WL]] matches reply = true .
eq [O . initialized] matches initialized = true .
eq [O . release] matches release = true .
eq [K % O . reenter] matches reenter = true .
eq EV matches control = EV matches (initialized | release | reenter) .
```

```
  eq [O -> O' . new C[VL]] matches [O -> *] = true .
  eq [K % O -> O' . M @ C[VL]] matches [O -> *] = true .
  eq [K % O <- O' . M @ C[VL ; WL]] matches [O <- *] = true .
  eq [O -> O' . new C[VL]] matches [* -> O'] = true .
  eq [K % O -> O' . M @ C[VL]] matches [* -> O'] = true .
  eq [K % O <- O' . M @ C[VL ; WL]] matches [* <- O'] = true .
  eq [O -> O' . new C[VL]] matches [* -> O' . new C[*]] = true .
  eq [K % O -> O' . M @ C[VL]] matches [* -> O' . M @ C[*]] = true .
  eq [K % O <- O' . M @ C[VL ; WL]] matches [* <- O' . M @ C[*]] = true .
  eq [K % O -> O' . M @ C[VL]] matches [K % O -> O' . *] = true .
  eq [K % O -> O' . M @ C[VL]] matches [K % O -> * . *] = true .
  eq [K % O <- O' . M @ C[VL ; WL]] matches [K % O <- O' . *] = true .
  eq [K % O <- O' . M @ C[VL ; WL]] matches [K % O <- * . *] = true .
  eq [K % O . reenter] matches [O . reenter] = true .
  eq EV matches in[O]  = EV matches ([* -> O] | [O <- *]) .
  eq EV matches out[O] = EV matches ([O -> *] | [* <- O]) .
  eq EV matches ctl[O] =
     EV matches ([O . initialized] | [O . release] | [O . reenter]) .

  eq EV matches ~ EVPAT = not EV matches EVPAT .
  eq EV matches EVPAT1 & EVPAT2 =
     EV matches EVPAT1 and EV matches EVPAT2 .
  eq EV matches EVPAT1 | EVPAT2 =
     EV matches EVPAT1 or EV matches EVPAT2 .

  eq EV matches EVPAT = false [otherwise] .

  eq emptyHistory / EVPAT = emptyHistory .
  eq H ^^ EV / EVPAT =
     (H / EVPAT) ^^ if EV matches EVPAT then EV else emptyHistory fi .

  eq #[emptyHistory] = 0 .
  eq #[H ^^ EV] = 1 + #[H] .

  eq EVPAT in emptyHistory = false .
  eq EVPAT in (H ^^ EV) = (EV matches EVPAT) or (EVPAT in H) .

  eq H bw emptyHistory = true .
  eq emptyHistory bw HPAT ^^ EVPAT = false .
  eq EV ^^ H bw EVPAT ^^ HPAT = (EV matches EVPAT) and (H bw HPAT) .

  eq H ew emptyHistory = true .
  eq emptyHistory ew H ^^ EVPAT = false .
  eq H ^^ EV ew H' ^^ EVPAT = (EV matches EVPAT) and (H ew H') .

  eq emptyHistory pr H = true .
  eq HPAT ^^ EVPAT pr emptyHistory = false .
  eq EVPAT ^^ HPAT pr EV ^^ H = (EV matches EVPAT) and (H bw HPAT) .
endfm

fmod CREOL-HISTORY-EXPRESSION is
  including CREOL-EXPRESSION .
  including CREOL-HISTORY .

  *** Event expression
```

```
sort EventExp .
subsort BasicExp < EventExp .
subsort Event < EventExp .
subsort EventTypedId < EventExp .

op [_->_. new_[_]] : OExp OExp Id ExpList -> EventExp [ctor] .
op [_%_->_._[_]] : AExp OExp OExp QualifiedId ExpList -> EventExp
                                                              [ctor] .
op [_%_<-_._[_;_]] :
   AExp OExp OExp QualifiedId ExpList ExpList -> EventExp [ctor] .
op [_. initialized] : OExp -> EventExp [ctor] .
op [_. release] : OExp -> EventExp [ctor] .
op [_%_. reenter] : AExp OExp -> EventExp [ctor] .
op if_th_el_fi : BExp EventExp EventExp -> EventExp [ctor] .
op [_] : EventExp -> EventExp [ctor] .

*** auxiliary operators
op _must match_ : EventExp EventPatExp -> Bool .
op _cannot match_ : EventExp EventPatExp -> Bool .

*** Event pattern expression
sort EventPatExp .
subsort EventExp < EventPatExp .
subsort OExp < EventPatExp .
subsort EventPat < EventPatExp .

op [_-> *] : OExp -> EventPatExp [ctor] .
op [_<- *] : OExp -> EventPatExp [ctor] .
op [* ->_] : OExp -> EventPatExp [ctor] .
op [* <-_] : OExp -> EventPatExp [ctor] .
op [* ->_. new_[*]] : OExp Id -> EventPatExp [ctor] .
op [* -> _._[*]] : OExp QualifiedId -> EventPat [ctor] .
op [* <- _._[*]] : OExp QualifiedId -> EventPat [ctor] .
op [_%_->_. *] : AExp OExp OExp -> EventPatExp [ctor] .
op [_%_-> * . *] : AExp OExp -> EventPatExp [ctor] .
op [_%_<-_. *] : AExp OExp OExp -> EventPatExp [ctor] .
op [_%_<- * . *] : AExp OExp -> EventPat [ctor] .
op [_. reenter] : AExp -> EventPatExp [ctor] .
op in[_] : OExp -> EventPatExp [ctor] .
op out[_] : OExp -> EventPatExp [ctor] .
op ctl[_] : OExp -> EventPatExp [ctor] .
op ~_ : EventPatExp -> EventPatExp [ctor ditto] .
op _&_ : EventPatExp EventPatExp -> EventPatExp [ctor ditto] .
op _|_ : EventPatExp EventPatExp -> EventPatExp [ctor ditto] .
op if_th_el_fi : BExp EventPatExp EventPatExp -> EventPatExp [ctor] .
op [_] : EventPatExp -> EventPatExp [ctor] .

*** History expression
sort HistoryExp .
subsort EventExp < HistoryExp .
subsort History < HistoryExp .
subsort HistoryTypedId < HistoryExp .

op _^^_ : HistoryExp HistoryExp -> HistoryExp [ctor ditto] .
op if_th_el_fi : BExp HistoryExp HistoryExp -> HistoryExp [ctor] .
```

```
op [_] : HistoryExp -> HistoryExp [ctor] .

*** expressions involving histories
op _/_ : HistoryExp EventPatExp -> HistoryExp [ctor ditto] .
op #[_] : HistoryExp -> AExp [ctor ditto] .
op _eq_ : HistoryExp HistoryExp -> BasicAssn [ctor ditto] .
op _ne_ : HistoryExp HistoryExp -> BasicAssn [ctor ditto] .
op _in_ : EventPatExp HistoryExp -> BasicAssn [ctor ditto] .

*** History pattern expression
sort HistoryPatExp .
subsort HistoryPat < HistoryPatExp .
subsort HistoryExp < HistoryPatExp .
subsort EventPatExp < HistoryPatExp .
subsort HistoryPatExp < Exp .

op _^^_ : HistoryPatExp HistoryPatExp -> HistoryPatExp [ctor ditto] .
op if_th_el_fi :
   BExp HistoryPatExp HistoryPatExp -> HistoryPatExp [ctor] .
op [_] : HistoryPatExp -> HistoryPatExp [ctor] .

*** expressions involving history patterns
op _bw_ : HistoryExp HistoryPatExp -> BasicAssn [ctor ditto] .
op _ew_ : HistoryExp HistoryPatExp -> BasicAssn [ctor ditto] .
op _pr_ : HistoryPatExp HistoryExp -> BasicAssn [ctor ditto] .

vars A A1 A2 : AExp .
vars C C1 C2 : Id .
var EEXP : EventExp .
vars EL EL' EL1 EL1' EL2 EL2' : ExpList .
vars EPEXP EPEXP1 EPEXP2 : EventPatExp .
var HEXP : HistoryExp .
var HPEXP : HistoryPatExp .
vars M M1 M2 : Id .
vars OEXP OEXP' OEXP1 OEXP1' OEXP2 OEXP2' : OExp .
var RHO : Subst .

eq EEXP must match EEXP = true .
eq [OEXP -> OEXP' . new C[EL]] must match new = true .
eq [A % OEXP -> OEXP' . M @ C[EL]] must match invoke = true .
eq [A % OEXP <- OEXP' . M @ C[EL ; EL']] must match reply = true .
eq [OEXP . initialized] must match initialized = true .
eq [OEXP . release] must match release = true .
eq [A % OEXP . reenter] must match reenter = true .
eq EEXP must match control =
   EEXP must match (initialized | release | reenter) .
eq [OEXP -> OEXP' . new C[EL]] must match [OEXP -> *] = true .
eq [A % OEXP -> OEXP' . M @ C[EL]] must match [OEXP -> *] = true .
eq [A % OEXP <- OEXP' . M @ C[EL ; EL']] must match [OEXP <- *] =
   true .
eq [OEXP -> OEXP' . new C[EL]] must match [* -> OEXP'] = true .
eq [A % OEXP -> OEXP' . M @ C[EL]] must match [* -> OEXP'] = true .
eq [A % OEXP <- OEXP' . M @ C[EL ; EL']] must match [* <- OEXP'] =
   true .
eq [OEXP -> OEXP' . new C[EL]] must match [* -> OEXP' . new C[*]] =
```

```
   true .
eq [A % OEXP -> OEXP' . M @ C[EL]] must match [* -> OEXP' . M @ C[*]] =
   true .
eq [A % OEXP <- OEXP' . M @ C[EL ; EL']] must match
   [* <- OEXP' . M @ C[*]] =
   true .
eq [A % OEXP -> OEXP' . M @ C[EL]] must match [A % OEXP -> OEXP' . *] =
   true .
eq [A % OEXP <- OEXP' . M @ C[EL ; EL']] must match
   [A % OEXP <- OEXP' . *] =
   true .
eq [A % OEXP <- OEXP' . M @ C[EL ; EL']] must match
   [A % OEXP <- * . *] =
   true .
eq [A % OEXP . reenter] must match [OEXP . reenter] = true .
eq [OEXP -> OEXP' . new C[EL]] must match in[OEXP'] = true .
eq [A % OEXP -> OEXP' . M @ C[EL]] must match in[OEXP'] = true .
eq [A % OEXP <- OEXP' . M @ C[EL ; EL']] must match in[OEXP] = true .
eq [OEXP -> OEXP' . new C[EL]] must match out[OEXP] = true .
eq [A % OEXP -> OEXP' . M @ C[EL]] must match out[OEXP] = true .
eq [A % OEXP <- OEXP' . M @ C[EL ; EL']] must match out[OEXP'] = true .
eq [OEXP . initialized] must match ctl[OEXP] = true .
eq [OEXP . release] must match ctl[OEXP] = true .
eq [A % OEXP . reenter] must match ctl[OEXP] = true .
eq EEXP must match ~ EPEXP = EEXP cannot match EPEXP .
eq EEXP must match EPEXP1 & EPEXP2 =
   EEXP must match EPEXP1 and EEXP must match EPEXP2 .
eq EEXP must match EPEXP1 | EPEXP2 =
   EEXP must match EPEXP1 or EEXP must match EPEXP2 .
eq EEXP must match EPEXP = false [otherwise] .

eq EEXP cannot match new = EEXP must match (invoke | reply | control) .
eq EEXP cannot match invoke = EEXP must match (new | reply | control) .
eq EEXP cannot match reply = EEXP must match (new | invoke | control) .
eq EEXP cannot match initialized =
   EEXP must match (new | invoke | reply | release | reenter) .
eq EEXP cannot match release =
   EEXP must match (new | invoke | reply | initialized | reenter) .
eq EEXP cannot match reenter =
   EEXP must match (new | invoke | reply | initialized | release) .
eq EEXP cannot match control =
   EEXP must match (new | invoke | reply) .
eq EEXP cannot match [OEXP -> OEXP' . new C[EL]] =
   EEXP cannot match new .
eq [A1 % OEXP1 -> OEXP1' . M1 @ C1[EL1]] cannot match
   [A2 % OEXP2 -> OEXP2' . M2 @ C2[EL2]] =
   M1 =/= M2 or C1 =/= C2 .
ceq EEXP cannot match [A % OEXP -> OEXP' . M @ C[EL]] = true
if EEXP cannot match invoke .
eq [A1 % OEXP1 <- OEXP1' . M1 @ C1[EL1 ; EL1']] cannot match
   [A2 % OEXP2 <- OEXP2' . M2 @ C2[EL2 ; EL2']] =
   M1 =/= M2 or C1 =/= C2 .
ceq EEXP cannot match [A % OEXP <- OEXP' . M @ C[EL ; EL']] = true
if EEXP cannot match reply .
eq EEXP cannot match [OEXP . initialized] =
```

```
   EEXP cannot match initialized .
eq EEXP cannot match [OEXP . release] = EEXP cannot match release .
eq EEXP cannot match [A % OEXP . reenter] = EEXP cannot match reenter .
eq EEXP cannot match [OEXP -> *] = EEXP cannot match (new | invoke) .
eq EEXP cannot match [OEXP <- *] = EEXP cannot match reply .
eq EEXP cannot match [* -> OEXP] = EEXP cannot match (new | invoke) .
eq EEXP cannot match [* <- OEXP] = EEXP cannot match reply .
eq [OEXP1 -> OEXP1' . new C1[EL1]] cannot match
   [* -> OEXP' . new C2[*]] =
   C1 =/= C2 .
ceq EEXP cannot match [* -> OEXP' . new C[*]] = true
if EEXP cannot match new .
eq [A1 % OEXP1 -> OEXP1' . M1 @ C1[EL1]] cannot match
   [* -> OEXP2' . M2 @ C2[*]] =
   M1 =/= M2 or C1 =/= C2 .
ceq EEXP cannot match [* -> OEXP' . M @ C[*]] = true
if EEXP cannot match invoke .
eq [A1 % OEXP1 <- OEXP1' . M1 @ C1[EL1 ; EL1']] cannot match
   [* <- OEXP2' . M2 @ C2[*]] =
   M1 =/= M2 or C1 =/= C2 .
ceq EEXP cannot match [* <- OEXP' . M @ C[*]] = true
if EEXP cannot match reply .
eq EEXP cannot match [A % OEXP -> OEXP' . *] =
   EEXP cannot match invoke .
eq EEXP cannot match [A % OEXP -> * . *] = EEXP cannot match invoke .
eq EEXP cannot match [A % OEXP <- OEXP' . *] =
   EEXP cannot match reply .
eq EEXP cannot match [A % OEXP <- * . *] = EEXP cannot match reply .
eq EEXP cannot match [OEXP . reenter] = EEXP cannot match reenter .
eq EEXP cannot match in[OEXP] = EEXP must match control .
eq EEXP cannot match out[OEXP] = EEXP must match control .
eq EEXP cannot match ctl[OEXP] = EEXP cannot match control .
eq EEXP cannot match ~ EPEXP = EEXP must match EPEXP .
eq EEXP cannot match EPEXP1 & EPEXP2 =
   EEXP cannot match EPEXP1 or EEXP cannot match EPEXP2 .
eq EEXP cannot match EPEXP1 | EPEXP2 =
   EEXP cannot match EPEXP1 and EEXP cannot match EPEXP2 .
eq EEXP cannot match EPEXP = false [otherwise] .

eq ([OEXP -> OEXP' . new C[EL]]) RHO =
   [(OEXP) RHO -> (OEXP') RHO . new C[(EL) RHO]] .
eq ([A % OEXP -> OEXP' . M @ C[EL]]) RHO =
   [(A) RHO % (OEXP) RHO -> (OEXP') RHO . M @ C[(EL) RHO]] .
eq ([A % OEXP <- OEXP' . M @ C[EL ; EL']]) RHO =
   [(A) RHO % (OEXP) RHO <- (OEXP') RHO
    . M @ C[(EL) RHO ; (EL') RHO]] .
eq ([OEXP . initialized]) RHO = [(OEXP) RHO . initialized] .
eq ([OEXP . release]) RHO = [(OEXP) RHO . release] .
eq ([A % OEXP . reenter]) RHO = [(A) RHO % (OEXP) RHO . reenter] .

eq (new) RHO = new .
eq (invoke) RHO = invoke .
eq (reply) RHO = reply .
eq (initialized) RHO = initialized .
eq (release) RHO = release .
```

```
  eq (reenter) RHO = reenter .
  eq (control) RHO = control .
  eq ([OEXP -> *]) RHO = [(OEXP) RHO -> *] .
  eq ([OEXP <- *]) RHO = [(OEXP) RHO <- *] .
  eq ([* -> OEXP]) RHO = [* -> (OEXP) RHO] .
  eq ([* <- OEXP]) RHO = [* <- (OEXP) RHO] .
  eq ([* -> OEXP . new C[*]]) RHO = [* -> (OEXP) RHO . new C[*]] .
  eq ([* -> OEXP . M @ C[*]]) RHO = [* -> (OEXP) RHO . M @ C[*]] .
  eq ([* <- OEXP . M @ C[*]]) RHO = [* <- (OEXP) RHO . M @ C[*]] .
  eq ([A % OEXP -> OEXP' . *]) RHO =
     [(A) RHO % (OEXP) RHO -> (OEXP') RHO . *] .
  eq ([A % OEXP -> * . *]) RHO = [(A) RHO % (OEXP) RHO -> * . *] .
  eq ([A % OEXP <- OEXP' . *]) RHO =
     [(A) RHO % (OEXP) RHO <- (OEXP') RHO . *] .
  eq ([A % OEXP <- * . *]) RHO = [(A) RHO % (OEXP) RHO <- * . *] .
  eq ([OEXP . reenter]) RHO = [(OEXP) RHO . reenter] .
  eq (in[OEXP]) RHO = in[(OEXP) RHO] .
  eq (out[OEXP]) RHO = out[(OEXP) RHO] .
  eq (ctl[OEXP]) RHO = ctl[(OEXP) RHO] .

  eq (~ EPEXP) RHO = ~ (EPEXP) RHO .
  eq (EPEXP1 & EPEXP2) RHO = (EPEXP1) RHO & (EPEXP2) RHO .
  eq (EPEXP1 | EPEXP2) RHO = (EPEXP1) RHO | (EPEXP2) RHO .

  eq (emptyHistory) RHO = emptyHistory .
  ceq (HPEXP ^^ EPEXP) RHO = (HPEXP) RHO ^^ (EPEXP) RHO
  if HPEXP =/= emptyHistory .

  eq (HEXP / EPEXP) RHO = (HEXP) RHO / (EPEXP) RHO .
  eq (#[HEXP]) RHO = #[(HEXP) RHO] .
  eq (EPEXP in HEXP) RHO = (EPEXP) RHO in (HEXP) RHO .

  eq (HEXP bw HPEXP) RHO = (HEXP) RHO bw (HPEXP) RHO .
  eq (HEXP bw HPEXP) RHO = (HEXP) RHO bw (HPEXP) RHO .

  eq (HEXP ew HPEXP) RHO = (HEXP) RHO ew (HPEXP) RHO .
  eq (HEXP ew HPEXP) RHO = (HEXP) RHO ew (HPEXP) RHO .

  eq (HPEXP pr HEXP) RHO = (HPEXP) RHO pr (HEXP) RHO .
  eq (HPEXP pr HEXP) RHO = (HPEXP) RHO pr (HEXP) RHO .
endfm

fmod CREOL-GUARD is
  including CREOL-EXPRESSION .

  *** Conditional guard
  sort Guard .
  subsort BExp < Guard .

  op _? : Id -> Guard [ctor prec 13] .
  op wait : -> Guard [ctor] .
  op _&&&_ : Guard Guard -> Guard [ctor assoc comm prec 15] .

  *** application of a substitution on a conditional guard
  op __ : Guard Subst -> Guard [ditto] .
```

```
  vars G1 G2 : Guard .
  var LL : TypedId .
  var RHO : Subst .

  eq (LL ?) RHO = LL ? .
  eq (wait) RHO = wait .
  eq (G1 &&& G2) RHO = (G1) RHO &&& (G2) RHO .
endfm

fmod CREOL-STATEMENT is
  including CREOL-GUARD .
  including CREOL-TYPED-IDENTIFIER .

  *** Argument list to synchronous call
  sort SyncCallArgs .

  *** canonical argument list
  op [_;_] : ExpList TypedQualifiedIdList -> SyncCallArgs [ctor] .

  *** non-canonical argument lists
  op [_;] : ExpList -> SyncCallArgs .
  op [;_] : TypedQualifiedIdList -> SyncCallArgs .
  op [;] : -> SyncCallArgs .
  op [] : -> SyncCallArgs .

  *** Statement that may serve as an await guard
  sort SingleStmtAllowedInAwait .

  *** Statement other than a statement list
  sort SingleStmt .
  subsort SingleStmtAllowedInAwait < SingleStmt .

  *** Any statement
  sort Stmt .
  subsort SingleStmt < Stmt .

  *** canonical statements
  op skip : -> SingleStmt [ctor] .
  op abort : -> SingleStmt [ctor] .
  op prove_ : Assn -> SingleStmt [ctor prec 23] .
  op _:=_ : TypedQualifiedIdList ExpList -> SingleStmt [ctor prec 23] .
  op _:= new_ : TypedQualifiedId IdWithArgs -> SingleStmt
                                                    [ctor prec 23] .
  op _!_._[_] : TypedId OExp Id ExpList -> SingleStmt [ctor prec 5] .
  op _!_[_] : TypedId TypedQualifiedId ExpList -> SingleStmt
                                                     [ctor prec 5] .
  op _?[_] : TypedId TypedQualifiedIdList -> SingleStmtAllowedInAwait
                                                     [ctor prec 5] .

  op await_ : Guard -> SingleStmt [ctor prec 19] .
  op if_th_el_fi : BExp Stmt Stmt -> SingleStmt [ctor] .
  op while_do_od : BExp Stmt -> SingleStmt [ctor] .
  op inv_while_do_od : Assn BExp Stmt -> SingleStmt [ctor] .
  op _[]_ : Stmt Stmt -> SingleStmt
```

```
                          [ctor assoc comm prec 27 format (d s d s d)] .
op _|||_ : Stmt Stmt -> SingleStmt [ctor assoc comm prec 29] .
op [_] : Stmt -> SingleStmt [ctor] .

*** synthetic statements
op !_._[_] : OExp Id ExpList -> SingleStmt [ctor] .
op !_[_] : QualifiedId ExpList -> SingleStmt [ctor] .
op _.__ : OExp Id SyncCallArgs -> SingleStmtAllowedInAwait
                                                        [ctor prec 1] .
op __ : QualifiedId SyncCallArgs -> SingleStmtAllowedInAwait
                                                        [ctor prec 1] .
op await_&&&_ : Guard SingleStmtAllowedInAwait -> SingleStmt
                                                      [ctor prec 19] .
op await_ : SingleStmtAllowedInAwait -> SingleStmt [ctor prec 19] .
op if_th_fi : BExp Stmt -> SingleStmt [ctor] .

*** statements with empty argument list
op _!_._[] : TypedId OExp Id -> SingleStmt [prec 5] .
op _!_[] : TypedId QualifiedId -> SingleStmt [prec 5] .
op !_._[] : OExp Id -> SingleStmt .
op !_[] : QualifiedId -> SingleStmt .
op _?[] : TypedId -> SingleStmtAllowedInAwait [prec 5] .

*** sequence of statements
op emptyStmt : -> Stmt [ctor] .
op _;_ : Stmt Stmt -> Stmt [ctor assoc prec 25 id: emptyStmt] .

*** application of a substitution on a statement
op __ : Stmt Subst -> Stmt [prec 1] .

*** other operators
op qualified : Stmt TypedIdList Id -> Stmt .
op clearWait : Stmt -> Stmt .
op cleared : Guard -> Guard .
op simpleBranch : Stmt -> Bool .

var B : BExp .
var C : Id .
var EL : ExpList .
vars G G1 G2 : Guard .
var LL : TypedId .
var M : Id .
var OEXP : OExp .
var PHI : Assn .
var RHO : Subst .
vars S S' S1 S2 S1..SK SK+1..SN : Stmt .
var SS : SingleStmt .
var XXL : TypedIdList .
var ZZ : TypedQualifiedId .
var ZZL : TypedQualifiedIdList .

eq [EL ;] = [EL ; epsilon] .
eq [; ZZL] = [epsilon ; ZZL] .
eq [;] = [epsilon ; epsilon] .
eq [] = [epsilon ; epsilon] .
```

```
eq LL ! OEXP . M[] = LL ! OEXP . M[epsilon] .
eq LL ! M @ C[] = LL ! M @ C[epsilon] .
eq ! OEXP . M[] = ! OEXP . M[epsilon] .
eq ! M @ C[] = ! M @ C[epsilon] .
eq LL ?[] = LL ?[epsilon] .

eq (skip) RHO = skip .
eq (abort) RHO = abort .
eq (prove PHI) RHO = prove (PHI) RHO .
eq (ZZL := EL) RHO = (ZZL) RHO := (EL) RHO .
eq (ZZ := new C[EL]) RHO = (ZZ) RHO := new C[(EL) RHO] .
eq (LL ! OEXP . M[EL]) RHO = LL ! (OEXP) RHO . M[(EL) RHO] .
eq (LL ! M @ C[EL]) RHO = LL ! M @ C[(EL) RHO] .
eq (LL ?[ZZL]) RHO = LL ?[(ZZL) RHO] .
eq (await G) RHO = await (G) RHO .
eq (if B th S1 el S2 fi) RHO =
   if (B) RHO th (S1) RHO el (S2) RHO fi .
eq (while B do S od) RHO = while (B) RHO do (S) RHO od .
eq (inv PHI while B do S od) RHO =
   inv (PHI) RHO while (B) RHO do (S) RHO od .
eq (S1 [] S2) RHO = (S1) RHO [] (S2) RHO .
eq (S1..SK ||| SK+1..SN) RHO = (S1..SK) RHO ||| (SK+1..SN) RHO .
eq ([S]) RHO = [(S) RHO] .

eq (! OEXP . M[EL]) RHO = ! (OEXP) RHO . M[(EL) RHO] .
eq (! M @ C[EL]).Stmt RHO = (! M @ C[(EL) RHO]).Stmt .
eq (OEXP . M[EL ; ZZL]) RHO = (OEXP) RHO . M[(EL) RHO ; (ZZL) RHO] .
eq (M @ C[EL ; ZZL]) RHO = M @ C[(EL) RHO ; (ZZL) RHO] .
eq (await G &&& SS) RHO = await (G) RHO &&& (SS) RHO .
eq (await SS) RHO = await (SS) RHO .
eq (if B th S1 fi) RHO = if (B) RHO th (S1) RHO fi .

eq (emptyStmt) RHO = emptyStmt .
ceq (SS ; S) RHO = (SS) RHO ; (S) RHO if S =/= emptyStmt .

eq qualified(S, XXL, C) = (S) { XXL |-> XXL @@ C } .

*** Definition T9 (Wait Guard Clearer)
*** slightly adapted to cover synthetic await statements properly
eq clearWait(await G ; S) = await cleared(G) ; S .
eq clearWait(await G &&& SS ; S) = await cleared(G) &&& SS ; S .
eq clearWait((S1 [] S2) ; S) = (clearWait(S1) [] clearWait(S2)) ; S .
eq clearWait((S1..SK ||| SK+1..SN) ; S) =
   (clearWait(S1..SK) ||| clearWait(SK+1..SN)) ; S .
eq clearWait([S] ; S') = [clearWait(S)] ; S' .
eq clearWait(S) = S [otherwise] .

*** Definition T10 (Cleared Guard)
eq cleared(wait) = true .
eq cleared(G1 &&& G2) = cleared(G1) &&& cleared(G2) .
eq cleared(G) = G [otherwise] .

eq simpleBranch(S1..SK ||| SK+1..SN) = false .
eq simpleBranch(S) = true [otherwise] .
```

```
endfm

fmod CREOL-METHOD is
  including CREOL-STATEMENT .

  *** Creol method
  sort Mtd .

  op <_: Method | In:_, Out:_, LVar:_, Code:_> :
     Id TypedIdList TypedIdList TypedIdList Stmt -> Mtd
              [ctor format (c c! oc c c sc! oc c sc! oc c sc! oc c sc! oc
                            c n)] .

  *** Multiset of methods
  sort MtdMSet .
  subsort Mtd < MtdMSet .
  subsort Mtd < MSetElem .
  subsort MtdMSet < MSet .
  subsort EmptyMSet < MtdMSet .

  op _++_ : MtdMSet MtdMSet -> MtdMSet [ctor ditto] .

  *** method multiset membership
  op <_: Method | * > in mset_ : Id MtdMSet -> Bool [prec 11] .

  vars M M' : Id .
  var MM : MtdMSet .
  var S : Stmt .
  var VVL : TypedIdList .
  var XXL : TypedIdList .
  var YYL : TypedIdList .

  eq < M : Method | * > in mset emptyMSet = false .
  eq < M : Method | * > in mset
     < M' : Method | In: XXL, Out: YYL, LVar: VVL, Code: S > ++ MM =
     (M == M') or (< M : Method | * > in mset MM) .
endfm

fmod CREOL-SUPER is
  including CREOL-STATEMENT .

  *** Creol supertype in interface or class declaration
  sort Super .
  subsort IdWithArgs < Super .

  *** Comma-separated list of supertypes
  sort SuperList .
  subsort Super < SuperList .
  subsort Super < ListElem .
  subsort SuperList < ExpList .
  subsort IdList < SuperList .

  op _,_ : SuperList SuperList -> SuperList [ctor ditto] .

  *** auxiliary function
```

```
  op qualifiedSuperList : SuperList TypedIdList Id -> SuperList .

  vars C C' : Id .
  var EL : ExpList .
  var SUPERL : SuperList .
  var XXL : TypedIdList .

  eq qualifiedSuperList(epsilon, XXL, C) = epsilon .
  eq qualifiedSuperList((C'[EL], SUPERL), XXL, C) =
     C'[EL {XXL |-> XXL @@ C}], qualifiedSuperList(SUPERL, XXL, C) .
endfm

fmod CREOL-SIGNATURE is
  including CREOL-TYPED-IDENTIFIER .

  *** Creol method signature preceded by the op keyword
  sort OpSig .

  *** canonical form of method signature
  op op_[in_out_] : Id TypedIdList TypedIdList -> OpSig
                                      [ctor format (ss d d d d d d d d)] .

  *** non-canonical forms
  op op_[in_] : Id TypedIdList -> OpSig .
  op op_[out_] : Id TypedIdList -> OpSig .
  op op_[] : Id -> OpSig .
  op op_ : Id -> OpSig [prec 1] .

  var M : Id .
  var XXL : TypedIdList .
  var YYL : TypedIdList .

  eq op M[in XXL] = op M[in XXL out epsilon] .
  eq op M[out YYL] = op M[in epsilon out YYL] .
  eq op M[] = op M[in epsilon out epsilon] .
  eq op M = op M[in epsilon out epsilon] .
endfm

fmod CREOL-INTERFACE is
  including CREOL-HISTORY-EXPRESSION .
  including CREOL-METHOD .
  including CREOL-SIGNATURE .
  including CREOL-SUPER .

  *** Clause in a Creol interface declaration head
  sort InterfaceHeadClause .

  op inherits_ : SuperList -> InterfaceHeadClause [ctor prec 23] .

  *** Interface declaration head
  *** e.g., interface 'I['a] inherits 'J, 'K['a]
  sort InterfaceHead .

  op interface_ : IdWithParams -> InterfaceHead [ctor prec 3] .
  op __ : InterfaceHead InterfaceHeadClause -> InterfaceHead
```

```
                                                        [ctor prec 25] .

*** Interface or class declaration tail
sort Tail .

op asum_guar_end : Assn Assn -> Tail [ctor] .

op asum_end : Assn -> Tail .
op guar_end : Assn -> Tail .
op end : -> Tail .

*** Interface declaration body
sort InterfaceMtds .
subsort OpSig < InterfaceMtds .

op noMtds : -> InterfaceMtds [ctor format (ss d)] .
op __ : InterfaceMtds InterfaceMtds -> InterfaceMtds
                  [ctor assoc comm prec 25 id: noMtds format (d n d)] .

*** Creol interface
sort Interface .

*** canonical representation of interfaces
op <_: Interface | Inh:_, Param:_, Asum:_, Guar:_> :
   Id SuperList TypedIdList Assn Assn -> Interface
           [ctor format (m m! om m m m! om m sm! om m sm! om m sm! om m
                         n)] .

*** interface declaration syntax
op _begin with_:__ : InterfaceHead Type InterfaceMtds Tail -> Interface
                                                          [prec 27] .
op _begin_ : InterfaceHead Tail -> Interface [prec 27] .

*** auxiliary functions
op interfaceName : InterfaceHead -> Id .
op interfaceParams : InterfaceHead -> TypedIdList .
op interfaceInherits : InterfaceHead -> SuperList .

var ASUM : Assn .
var GUAR : Assn .
var I : Id .
var IHEAD : InterfaceHead .
var IHEADCLAUSE : InterfaceHeadClause .
var IMTDS : InterfaceMtds .
var SUPERL : SuperList .
var T : Type .
var XXL : TypedIdList .

eq asum ASUM end = asum ASUM guar true end .
eq guar GUAR end = asum true guar GUAR end .
eq end = asum true guar true end .

*** Definition T1' (Interface Declaration)
eq IHEAD begin with T : IMTDS asum ASUM guar GUAR end =
   < interfaceName(IHEAD) : Interface |
```

```
        Inh: interfaceInherits(IHEAD), Param: interfaceParams(IHEAD),
        Asum: ASUM, Guar: GUAR > .

  eq IHEAD begin end = IHEAD begin with any : noMtds end .

  eq interfaceName(interface I[XXL]) = I .
  eq interfaceName(IHEAD IHEADCLAUSE) = interfaceName(IHEAD) .

  eq interfaceParams(interface I[XXL]) = XXL .
  eq interfaceParams(IHEAD IHEADCLAUSE) = interfaceParams(IHEAD) .

  eq interfaceInherits(interface I[XXL]) = epsilon .
  eq interfaceInherits(IHEAD inherits SUPERL) =
     interfaceInherits(IHEAD), SUPERL .
endfm

fmod CREOL-CLASS is
  including CREOL-INTERFACE .
  including CREOL-METHOD .

  *** Clause in a class declaration head
  sort ClassHeadClause .
  subsort InterfaceHeadClause < ClassHeadClause .

  op implements_ : SuperList -> ClassHeadClause [ctor prec 23] .
  op contracts_ : SuperList -> ClassHeadClause [ctor prec 23] .

  *** Class declaration head
  *** e.g., class 'C['a] implements 'J contracts 'K['a] inherits 'B
  sort ClassHead .

  op class_ : IdWithParams -> ClassHead [ctor prec 3] .
  op __ : ClassHead ClassHeadClause -> ClassHead [ctor prec 25] .

  *** Class attribute declaration
  sort VarClause .

  op var_ : TypedIdList -> VarClause [ctor prec 23] .

  *** Method body
  *** e.g., var 'i : int ; 'i := x plus 1 ; 'y := 'i
  sort MtdBody .
  subsort Stmt < MtdBody .

  op _;_ : VarClause Stmt -> MtdBody
                                  [ctor prec 27 left id: (var epsilon)] .

  *** Method declaration
  *** e.g., op 'init is 'pi := 31416
  sort MtdDecl .

  op _is_ : OpSig MtdBody -> MtdDecl [ctor prec 29] .

  *** Group of method declarations
  sort MtdDeclGroup .
```

```
subsort MtdDecl < MtdDeclGroup .

op noMtds : -> MtdDeclGroup [ctor format (ss d)] .
op __ : MtdDeclGroup MtdDeclGroup -> MtdDeclGroup
                  [ctor assoc comm prec 31 id: noMtds format (d n d)] .

*** All method declarations in a class
sort ClassMtds .
subsort MtdDeclGroup < ClassMtds .

op _with_:_ : MtdDeclGroup Type ClassMtds -> ClassMtds [ctor prec 33] .

*** Class declaration body
sort ClassBody .
subsort ClassMtds < ClassBody .

op __ : VarClause ClassMtds -> ClassBody
                                [ctor prec 35 left id: (var epsilon)] .

*** Creol class
sort Class .

*** canonical representation of classes
op <_: Class | Impl:_, Ctrc:_, Inh:_, Param:_, Att:_, Mtd:_, ObjCnt:_,
               Asum:_, Guar:_> :
   Id SuperList SuperList SuperList TypedIdList TypedIdList MtdMSet Int
   Assn Assn
   -> Class
          [ctor format (b b! ob b b b! ob b sb! ob b sb! ob b sb! ob b
                        sb! ob b sb! onssssb ssb sb! ob b sb! ob b sb!
                        ob b n)] .

*** class declaration syntax
op _begin__ : ClassHead ClassBody Tail -> Class [prec 37] .
op _begin_with_:__ : ClassHead VarClause Type ClassMtds Tail -> Class
                                                          [prec 37] .
op _begin with_:__ : ClassHead Type ClassMtds Tail -> Class [prec 37] .
op _begin__ : ClassHead VarClause Tail -> Class [prec 37] .
op _begin_ : ClassHead Tail -> Class [prec 37] .

*** auxiliary functions
op className : ClassHead -> Id .
op classParams : ClassHead -> TypedIdList .
op classImplements : ClassHead -> SuperList .
op classContracts : ClassHead -> SuperList .
op classInherits : ClassHead -> SuperList .
op classMethods : ClassMtds -> MtdMSet .
op labels : Stmt -> TypedIdList .

var ASUM : Assn .
var C : Id .
var CHEAD : ClassHead .
var CHEADCLAUSE : ClassHeadClause .
var CMTDS : ClassMtds .
var EL : ExpList .
```

```
var GUAR : Assn .
var L : Id .
var M : Id .
vars MGRP MGRP1 MGRP2 : MtdDeclGroup .
var OEXP : OExp .
var S : Stmt .
var SS : SingleStmt .
var SUPERL : SuperList .
var T : Type .
var TAIL : Tail .
var VVL : TypedIdList .
var WWL : TypedIdList .
var XXL : TypedIdList .
var YYL : TypedIdList .
var ZZL : TypedQualifiedIdList .

*** Definition T2' (Class Declaration)
eq CHEAD begin var WWL CMTDS asum ASUM guar GUAR end =
   < className(CHEAD) : Class | Impl: classImplements(CHEAD),
       Ctrc: classContracts(CHEAD), Inh: classInherits(CHEAD),
       Param: classParams(CHEAD), Att: WWL, Mtd: classMethods(CMTDS),
       ObjCnt: 0, Asum: ASUM, Guar: GUAR > .

eq CHEAD begin var WWL with T : CMTDS TAIL =
   CHEAD begin var WWL noMtds with T : CMTDS TAIL .
eq CHEAD begin with T : CMTDS TAIL =
   CHEAD begin noMtds with T : CMTDS TAIL .
eq CHEAD begin var WWL TAIL = CHEAD begin var WWL noMtds TAIL .
eq CHEAD begin TAIL = CHEAD begin noMtds TAIL .

eq className(class C[XXL]) = C .
eq className(CHEAD CHEADCLAUSE) = className(CHEAD) .

eq classParams(class C[XXL]) = XXL .
eq classParams(CHEAD CHEADCLAUSE) = classParams(CHEAD) .

eq classImplements(class C[XXL]) = epsilon .
eq classImplements(CHEAD implements SUPERL) =
   classImplements(CHEAD), SUPERL .
eq classImplements(CHEAD CHEADCLAUSE) = classImplements(CHEAD)
                                                        [otherwise] .

eq classContracts(class C[XXL]) = epsilon .
eq classContracts(CHEAD contracts SUPERL) =
   classContracts(CHEAD), SUPERL .
eq classContracts(CHEAD CHEADCLAUSE) = classContracts(CHEAD)
                                                        [otherwise] .

eq classInherits(class C[XXL]) = epsilon .
eq classInherits(CHEAD inherits SUPERL) =
   classInherits(CHEAD), SUPERL .
eq classInherits(CHEAD CHEADCLAUSE) = classInherits(CHEAD)
                                                        [otherwise] .

eq classMethods(op M[in XXL out YYL] is var VVL ; S) =
```

```
     < M : Method | In: XXL, Out: YYL, LVar: VVL, labels(S), Code: S > .
  eq classMethods(noMtds) = emptyMSet .
  eq classMethods(MGRP1 MGRP2) =
     classMethods(MGRP1) ++ classMethods(MGRP2) .
  eq classMethods(MGRP with T : CMTDS) =
     classMethods(MGRP) ++ classMethods(CMTDS) .

  eq labels(L : T ! OEXP . M[EL]) = L : int .
  eq labels(L : T ! M @ C[EL]) = L : int .
  eq labels(L : T ?[ZZL]) = L : int .
  eq labels(SS) = epsilon [otherwise] .

  eq labels(emptyStmt) = epsilon .
  ceq labels(SS ; S) = labels(SS), labels(S) if S =/= emptyStmt .
endfm

fmod CREOL-PROGRAM is
  including CREOL-CLASS .

  *** System subconfiguration (multiset of interfaces, classes,
  *** objects, and messages)
  sort Config .
  subsort Interface < Config .
  subsort Class < Config .

  op emptyConfig : -> Config [ctor] .
  op __ : Config Config -> Config [ctor assoc comm id: emptyConfig] .

  *** auxiliary functions
  op qualified : Stmt SuperList TypedIdList Config ~> Stmt .
  op qualifiedAndTyped :
     Stmt SuperList TypedQualifiedIdList TypedIdList Config ~> Stmt .
  op classParams : Id Config ~> TypedIdList .
  op initialPr : Super Config ~> Stmt .
  op initialPr : SuperList Stmt Stmt Stmt Config ~> Stmt .

  *** Global system configuration
  sort GlobalConfig .

  op {_} : Config -> GlobalConfig [ctor format (!w on !w o)] .

  var AAL : TypedQualifiedIdList .
  var ASUM : Assn .
  var C : Id .
  var CONFIG : Config .
  var EL : ExpList .
  var GUAR : Assn .
  var MM : MtdMSet .
  var N : Int .
  var PPL : TypedIdList .
  vars S S1 S2 S3 : Stmt .
  vars SUPERL SUPERL' SUPERL'' SUPERL''' : SuperList .
  var VVL : TypedIdList .
  var WWL : TypedIdList .
  var XXL : TypedIdList .
```

```
  eq qualified(S, epsilon, XXL, CONFIG) = S .
  eq qualified(S, (SUPERL''', C[EL]), XXL,
               < C : Class | Impl: SUPERL, Ctrc: SUPERL', Inh: SUPERL'',
                             Param: PPL, Att: WWL, Mtd: MM, ObjCnt: N,
                             Asum: ASUM, Guar: GUAR >
               CONFIG) =
     qualified(qualified(S, (PPL, WWL) without XXL, C),
               (SUPERL''', SUPERL''), XXL,
               < C : Class | Impl: SUPERL, Ctrc: SUPERL', Inh: SUPERL'',
                             Param: PPL, Att: WWL, Mtd: MM, ObjCnt: N,
                             Asum: ASUM, Guar: GUAR >
               CONFIG) .

  eq qualifiedAndTyped(S, C, AAL, VVL, CONFIG) =
     (qualified(S, C, VVL, CONFIG))
         { AAL, VVL |-> AAL, VVL }
         typedBuiltInVarSubst .

  eq classParams(C,
                 < C : Class | Impl: SUPERL, Ctrc: SUPERL',
                               Inh: SUPERL'', Param: PPL, Att: WWL,
                               Mtd: MM, ObjCnt: N, Asum: ASUM,
                               Guar: GUAR >
                 CONFIG) = PPL .

  eq initialPr(C[EL], CONFIG) =
     initialPr(C[EL], emptyStmt, emptyStmt, emptyStmt, CONFIG) .

  eq initialPr(epsilon, S1, S2, S3, CONFIG) = S1 ; S2 ; S3 .
  eq initialPr((SUPERL''', C[EL]), S1, S2, S3,
               < C : Class | Impl: SUPERL, Ctrc: SUPERL', Inh: SUPERL'',
                             Param: PPL, Att: WWL, Mtd: MM, ObjCnt: N,
                             Asum: ASUM, Guar: GUAR >
               CONFIG) =
     initialPr((SUPERL''',
                qualifiedSuperList(SUPERL'', asIdList(PPL), C)),
               S1 ; PPL := EL,
               if < 'init : Method | * > in mset MM then
                   'init @ C[epsilon ; epsilon]
               else
                   emptyStmt
               fi ; S2,
               if < 'run : Method | * > in mset MM then
                   'run[epsilon ; epsilon]
               else
                   S3
               fi,
               < C : Class | Impl: SUPERL, Ctrc: SUPERL', Inh: SUPERL'',
                             Param: PPL, Att: WWL, Mtd: MM, ObjCnt: N,
                             Asum: ASUM, Guar: GUAR >
               CONFIG) .
endfm
```

B.2 Assertion Utilities

```
***(
    creol-assertion-utilities.maude

    This file contains modules shared by the Creol assertion analyzer
    and the interpreter for open systems.
)

fmod CREOL-ASSERTION-WITH-SIDE-CONDITION is
  including CREOL-HISTORY-EXPRESSION .

  *** assertion with side condition
  op _{{_}} : Assn Assn -> Assn [ctor prec 23 format (d s d s s d d)] .

  vars PHI PHI' PHI'' PHI1 PHI2 PHI3 : Assn .
  var QUANT : Quantifier .
  var RHO : Subst .
  var XXL : TypedIdList .

  eq ! (PHI {{ PHI' }}) = (! PHI) {{ PHI' }} .
  eq if PHI1 {{ PHI' }} th PHI2 el PHI3 fi =
     (if PHI1 th PHI2 el PHI3 fi) {{ PHI' }} .
  eq if PHI1 th PHI2 {{ PHI' }} el PHI3 fi =
     (if PHI1 th PHI2 el PHI3 fi) {{ PHI' }} .
  eq if PHI1 th PHI2 el PHI3 {{ PHI' }} fi =
     (if PHI1 th PHI2 el PHI3 fi) {{ PHI' }} .
  eq (PHI1 {{ PHI' }}) && PHI2 = (PHI1 && PHI2) {{ PHI' }} .
  eq (PHI1 {{ PHI' }}) || PHI2 = (PHI1 || PHI2) {{ PHI' }} .
  eq (PHI1 {{ PHI' }}) ==> PHI2 = (PHI1 ==> PHI2) {{ PHI' }} .
  eq PHI1 ==> (PHI2 {{ PHI' }}) = (PHI1 ==> PHI2) {{ PHI' }} .
  eq (PHI1 {{ PHI' }}) <==> PHI2 = (PHI1 <==> PHI2) {{ PHI' }} .
  eq [PHI {{ PHI' }}] = [PHI] {{ PHI' }} .
  eq QUANT XXL . (PHI {{ PHI' }}) = (QUANT XXL . PHI) {{ PHI' }} .

  eq PHI {{ true }} = PHI .
  eq PHI {{ PHI' {{ PHI'' }} }} = PHI {{ PHI' && PHI'' }} .
  eq PHI {{ PHI' }} {{ PHI'' }} = PHI {{ PHI' && PHI'' }} .

  *** substitutions leave the side condition alone
  eq (PHI {{ PHI' }}) RHO = ((PHI) RHO) {{ PHI' }} .
endfm

mod CREOL-LOGICAL-SIMPLIFICATION-RULES is
  including CREOL-ASSERTION-WITH-SIDE-CONDITION .

  var BASIC : BasicAssn .
  vars PHI PHI1 PHI2 PHI3 : Assn .
  var QUANT : Quantifier .
  var XX : TypedId .
  vars XXL XXL' : TypedIdList .

  rl PHI && PHI => PHI .
  rl true && PHI => PHI .
```

```
  rl false && PHI => false .
  rl PHI && (! PHI) => false .
  rl PHI || PHI => PHI .
  rl true || PHI => true .
  rl false || PHI => PHI .
  rl PHI || (! PHI) => true .
  rl (true ==> PHI) => PHI .
  rl (false ==> PHI) => true .
  rl (PHI ==> true) => true .
  rl (PHI ==> PHI) => true .
  rl (true <==> PHI) => PHI .
  rl (false <==> PHI) => ! PHI .
  rl ((PHI1 && PHI2) ==> PHI1) => true .
  rl ((BASIC && PHI1) ==> (BASIC && PHI2)) =>
     ((BASIC && PHI1) ==> PHI2) .
  rl ((BASIC && PHI1) ==> (BASIC || PHI2)) => true .
  rl ((BASIC && PHI1) ==> ((BASIC || PHI2) && PHI3)) =>
     ((BASIC && PHI1) ==> PHI3) .
  rl ((BASIC && PHI1) ==> ((BASIC && PHI2) || PHI3)) =>
     ((BASIC && PHI1) ==> (PHI2 || PHI3)) .
  rl if true th PHI1 el PHI2 fi => PHI1 .
  rl if false th PHI1 el PHI2 fi => PHI2 .
  rl if PHI1 th true el PHI2 fi => ! PHI1 ==> PHI2 .
  rl if PHI1 th false el PHI2 fi => ! PHI1 && PHI2 .
  rl if PHI1 th PHI2 el true fi => PHI1 ==> PHI2 .
  rl if PHI1 th PHI2 el false fi => PHI1 && PHI2 .
  rl ! false => true .
  rl ! true => false .
  rl ! ! PHI => PHI .
  crl QUANT XXL, XX, XXL' . PHI => QUANT XXL, XXL' . PHI
  if not XX occurs free in PHI .
endm

fmod CREOL-LOGICAL-SIMPLIFICATION is
  including CREOL-HISTORY-EXPRESSION .

  *** simplifies an assertion using the rules declared in
  *** CREOL-LOGICAL-SIMPLIFICATION-RULES
  op logicallySimplified : Assn -> Assn .

  *** auxiliary function
  op rewritten : Assn Module -> Assn .

  var MOD : Module .
  var PHI : Assn .

  eq logicallySimplified(PHI) =
     rewritten(PHI, upModule('CREOL-LOGICAL-SIMPLIFICATION-RULES,
                             false)) .

  eq rewritten(PHI, MOD) =
     downTerm(getTerm(metaRewrite(MOD, upTerm(PHI), unbounded)), PHI) .
endfm

fmod CREOL-ASSUME-GUARANTEE-SPECIFICATION is
```

```
including CREOL-LOGICAL-SIMPLIFICATION .
including CREOL-PROGRAM .

*** Assume--guarantee specification
sort AGSpec .

op <_,_> : Assn Assn -> AGSpec [ctor] .

*** auxiliary operator and functions
op _&&_ : AGSpec AGSpec -> AGSpec [assoc comm prec 13] .
op logicallySimplified : AGSpec -> AGSpec .
op classAGSpec : Id Config ~> AGSpec .
op inheritedAGSpec : SuperList Config ~> AGSpec .

vars ASUM ASUM' : Assn .
var C : Id .
var CONFIG : Config .
var EL : ExpList .
vars GUAR GUAR' : Assn .
var I : Id .
var MM : MtdMSet .
var N : Int .
var PPL : TypedIdList .
vars SUPERL SUPERL' SUPERL'' SUPERL''' : SuperList .
var WWL : TypedIdList .

eq < ASUM, GUAR > && < ASUM', GUAR' > =
   < ASUM && ASUM', GUAR && GUAR' > .

eq logicallySimplified(< ASUM, GUAR >) =
   < logicallySimplified(ASUM), logicallySimplified(GUAR) > .

eq classAGSpec(C,
               < C : Class | Impl: SUPERL, Ctrc: SUPERL',
                             Inh: SUPERL'', Param: PPL, Att: WWL,
                             Mtd: MM, ObjCnt: N, Asum: ASUM,
                             Guar: GUAR >
               CONFIG) =
   logicallySimplified(
       < qualifiedAndTyped(ASUM, (PPL, WWL), C),
         qualifiedAndTyped(GUAR, (PPL, WWL), C) >
       && inheritedAGSpec((SUPERL, SUPERL', SUPERL''), CONFIG)) .

eq inheritedAGSpec(epsilon, CONFIG) = < true, true > .
eq inheritedAGSpec((SUPERL''', C[EL]),
                   < C : Class | Impl: SUPERL, Ctrc: SUPERL',
                                 Inh: SUPERL'', Param: PPL, Att: WWL,
                                 Mtd: MM, ObjCnt: N, Asum: ASUM,
                                 Guar: GUAR >
                   CONFIG) =
   inheritedAGSpec((SUPERL''', SUPERL', SUPERL''),
                   < C : Class | Impl: SUPERL, Ctrc: SUPERL',
                                 Inh: SUPERL'', Param: PPL, Att: WWL,
                                 Mtd: MM, ObjCnt: N, Asum: ASUM,
                                 Guar: GUAR >
```

```
                    CONFIG) .
  eq inheritedAGSpec((SUPERL', I[EL]),
                    < I : Interface | Inh: SUPERL, Param: PPL,
                                      Asum: ASUM, Guar: GUAR >
                    CONFIG) =
     < qualifiedAndTyped(ASUM, PPL, I), qualifiedAndTyped(GUAR, PPL, I) >
     && inheritedAGSpec((SUPERL', SUPERL),
                       < I : Interface | Inh: SUPERL, Param: PPL,
                                         Asum: ASUM, Guar: GUAR >
                       CONFIG) .
endfm
```

B.3 Assertion Analyzer

```
***(
    creol-assertion-analyzer.maude

    This file implements the Creol assertion analyzer described in
    Chapter 6 of Verification of Assertions in Creol Programs. See
    Appendix A for a user's guide.
)

fmod CREOL-ASSERTION-PRETTY-PRINTING is
  including CREOL-ASSERTION-WITH-SIDE-CONDITION .

  *** Pretty-printed assertion
  sort PrettyAssn .
  subsort Assn < PrettyAssn .

  op _And:_ : PrettyAssn PrettyAssn -> PrettyAssn
                [ctor assoc id: true prec 35 format (y !ynssss
                                                     oynsssssssssss y)] .
  op Forall_._ : TypedIdList PrettyAssn -> PrettyAssn
                          [ctor prec 33 format (!y oy y nsssssssssss y)] .
  op Exists_._ : TypedIdList PrettyAssn -> PrettyAssn
                          [ctor prec 33 format (!y oy y nsssssssssss y)] .
  op _And_ : PrettyAssn PrettyAssn -> PrettyAssn
         [ctor assoc comm prec 31 format (y !ynssssss oynsssssssssss y)] .
  op _===>_ : PrettyAssn PrettyAssn -> PrettyAssn
            [ctor prec 29 gather (e E) format (y !ynssssss oynsssssssssss
                                               y)] .
  op _\/_ : PrettyAssn PrettyAssn -> PrettyAssn
             [ctor assoc comm prec 27 format (y !ynsssssss oynsssssssssss
                                              y)] .
  op _/\_ : PrettyAssn PrettyAssn -> PrettyAssn
                  [ctor assoc comm prec 25 format (y !ynsssssss oy y)] .

  op pretty : Assn -> PrettyAssn .
  op prettyQuantifier : Assn -> PrettyAssn .
  op prettyMiddleAnd : Assn -> PrettyAssn .
  op prettyImplies : Assn -> PrettyAssn .
  op prettyOr : Assn -> PrettyAssn .
  op prettyInnerAnd : Assn -> PrettyAssn .
```

```
  vars PHI PHI' PHI1 PHI2 PHI3 : Assn .
  var XXL : TypedIdList .

  eq pretty(PHI {{ PHI' }}) = pretty(PHI) And: prettyQuantifier(PHI') .
  eq pretty(PHI) = prettyQuantifier(PHI) [otherwise] .

  eq prettyQuantifier(forall XXL . PHI) =
     Forall XXL . prettyQuantifier(PHI) .
  eq prettyQuantifier(exists XXL . PHI) =
     Exists XXL . prettyQuantifier(PHI) .
  eq prettyQuantifier(PHI) = prettyMiddleAnd(PHI) [otherwise] .

  eq prettyMiddleAnd((PHI1 ==> PHI2) && PHI3) =
     prettyImplies(PHI1 ==> PHI2) And prettyMiddleAnd(PHI3) .
  eq prettyMiddleAnd(PHI) = prettyImplies(PHI) [otherwise] .

  eq prettyImplies(PHI1 ==> PHI2) =
     prettyOr(PHI1) ===> prettyImplies(PHI2) .
  eq prettyImplies(PHI) = prettyOr(PHI) [otherwise] .

  eq prettyOr(PHI1 || PHI2) = prettyOr(PHI1) \/ prettyOr(PHI2) .
  eq prettyOr(PHI) = prettyInnerAnd(PHI) [otherwise] .

  eq prettyInnerAnd(PHI1 && PHI2) =
     prettyInnerAnd(PHI1) /\ prettyInnerAnd(PHI2) .
  eq prettyInnerAnd(PHI) = PHI [otherwise] .
endfm

mod CREOL-NORMALIZATION-RULES-1 is
  including CREOL-ASSERTION-WITH-SIDE-CONDITION .

  vars PHI PHI1 PHI2 PHI3 PHI4 : Assn .
  var QUANT : Quantifier .
  var XX0 : TypedId .
  var XXL : TypedIdList .
  var YY0 : TypedId .

  *** eliminate square brackets
  rl [PHI] => PHI .

  *** eliminate bi-implications
  rl (PHI1 <==> PHI2) => (PHI1 ==> PHI2) && (PHI2 ==> PHI1) .

  *** move negations inward
  rl ! (PHI1 && PHI2) => (! PHI1 || ! PHI2) .
  rl ! (PHI1 || PHI2) => (! PHI1 && ! PHI2) .
  rl ! (PHI1 ==> PHI2) => (PHI1 && ! PHI2) .
  rl ! if PHI1 th PHI2 el PHI3 fi => if PHI1 th ! PHI2 el ! PHI3 fi .
  rl ! (QUANT XXL . PHI) => opposite(QUANT) XXL . (! PHI) .

  *** move quantifiers outward
  crl PHI1 && (QUANT XX0, XXL . PHI2) =>
      QUANT XX0 . PHI1 && (QUANT XXL . PHI2)
  if not XX0 occurs free in PHI1 .
```

```
  crl PHI1 && (QUANT XX0, XXL . PHI2) =>
      QUANT YY0 . PHI1 && (QUANT XXL . (PHI2) { XX0 |-> YY0 })
  if XX0 occurs free in PHI1
     /\ YY0 := freshLogicalVar(XX0, (XXL, PHI1, PHI2)) .
  crl PHI1 || (QUANT XX0, XXL . PHI2) =>
      QUANT XX0 . PHI1 || (QUANT XXL . PHI2)
  if not XX0 occurs free in PHI1 .
  crl PHI1 || (QUANT XX0, XXL . PHI2) =>
      QUANT YY0 . PHI1 || (QUANT XXL . (PHI2) { XX0 |-> YY0 })
  if XX0 occurs free in PHI1
     /\ YY0 := freshLogicalVar(XX0, (XXL, PHI1, PHI2)) .
  crl (PHI1 ==> (QUANT XX0, XXL . PHI2)) =>
      (QUANT XX0 . PHI1 ==> (QUANT XXL . PHI2))
  if not XX0 occurs free in PHI1 .
  crl (PHI1 ==> (QUANT XX0, XXL . PHI2)) =>
      (QUANT YY0 . PHI1 ==> (QUANT XXL . (PHI2) { XX0 |-> YY0 }))
  if XX0 occurs free in PHI1
     /\ YY0 := freshLogicalVar(XX0, (XXL, PHI1, PHI2)) .
  crl ((QUANT XX0, XXL . PHI1) ==> PHI2) =>
      (opposite(QUANT) XX0 . ((QUANT XXL . PHI1) ==> PHI2))
  if not XX0 occurs free in PHI2 .
  crl ((QUANT XX0, XXL . PHI1) ==> PHI2) =>
      (opposite(QUANT) YY0 .
           ((QUANT XXL . (PHI1) { XX0 |-> YY0 }) ==> PHI2))
  if XX0 occurs free in PHI2
     /\ YY0 := freshLogicalVar(XX0, (XXL, PHI1, PHI2)) .
  crl if PHI1 th (QUANT XX0, XXL . PHI2) el PHI3 fi =>
      QUANT XX0 . if PHI1 th (QUANT XXL . PHI2) el PHI3 fi
  if not XX0 occurs free in (PHI1, PHI3) .
  crl if PHI1 th (QUANT XX0, XXL . PHI2) el PHI3 fi =>
      QUANT YY0 . if PHI1 th (QUANT XXL . (PHI2) { XX0 |-> YY0 })
                           el PHI3 fi
  if XX0 occurs free in (PHI1, PHI3)
     /\ YY0 := freshLogicalVar(XX0, (XXL, PHI1, PHI2, PHI3)) .
  crl if PHI1 th PHI2 el (QUANT XX0, XXL . PHI3) fi =>
      QUANT XX0 . if PHI1 th PHI2 el (QUANT XXL . PHI3) fi
  if not XX0 occurs free in (PHI1, PHI2) .
  crl if PHI1 th PHI2 el (QUANT XX0, XXL . PHI3) fi =>
      QUANT YY0 . if PHI1 th PHI2
                           el (QUANT XXL . (PHI3) { XX0 |-> YY0 }) fi
  if XX0 occurs free in (PHI1, PHI2)
     /\ YY0 := freshLogicalVar(XX0, (XXL, PHI1, PHI2, PHI3)) .

  *** move conditional expressions outward
  rl PHI1 && (if PHI2 th PHI3 el PHI4 fi) =>
     if PHI2 th PHI1 && PHI3 el PHI1 && PHI4 fi .
  rl PHI1 || (if PHI2 th PHI3 el PHI4 fi) =>
     if PHI2 th PHI1 || PHI3 el PHI1 || PHI4 fi .
  rl if PHI1 th PHI2 el PHI3 fi ==> PHI4 =>
     if PHI1 th PHI2 ==> PHI4 el PHI3 ==> PHI4 fi .
  rl PHI1 ==> if PHI2 th PHI3 el PHI4 fi =>
     if PHI2 th PHI1 ==> PHI3 el PHI1 ==> PHI4 fi .
endm

mod CREOL-NORMALIZATION-RULES-2 is
```

```
  including CREOL-ASSERTION-WITH-SIDE-CONDITION .

  vars PHI1 PHI2 PHI3 PHI4 PHI5 : Assn .

  rl if PHI1 th PHI2 el PHI3 fi =>
     ((PHI1 ==> PHI2) && (! PHI1 ==> PHI3)) .

  *** uncurry nested implications
  rl (PHI1 ==> (PHI2 ==> PHI3)) => ((PHI1 && PHI2) ==> PHI3) .
  rl (PHI1 ==> (PHI2 && (PHI3 ==> PHI4))) =>
     ((PHI1 ==> PHI2) && ((PHI1 && PHI3) ==> PHI4)) .

  rl (PHI1 || PHI2) && PHI3 => (PHI1 && PHI3) || (PHI2 && PHI3) .
  rl ((PHI1 || PHI2) ==> PHI3) => (PHI1 ==> PHI3) && (PHI2 ==> PHI3) .
endm

mod CREOL-SIMPLIFICATION-RULES is
  including CREOL-LOGICAL-SIMPLIFICATION-RULES .

  op isPrefix : HistoryExp HistoryExp Assn -> Bool .

  vars A A1 A2 A3 A4 : AExp .
  var BASIC : BasicAssn .
  vars E E1 E2 : Exp .
  var EEXP : EventExp .
  vars EPEXP EPEXP' : EventPatExp .
  vars HEXP HEXP' HEXP'' HEXP''' : HistoryExp .
  vars N N1 N2 : Int .
  var OEXP : OExp .
  vars PHI PHI' PHI1 PHI2 PHI3 : Assn .
  var V : Value .
  var XX : TypedId .
  var YY : TypedId .

  *** auxiliary predicate
  eq isPrefix(HEXP, HEXP', HEXP ^^ HEXP'' pr HEXP) = true .
  eq isPrefix(HEXP, HEXP', HEXP ^^ HEXP'' pr HEXP''' && PHI) =
     (HEXP''' == HEXP') or isPrefix(HEXP''', HEXP', PHI) .
  eq isPrefix(HEXP, HEXP', PHI) = false [otherwise] .

  rl if true th E1 el E2 fi => E1 .
  rl if false th E1 el E2 fi => E2 .
  rl [E] => E .

  rl plus A => A .
  rl minus N => - N .
  rl N1 times N2 => N1 * N2 .
  rl A times 1 => A .
  rl N1 div N2 => if N2 == 0 then 0 else N1 quo N2 fi .
  rl N1 plus N2 => N1 + N2 .
  rl A plus 0 => A .
  rl N1 minus N2 => N1 - N2 .
  rl A minus 0 => A .
  rl A minus A => 0 .
  rl 0 minus A => minus A .
```

```
rl A1 plus (A2 minus A3) => A1 plus A2 minus A3 .
rl A1 minus (A2 plus A3) => A1 minus A2 minus A3 .
rl A1 minus (A2 minus A3) => A1 minus A2 plus A3 .
rl A1 plus A2 minus A2 => A1 .
rl 0 eq A1 minus A2 => A1 eq A2 .
rl 0 lt A1 minus A2 => A2 lt A1 .
rl 0 le A1 minus A2 => A2 le A1 .
rl N1 eq A plus N2 => (N1 - N2) eq A .
rl N1 eq A minus N2 => (N1 + N2) eq A .
rl A1 minus A2 eq 0 => A1 eq A2 .
rl A1 minus A2 lt 0 => A1 lt A2 .
rl A1 minus A2 le 0 => A1 le A2 .
rl A1 plus A2 eq A1 plus A3 plus A4 => A2 eq A3 plus A4 .
rl A1 plus A2 eq A1 plus A3 minus A4 => A2 eq A3 minus A4 .
rl A1 minus A2 eq A3 minus A2 plus A4 => A1 eq A3 plus A4 .
rl A1 minus A2 eq A3 minus A2 minus A4 => A1 eq A3 minus A4 .
rl A1 plus A2 eq A1 => A2 eq 0 .
rl A1 plus A2 eq A2 => A1 eq 0 .

*** eliminate ne, gt, and ge
rl E1 ne E2 => ! (E1 eq E2) .
rl A1 gt A2 => A2 lt A1 .
rl A1 ge A2 => A2 le A1 .

rl ! (A1 lt A2) => A2 le A1 .
rl ! (A1 le A2) => A2 lt A1 .

rl N1 eq N2 => N1 == N2 .
rl E eq E => true .
rl N1 lt N2 => N1 < N2 .
rl A lt A => false .
rl N1 le N2 => N1 <= N2 .
rl A le A => true .
crl A minus N le A => true if N >= 0 .
crl A minus N lt A => true if N >= 1 .
crl A le A plus N => true if N >= 0 .
crl A lt A plus N => true if N >= 1 .
rl A1 lt A2 && A1 le A2 => A1 lt A2 .
rl A1 lt A2 plus 1 => A1 le A2 .
rl A1 minus 1 lt A2 => A1 le A2 .
rl A1 plus 1 le A2 => A1 lt A2 .
rl A1 le A2 minus 1 => A1 lt A2 .
rl A1 lt A2 && A2 lt A1 => false .
rl A1 le A2 && A2 le A1 => A1 eq A2 .
rl A1 lt A2 && A2 lt A3 && A3 lt A1 => false .
rl A1 lt A2 && A2 lt A3 && A3 le A1 => false .
rl A1 lt A2 && A2 le A3 && A3 le A1 => false .
rl A1 lt A2 || A2 le A1 => true .

rl A1 eq A2 && A1 le A2 => A1 eq A2 .
rl A1 eq A2 && A2 le A1 => A1 eq A2 .
crl N1 eq A && N2 lt A => N1 eq A if N2 < N1 .
crl A eq N1 && A lt N2 => A eq N1 if N1 < N2 .

rl A lt N1 && A lt N2 => A lt min(N1, N2) .
```

```
rl A le N1 && A le N2 => A le min(N1, N2) .
rl N1 lt A && N2 lt A => max(N1, N2) lt A .
rl N1 le A && N2 le A => max(N1, N2) le A .

crl N1 lt A ==> N2 lt A => true
if N1 >= N2 .
crl N1 le A ==> N2 le A => true
if N1 >= N2 .
crl A lt N1 ==> A lt N2 => true
if N1 <= N2 .
crl A le N1 ==> A le N2 => true
if N1 <= N2 .

rl (A1 eq A2 ==> A1 le A2) => true .
rl (A1 eq A2 ==> A2 le A1) => true .
crl (N1 eq A ==> N2 lt A) => true
if N2 < N1 .
crl (A eq N1 ==> A lt N2) => true
if N1 < N2 .

rl ((A1 eq A2 && PHI) ==> A1 le A2) => true .
rl ((A1 eq A2 && PHI) ==> A2 le A1) => true .
crl ((N1 eq A && PHI) ==> N2 lt A) => true
if N2 < N1 .
crl ((A eq N1 && PHI) ==> A lt N2) => true
if N1 < N2 .

rl ((A1 eq A2 && PHI1) ==> (A1 le A2 && PHI2)) =>
   ((A1 eq A2 && PHI1) ==> PHI2) .
rl ((A1 eq A2 && PHI1) ==> (A2 le A1 && PHI2)) =>
   ((A1 eq A2 && PHI1) ==> PHI2) .
crl ((N1 eq A && PHI1) ==> (N2 lt A && PHI2)) =>
    ((N1 eq A && PHI1) ==> PHI2)
if N2 < N1 .
crl ((A eq N1 && PHI1) ==> (A lt N2 && PHI2)) =>
    ((A eq N1 && PHI1) ==> PHI2)
if N1 < N2 .

crl (A eq N1 && A eq N2) => false if N1 =/= N2 .
crl ((A eq N1 && PHI) ==> A eq N2) =>
    ((A eq N1 && PHI) ==> false)
if N1 =/= N2 .
crl ((A eq N1 && PHI1) ==> (A eq N2 && PHI2)) =>
    ((A eq N1 && PHI1) ==> false)
if N1 =/= N2 .
crl ((A eq N1 && PHI1) ==> ((A eq N2 && PHI2) || PHI3)) =>
    ((A eq N1 && PHI1) ==> PHI3)
if N1 =/= N2 .

crl ((N1 le A && PHI1) ==> A eq N2) => ((N1 le A && PHI1) ==> false)
if N1 > N2 .
crl ((N1 le A && PHI1) ==> (A eq N2 && PHI3)) =>
    ((N1 le A && PHI1) ==> false)
if N1 > N2 .
crl ((N1 le A && PHI1) ==> (A eq N2 || PHI3)) =>
```

```
    ((N1 le A && PHI1) ==> PHI3)
if N1 > N2 .
crl ((N1 le A && PHI1) ==> ((A eq N2 && PHI2) || PHI3)) =>
    ((N1 le A && PHI1) ==> PHI3)
if N1 > N2 .

rl ((A1 lt A2 && PHI1) ==> A1 le A2) => true .
rl ((A1 le A2 && PHI1) ==> A1 le A2 plus 1) => true .
rl ((A1 lt A2 && PHI1) ==> A1 minus 1 lt A2) => true .
rl ((A1 le A2 && PHI1) ==> A1 minus 1 le A2) => true .
rl ((A1 lt A2 && PHI1) ==> (A1 le A2 && PHI2)) =>
   ((A1 lt A2 && PHI1) ==> PHI2) .
rl ((A1 le A2 && PHI1) ==> (A1 le A2 plus 1 && PHI2)) =>
   ((A1 le A2 && PHI1) ==> PHI2) .
rl ((A1 lt A2 && PHI1) ==> (A1 minus 1 lt A2 && PHI2)) =>
   ((A1 lt A2 && PHI1) ==> PHI2) .
rl ((A1 le A2 && PHI1) ==> (A1 minus 1 le A2 && PHI2)) =>
   ((A1 le A2 && PHI1) ==> PHI2) .

crl XX eq V && BASIC => XX eq V && (BASIC) { XX |-> V }
if XX occurs free in BASIC .

rl ((XX eq V && PHI) ==> XX eq E) => ((XX eq V && PHI) ==> V eq E) .

crl ((XX eq E && PHI1) ==> PHI2) =>
    ((XX eq E && (PHI1) { XX |-> E }) ==> (PHI2) { XX |-> E })
if XX occurs free in (PHI1, PHI2)
   and not XX occurs free in E
   and not (E :: TypedQualifiedId) .

crl ((XX eq YY && PHI1) ==> PHI2) =>
     (XX eq YY && (PHI1) { XX |-> YY }) ==> (PHI2) { XX |-> YY }
if XX occurs free in (PHI1, PHI2) and YY occurs free in (PHI1, PHI2) .

rl #[emptyHistory] => 0 .
rl #[EEXP] => 1 .
crl #[HEXP ^^ HEXP'] => #[HEXP] plus #[HEXP']
if HEXP =/= emptyHistory and HEXP' =/= emptyHistory .

rl 'lwf[HEXP, OEXP] && 'lwf[HEXP', OEXP] && HEXP ^^ HEXP'' pr HEXP' =>
   'lwf[HEXP', OEXP] && HEXP ^^ HEXP'' pr HEXP' .

crl (HEXP ^^ EEXP) / EPEXP => HEXP / EPEXP
if EEXP cannot match EPEXP .

crl (HEXP ^^ EEXP) / EPEXP => (HEXP / EPEXP) ^^ EEXP
if EEXP must match EPEXP .

crl EEXP / EPEXP ew EPEXP' => true
if EEXP must match EPEXP' .

crl EEXP / EPEXP ew EPEXP' => false
if EEXP cannot match EPEXP' .

crl (HEXP ^^ EEXP) ew EPEXP => false
```

```
  if EEXP cannot match EPEXP .

  crl (HEXP ^^ EEXP) ew EPEXP => true
  if EEXP must match EPEXP .

  crl EPEXP in HEXP ^^ EEXP => EPEXP in HEXP
  if EEXP cannot match EPEXP .

  crl EPEXP in HEXP ^^ EEXP => true
  if EEXP must match EPEXP .

  ceq ((EPEXP in HEXP) && (EPEXP in HEXP') && PHI) ==> PHI' =
      ((EPEXP in HEXP') && PHI) ==> PHI'
  if isPrefix(HEXP, HEXP', PHI) .
  ceq (! (EPEXP in HEXP) && ! (EPEXP in HEXP') && PHI) ==> PHI' =
      (! (EPEXP in HEXP') && PHI) ==> PHI'
  if isPrefix(HEXP, HEXP', PHI) .
endm

fmod CREOL-ASSERTION-MASSAGING is
  including CREOL-ASSERTION-PRETTY-PRINTING .
  including CREOL-LOGICAL-SIMPLIFICATION .

  *** convenience functions
  op massaged : Assn Module -> PrettyAssn .
  op massaged : Assn -> PrettyAssn .

  *** low-level massaging functions
  op untyped : Assn -> Assn .
  op metaUntyped : TermList -> TermList .
  op unqualified : Assn -> Assn .
  op metaUnqualified : TermList -> TermList .
  op simplifiedAndNormalized : Assn Module Int -> Assn .
  op normalized : Assn -> Assn .
  op normalized1 : Assn -> Assn .
  op normalized2 : Assn -> Assn .
  op simplified : Assn Module -> Assn .

  var CONST : Constant .
  var MOD : Module .
  var N : Int .
  vars PHI PHI' : Assn .
  var QID : Qid .
  vars TERM0 TERM1 TERM2 TERM3 : Term .
  var TERMLIST : TermList .
  var VAR : Variable .

  eq massaged(PHI, MOD) =
     pretty(untyped(simplifiedAndNormalized(normalized(PHI), MOD, 5))) .

  eq massaged(PHI) =
     massaged(PHI, upModule('CREOL-SIMPLIFICATION-RULES, false)) .

  eq untyped(PHI) = downTerm(metaUntyped(upTerm(PHI)), PHI) .
```

```
  eq metaUntyped(CONST) = CONST .
  eq metaUntyped(VAR) = VAR .
  eq metaUntyped('_:_[TERM1, TERM2]) = TERM1 .
  eq metaUntyped('__._[TERM1, TERM2, TERM3]) =
     '__._[TERM1, TERM2, metaUntyped(TERM3)] .
  eq metaUntyped(QID[TERMLIST]) = QID[metaUntyped(TERMLIST)]
                                                          [otherwise] .
  ceq metaUntyped((TERM0, TERMLIST)) =
      metaUntyped(TERM0), metaUntyped(TERMLIST)
  if TERMLIST =/= empty .

  *** unused, but comes in handy at times
  eq unqualified(PHI) = downTerm(metaUnqualified(upTerm(PHI)), PHI) .

  eq metaUnqualified(CONST) = CONST .
  eq metaUnqualified(VAR) = VAR .
  eq metaUnqualified('_@_[TERM1, TERM2]) = TERM1 .
  eq metaUnqualified(QID[TERMLIST]) = QID[metaUnqualified(TERMLIST)]
                                                          [otherwise] .
  ceq metaUnqualified((TERM0, TERMLIST)) =
      metaUnqualified(TERM0), metaUnqualified(TERMLIST)
  if TERMLIST =/= empty .

  eq simplifiedAndNormalized(PHI, MOD, 0) = PHI .
  ceq simplifiedAndNormalized(PHI, MOD, N) =
      if PHI == PHI' then
          PHI'
      else
          simplifiedAndNormalized(PHI', MOD, N - 1)
      fi
  if PHI' := normalized(simplified(PHI, MOD)) [otherwise] .

  eq normalized(PHI) = normalized2(normalized1(PHI)) .

  eq normalized1(PHI) =
     rewritten(PHI, upModule('CREOL-NORMALIZATION-RULES-1, false)) .

  eq normalized2(PHI) =
     rewritten(PHI, upModule('CREOL-NORMALIZATION-RULES-2, false)) .

  eq simplified(PHI, MOD) = rewritten(PHI, MOD) .
endfm

fmod CREOL-WEAKEST-LIBERAL-PRECONDITION is
  including CREOL-ASSERTION-MASSAGING .
  including CREOL-ASSUME-GUARANTEE-SPECIFICATION .

  op _all eq_ : ExpList ExpList ~> BasicBExp [ctor comm prec 11] .
  op perms : Stmt -> Stmt .
  op perms : Stmt Stmt -> Stmt .
  op awaitFree : Stmt -> Bool .
  op satisfied :
     Guard TypedQualifiedIdList HistoryExp TypedQualifiedIdList -> Assn .
  op enabled :
     Stmt TypedQualifiedIdList HistoryExp TypedQualifiedIdList -> Assn .
```

```
op ready :
   Stmt TypedQualifiedIdList HistoryExp TypedQualifiedIdList -> Assn .
op pickReadyBranch :
   Stmt Stmt Assn Assn TypedQualifiedIdList HistoryExp
   TypedQualifiedIdList
   -> Assn .
op pending : HistoryExp OExp OExp AExp -> Assn .
op release :
   HistoryExp HistoryExp TypedQualifiedIdList AGSpec
   TypedQualifiedIdList
   -> Assn .
op reenter :
   HistoryExp HistoryExp TypedQualifiedIdList AExp AGSpec
   TypedQualifiedIdList
   -> Assn .
op interleave : HistoryExp HistoryExp AGSpec -> Assn .
op interleaved : Assn AGSpec -> Assn .
op returnVals : HistoryExp OExp TypedId Int -> ExpList .
op wlp : Stmt Assn AGSpec TypedQualifiedIdList ~> Assn .

var A : AExp .
vars AAL AAL' : TypedQualifiedIdList .
var AG : AGSpec .
var ASUM : Assn .
var B : BExp .
var C : Id .
vars E1 E2 : Exp .
vars EL EL1 EL2 : ExpList .
vars G G1 G2 : Guard .
var GUAR : Assn .
vars HEXP HEXP' : HistoryExp .
vars HH HH' : TypedId .
var I : Assn .
var K : Int .
var KK : TypedId .
var LL : TypedId .
var M : Id .
vars OEXP OEXP' : OExp .
var OO : TypedId .
vars P P1 P2 : Assn .
var Q : Assn .
vars S S' S1 S1* S2 S2* S1..SK SK+1..SN S2..SN : Stmt .
var ZZ : TypedQualifiedId .
var ZZL : TypedQualifiedIdList .

eq epsilon all eq epsilon = true .
eq (E1, EL1) all eq (E2, EL2) = E1 eq E2 && EL1 all eq EL2 .

eq perms(S) = perms(S, S) .

ceq perms(S1, S1) = S1
if simpleBranch(S1) .
ceq perms(S1, S1 ||| S') = S1 ; perms(S')
if simpleBranch(S1) .
ceq perms(S1 ||| S', S1 ||| S2..SN) =
```

```
    S1 ; perms(S2..SN, S2..SN) [] perms(S', S1 ||| S2..SN)
if simpleBranch(S1) .

*** Definition Q9 (Await-Freedom)
eq awaitFree(await G) = false .
eq awaitFree(if B th S1 el S2 fi) = awaitFree(S1) and awaitFree(S2) .
eq awaitFree(while B do S od) = awaitFree(S) .
eq awaitFree(inv I while B do S od) = awaitFree(S) .
ceq awaitFree(S1 ; S2) = awaitFree(S1) and awaitFree(S2)
if S1 =/= emptyStmt and S2 =/= emptyStmt .
eq awaitFree(S1 [] S2) = awaitFree(S1) and awaitFree(S2) .
eq awaitFree(S1..SK ||| SK+1..SN) =
   awaitFree(S1..SK) and awaitFree(SK+1..SN) .
eq awaitFree([S]) = awaitFree(S) .
eq awaitFree(S) = true [otherwise] .

*** Definition Q8 (Guard Satisfaction Assertion)
*** extended with AAL parameter
eq satisfied(B, AAL', HEXP, AAL) =
   (B) { AAL |-> AAL' }
       { ~H~ : history |-> HEXP } .
eq satisfied(LL ?, AAL', HEXP, AAL) =
   [LL % self : any <- * . *] in HEXP .
eq satisfied(wait, AAL', HEXP, AAL) = false .
eq satisfied(G1 &&& G2, AAL', HEXP, AAL) =
   satisfied(G1, AAL', HEXP, AAL) && satisfied(G2, AAL', HEXP, AAL) .

*** Definition Q7 (Statement Enabledness Assertion)
*** extended with AAL parameter
eq enabled(await G ; S, AAL', HEXP, AAL) =
   satisfied(G, AAL', HEXP, AAL) .
eq enabled((S1 [] S2) ; S, AAL', HEXP, AAL) =
   enabled(S1, AAL', HEXP, AAL) || enabled(S2, AAL', HEXP, AAL) .
eq enabled((S1..SK ||| SK+1..SN) ; S, AAL', HEXP, AAL) =
   enabled(S1..SK, AAL', HEXP, AAL)
   || enabled(SK+1..SN, AAL', HEXP, AAL) .
eq enabled([S] ; S', AAL', HEXP, AAL) = enabled(S, AAL', HEXP, AAL) .
eq enabled(S, AAL', HEXP, AAL) = true [otherwise] .

*** Definition Q6 (Statement Readiness Assertion)
*** extended with AAL parameter
eq ready(LL ?[ZZL] ; S, AAL', HEXP, AAL) =
   satisfied(LL ?, AAL', HEXP, AAL) .
eq ready(await G ; S, AAL', HEXP, AAL) =
   satisfied(G, AAL', HEXP, AAL) .
eq ready((S1 [] S2) ; S, AAL', HEXP, AAL) =
   ready(S1, AAL', HEXP, AAL) || ready(S2, AAL', HEXP, AAL) .
eq ready((S1..SK ||| SK+1..SN) ; S, AAL', HEXP, AAL) =
   ready(S1..SK, AAL', HEXP, AAL) || ready(SK+1..SN, AAL', HEXP, AAL) .
eq ready([S] ; S', AAL', HEXP, AAL) = ready(S, AAL', HEXP, AAL) .
eq ready(S, AAL', HEXP, AAL) = true [otherwise] .

*** Definition Q5 (Ready Branch Choice Assertion)
*** extended with AAL parameter
eq pickReadyBranch(S1, S2, P1, P2, AAL', HEXP, AAL) =
```

```
   [ready(S1, AAL', HEXP, AAL) ==>
   (P1) { AAL |-> AAL' }
        { ~H~ : history |-> HEXP }]
   && [ready(S2, AAL', HEXP, AAL) ==>
       (P2) { AAL |-> AAL' }
            { ~H~ : history |-> HEXP }] .

*** Definition T22 (Pending Call Predicate)
eq pending(HEXP, OEXP, OEXP', A) =
   [A % OEXP -> OEXP' . *] in HEXP
   && ! [[A % self : any <- self : any . *] in HEXP] .

*** Definition Q4 (Processor Release Assertion)
*** extended with < ASUM, GUAR > and AAL parameters
eq release(HEXP, HEXP', AAL', < ASUM, GUAR >, AAL) =
   'lwf[HEXP', self : any]
   && HEXP ^^ [self : any . release] pr HEXP'
   && 'mayAcquireProcessor[HEXP', self : any, caller : any,
                           label : int]
   && ! [[label : int % caller : any <- self : any . *] in HEXP']
   && [[HEXP ^^ [self : any . release]] /
       (out[self : any] | ctl[self : any]) eq
       HEXP' / (out[self : any] | ctl[self : any]) ==>
       AAL all eq AAL']
   && (ASUM && GUAR) { AAL |-> AAL' }
                     { ~H~ : history |-> HEXP' } .

*** Definition Q2 (Local Reentry Assertion)
*** extended with < ASUM, GUAR > and AAL parameters
eq reenter(HEXP, HEXP', AAL', A, < ASUM, GUAR >, AAL) =
   'lwf[HEXP', self : any]
   && HEXP ^^ [A % self : any . reenter] pr HEXP'
   && HEXP' ew [A % self : any <- self : any . *]
   && ! [[label : int % caller : any <- self : any . *] in HEXP']
   && (ASUM && GUAR) { AAL |-> AAL' }
                     { ~H~ : history |-> HEXP' } .

*** Definition Q1 (Parallel Activity Interleaving Assertion)
*** extended with < ASUM, GUAR >
eq interleave(HEXP, HEXP', < ASUM, GUAR >) =
   'lwf[HEXP', self : any]
   && HEXP pr HEXP'
   && 'agreeOnOutAndCtl[HEXP, HEXP', self : any]
   && ASUM { ~H~ : history |-> HEXP' } .

ceq interleaved(Q, AG) =
    forall HH .
        interleave(~H~ : history, HH, AG) ==>
        (Q) { ~H~ : history |-> HH }
if HH := freshLogicalVar('h : history, Q) .

eq returnVals(HEXP, OEXP, LL, 0) = epsilon .
eq returnVals(HEXP, OEXP, LL, K) =
   returnVals(HEXP, OEXP, LL, (K - 1)),
   'returnVal $ K[HEXP, OEXP, LL] [otherwise] .
```

```
*** Definition Q10 (WLP for Most Creol Statements)
*** extended with AG and AAL
eq wlp(skip, Q, AG, AAL) = Q .
eq wlp(abort, Q, AG, AAL) = true .
eq wlp(prove P, Q, AG, AAL) = P && Q .
eq wlp(ZZL := EL, Q, AG, AAL) = (Q) { ZZL |-> EL } .
eq wlp([S], Q, AG, AAL) = wlp(S, Q, AG, AAL) .
eq wlp(if B th S1 el S2 fi, Q, AG, AAL) =
   if B th wlp(S1, Q, AG, AAL) el wlp(S2, Q, AG, AAL) fi .
ceq wlp(S1..SK ||| SK+1..SN, Q, AG, AAL) =
    wlp(perms(S1..SK ||| SK+1..SN), Q, AG, AAL)
if awaitFree(S1..SK ||| SK+1..SN) .
ceq wlp(S1 ; S2, Q, AG, AAL) = wlp(S1, wlp(S2, Q, AG, AAL), AG, AAL)
if S1 =/= emptyStmt and S2 =/= emptyStmt .

*** Precond. of Axiom P12 (Object Creation)
ceq wlp(ZZ := new C[EL], Q, AG, AAL) =
    interleaved(
        forall OO .
            ['isFreshObjectId[OO, ~H~ : history]
             && 'parent[OO] eq self : any] ==>
            (Q) { ZZ |-> OO }
                { ~H~ : history |-> ~H~ : history
                                    ^^ [self : any -> OO . new C[EL]] },
        AG)
if OO := freshLogicalVar('o : any, Q) .

*** Precond. of Axiom P13 (Asynchronous Invocation)
ceq wlp(LL ! OEXP . M[EL], Q, AG, AAL) =
    interleaved(
        forall KK .
            'isFreshSequenceNum[KK, self : any, ~H~ : history] ==>
            (Q) { LL |-> KK }
                { ~H~ : history |-> ~H~ : history
                                ^^ [KK % self : any -> OEXP . M[EL]] },
        AG)
if KK := freshLogicalVar('k : int, Q) .

*** Precond. of Axiom P14 (Local Asynchronous Invocation)
ceq wlp(LL ! M @ C[EL], Q, AG, AAL) =
    interleaved(
        forall KK .
            'isFreshSequenceNum[KK, self : any, ~H~ : history] ==>
            (Q) { LL |-> KK }
                { ~H~ : history |-> ~H~ : history
                          ^^ [KK % self : any -> self : any . M[EL]] },
        AG)
if KK := freshLogicalVar('k : int, Q) .

*** Precond. of Axiom P15 (Asynchronous Reply)
ceq wlp(LL ?[ZZL], Q, < ASUM, GUAR >, AAL) =
    if pending(~H~ : history, self : any, self : any, LL) th
        GUAR { ~H~ : history |->
               ~H~ : history ^^ [LL % self : any . reenter] }
```

```
        && [forall AAL', HH .
                reenter(~H~ : history, HH, AAL', LL, < ASUM, GUAR >,
                        AAL) ==>
                (Q) { AAL |-> AAL' }
                    { ~H~ : history |-> HH }
                    { LL |-> -1 }
                    { ZZL |-> returnVals(HH, self : any, LL,
                                         length(ZZL)) }]
    el
        interleaved(
            [LL % self : any <- * . *] in ~H~ : history ==>
            (Q) { LL |-> -1 }
                { ZZL |-> returnVals(~H~, self : any, LL,
                                     length(ZZL)) },
            < ASUM, GUAR >)
    fi
if HH := freshLogicalVar('h : history, Q)
   /\ AAL' := freshLogicalVarList(AAL, (HH, Q)) .

*** Precond. of Axiom P16 (Conditional Wait with Boolean Guards)
*** Precond. of Axiom P17 (Conditional Wait with Reply Guards)
*** Precond. of Axiom P18 (Unconditional Wait)
*** slightly altered to avoid interleaved in Boolean case and to share
*** common code
ceq wlp(await G, Q, < ASUM, GUAR >, AAL) =
    if cleared(G) == G then
        if P :: BExp then
            *** Boolean guards
            if P th
                Q
            el
                GUAR { ~H~ : history |->
                       ~H~ : history ^^ [self : any . release] }
                && [forall AAL', HH .
                        release(~H~ : history, HH, AAL',
                                < ASUM, GUAR >, AAL) ==>
                        (P ==> Q) { AAL |-> AAL' }
                                  { ~H~ : history |-> HH }]
            fi
        else
            *** reply guards with optional Boolean guards
            forall HH .
                interleave(~H~ : history, HH, < ASUM, GUAR >) ==>
                if (P) { ~H~ : history |-> HH } th
                    (Q) { ~H~ : history |-> HH }
                el
                    GUAR { ~H~ : history |->
                           HH ^^ [self : any . release] }
                    && [forall AAL', HH' .
                            release(HH, HH', AAL', < ASUM, GUAR >,
                                    AAL) ==>
                            (P ==> Q) { AAL |-> AAL' }
                                      { ~H~ : history |-> HH' }]
                fi
        fi
```

```
    else
        *** wait guard with optional reply and Boolean guards
        GUAR { ~H~ : history |->
               ~H~ : history ^^ [self : any . release] }
        && [forall AAL', HH .
                release(~H~ : history, HH, AAL', < ASUM, GUAR >,
                        AAL) ==>
                (P ==> Q) { AAL |-> AAL' }
                          { ~H~ : history |-> HH }]
    fi
if P := satisfied(cleared(G), AAL, ~H~, AAL)
   /\ HH := freshLogicalVar('h : history, Q)
   /\ HH' := freshLogicalVar('h : history, (HH, Q))
   /\ AAL' := freshLogicalVarList(AAL, (HH, HH', Q)) .

*** Definition Q12 (WLP for While Loop)
*** Proof Rule P9 (While Loop)
*** altered versions that doesn't use Dijkstra's trick, which is hard
*** to implement and has little practical value
eq wlp(while B do S od, Q, AG, AAL) = false .
eq wlp(inv I while B do S od, Q, AG, AAL) =
   I {{ [I && B] ==> wlp(S, I, AG, AAL) }}
     {{ [I && ! B] ==> Q }} .

*** Definition Q11 (WLP for Nondeterministic Choice)
*** Proof Rule P20 (Nondeterministic Choice, Incomplete)
*** altered version that optimizes the common case
ceq wlp(S1 [] S2, Q, < ASUM, GUAR >, AAL) =
    if logicallySimplified(ready(S1, AAL, ~H~ : history, AAL)
                           && ready(S2, AAL, ~H~ : history, AAL))
           == true then
        *** S1 and S2 are always ready
        wlp(S1, Q, < ASUM, GUAR >, AAL) && wlp(S2, Q, < ASUM, GUAR >,
            AAL)
    else
        *** general case
        interleaved(
            if ready(S1 [] S2, AAL, ~H~ : history, AAL) th
                pickReadyBranch(S1, S2,
                                wlp(S1, Q, < ASUM, GUAR >, AAL),
                                wlp(S2, Q, < ASUM, GUAR >, AAL),
                                AAL, ~H~ : history, AAL)
            el if enabled(S1 [] S2, AAL, ~H~ : history, AAL) th
                interleaved(
                    ready(S1 [] S2, AAL, ~H~ : history, AAL) ==>
                    pickReadyBranch(S1, S2,
                                    wlp(S1, Q, < ASUM, GUAR >, AAL),
                                    wlp(S2, Q, < ASUM, GUAR >, AAL),
                                    AAL, ~H~ : history, AAL),
                    < ASUM, GUAR >)
            el
                GUAR { ~H~ : history |->
                       ~H~ : history ^^ [self : any . release] }
                && [forall AAL', HH .
                        [release(~H~ : history, HH, AAL',
```

```
                                < ASUM, GUAR >, AAL)
                         && ready(S1* [] S2*, AAL', HH, AAL)] ==>
                        pickReadyBranch(S1*, S2*,
                                        wlp(S1*, Q,
                                            < ASUM, GUAR >, AAL),
                                        wlp(S2*, Q,
                                            < ASUM, GUAR >, AAL),
                                        AAL', HH, AAL)]
            fi fi,
            < ASUM, GUAR >)
    fi
if HH := freshLogicalVar('h : history, Q)
   /\ AAL' := freshLogicalVarList(AAL, (HH, Q))
   /\ S1* := clearWait(S1)
   /\ S2* := clearWait(S2) .

*** WLPs for synthetic statements; some of these are optimized

eq wlp(! OEXP . M[EL], Q, AG, AAL) =
   wlp(nu : int ! OEXP . M[EL], Q, AG, AAL) .

eq wlp(! M @ C[EL], Q, AG, AAL) =
   wlp(nu : int ! M @ C[EL], Q, AG, AAL) .

eq wlp(OEXP . M[EL ; ZZL], Q, < ASUM, GUAR >, AAL) =
   if OEXP == self : any then
       wlp(M[EL ; ZZL], Q, < ASUM, GUAR >, AAL)
   else
       wlp(nu : int ! OEXP . M[EL] ; nu : int ?[ZZL], Q,
           < ASUM, GUAR >, AAL)
   fi .

ceq wlp(M @ C[EL ; ZZL], Q, < ASUM, GUAR >, AAL) =
    forall KK .
        'isFreshSequenceNum[KK, self : any, ~H~ : history] ==>
        (GUAR { ~H~ : history |->
                ~H~ : history ^^ [KK % self : any . reenter] }
         && [forall AAL', HH .
                 reenter(~H~ : history, HH, AAL', KK, < ASUM, GUAR >,
                         AAL) ==>
                 (Q) { AAL |-> AAL' }
                     { ~H~ : history |-> HH }
                     { ZZL |-> returnVals(HH, self : any, KK,
                                          length(ZZL)) }])
        { ~H~ : history |-> ~H~ : history
                            ^^ [KK % self : any -> self : any . M[EL]] }
if KK := freshLogicalVar('k : int, Q)
   /\ HH := freshLogicalVar('h : history, Q)
   /\ AAL' := freshLogicalVarList(AAL, (KK, HH, Q)) .

eq wlp(await G &&& LL ?[ZZL], Q, AG, AAL) =
   wlp(await G &&& LL ?,
       (Q) { LL |-> -1 }
           { ZZL |-> returnVals(~H~ : history, self : any, LL,
                                length(ZZL)) },
```

```
        AG, AAL) .

  eq wlp(await G &&& OEXP . M[EL ; ZZL], Q, AG, AAL) =
     wlp(nu : int ! OEXP . M[EL] ; await G &&& nu : int ?[ZZL], Q, AG,
         AAL) .

  eq wlp(await G &&& M @ C[EL ; ZZL], Q, AG, AAL) =
     wlp(nu : int ! M @ C[EL] ; await wait &&& G &&& nu : int ?[ZZL], Q,
         AG, AAL) .

  eq wlp(await LL ?[ZZL], Q, AG, AAL) =
     wlp(await true &&& LL ?[ZZL], Q, AG, AAL) .

  eq wlp(await OEXP . M[EL ; ZZL], Q, AG, AAL) =
     wlp(await true &&& OEXP . M[EL ; ZZL], Q, AG, AAL) .

  eq wlp(await M @ C[EL ; ZZL], Q, AG, AAL) =
     wlp(await true &&& M @ C[EL ; ZZL], Q, AG, AAL) .

  eq wlp(if B th S1 fi, Q, AG, AAL) =
     wlp(if B th S1 el skip fi, Q, AG, AAL) .

  *** isn't really needed
  eq wlp(emptyStmt, Q, AG, AAL) = Q .
endfm

fmod CREOL-ASSERTION-ANALYZER-REPORT is
  including CREOL-ASSERTION-MASSAGING .
  including CREOL-PROGRAM .
  including CREOL-WEAKEST-LIBERAL-PRECONDITION .

  *** Head of a judgment
  sort JudgmentHead .

  op Initialization code : -> JudgmentHead [ctor format (b b o)] .
  op Method_of_ : Id Id -> JudgmentHead
                                        [ctor prec 3 format (b b b b o)] .

  *** Body of a judgment
  sort JudgmentBody .

  op Maintains the guarantee : -> JudgmentBody [ctor format (g g g o)] .
  op Maintains the guarantee iff_holds :
     PrettyAssn -> JudgmentBody [ctor format (y y y y nssssy nssssy o)] .
  op Maintains the guarantee if_holds :
     PrettyAssn -> JudgmentBody [ctor format (y y y y nssssy nssssy o)] .
  op Breaks the guarantee : -> JudgmentBody [ctor format (r r r o)] .
  op Don't know : -> JudgmentBody [ctor format (y y o)] .
  op Establishes the guarantee : -> JudgmentBody
                                                 [ctor format (g g g o)] .
  op Establishes the guarantee iff_holds :
     PrettyAssn -> JudgmentBody [ctor format (y y y y nssssy nssssy o)] .
  op Establishes the guarantee if_holds :
     PrettyAssn -> JudgmentBody [ctor format (y y y y nssssy nssssy o)] .
  op Fails to establish the guarantee : -> JudgmentBody
```

```
                                          [ctor format (r r r r r o)] .

*** auxiliary functions
op judgmentBody : Stmt Assn Module -> JudgmentBody .
op wlpIsComplete : Stmt -> Bool .

*** Judgment produced by the assertion analyzer
sort Judgment .

op _:_ : JudgmentHead JudgmentBody -> Judgment
                                 [ctor prec 7 format (b b bnssss on)] .

*** List of judgments
sort JudgmentList .
subsort Judgment < JudgmentList .

op Nothing to verify : -> JudgmentList [ctor format (b b b on)] .
op __ : JudgmentList JudgmentList -> JudgmentList
            [ctor assoc prec 9 id: Nothing to verify format (b bn o)] .

*** Verification report
sort Report .
subsort Report < Config .

*** result of verification
op Verification of class__ : Id JudgmentList -> Report
                         [ctor gather (e &) format (b b osb b bnn o)] .

*** canonical verify command
op verify class_with simplifications_ : Id Qid -> Report
                                [ctor prec 3 format (b b b b b b on)] .

*** non-canonical verify command
op verify class_ : Id -> Report [prec 3] .

var B : BExp .
var C : Id .
var I : Assn .
var MOD : Module .
var PRETTY : PrettyAssn .
var P : Assn .
var Q : Assn .
vars S S1 S2 S1..SK SK+1..SN : Stmt .

eq Initialization code : Maintains the guarantee =
   Initialization code : Establishes the guarantee .
eq Initialization code : Maintains the guarantee iff PRETTY holds =
   Initialization code : Establishes the guarantee iff PRETTY holds .
eq Initialization code : Maintains the guarantee if PRETTY holds =
   Initialization code : Establishes the guarantee if PRETTY holds .
eq Initialization code : Breaks the guarantee =
   Initialization code : Fails to establish the guarantee .

eq Maintains the guarantee iff true holds = Maintains the guarantee .
eq Maintains the guarantee iff false holds = Breaks the guarantee .
```

```
  eq Maintains the guarantee if true holds = Maintains the guarantee .
  eq Maintains the guarantee if false holds = Don't know .
  eq Establishes the guarantee iff true holds =
     Establishes the guarantee .
  eq Establishes the guarantee iff false holds =
     Fails to establish the guarantee .
  eq Establishes the guarantee if true holds =
     Establishes the guarantee .
  eq Establishes the guarantee if false holds = Don't know .

  ceq judgmentBody(S, Q, MOD) =
      if wlpIsComplete(S) then
          Maintains the guarantee iff PRETTY holds
      else
          Maintains the guarantee if PRETTY holds
      fi
  if PRETTY := massaged(Q, MOD) .

  eq wlpIsComplete(prove P) = false .
  eq wlpIsComplete(while B do S od) = false .
  eq wlpIsComplete(inv I while B do S od) = false .
  eq wlpIsComplete(if B th S1 el S2 fi) =
     wlpIsComplete(S1) and wlpIsComplete(S2) .
  ceq wlpIsComplete(S1 ; S2) = wlpIsComplete(S1) and wlpIsComplete(S2)
  if S1 =/= emptyStmt and S2 =/= emptyStmt .
  eq wlpIsComplete(S1 [] S2) = wlpIsComplete(S1) and wlpIsComplete(S2) .
  eq wlpIsComplete(S1..SK ||| SK+1..SN) =
     wlpIsComplete(S1..SK) and wlpIsComplete(SK+1..SN) .
  eq wlpIsComplete([S]) = wlpIsComplete(S) .
  eq wlpIsComplete(S) = true [otherwise] .

  eq verify class C =
     verify class C with simplifications 'CREOL-SIMPLIFICATION-RULES .
endfm

mod CREOL-ASSERTION-ANALYZER is
  including CREOL-ASSERTION-ANALYZER-REPORT .

  *** auxiliary functions needed by the assertion analyzer
  op classInitializationJudgment :
     SuperList AGSpec TypedQualifiedIdList Config Module ~> Judgment .
  op classMethodJudgments :
     SuperList AGSpec TypedQualifiedIdList Config Module
     ~> JudgmentList .
  op methodJudgments :
     Id AGSpec TypedQualifiedIdList MtdMSet Config Module
     ~> JudgmentList .
  op methodJudgment :
     Id AGSpec TypedQualifiedIdList Id TypedIdList TypedIdList
     TypedIdList Stmt Config Module
     ~> Judgment .
  op classWritableAttributes : SuperList Config ~> TypedQualifiedIdList .
  op initializeVars : TypedQualifiedIdList -> Stmt .

  var AAL : TypedQualifiedIdList .
```

```
var AG : AGSpec .
var ASUM : Assn .
var C : Id .
var CONFIG : Config .
var EL : ExpList .
var GUAR : Assn .
var M : Id .
var MM : MtdMSet .
var MOD : Module .
var N : Int .
var PPL : TypedIdList .
var Q : Assn .
var QID : Qid .
vars S S' : Stmt .
vars SUPERL SUPERL' SUPERL'' SUPERL''' : SuperList .
var T0 : Type .
var VVL : TypedIdList .
var WWL : TypedIdList .
var XXL : TypedIdList .
var YYL : TypedIdList .
var Z0 : QualifiedId .
var ZZL : TypedQualifiedIdList .

ceq classInitializationJudgment(C, < ASUM, GUAR >, AAL, CONFIG, MOD) =
    Initialization code :
        judgmentBody(S,
            (ASUM ==> wlp(S, Q, < ASUM, GUAR >, AAL))
                { ~H~ : history |->
                    ['parent[self : any] -> self : any . new C[PPL]] },
            MOD)
if Q := GUAR { ~H~ : history |-> ~H~ : history
                                 ^^ [self : any . initialized] }
   /\ PPL := freshLogicalVarList(classParams(C, CONFIG), Q)
   /\ S := (initializeVars(AAL) ; initialPr(C[PPL], CONFIG)) .

eq classMethodJudgments(epsilon, AG, AAL, CONFIG, MOD) =
   Nothing to verify .
eq classMethodJudgments((SUPERL''', C[EL]), AG, AAL,
                        < C : Class | Impl: SUPERL, Ctrc: SUPERL',
                                      Inh: SUPERL'', Param: PPL,
                                      Att: WWL, Mtd: MM, ObjCnt: N,
                                      Asum: ASUM, Guar: GUAR >
                        CONFIG,
                        MOD) =
   methodJudgments(C, AG, AAL, MM,
                   < C : Class | Impl: SUPERL, Ctrc: SUPERL',
                                 Inh: SUPERL'', Param: PPL, Att: WWL,
                                 Mtd: MM, ObjCnt: N, Asum: ASUM,
                                 Guar: GUAR >
                   CONFIG,
                   MOD)
   classMethodJudgments((SUPERL''', SUPERL''), AG, AAL,
                        < C : Class | Impl: SUPERL, Ctrc: SUPERL',
                                      Inh: SUPERL'', Param: PPL,
                                      Att: WWL, Mtd: MM, ObjCnt: N,
```

```
                                        Asum: ASUM, Guar: GUAR >
                        CONFIG,
                        MOD) .

eq methodJudgments(C, AG, AAL, emptyMSet, CONFIG, MOD) =
   Nothing to verify .
eq methodJudgments(C, < ASUM, GUAR >, AAL,
                   < M : Method | In: XXL, Out: YYL,
                                  LVar: VVL, Code: S >
                   ++ MM, CONFIG, MOD) =
   methodJudgment(C, < ASUM, GUAR >, AAL, M, XXL, YYL, VVL, S, CONFIG,
                  MOD)
   methodJudgments(C, < ASUM, GUAR >, AAL, MM, CONFIG, MOD) .

ceq methodJudgment(C, < ASUM, GUAR >, AAL, M, XXL, YYL, VVL, S, CONFIG,
                   MOD) =
    Method M of C :
        judgmentBody(S',
            [ASUM
             && GUAR
             && 'lwf[~H~ : history, self : any]
             && 'mayAcquireProcessor[~H~ : history, self : any,
                                     caller : any, label : int]
             && [label : int % caller : any -> self : any . M[XXL]] in
                ~H~ : history
             && ! [[label : int % caller : any <- self : any . *] in
                   ~H~ : history]] ==>
            wlp(S', (GUAR) { ~H~ : history |-> ~H~ : history
                                ^^ [label : int % caller : any
                                    <- self : any . M[XXL ; YYL]] },
                < ASUM, GUAR >, AAL),
            MOD)
if S' := (initializeVars(YYL, VVL) ;
          qualifiedAndTyped(S, C, AAL, (XXL, YYL, VVL), CONFIG)) .

eq classWritableAttributes(epsilon, CONFIG) = epsilon .
eq classWritableAttributes((SUPERL'', C[EL]),
                           < C : Class | Impl: SUPERL, Ctrc: SUPERL',
                                         Inh: SUPERL'', Param: PPL,
                                         Att: WWL, Mtd: MM, ObjCnt: N,
                                         Asum: ASUM, Guar: GUAR >
                           CONFIG) =
   WWL @@ C,
   classWritableAttributes((SUPERL''', SUPERL''),
                           < C : Class | Impl: SUPERL, Ctrc: SUPERL',
                                         Inh: SUPERL'', Param: PPL,
                                         Att: WWL, Mtd: MM, ObjCnt: N,
                                         Asum: ASUM, Guar: GUAR >
                           CONFIG) .

eq initializeVars(epsilon) = emptyStmt .
eq initializeVars(Z0 : T0, ZZL) =
   Z0 : T0 := defaultValue(T0) ; initializeVars(ZZL) .

crl [start-verification] :
```

```
  {
     verify class C with simplifications QID
     CONFIG
  }
   =>
  {
     Verification of class C
     classInitializationJudgment(C, AG, AAL, CONFIG, MOD)
     classMethodJudgments(C, AG, AAL, CONFIG, MOD)
  }
  if AG := classAGSpec(C, CONFIG)
     /\ AAL := classWritableAttributes(C, CONFIG)
     /\ MOD := upModule(QID, false) .
endm
```

B.4 Interpreter Core

```
***(
    creol-interpreter-core.maude

    This file contains modules shared by the Creol interpreter for closed
    systems and the interpreter for open systems.
)

fmod CREOL-STATE is
  including CREOL-TYPED-QUALIFIED-IDENTIFIER .
  including CREOL-VALUE .

  *** Variable state
  sort State .

  op emptyState : -> State [ctor] .
  op [_|->_] : QualifiedId Value -> State [ctor] .
  op __ : State State -> State [ctor assoc prec 1 id: emptyState] .

  *** state from two lists
  op [_|->_] : QualifiedIdList ValueList -> State .

  *** state membership
  op _in_ : QualifiedId State -> Bool [prec 11] .

  *** qualification of a state
  *** e.g., ['x |-> 1]['y |-> 2] @@ 'C == ['x @ 'C |-> 1]['y @ 'C |-> 2]
  op _@@_ : State Id -> State [prec 3] .

  op asState : TypedQualifiedIdList -> State .

  var C : Id .
  var SIGMA : State .
  var T0 : Type .
  vars V V' V0 : Value .
  var VL : ValueList .
  var X : Id .
```

```
  vars Z Z' Z0 : QualifiedId .
  var ZL : QualifiedIdList .
  var ZZL : TypedQualifiedIdList .

  eq [epsilon |-> epsilon] = emptyState .
  ceq [Z0, ZL |-> V0, VL] = [Z0 |-> V0][ZL |-> VL]
  if ZL =/= epsilon .

  *** Definition T5 (State Membership)
  eq Z in emptyState = false .
  eq Z in SIGMA [Z' |-> V] = (Z == Z') or (Z in SIGMA) .

  eq [Z |-> V] SIGMA [Z |-> V'] = SIGMA [Z |-> V'] .

  eq emptyState @@ C = emptyState .
  eq SIGMA [X |-> V] @@ C = (SIGMA @@ C) [X @ C |-> V] .

  eq asState(epsilon) = emptyState .
  eq asState(Z0 : T0, ZZL) = [Z0 |-> defaultValue(T0)] asState(ZZL) .
endfm

fmod CREOL-EXPRESSION-EVALUATION is
  including CREOL-HISTORY-EXPRESSION .
  including CREOL-STATE .

  *** evaluation of expressions
  op {_}_ : ExpList State ~> ValueList [prec 3] .

  vars A A1 A2 : AExp .
  var B : BExp .
  var C : Id .
  vars E E0 E1 E2 : Exp .
  vars EL EL' : ExpList .
  vars EPEXP EPEXP1 EPEXP2 : EventPatExp .
  var F : Id .
  var HEXP : HistoryExp .
  var HPEXP : HistoryPatExp .
  var M : Id .
  var N : Int .
  var O : OId .
  vars OEXP OEXP' : OExp .
  vars PHI PHI1 PHI2 : Assn .
  var SIGMA : State .
  var T : Type .
  vars V V1 V2 : Value .
  var VL : ValueList .
  vars Z Z' : QualifiedId .

  *** Definition T12 (Evaluation of Variable)
  *** extended to deal with typed identifiers
  eq {Z : T} SIGMA [Z' |-> V] = if Z == Z' then V else {Z} SIGMA fi .

  *** Definition T13 (Evaluation of Generic Expression)
  eq {if B th E1 el E2 fi} SIGMA =
     if {B} SIGMA then {E1} SIGMA else {E2} SIGMA fi .
```

```
eq {[E]} SIGMA = {E} SIGMA .

*** evaluation of custom functions
eq {F[EL]} SIGMA = @ F[{EL} SIGMA] .
eq {@ F[VL]} SIGMA = @ F[VL] .

*** Definition T14 (Evaluation of Arithmetic Expression)
eq {N} SIGMA = N .
eq {plus A} SIGMA = {A} SIGMA .
eq {minus A} SIGMA = - {A} SIGMA .
eq {A1 times A2} SIGMA = {A1} SIGMA * {A2} SIGMA .
eq {A1 div A2} SIGMA = {A1} SIGMA quo {A2} SIGMA .
eq {A1 plus A2} SIGMA = {A1} SIGMA + {A2} SIGMA .
eq {A1 minus A2} SIGMA = {A1} SIGMA - {A2} SIGMA .

*** Definition T15 (Evaluation of Boolean Expression)
*** altered to ensure that junk terms don't evaluate to valid terms and
*** to handle quantifier-free assertions
eq {true} SIGMA = true .
eq {false} SIGMA = false .
ceq {E1 eq E2} SIGMA = V1 == V2
if V1 := {E1} SIGMA /\ V2 := {E2} SIGMA .
ceq {E1 ne E2} SIGMA = V1 =/= V2
if V1 := {E1} SIGMA /\ V2 := {E2} SIGMA .
eq {E1 lt E2} SIGMA = {E1} SIGMA < {E2} SIGMA .
eq {E1 gt E2} SIGMA = {E1} SIGMA > {E2} SIGMA .
eq {E1 le E2} SIGMA = {E1} SIGMA <= {E2} SIGMA .
eq {E1 ge E2} SIGMA = {E1} SIGMA >= {E2} SIGMA .
eq {! PHI} SIGMA = not {PHI} SIGMA .
eq {PHI1 && PHI2} SIGMA = {PHI1} SIGMA and {PHI2} SIGMA .
eq {PHI1 || PHI2} SIGMA = {PHI1} SIGMA or {PHI2} SIGMA .
eq {PHI1 ==> PHI2} SIGMA = {PHI1} SIGMA implies {PHI2} SIGMA .
eq {PHI1 <==> PHI2} SIGMA = {PHI1} SIGMA == {PHI2} SIGMA .

*** Definition T16 (Evaluation of Object Expression)
eq {O} SIGMA = O .

*** Definition T17 (Evaluation of Expression List)
eq {epsilon} SIGMA = epsilon .
ceq {E0, EL} SIGMA = {E0} SIGMA, {EL} SIGMA if EL =/= epsilon .

eq {[OEXP -> OEXP' . new C[EL]]} SIGMA =
   [{OEXP} SIGMA -> {OEXP'} SIGMA . new C[{EL} SIGMA]] .
eq {[A % OEXP -> OEXP' . M @ C[EL]]} SIGMA =
   [{A} SIGMA % {OEXP} SIGMA -> {OEXP'} SIGMA . M @ C[{EL} SIGMA]] .
eq {[A % OEXP <- OEXP' . M @ C[EL ; EL']]} SIGMA =
   [{A} SIGMA % {OEXP} SIGMA <- {OEXP'} SIGMA
    . M @ C[{EL} SIGMA ; {EL'} SIGMA]] .
eq {[OEXP . initialized]} SIGMA = [{OEXP} SIGMA . initialized] .
eq {[OEXP . release]} SIGMA = [{OEXP} SIGMA . release] .
eq {[A % OEXP . reenter]} SIGMA =
   [{A} SIGMA % {OEXP} SIGMA . reenter] .

eq {[OEXP -> *]} SIGMA = [{OEXP} SIGMA -> *] .
eq {[OEXP <- *]} SIGMA = [{OEXP} SIGMA <- *] .
```

```
  eq {[* -> OEXP]} SIGMA = [* -> {OEXP} SIGMA] .
  eq {[* <- OEXP]} SIGMA = [* <- {OEXP} SIGMA] .
  eq {[OEXP . reenter]} SIGMA = [{OEXP} SIGMA . reenter] .
  eq {in[OEXP]} SIGMA = in[{OEXP} SIGMA] .
  eq {out[OEXP]} SIGMA = out[{OEXP} SIGMA] .
  eq {ctl[OEXP]} SIGMA = ctl[{OEXP} SIGMA] .

  eq {~ EPEXP} SIGMA = ~ {EPEXP} SIGMA .
  eq {EPEXP1 & EPEXP2} SIGMA = {EPEXP1} SIGMA & {EPEXP2} SIGMA .
  eq {EPEXP1 | EPEXP2} SIGMA = {EPEXP1} SIGMA | {EPEXP2} SIGMA .

  eq {HEXP / EPEXP} SIGMA = {HEXP} SIGMA / {EPEXP} SIGMA .
  eq {EPEXP in HEXP} SIGMA = {EPEXP} SIGMA in {HEXP} SIGMA .

  eq {emptyHistory} SIGMA = emptyHistory .
  ceq {HPEXP ^^ EPEXP} SIGMA = {HPEXP} SIGMA ^^ {EPEXP} SIGMA
  if HPEXP =/= emptyHistory .

  eq {HEXP bw HPEXP} SIGMA = {HEXP} SIGMA bw {HPEXP} SIGMA .
  eq {HEXP bw HPEXP} SIGMA = {HEXP} SIGMA bw {HPEXP} SIGMA .

  eq {HEXP ew HPEXP} SIGMA = {HEXP} SIGMA ew {HPEXP} SIGMA .
  eq {HEXP ew HPEXP} SIGMA = {HEXP} SIGMA ew {HPEXP} SIGMA .

  eq {HPEXP pr HEXP} SIGMA = {HPEXP} SIGMA pr {HEXP} SIGMA .
  eq {HPEXP pr HEXP} SIGMA = {HPEXP} SIGMA pr {HEXP} SIGMA .
endfm

fmod CREOL-PROCESS is
  including CREOL-STATE .
  including CREOL-STATEMENT .

  *** Suspended process
  sort Process .

  op <_,_> : Stmt State -> Process [ctor format (d d d s d d)] .

  op pr : Process -> Stmt .
  op lvar : Process -> State .

  *** Multiset of suspended processes
  sort ProcessMSet .
  subsort Process < ProcessMSet .
  subsort Process < MSetElem .
  subsort ProcessMSet < MSet .
  subsort EmptyMSet < ProcessMSet .

  op _++_ : ProcessMSet ProcessMSet -> ProcessMSet [ctor ditto] .

  var BETA : State .
  var S : Stmt .

  eq pr(< S, BETA >) = S .
  eq lvar(< S, BETA >) = BETA .
endfm
```

```
fmod CREOL-MESSAGE is
  including CREOL-EXPRESSION .
  including CREOL-PROGRAM .

  *** Message body
  sort MsgBody .
  subsort Msg < Config .

  op Invoke : OId Int QualifiedId ValueList -> MsgBody
                                                  [ctor format (!r or)] .
  op Reply : Int ValueList -> MsgBody [ctor format (!r or)] .

  *** Message with a target
  sort Msg .

  op _to_ : MsgBody OId -> Msg [ctor prec 3 format (r r! or n)] .

  *** Multiset of message bodies
  sort MsgBodyMSet .
  subsort MsgBody < MsgBodyMSet .
  subsort MsgBody < MSetElem .
  subsort MsgBodyMSet < MSet .
  subsort EmptyMSet < MsgBodyMSet .

  op _++_ : MsgBodyMSet MsgBodyMSet -> MsgBodyMSet [ctor ditto] .
endfm

fmod CREOL-OBJECT is
  including CREOL-MESSAGE .
  including CREOL-PROCESS .

  *** Field of an object term
  sort ObjectField .

  op Pr:_ : Stmt -> ObjectField [ctor prec 35 format (y! oy y)] .
  op LVar:_ : State -> ObjectField [ctor prec 35 format (y! oy y)] .
  op Att:_ : State -> ObjectField [ctor prec 35 format (y! oy y)] .
  op PrQ:_ : ProcessMSet -> ObjectField [ctor prec 35 format (y! oy y)] .
  op MsgQ:_ : MsgBodyMSet -> ObjectField
                                        [ctor prec 35 format (y! oy y)] .
  op LabCnt:_ : Int -> ObjectField [ctor prec 35 format (y! oy y)] .

  *** Comma-separated multiset of object fields
  sort ObjectFieldMSet .
  subsort ObjectField < ObjectFieldMSet .

  op noFields : -> ObjectField [ctor] .
  op _,_ : ObjectFieldMSet ObjectFieldMSet -> ObjectFieldMSet
               [ctor assoc comm prec 37 id: noFields format (d d s d)] .

  *** Creol object
  sort Object .
  subsort Object < Config .
```

```
  op <_:_|_> : OId Id ObjectFieldMSet -> Object
                                        [ctor format (y y! oy y y oy y n)] .
endfm

mod CREOL-INTERPRETER-CORE is
  including CREOL-EXPRESSION-EVALUATION .
  including CREOL-OBJECT .

  *** used internally as part of the operational semantics
  op return_ : ExpList -> SingleStmt [ctor prec 23] .
  op continue_ : Int -> SingleStmt [ctor prec 23] .
  op _///_ : Stmt Stmt -> SingleStmt [ctor prec 31] .

  *** auxiliary functions
  op boundMtd :
     SuperList OId Int QualifiedId ValueList Config ~> Process .
  op initialAtt : SuperList Config ~> State .
  op nextOId : Id Config ~> OId .
  op incrementObjCnt : Id Config ~> Config .
  op satisfied : Guard State MsgBodyMSet -> Bool .
  op replied : Int MsgBodyMSet -> Bool .
  op enabled : Stmt State MsgBodyMSet -> Bool .
  op ready : Stmt State MsgBodyMSet -> Bool .
  op shadowed : TypedIdList IdList -> ExpList .

  var ALPHA : State .
  var ASUM : Assn .
  var B : BExp .
  var BETA : State .
  var C : Id .
  var CONFIG : Config .
  var E0 : Exp .
  var EL : ExpList .
  var ETC : ObjectFieldMSet .
  vars G G1 G2 : Guard .
  var GUAR : Assn .
  vars K K' : Int .
  var LL : TypedId .
  var M : Id .
  var MM : MtdMSet .
  var N : Int .
  vars O O' : OId .
  var OEXP : OExp .
  var PHI : Assn .
  var PPL : TypedIdList .
  var Q : MsgBodyMSet .
  vars S S' S1 S1' S2 S1..SK SK+1..SN S2..SN : Stmt .
  var SIGMA : State .
  var SS : SingleStmt .
  vars SUPERL SUPERL' SUPERL'' SUPERL''' : SuperList .
  vars T T' T0 : Type .
  var VL : ValueList .
  var VVL : TypedIdList .
  var WL : ValueList .
  var WWL : TypedIdList .
```

```
vars XXL XXL' : TypedIdList .
vars Y Y0 : Id .
var YYL : TypedIdList .
var Z0 : QualifiedId .
var ZZL : TypedQualifiedIdList .

*** Definition T3 (Synthetic Statements)
eq ! OEXP . M[EL] = nu ! OEXP . M[EL] .
eq ! M @ C[EL] = nu ! M @ C[EL] .
eq OEXP . M[EL ; ZZL] = nu ! OEXP . M[EL] ; nu ?[ZZL] .
eq M @ C[EL ; ZZL] = nu ! M @ C[EL] ; nu ?[ZZL] .
eq await G &&& LL ?[ZZL] = await G &&& LL ? ; LL ?[ZZL] .
eq await G &&& OEXP . M[EL ; ZZL] =
   nu ! OEXP . M[EL] ; await G &&& nu ? ; nu ?[ZZL] .
eq await G &&& M @ C[EL ; ZZL] =
   nu ! M @ C[EL] ; await G &&& nu ? ; nu ?[ZZL] .
eq await LL ?[ZZL] = await LL ? ; LL ?[ZZL] .
eq await OEXP . M[EL ; ZZL] =
   nu ! OEXP . M[EL] ; await nu ? ; nu ?[ZZL] .
eq await M @ C[EL ; ZZL] = nu ! M @ C[EL] ; await nu ? ; nu ?[ZZL] .
eq if B th S1 fi = if B th S1 el skip fi .

*** Definition T4 (Residual Statements)
eq epsilon := epsilon = emptyStmt .
eq emptyStmt /// S = S .

*** Definition T6 (Guard Satisfaction)
eq satisfied(B, SIGMA, Q) = {B} SIGMA .
eq satisfied(LL ?, SIGMA, Q) = replied({LL} SIGMA, Q) .
eq satisfied(wait, SIGMA, Q) = false .
eq satisfied(G1 &&& G2, SIGMA, Q) =
   satisfied(G1, SIGMA, Q) and satisfied(G2, SIGMA, Q) .

*** Definition T7 (Replied Predicate)
eq replied(K, emptyMSet) = false .
eq replied(K, Invoke(O, K', M @ C, VL) ++ Q) = replied(K, Q) .
eq replied(K, Reply(K', WL) ++ Q) = (K == K') or replied(K, Q) .

*** Definition T8 (Statement Enabledness)
eq enabled(await G ; S, SIGMA, Q) = satisfied(G, SIGMA, Q) .
eq enabled((S1 [] S2) ; S, SIGMA, Q) =
   enabled(S1, SIGMA, Q) or enabled(S2, SIGMA, Q) .
eq enabled((S1..SK ||| SK+1..SN) ; S, SIGMA, Q) =
   enabled(S1..SK, SIGMA, Q) or enabled(SK+1..SN, SIGMA, Q) .
eq enabled([S] ; S', SIGMA, Q) = enabled(S, SIGMA, Q) .
eq enabled(S, SIGMA, Q) = true [otherwise] .

*** Definition T11 (Statement Readiness)
eq ready(LL ?[ZZL] ; S, SIGMA, Q) = satisfied(LL ?, SIGMA, Q) .
eq ready(await G ; S, SIGMA, Q) = satisfied(G, SIGMA, Q) .
eq ready((S1 [] S2) ; S, SIGMA, Q) =
   ready(S1, SIGMA, Q) or ready(S2, SIGMA, Q) .
eq ready((S1..SK ||| SK+1..SN) ; S, SIGMA, Q) =
   ready(S1..SK, SIGMA, Q) or ready(SK+1..SN, SIGMA, Q) .
eq ready([S] ; S', SIGMA, Q) = ready(S, SIGMA, Q) .
```

```
eq ready(S, SIGMA, Q) = true [otherwise] .

eq shadowed(epsilon, XXL) = epsilon .
eq shadowed(Y : T, (XXL, Y : T', XXL')) = defaultValue(T) .
eq shadowed(Y : T, XXL) = Y [otherwise] .
ceq shadowed((Y0 : T0, YYL), XXL) =
    shadowed(Y0 : T0, XXL), shadowed(YYL, XXL)
if YYL =/= epsilon .

eq initialAtt(epsilon, CONFIG) = emptyState .
eq initialAtt((SUPERL''', C[EL]),
              < C : Class | Impl: SUPERL, Ctrc: SUPERL', Inh: SUPERL'',
                            Param: PPL, Att: WWL, Mtd: MM, ObjCnt: N,
                            Asum: ASUM, Guar: GUAR >
              CONFIG) =
   initialAtt((SUPERL''', SUPERL''),
              < C : Class | Impl: SUPERL, Ctrc: SUPERL', Inh: SUPERL'',
                            Param: PPL, Att: WWL, Mtd: MM, ObjCnt: N,
                            Asum: ASUM, Guar: GUAR >
              CONFIG)
   asState((PPL, WWL) @@ C) .

eq nextOId(C,
           < C : Class | Impl: SUPERL, Ctrc: SUPERL', Inh: SUPERL'',
                         Param: PPL, Att: WWL, Mtd: MM, ObjCnt: N,
                         Asum: ASUM, Guar: GUAR >
           CONFIG) =
   C # N .

eq incrementObjCnt(C,
                   < C : Class | Impl: SUPERL, Ctrc: SUPERL',
                         Inh: SUPERL'', Param: PPL, Att: WWL, Mtd: MM,
                         ObjCnt: N, Asum: ASUM, Guar: GUAR >
                   CONFIG) =
   < C : Class | Impl: SUPERL, Ctrc: SUPERL', Inh: SUPERL'',
                 Param: PPL, Att: WWL, Mtd: MM, ObjCnt: N + 1,
                 Asum: ASUM, Guar: GUAR >
   CONFIG .

*** can't find method in class; proceed with superclasses
ceq boundMtd((SUPERL''', C[EL]), O', K, M, VL,
             < C : Class | Impl: SUPERL, Ctrc: SUPERL', Inh: SUPERL'',
                           Param: PPL, Att: WWL, Mtd: MM, ObjCnt: N,
                           Asum: ASUM, Guar: GUAR >
             CONFIG) =
    boundMtd((SUPERL''', SUPERL''), O', K, M, VL,
             < C : Class | Impl: SUPERL, Ctrc: SUPERL', Inh: SUPERL'',
                           Param: PPL, Att: WWL, Mtd: MM, ObjCnt: N,
                           Asum: ASUM, Guar: GUAR >
             CONFIG)
if not < M : Method | * > in mset MM .

*** found method in class
eq boundMtd((SUPERL''', C[EL]), O', K, M, VL,
            < C : Class | Impl: SUPERL, Ctrc: SUPERL', Inh: SUPERL'',
```

```
                          Param: PPL, Att: WWL,
                          Mtd: < M : Method | In: XXL, Out: YYL,
                                              LVar: VVL,
                                              Code: S > ++ MM,
                          ObjCnt: N, Asum: ASUM, Guar: GUAR >
           CONFIG) =
   < qualified(S, (SUPERL'', C[EL]), (XXL, YYL, VVL),
               < C : Class | Impl: SUPERL, Ctrc: SUPERL',
                             Inh: SUPERL'', Param: PPL, Att: WWL,
                             Mtd: < M : Method | In: XXL, Out: YYL,
                                                 LVar: VVL,
                                                 Code: S > ++ MM,
                             ObjCnt: N, Asum: ASUM, Guar: GUAR >
               CONFIG) ;
     return shadowed(YYL, VVL),
     [caller |-> O'][label |-> K] [asIdList(XXL) |-> VL]
     asState(YYL, VVL) > .

*** eliminate invariant clause in while loop
eq inv PHI while B do S od = while B do S od .

*** Rewrite Rule S2 (Null Statement)
rl [null-statement] :
< O : C | Pr: skip ; S, ETC >
 =>
< O : C | Pr: S, ETC > .

*** Rewrite Rule S3 (Abnormal Termination)
rl [abnormal-termination] :
< O : C | Pr: abort ; S, ETC >
 =>
emptyConfig .

*** Rewrite Rule S4 (Inline Assertion)
rl [inline-assertion] :
< O : C | Pr: prove PHI ; S, ETC >
 =>
< O : C | Pr: S, ETC > .

*** Rewrite Rule S5 (Assignment)
rl [assignment] :
< O : C | Pr: (Z0 : T0, ZZL) := (E0, EL) ; S, LVar: BETA, Att: ALPHA,
          ETC >
 =>
if Z0 in BETA then
    < O : C | Pr: ZZL := {EL} ALPHA BETA ; S,
              LVar: BETA [Z0 |-> {E0} ALPHA BETA], Att: ALPHA, ETC >
else
    < O : C | Pr: ZZL := {EL} ALPHA BETA ; S, LVar: BETA,
              Att: ALPHA [Z0 |-> {E0} ALPHA BETA], ETC >
fi .

*** Rewrite Rule S6 (If Statement)
rl [if-statement] :
< O : C | Pr: if B th S1 el S2 fi ; S, LVar: BETA, Att: ALPHA, ETC >
```

```
 =>
if {B} ALPHA BETA then
    < O : C | Pr: S1 ; S, LVar: BETA, Att: ALPHA, ETC >
else
    < O : C | Pr: S2 ; S, LVar: BETA, Att: ALPHA, ETC >
fi .

*** Rewrite Rule S7 (While Loop)
rl [while-loop] :
< O : C | Pr: while B do S od ; S', LVar: BETA, Att: ALPHA, ETC >
 =>
if {B} ALPHA BETA then
    < O : C | Pr: S ; while B do S od ; S', LVar: BETA, Att: ALPHA,
              ETC >
else
    < O : C | Pr: S', LVar: BETA, Att: ALPHA, ETC >
fi .

*** Rewrite Rule S8 (Guard Crossing)
crl [guard-crossing] :
< O : C | Pr: await G ; S, LVar: BETA, Att: ALPHA, MsgQ: Q, ETC >
 =>
< O : C | Pr: S, LVar: BETA, Att: ALPHA, MsgQ: Q, ETC >
if satisfied(G, ALPHA BETA, Q) .

*** Rewrite Rule S9 (Nondeterministic Choice)
crl [nondeterministic-choice] :
< O : C | Pr: (S1 [] S2) ; S, LVar: BETA, Att: ALPHA, MsgQ: Q, ETC >
 =>
< O : C | Pr: S1 ; S, LVar: BETA, Att: ALPHA, MsgQ: Q, ETC >
if ready(S1, ALPHA BETA, Q) .

*** Rewrite Rule S10 (Nondeterministic Merge)
crl [nondeterministic-merge] :
< O : C | Pr: (S1 ||| S2..SN) ; S, LVar: BETA, Att: ALPHA, MsgQ: Q,
          ETC >
 =>
< O : C | Pr: (S1 /// S2..SN) ; S, LVar: BETA, Att: ALPHA, MsgQ: Q,
          ETC >
if ready(S1, ALPHA BETA, Q) and simpleBranch(S1) .

*** Rewrite Rule S11 (Left Merge)
rl [left-merge] :
< O : C | Pr: ((SS ; S1') /// S2..SN) ; S, LVar: BETA, Att: ALPHA,
          MsgQ: Q, ETC >
 =>
if enabled(SS, ALPHA BETA, Q) then
    < O : C | Pr: SS ; (S1' /// S2..SN) ; S, LVar: BETA, Att: ALPHA,
              MsgQ: Q, ETC >
else
    < O : C | Pr: ((SS ; S1') ||| S2..SN) ; S, LVar: BETA,
              Att: ALPHA, MsgQ: Q, ETC >
fi .

*** Rewrite Rule S12 (Parenthesized Statement)
```

```
  rl [parenthesized-statement] :
  < O : C | Pr: [S] ; S', ETC >
   =>
  < O : C | Pr: S ; S', ETC > .

  *** Rewrite Rule S22 (Asynchronous Reply)
  crl [asynchronous-reply] :
  < O : C | Pr: LL ?[ZZL] ; S, LVar: BETA, MsgQ: Reply(K, WL) ++ Q, ETC >
   =>
  < O : C | Pr: ZZL := WL ; S, LVar: BETA [LL |-> -1], MsgQ: Q, ETC >
  if {LL} BETA == K .
endm
```

B.5 Interpreter for Closed System

```
***(
    creol-closed-interpreter.maude

    This file implements the Creol interpreter for closed systems, based
    on the operational semantics described in Section 4.3 of Verification
    of Assertions in Creol Programs. See Appendix A for a user's guide.
)

mod CREOL-INTERPRETER-FOR-CLOSED-SYSTEMS is
  including CREOL-INTERPRETER-CORE .

  var ALPHA : State .
  vars BETA BETA' : State .
  vars C C' : Id .
  var CONFIG : Config .
  var EL : ExpList .
  var ETC : ObjectFieldMSet .
  var K : Int .
  var LL : TypedId .
  var M : Id .
  vars O O' : OId .
  var OEXP : OExp .
  var P : ProcessMSet .
  var Q : MsgBodyMSet .
  vars S S' : Stmt .
  var VL : ValueList .
  var WL : ValueList .
  var ZZ : TypedQualifiedId .
  var ZZL : TypedQualifiedIdList .

  *** closed system interpreter bootstrap command
  op bootstrap system_ : IdWithArgs -> Object
                                        [ctor prec 3 format (b b b on)] .

  *** Rewrite Rule S1 (System Bootstrapping)
  rl [system-bootstrapping] :
  {
     bootstrap system C[EL]
```

```
   CONFIG
}
 =>
{
   < C # 0 : C | Pr: initialPr(C[{EL} emptyState], CONFIG),
                 LVar: emptyState,
                 Att: initialAtt(C[{EL} emptyState], CONFIG)
                      [self |-> C # 0],
                PrQ: emptyMSet, MsgQ: emptyMSet, LabCnt: 0 >
   incrementObjCnt(C, CONFIG)
} .

*** Rewrite Rule S13 (Object Creation)
crl [object-creation] :
{
   < O : C | Pr: ZZ := new C'[EL] ; S, LVar: BETA, Att: ALPHA, ETC >
   CONFIG
}
 =>
{
   < O : C | Pr: ZZ := O' ; S, LVar: BETA, Att: ALPHA, ETC >
   < O' : C' | Pr: initialPr(C'[{EL} ALPHA BETA], CONFIG),
               LVar: emptyState,
               Att: initialAtt(C', CONFIG) [self |-> O'],
               PrQ: emptyMSet, MsgQ: emptyMSet, LabCnt: 0 >
   incrementObjCnt(C', CONFIG)
}
if O' := nextOId(C', CONFIG) .

*** Rewrite Rule S14 (Process Suspension)
crl [process-suspension] :
< O : C | Pr: S, LVar: BETA, Att: ALPHA, PrQ: P, MsgQ: Q, ETC >
 =>
< O : C | Pr: emptyStmt, LVar: emptyState, Att: ALPHA,
          PrQ: P ++ < clearWait(S), BETA >, MsgQ: Q, ETC >
if not enabled(S, ALPHA BETA, Q) .

*** Rewrite Rule S15 (Process Activation)
crl [process-activation] :
< O : C | Pr: emptyStmt, LVar: BETA, Att: ALPHA,
          PrQ: < S', BETA' > ++ P, MsgQ: Q, ETC >
 =>
< O : C | Pr: S', LVar: BETA', Att: ALPHA, PrQ: P, MsgQ: Q, ETC >
if ready(S', ALPHA BETA', Q) .

*** Rewrite Rule S16 (Asynchronous Invocation)
rl [asynchronous-invocation] :
< O : C | Pr: LL ! OEXP . M[EL] ; S, LVar: BETA, Att: ALPHA, LabCnt: K,
          ETC >
 =>
< O : C | Pr: S, LVar: BETA [LL |-> K], Att: ALPHA, LabCnt: K + 1,
          ETC >
Invoke(O, K, M, {EL} ALPHA BETA) to {OEXP} ALPHA BETA .

*** Rewrite Rule S17 (Local Asynchronous Invocation)
```

```
rl [local-asynchronous-invocation] :
< O : C | Pr: LL ! M @ C'[EL] ; S, LVar: BETA, Att: ALPHA, LabCnt: K,
          ETC >
 =>
< O : C | Pr: S, LVar: BETA [LL |-> K], Att: ALPHA, LabCnt: K + 1,
          ETC >
Invoke(O, K, M @ C', {EL} ALPHA BETA) to O .

*** Rewrite Rule S18 (Transport of Invocation Message)
rl [transport-of-invocation-message] :
< O : C | MsgQ: Q, ETC >
Invoke(O', K, M @ C', VL) to O
 =>
< O : C | MsgQ: Q ++ Invoke(O', K, M @ C', VL), ETC > .

*** Rewrite Rule S19 (Method Binding)
rl [method-binding] :
{
   < O : C | PrQ: P, MsgQ: Invoke(O', K, M @ C', VL) ++ Q, ETC >
   CONFIG
}
 =>
{
   < O : C | PrQ: P ++ boundMtd(if C' == none then C else C' fi, O', K,
                               M, VL, CONFIG),
             MsgQ: Q, ETC >
   CONFIG
} .

*** Rewrite Rule S20 (Method Return)
rl [method-return] :
< O : C | Pr: return EL ; S, LVar: BETA, Att: ALPHA, ETC >
 =>
< O : C | Pr: S, LVar: BETA, Att: ALPHA, ETC >
Reply({label} BETA, {EL} ALPHA BETA) to {caller} BETA .

*** Rewrite Rule S21 (Transport of Reply Message)
rl [transport-of-reply-message] :
< O : C | MsgQ: Q, ETC >
Reply(K, WL) to O
 =>
< O : C | MsgQ: Q ++ Reply(K, WL), ETC > .

*** Rewrite Rule S23 (Local Reentry)
crl [local-reentry] :
< O : C | Pr: LL ?[ZZL] ; S, LVar: BETA, PrQ: < S', BETA' > ++ P, ETC >
 =>
< O : C | Pr: S' ; continue {LL} BETA, LVar: BETA',
          PrQ: P ++ < LL ?[ZZL] ; S, BETA >, ETC >
if {caller} BETA' == O and {label} BETA' == {LL} BETA .

*** Rewrite Rule S24 (Local Continuation)
crl [local-continuation] :
< O : C | Pr: continue K, LVar: BETA,
          PrQ: < LL ?[ZZL] ; S, BETA' > ++ P, ETC >
```

```
   =>
  < O : C | Pr: LL ?[ZZL] ; S, LVar: BETA', PrQ: P, ETC >
  if {LL} BETA' == K .
endm
```

B.6 Interpreter for Open System

```
***(
    creol-open-interpreter.maude

    This file implements the Creol interpreter for open systems, based on
    the operational semantics described in Section 4.4 of Verification of
    Assertions in Creol Programs. See Appendix A for a user's guide.
)

fmod CREOL-RANDOM-DATA is
  including CREOL-HISTORY .
  including CREOL-PROGRAM .
  including CREOL-STATE .

  *** User-supplied random data command
  sort RandomData .
  subsort RandomData < Config .

  op random data_ : History -> RandomData
                                      [ctor prec 23 format (b b b on)] .
  op random data_ : State -> RandomData
                                      [ctor prec 23 format (b b b on)] .
endfm

mod CREOL-INTERPRETER-FOR-OPEN-SYSTEMS is
  including CREOL-ASSUME-GUARANTEE-SPECIFICATION .
  including CREOL-INTERPRETER-CORE .
  including CREOL-RANDOM-DATA .
  including CREOL-TYPED-QUALIFIED-IDENTIFIER .

  *** canonical open system interpreter bootstrap commands
  op bootstrap object_:= new_with parent_ :
     OId IdWithArgs OId -> Object
                           [ctor prec 3 format (b b b b b b b b b on)] .
  op bootstrap method_._[_]
               with class_,caller_,label_,history_,attributes_:=_ :
     OId QualifiedId ValueList Id OId Int History QualifiedIdList
     ValueList -> Object
             [ctor prec 23 format (b b b b b b b b sb b b b sb b b sb b b
                                   sb b b sb b b b on)] .

  *** non-canonical bootstrap command
  op bootstrap method_._[_]
               with class_,caller_,label_,history_ :
     OId QualifiedId ValueList Id OId Int History -> Object
              [format (b b b b b b b b sb b b b sb b b sb b b sb b on)] .
```

```
*** extra object fields needed by this semantics
op Asum:_ : Assn -> ObjectField [ctor prec 35 format (y! oy y)] .
op Guar:_ : Assn -> ObjectField [ctor prec 35 format (y! oy y)] .
op ROAtt:_ : TypedQualifiedIdList -> ObjectField
                                      [ctor prec 35 format (y! oy y)] .

*** auxiliary functions
op classROAtt : Id Config ~> TypedQualifiedIdList .
op inheritedROAtt : SuperList Config ~> TypedQualifiedIdList .
op release :
   Assn Assn OId OId Int TypedQualifiedIdList State State -> Bool .
op reenter :
   Assn Assn OId OId Int Int TypedQualifiedIdList State State -> Bool .
op interleave : Assn OId State History -> Bool .
op compatibleStates : State State -> Bool .
op lwf : History OId -> Bool .
op wf : History -> Bool .
op pending : History OId OId Int -> Bool .
op readOnly : TypedQualifiedIdList State State -> Bool .
op mayAcquireProcessor : History OId OId Int -> Bool .
op replyEvent : History OId OId Int ValueList ~> Event .
op replies : History OId -> MsgBodyMSet .
op trailingReplySeqNum : History -> Int .
op nextLabel : History OId -> Int .
op nextChild : History OId -> Int .

vars ALPHA ALPHA' : State .
var ASUM : Assn .
var BETA : State .
vars C C' : Id .
var CONFIG : Config .
var EL : ExpList .
var ETC : ObjectFieldMSet .
var EV : Event .
var GUAR : Assn .
vars H H' : History .
vars K K' : Int .
var LL : TypedId .
var M : Id .
var MM : MtdMSet .
var N : Int .
vars O O' : OId .
var OEXP : OExp .
var PPL : TypedIdList .
var Q : MsgBodyMSet .
var S : Stmt .
vars SIGMA SIGMA' SIGMA'' : State .
vars SUPERL SUPERL' SUPERL'' SUPERL''' : SuperList .
vars V V' : Value .
var VL : ValueList .
var WL : ValueList .
var WWL : TypedIdList .
var Z : QualifiedId .
var ZL : QualifiedIdList .
vars ZZ ZZ0 : TypedQualifiedId .
```

```
var ZZL : TypedQualifiedIdList .

eq bootstrap method O . M @ C'[VL]
            with class C, caller O', label K, history H =
   bootstrap method O . M @ C'[VL]
            with class C, caller O', label K, history H,
                 attributes epsilon := epsilon .

eq classROAtt(C, CONFIG) = inheritedROAtt(C, CONFIG), self : C .

eq inheritedROAtt(epsilon, CONFIG) = epsilon .
eq inheritedROAtt((SUPERL''', C[EL]),
                  < C : Class | Impl: SUPERL, Ctrc: SUPERL',
                                Inh: SUPERL'', Param: PPL, Att: WWL,
                                Mtd: MM, ObjCnt: N, Asum: ASUM,
                                Guar: GUAR >
                  CONFIG) =
   inheritedROAtt((SUPERL''', SUPERL''),
                  < C : Class | Impl: SUPERL, Ctrc: SUPERL',
                                Inh: SUPERL'', Param: PPL, Att: WWL,
                                Mtd: MM, ObjCnt: N, Asum: ASUM,
                                Guar: GUAR >
                  CONFIG),
   (PPL @@ C) .

*** Definition T18 (Processor Release Predicate)
eq release(ASUM, GUAR, O, O', K, ZZL, ALPHA, ALPHA') =
   lwf({~H~} ALPHA', O)
   and {~H~} ALPHA ^^ [O . release] pr {~H~} ALPHA'
   and mayAcquireProcessor({~H~} ALPHA', O, O', K)
   and pending({~H~} ALPHA', O', O, K)
   and compatibleStates(ALPHA, ALPHA')
   and readOnly(ZZL, ALPHA, ALPHA')
   and ((({~H~} ALPHA ^^ [O . release]) / (out[O] | ctl[O]) ==
         {~H~} ALPHA' / (out[O] | ctl[O])) implies
        (ALPHA [~H~ |-> {~H~} ALPHA'] == ALPHA'))
   and ({ASUM && GUAR} ALPHA implies {ASUM && GUAR} ALPHA') .

*** Definition T24 (Local Reentry Predicate)
eq reenter(ASUM, GUAR, O, O', K, K', ZZL, ALPHA, ALPHA') =
   lwf({~H~} ALPHA', O)
   and {~H~} ALPHA ^^ [K' % O . reenter] pr {~H~} ALPHA'
   and {~H~} ALPHA' ew [K' % O <- O . *]
   and pending({~H~} ALPHA', O', O, K)
   and compatibleStates(ALPHA, ALPHA')
   and readOnly(ZZL, ALPHA, ALPHA')
   and ({ASUM && GUAR} ALPHA implies {ASUM && GUAR} ALPHA') .

*** Definition T25 (Parallel Activity Interleaving Predicate)
eq interleave(ASUM, O, ALPHA, H) =
   lwf(H, O)
   and {~H~} ALPHA pr H
   and {~H~} ALPHA / (out[O] | ctl[O]) == H / (out[O] | ctl[O])
   and {ASUM} ALPHA implies {ASUM} ALPHA [~H~ |-> H] .
```

```
eq compatibleStates(emptyState, emptyState) = true .
eq compatibleStates([Z |-> V] SIGMA, SIGMA' [Z |-> V'] SIGMA'') =
   typeOf(V) == typeOf(V')
   and compatibleStates(SIGMA, SIGMA' SIGMA'') .
eq compatibleStates(SIGMA, SIGMA') = false [otherwise] .

*** Definition T20 (Local History Well-Formedness)
eq lwf(H, O) = wf(H) and H == H / O .

*** Definition T19 (History Well-Formedness)
*** slightly altered to take advantage of the naming scheme enforced by
*** the interpreter
eq wf(emptyHistory) = true .
eq wf(H ^^ [O -> O' . new C[VL]]) =
   wf(H) and parent(O') == O and seq(O') >= nextChild(H, O) .
eq wf(H ^^ [K % O -> O' . M @ C[VL]]) =
   wf(H) and K >= nextLabel(H, O) .
eq wf(H ^^ [K % O <- O' . M @ C[VL ; WL]]) =
   wf(H) and pending(H, O, O', K) .
eq wf(H ^^ [O . initialized]) =
   wf(H) and not [O . initialized] in H .
eq wf(H ^^ [O . release]) = wf(H) .
eq wf(H ^^ [K % O . reenter]) =
   wf(H) and pending(H, O, O, K) and not [K % O . reenter] in H .

*** Definition T22 (Pending Call Predicate)
eq pending(H, O, O', K) =
   [K % O -> O' . *] in H and not [K % O <- O' . *] in H .

*** Definition T23 (Read-Only Predicate)
eq readOnly(epsilon, ALPHA, ALPHA') = true .
eq readOnly((ZZ0, ZZL), ALPHA, ALPHA') =
   {ZZ0} ALPHA == {ZZ0} ALPHA' and readOnly(ZZL, ALPHA, ALPHA') .

*** Definition T21 (Processor Acquisition Predicate)
*** slightly altered to make it executable
eq mayAcquireProcessor(H, O, O', K) =
   H / (out[O] | ctl[O]) ew ([O . initialized] | [O . release])
   or (O == O' and H / (out[O] | ctl[O]) ew [K % O . reenter])
   or (H / (out[O] | ctl[O]) ew [* <- O]
       and not [trailingReplySeqNum(H) % O . reenter] in H) .

eq replyEvent(H ^^ [K % O' -> O . M @ C[VL]] ^^ H', O, O', K, WL) =
   [K % O' <- O . M @ C[VL ; WL]] .

eq replies(emptyHistory, O) = emptyMSet .
eq replies(H ^^ [K % O <- O' . M[VL ; WL]], O) =
   Reply(K, WL) ++ replies(H, O) .
eq replies(H ^^ EV, O) = replies(H, O) [otherwise] .

eq trailingReplySeqNum(emptyHistory) = -1 .
eq trailingReplySeqNum(H ^^ [K % O <- O' . M @ C[VL ; WL]]) = K .
eq trailingReplySeqNum(H ^^ EV) = trailingReplySeqNum(H) [otherwise] .

eq nextLabel(emptyHistory, O) = 0 .
```

```
eq nextLabel(H ^^ [K % O -> O' . M @ C[VL]], O) = K + 1 .
eq nextLabel(H ^^ EV, O) = nextLabel(H, O) [otherwise] .

eq nextChild(emptyHistory, O) = 0 .
eq nextChild(H ^^ [O -> O # N . new C[VL]], O) = N + 1 .
eq nextChild(H ^^ EV, O) = nextChild(H, O) [otherwise] .

*** Rewrite Rule S1'alpha (Object Bootstrapping)
crl [object-bootstrapping'] :
{
   bootstrap object O := new C[VL] with parent O'
   CONFIG
}
 =>
{
   < O : C | Pr: initialPr(C[VL], CONFIG) ;
                 ~H~ := ~H~ ^^ [O . initialized],
             LVar: emptyState,
             Att: initialAtt(C, CONFIG) [~H~ |-> H][self |-> O],
             MsgQ: emptyMSet, Asum: ASUM, Guar: GUAR,
             ROAtt: classROAtt(C, CONFIG) >
   CONFIG
}
if H := [O' -> O . new C[VL]]
   /\ parent(O) == O'
   /\ < ASUM, GUAR > := classAGSpec(C, CONFIG) .

*** Rewrite Rule S1'beta (Method Bootstrapping)
crl [method-bootstrapping'] :
{
   bootstrap method O . M @ C'[VL]
             with class C, caller O', label K, history H,
                  attributes ZL := WL
   CONFIG
}
 =>
{
   < O : C | Pr: S, LVar: BETA,
             Att: [~H~ |-> H][self |-> O][ZL |-> WL],
             MsgQ: replies(H, O), Asum: ASUM, Guar: GUAR,
             ROAtt: classROAtt(C, CONFIG) >
   CONFIG
}
if lwf(H, O)
   /\ H bw [* -> O . new C[*]]
   /\ mayAcquireProcessor(H, O, O', K)
   /\ [K % O' -> O . M @ C'[VL]] in H
   /\ pending(H, O', O, K)
   /\ < S, BETA > := boundMtd(if C' == none then C else C' fi, O',
                              K, M, VL, CONFIG)
   /\ < ASUM, GUAR > := classAGSpec(C, CONFIG) .

*** Rewrite Rule S13' (Object Creation)
*** slightly altered to make it executable
crl [object-creation'] :
```

```
< O : C | Pr: ZZ := new C'[EL] ; S, LVar: BETA, Att: ALPHA, ETC >
 =>
< O : C | Pr: ZZ := O' ; S, LVar: BETA, Att: ALPHA [~H~ |-> H], ETC >
if O' := O # nextChild({~H~} ALPHA, O)
   /\ H := {~H~} ALPHA ^^ [O -> O' . new C'[{EL} ALPHA BETA]] .

*** Rewrite Rule S14' (Process Suspension and Reactivation)
crl [process-suspension-and-reactivation'-nonexec] :
< O : C | Pr: S, LVar: BETA, Att: ALPHA, MsgQ: Q, Asum: ASUM,
          Guar: GUAR, ROAtt: ZZL, ETC >
 =>
< O : C | Pr: clearWait(S), LVar: BETA, Att: ALPHA',
          MsgQ: replies({~H~} ALPHA', O), Asum: true, Guar: GUAR,
          ROAtt: ZZL, ETC >
if not enabled(S, ALPHA BETA, Q)
   and release(ASUM, GUAR, O, {caller} BETA, {label} BETA, ZZL, ALPHA,
               ALPHA')
   and enabled(clearWait(S), ALPHA' BETA, replies({~H~} ALPHA', O))
                                                          [nonexec] .

*** Rewrite Rule S14' (Process Suspension and Reactivation)
*** alternative version that relies on user-supplied random data
crl [process-suspension-and-reactivation'] :
< O : C | Pr: S, LVar: BETA, Att: ALPHA, MsgQ: Q, Asum: ASUM,
          Guar: GUAR, ROAtt: ZZL, ETC >
random data ALPHA'
 =>
< O : C | Pr: clearWait(S), LVar: BETA, Att: ALPHA',
          MsgQ: replies({~H~} ALPHA', O), Asum: ASUM, Guar: GUAR,
          ROAtt: ZZL, ETC >
random data ALPHA'
if not enabled(S, ALPHA BETA, Q)
   and release(ASUM, GUAR, O, {caller} BETA, {label} BETA, ZZL, ALPHA,
               ALPHA')
   and enabled(clearWait(S), ALPHA' BETA, replies({~H~} ALPHA', O)) .

*** Rewrite Rule S16' (Asynchronous Invocation)
*** slightly altered to make it executable
crl [asynchronous-invocation'] :
< O : C | Pr: LL ! OEXP . M[EL] ; S, LVar: BETA, Att: ALPHA, ETC >
 =>
< O : C | Pr: S, LVar: BETA [LL |-> K], Att: ALPHA [~H~ |-> H], ETC >
if K := nextLabel({~H~} ALPHA, O)
   /\ H := {~H~} ALPHA
           ^^ [K % O -> {OEXP} ALPHA BETA . M[{EL} ALPHA BETA]] .

*** Rewrite Rule S17' (Local Asynchronous Invocation)
*** slightly altered to make it executable
crl [local-asynchronous-invocation'] :
< O : C | Pr: LL ! M @ C'[EL] ; S, LVar: BETA, Att: ALPHA, ETC >
 =>
< O : C | Pr: S, LVar: BETA [LL |-> K], Att: ALPHA [~H~ |-> H], ETC >
if K := nextLabel({~H~} ALPHA, O)
   /\ H := {~H~} ALPHA ^^ [K % O -> O . M @ C'[{EL} ALPHA BETA]] .
```

```
  *** Rewrite Rule S20' (Method Return)
  crl [method-return'] :
  < O : C | Pr: return EL ; S, LVar: BETA, Att: ALPHA, ETC >
   =>
  < O : C | Pr: S, LVar: BETA, Att: ALPHA [~H~ |-> H], ETC >
  if H := {~H~} ALPHA ^^ replyEvent({~H~} ALPHA, O, {caller} BETA,
                                    {label} BETA, {EL} ALPHA BETA) .

  *** Rewrite Rule S23' (Local Reentry and Continuation)
  crl [local-reentry-and-continuation'-nonexec] :
  < O : C | Pr: LL ?[ZZL] ; S, LVar: BETA, Att: ALPHA, Asum: ASUM,
            Guar: GUAR, ROAtt: ZZL, ETC >
   =>
  < O : C | Pr: LL ?[ZZL] ; S, LVar: BETA, Att: ALPHA', Asum: ASUM,
            Guar: GUAR, ROAtt: ZZL, ETC >
  if pending({~H~} ALPHA, O, O, {LL} BETA)
     and reenter(ASUM, GUAR, O, {caller} BETA, {label} BETA,
                 {LL} BETA, ZZL, ALPHA, ALPHA') [nonexec] .

  *** Rewrite Rule S23' (Local Reentry and Continuation)
  *** alternative version that relies on user-supplied random data
  crl [local-reentry-and-continuation'] :
  < O : C | Pr: LL ?[ZZL] ; S, LVar: BETA, Att: ALPHA, Asum: ASUM,
            Guar: GUAR, ROAtt: ZZL, ETC >
  random data ALPHA'
   =>
  < O : C | Pr: LL ?[ZZL] ; S, LVar: BETA, Att: ALPHA', Asum: ASUM,
            Guar: GUAR, ROAtt: ZZL, ETC >
  random data ALPHA'
  if pending({~H~} ALPHA, O, O, {LL} BETA)
     and reenter(ASUM, GUAR, O, {caller} BETA, {label} BETA, {LL} BETA,
                 ZZL, ALPHA, ALPHA') .

  *** Rewrite Rule S25' (Parallel Activity)
  crl [environment-activity'] :
  < O : C | Pr: S, Att: ALPHA, MsgQ: Q, Asum: ASUM, ETC >
   =>
  < O : C | Pr: S, Att: ALPHA [~H~ |-> H], MsgQ: replies(H, O), ETC >
  if interleave(ASUM, O, ALPHA, H) [nonexec] .

  *** Rewrite Rule S25' (Parallel Activity)
  *** alternative version that relies on user-supplied random data
  crl [environment-activity'] :
  < O : C | Pr: S, Att: ALPHA, MsgQ: Q, Asum: ASUM, ETC >
  random data H
   =>
  < O : C | Pr: S, Att: ALPHA [~H~ |-> H], MsgQ: replies(H, O),
            Asum: ASUM, ETC >
  random data H
  if interleave(ASUM, O, ALPHA, H)
     and H =/= {~H~} ALPHA .
endm
```

B.7 All Creol Tools

```
***(
    creol-tools.maude

    This file loads all the modules necessary to use the Creol assertion
    analyzer, the interpreter for closed systems, or the interpreter for
    open systems. See Appendix A for a user's guide.
)

load creol-program.maude .
load creol-assertion-utilities.maude .
load creol-assertion-analyzer.maude .
load creol-interpreter-core.maude .
load creol-closed-interpreter.maude .
load creol-open-interpreter.maude .

*** pas de jaloux
select CREOL-PROGRAM .
```

Appendix C

Specifications of the Case Studies

This appendix presents the complete Maude specification of the case studies that were presented in Chapter 7. The code is put in the public domain.

C.1 Bank Account

```
***(
    bank-account.maude

    This file specifies the 'NetBankAccount class presented in Section
    7.1 of Verification of Assertions in Creol Programs, together with
    the associated simplification rules.
)

load creol-tools.maude .

mod BANK-ACCOUNT-SIMPLIFICATION-RULES is
  including CREOL-SIMPLIFICATION-RULES .

  vars A A' A1 A2 A3 : AExp .
  var EEXP : EventExp .
  var HEXP : HistoryExp .
  var OEXP : OExp .
  var PHI : Assn .

  *** simplification rules derived from the definition of 'noNegatives
  rl 'noNegatives[emptyHistory] => true .
  rl 'noNegatives[HEXP ^^ [A % OEXP -> self : any . 'deposit[A']]] =>
     A' ge 0 && 'noNegatives[HEXP] .
  rl 'noNegatives[HEXP ^^ [A % OEXP -> self : any . 'payBill[A']]] =>
     A' ge 0 && 'noNegatives[HEXP] .
  crl 'noNegatives[HEXP ^^ EEXP] => 'noNegatives[HEXP]
  if EEXP cannot match invoke .

  *** simplification rules derived from the definition of 'sum
  rl 'sum[emptyHistory] => 0 .
  rl 'sum[HEXP ^^ [A % OEXP <- self : any . 'deposit[A' ; epsilon]]] =>
```

```
     'sum[HEXP] plus A' .
  rl 'sum[HEXP ^^ [A % OEXP <- self : any . 'payBill[A' ; epsilon]]] =>
     'sum[HEXP] minus A' .
  crl 'sum[HEXP ^^ EEXP] => 'sum[HEXP]
  if EEXP cannot match reply .

  *** additional simplification rule
  rl ((A1 le A2 && 'noNegatives[HEXP]
       && [A % OEXP -> self : any . 'deposit[A3]] in HEXP
       && PHI) ==> A1 le A2 plus A3) =>
     true .
endm

fmod BANK-ACCOUNT-PROGRAM is
  including CREOL-PROGRAM .

  op prog : -> Config .

  eq prog =
     interface 'BankAccount
     begin with any :
        op 'deposit[in 'amount : int]
        op 'payBill[in 'amount : int]

        asum 'noNegatives[~H~]
     end

     class 'NetBankAccount
        implements 'BankAccount
     begin
        var 'balance : int

     with any :
        op 'deposit[in 'amount : int] is
           'balance := 'balance plus 'amount

        op 'payBill[in 'amount : int] is
           await 'balance ge 'amount ;
           'balance := 'balance minus 'amount

        guar 'balance ge 0 && 'balance eq 'sum[~H~]
     end

     class 'Student ['account : 'BankAcount]
     begin
        op 'run is
           while true do
              'account . 'payBill[200 ;]
           od
     end

     class 'LoanAuthority ['account : 'BankAccount]
     begin
        op 'run is
           while true do
```

```
            'account . 'deposit[500 ;]
         od
   end

   *** test driver
   class 'Main
   begin
      op 'run is
         var 'account : 'BankAccount, 'student : any,
             'authority : any ;
         'account := new 'NetBankAccount ;
         'student := new 'Student['account] ;
         'authority := new 'LoanAuthority['account]
   end
   .
endfm

mod BANK-ACCOUNT-CLOSED-SYSTEM is
  including BANK-ACCOUNT-PROGRAM .
  including CREOL-INTERPRETER-FOR-CLOSED-SYSTEMS .

  op init : -> GlobalConfig .

  eq init =
     {
        prog
        bootstrap system 'Main
     } .
endm

mod BANK-ACCOUNT-VERIFICATION is
  including BANK-ACCOUNT-PROGRAM .
  including CREOL-ASSERTION-ANALYZER .

  op init : -> GlobalConfig .

  eq init =
     {
        prog
        verify class 'NetBankAccount
               with simplifications 'BANK-ACCOUNT-SIMPLIFICATION-RULES
     } .
endm
```

C.2 Read–Write Lock

```
***(
    read-write-lock.maude

    This file specifies the 'WriterFriendlyRWLock class presented in
    Section 7.2 of Verification of Assertions in Creol Programs, together
    with the associated simplification rules.
)
```

```
load creol-tools.maude .

mod READ-WRITE-LOCK-SIMPLIFICATION-RULES is
  including CREOL-SIMPLIFICATION-RULES .

  var A : AExp .
  var EEXP : EventExp .
  var HEXP : HistoryExp .
  var OEXP : OExp .
  var PHI : Assn .

  *** simplification rules derived from the definition of 'numReaders
  rl 'numReaders[emptyHistory] => 0 .
  rl 'numReaders[HEXP ^^ [A % OEXP <- self : any .
                          'beginRead[epsilon ; epsilon]]] =>
     'numReaders[HEXP] plus 1 .
  rl 'numReaders[HEXP ^^ [A % OEXP <- self : any .
                          'endRead[epsilon ; epsilon]]] =>
     'numReaders[HEXP] minus 1 .
  rl 'numReaders[HEXP ^^ [A % OEXP <- self : any .
                            'beginWrite[epsilon ; epsilon]]] =>
     'numReaders[HEXP] .
  rl 'numReaders[HEXP ^^ [A % OEXP <- self : any .
                          'endWrite[epsilon ; epsilon]]] =>
     'numReaders[HEXP] .
  crl 'numReaders[HEXP ^^ EEXP] => 'numReaders[HEXP]
  if EEXP cannot match reply .

  *** simplification rules derived from the definition of 'numWriters
  rl 'numWriters[emptyHistory] => 0 .
  rl 'numWriters[HEXP ^^ [A % OEXP <- self : any .
                          'beginRead[epsilon ; epsilon]]] =>
     'numWriters[HEXP] .
  rl 'numWriters[HEXP ^^ [A % OEXP <- self : any .
                          'endRead[epsilon ; epsilon]]] =>
     'numWriters[HEXP] .
  rl 'numWriters[HEXP ^^ [A % OEXP <- self : any .
                          'beginWrite[epsilon ; epsilon]]] =>
     'numWriters[HEXP] plus 1 .
  rl 'numWriters[HEXP ^^ [A % OEXP <- self : any .
                          'endWrite[epsilon ; epsilon]]] =>
     'numWriters[HEXP] minus 1 .
  crl 'numWriters[HEXP ^^ EEXP] => 'numWriters[HEXP]
  if EEXP cannot match reply .

  *** additional simplification rules
  rl [A % OEXP -> self : any . 'endRead[epsilon]] in HEXP
     && ! ([A % OEXP <- self : any . *] in HEXP)
     && 'lwf[HEXP, self : any]
     && #[HEXP / [* -> self : any . 'endRead[*]]] le
        #[HEXP / [* -> self : any . 'beginRead[*]]]
     && 0 eq 'numReaders[HEXP] =>
     false .
  rl [A % OEXP -> self : any . 'endWrite[epsilon]] in HEXP
```

```
     && ! ([A % OEXP <- self : any . *] in HEXP)
     && 'lwf[HEXP, self : any]
     && #[HEXP / [* -> self : any . 'endWrite[*]]] le
        #[HEXP / [* -> self : any . 'beginWrite[*]]]
     && 0 eq 'numWriters[HEXP] =>
     false .
  rl (([A % OEXP -> self : any . 'endRead[epsilon]] in HEXP
       && ! ([A % OEXP <- self : any . *] in HEXP)
       && 'lwf[HEXP, self : any]
       && #[HEXP / [* -> self : any . 'endRead[*]]] le
          #[HEXP / [* -> self : any . 'beginRead[*]]]
       && PHI) ==>
      0 lt 'numReaders[HEXP]) =>
     true .
endm

fmod READ-WRITE-LOCK-PROGRAM is
  including CREOL-PROGRAM .

  op prog : -> Config .

  eq prog =
     interface 'RWLock
     begin
     with any :
        op 'beginRead
        op 'endRead
        op 'beginWrite
        op 'endWrite

        asum #[~H~ / [* -> self . 'beginRead[*]]] ge
             #[~H~ / [* -> self . 'endRead[*]]]
             && #[~H~ / [* -> self . 'beginWrite[*]]] ge
                #[~H~ / [* -> self . 'endWrite[*]]]
        guar 'numWriters[~H~] eq 0
             || ['numWriters[~H~] eq 1 && 'numReaders[~H~] eq 0]
     end

     class 'WriterFriendlyRWLock
        implements 'RWLock
     begin
        var 'nr : int, 'nw : int, 'dw : int

     with any :
        op 'beginRead is
           await 'nw eq 0 && 'dw eq 0 ;
           'nr := 'nr plus 1

        op 'endRead is
           prove 'nr gt 0 ;
           'nr := 'nr minus 1

        op 'beginWrite is
           'dw := 'dw plus 1 ;
           await 'nr eq 0 && 'nw eq 0 ;
```

```
            'dw := 'dw minus 1 ;
            'nw := 'nw plus 1

         op 'endWrite is
            prove 'nw gt 0 ;
            'nw := 'nw minus 1

         guar 'nr eq 'numReaders[~H~] && 'nw eq 'numWriters[~H~]
      end

      class 'Reader ['lock : 'RWLock]
      begin
         var 'n : int

         op 'run is
            while true do
               await 'lock . 'beginRead[] ;
               'n := 'n plus 1 ;
               ! 'lock . 'endRead[]
            od
      end

      class 'Writer ['lock : 'RWLock]
      begin
         var 'n : int

         op 'run is
            while true do
               await 'lock . 'beginWrite[] ;
               'n := 'n plus 1 ;
               ! 'lock . 'endWrite[]
            od
      end

      *** test driver
      class 'Main
      begin
         op 'run is
            var 'lock : 'RWLock, 'o : any ;
            'lock := new 'WriterFriendlyRWLock ;
            'o := new 'Reader['lock] ;
            'o := new 'Reader['lock] ;
            'o := new 'Reader['lock] ;
            'o := new 'Writer['lock] ;
            'o := new 'Writer['lock] ;
            'o := new 'Writer['lock]
      end
      .
endfm

mod READ-WRITE-LOCK-CLOSED-SYSTEM is
  including CREOL-INTERPRETER-FOR-CLOSED-SYSTEMS .
  including READ-WRITE-LOCK-PROGRAM .

  op init : -> GlobalConfig .
```

```
  eq init =
     {
        prog
        bootstrap system 'Main
     } .
endm

mod READ-WRITE-LOCK-VERIFICATION is
  including CREOL-ASSERTION-ANALYZER .
  including READ-WRITE-LOCK-PROGRAM .

  op init : -> GlobalConfig .

  eq init =
     {
        prog
        verify class 'WriterFriendlyRWLock
               with simplifications 'READ-WRITE-LOCK-SIMPLIFICATION-RULES
     } .
endm
```

C.3 Factorial

```
***(
    factorial.maude

    This file specifies the 'IterativeFactorial, 'RecursiveFactorial, and
    'NonterminatingFactorial classes presented in Sections 7.3 and 7.4 of
    Verification of Assertions in Creol Programs, together with the
    associated simplification rules.
)

load creol-tools.maude .

mod FACTORIAL-SIMPLIFICATION-RULES is
  including CREOL-SIMPLIFICATION-RULES .

  vars A A' A1 A2 : AExp .
  var EEXP : EventExp .
  vars HEXP HEXP' HEXP'' : HistoryExp .
  var N : Int .
  var OEXP : OExp .
  var PHI : Assn .

  *** simplification rules derived from the definition of 'G
  rl 'G[emptyHistory] => true .
  rl 'G[HEXP ^^ [A % OEXP <- self : any . 'compute[A1 ; A2]]] =>
     'G[HEXP] && A2 eq 'fact[A1] .
  crl 'G[HEXP ^^ EEXP] => 'G[HEXP] if EEXP cannot match reply .

  *** simplification rules derived from the definition of 'fact
  rl 'fact[N] => if N > 1 then N * 'fact[N - 1] else 1 fi .
```

```
  *** additional simplification rules
  rl ((1 le A && PHI) ==>
      'fact[A plus 1] eq 'fact[A] times (A plus 1)) =>
     true .
  rl 'fact[A] eq 1 => A le 1 .
  rl (('G[HEXP']
       && 'lwf[HEXP', self : any]
       && HEXP ^^ [A % self : any -> self : any . 'compute[A' minus 1]]
          ^^ [A % self : any . reenter] pr HEXP'
       && PHI) ==>
       'fact[A'] eq A' times 'returnVal $ 1[HEXP', self : any, A]) =>
     true .
  rl HEXP ^^ [A % self : any -> self : any . 'compute[A']]
          ^^ HEXP'' pr HEXP'
     && HEXP' ew [A % self : any <- self : any . *]
     && ! ([* <- self : any . 'compute[*]] in HEXP') =>
     false .
endm

fmod FACTORIAL-PROGRAM is
  including CREOL-PROGRAM .

  op prog : -> Config .

  eq prog =
     interface 'Factorial
     begin
     with any :
        op 'compute[in 'x : int out 'y : int]

        guar 'G[~H~]
     end

     class 'IterativeFactorial
        implements 'Factorial
     begin
     with any :
        op 'compute[in 'x : int out 'y : int] is
           var 'i : int ;
           'i := 1 ;
           'y := 1 ;
           inv 'G[~H~] && 'i ge 1 && ['i le 'x || ['i eq 1 && 'x lt 1]]
               && 'y eq 'fact['i]
           while 'i lt 'x do
              'i := 'i plus 1 ;
              'y := 'y times 'i
           od
     end

     class 'RecursiveFactorial
        implements 'Factorial
     begin
     with any :
        op 'compute[in 'x : int out 'y : int] is
```

```
            if 'x le 1 th
               'y := 1
            el
               'compute['x minus 1 ; 'y] ;
               'y := 'y times 'x
            fi
      end

      class 'NonterminatingFactorial
         implements 'Factorial
      begin
      with any :
         op 'compute[in 'x : int out 'y : int] is
            'compute['x ; 'y]

         guar ! [[* <- self . 'compute[*]] in ~H~]
      end

      *** test driver
      class 'Main
      begin
         op 'run is
            var 'fact1 : 'Factorial, 'fact2 : 'Factorial, 'y1 : int,
                'y2 : int ;
            'fact1 := new 'IterativeFactorial ;
            'fact1 . 'compute[7 ; 'y1] ;
            'fact2 := new 'RecursiveFactorial ;
            'fact2 . 'compute[7 ; 'y2]
      end
      .
endfm

mod FACTORIAL-CLOSED-SYSTEM is
  including CREOL-INTERPRETER-FOR-CLOSED-SYSTEMS .
  including FACTORIAL-PROGRAM .

  op init : -> GlobalConfig .

  eq init =
     {
        prog
        bootstrap system 'Main
     } .
endm

mod FACTORIAL-VERIFICATION is
  including CREOL-ASSERTION-ANALYZER .
  including FACTORIAL-PROGRAM .

  op init1 : -> GlobalConfig .
  op init2 : -> GlobalConfig .
  op init3 : -> GlobalConfig .

  eq init1 =
     {
```

```
        prog
        verify class 'IterativeFactorial
               with simplifications 'FACTORIAL-SIMPLIFICATION-RULES
     } .

  eq init2 =
     {
        prog
        verify class 'RecursiveFactorial
               with simplifications 'FACTORIAL-SIMPLIFICATION-RULES
     } .

  eq init3 =
     {
        prog
        verify class 'NonterminatingFactorial
               with simplifications 'FACTORIAL-SIMPLIFICATION-RULES
     } .
endm
```

Bibliography

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[AJO04] Eyvind W. Axelsen, Einar Broch Johnsen, and Olaf Owe. Toward reflective application testing in open environments. *Norsk informatikkkonferanse NIK 2004*, 192–203, Tapir Forlag, 2004.

[And00] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[AO97] Krzysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs, Second Edition*. Springer-Verlag, 1997.

[Apt81] Krzysztof R. Apt. Ten years of Hoare's logic: A survey—Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, 1981.

[Apt84] Krzysztof R. Apt. Ten years of Hoare's logic: A survey—Part II: Nondeterminism. *Theoretical Computer Science*, 28(1–2):83–109, 1984.

[Arn03] Marte Arnestad. *En abstrakt maskin for Creol i Maude*. Cand. scient. thesis, University of Oslo, 2003.

[AS96] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1996.

[Axe04] Eyvind Wærsted Axelsen. *A Meta-Level Framework for Recording and Utilizing Communication Histories in Maude*. Cand. scient. thesis, University of Oslo, 2004.

[BH70] P. Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–242, 1970.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software—The KeY Approach*. LNCS 4334, Springer-Verlag, 2007.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[BO08] Jasmin Christian Blanchette and Olaf Owe. An open system operational semantics for an object-oriented and component-based language. To appear in *Electronic Notes in Theoretical Computer Science*, 2008.

[BP06] Lilian Burdy and Mariela Pavlova. Java bytecode specification and verification. *Proceedings of the 2006 ACM Symposium on Applied Computing*, 1835–1839, ACM Press, 2006.

[CDEL+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *Maude Manual (Version 2.3)*. Available at `http://maude.cs.uiuc.edu/maude2-manual/`, 2007.

[Cho05] Chyun Yung Chou. *En timeout-mekanisme for Creol*. Master's thesis, University of Oslo, 2005.

[CORSS95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. *A Tutorial Introduction to PVS.* Presented at the 1st Workshop on Industrial-Strength Formal Specification Techniques, 1995.

[Dah77] Ole-Johan Dahl. Can program proving be made practical? *Les fondements de la programmation*, 57–114, Institut de Recherche d'Informatique et d'Automatique, Toulouse, 1977.

[Dah92] Ole-Johan Dahl. *Verifiable Programming*. Prentice Hall, 1992.

[dBP04] Frank S. de Boer and Cees Pierik. How to cook a complete Hoare logic for your pet OO language. *Formal Methods for Components and Objects: Second International Symposium*, LNCS 3188, 111–133, Springer-Verlag, 2004.

[dH03] Robbert de Haan. *Using ASF+SDF for the Verification of Annotated Java Programs*. Master's thesis, University of Amsterdam, 2003.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[DJO05] Johan Dovland, Einar Broch Johnsen, and Olaf Owe. Verification of concurrent objects with asynchronous method calls. *Proceedings of the 2005 IEEE International Conference on Software—Science, Technology & Engineering*, 141–150, IEEE Press, 2005.

[DJO08] Johan Dovland, Einar Broch Johnsen, and Olaf Owe. *A Compositional Proof System for Dynamic Object Systems.* Research report, University of Oslo, Department of Informatics, 2008.

[DJOS08] Johan Dovland, Einar Broch Johnsen, Olaf Owe, and Martin Steffen. *Lazy Behavioral Subtyping*. Research report, University of Oslo, Department of Informatics, 2008.

[DLMSS78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, 1978.

[DO91] Ole-Johan Dahl and Olaf Owe. Formal Development with ABEL. *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development—Volume 2: Tutorials*, LNCS 552, 320–362, Springer-Verlag, 1991.

[dRdB+01] Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.

[Fje05] Jørgen Hermanrud Fjeld. *Compiling Creol Safely*. Master's thesis, University of Oslo, 2005.

[Flo67] R. W. Floyd. Assigning meanings to programs. *Proceedings of the Symposium in Applied Mathematics*, 19:19–32, AMS, 1967.

[Gal03] Jean Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Available at `http://www.cis.upenn.edu/~jean/gbooks/logic.html`, 2003.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210,405–431, 1935.

[GJSB05] James Gosling, Bill Joy, Guy Steele, Gilad Bracha. *The Java™ Language Specification, Third Edition*. Prentice Hall, 2005.

[Gor89] Michael J. C. Gordon. Mechanizing programming logics in higher order logic. *Current Trends in Hardware Verification and Automated Theorem Proving*, 387–439, Springer-Verlag, 1989.

[GvN46] Goldstine and von Neumann. *Planning and Coding of Problems for an Electronic Computing Instrument*. U.S. Army and Institute for Advanced Study report, 1946.

[Han07] Christian Mahesh Hansen. *INF3170/4170—Logikk*. Lecture notes, University of Oslo, Department of Informatics, 2007.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation, Third Edition*. Addison-Wesley, 2006.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa74] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hoa03] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, 50(1):63–69, 2003.

[Hol05] Bjarne Holen. *A Reflective Theorem Prover for the Connection Calculus*. Master's thesis, University of Oslo, 2005.

[Hus05] Are Husby. *Utvidelse av Creol-språket med synkronisert fletting*. Master's thesis, University of Oslo, 2005.

[IS97] Andrew Ireland and Jamie Stark. On the automatic discovery of loop invariants. *Proceedings of the 4th NASA Langley Formal Methods Workshop*, NASA Conference Publication 3356, 1997.

[JO02] Einar Broch Johnsen and Olaf Owe. A compositional formalism for object viewpoints. *Formal Methods for Open Object-Based Distributed Systems V*, Kluwer, 2002.

[JO04a] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Proceeding of the 2nd International Conference on Software Engineering and Formal Methods*, 188–197, IEEE Press, 2004.

[JO04b] Einar Broch Johnsen and Olaf Owe. Object-oriented specification and open distributed systems. *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, LNCS 2635, 137–164, Springer-Verlag, 2004.

[JO05] Einar Broch Johnsen and Olaf Owe. A dynamic binding strategy for multiple inheritance and asynchronously communicating objects. *Proceedings of the 3rd International Symposium on Formal Methods for Components and Objects*, LNCS 3657, 274–295, Springer-Verlag, 2005.

[JO07] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):39–58, Springer-Verlag, 2007.

[JOA03] Einar Broch Johnsen, Olaf Owe, and Marte Arnestad. Combining active and reactive behavior in concurrent objects. *Norsk informatikkkonferanse NIK 2003*, 193–204, Tapir Forlag, 2003.

[Joh02] Einar Broch Johnsen. *An Interleaving Evaluation Semantics for Object Oriented Method Calls.* Internal report, University of Oslo, Department of Informatics, 2002.

[Jon81] C. B. Jones. *Development Methods for Computer Programs Including a Notion of Interference.* Ph.D. thesis, Oxford University, 1981.

[JOT06] Einar Broch Johnsen, Olaf Owe, and Arild B. Torjusen. Validating behavioral component interfaces in rewriting logic. *Electronic Notes in Theoretical Computer Science*, 159:187–204, 2006.

[JOY06] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, 2006.

[Kal90] Anne Kaldewaij. *Programming: The Derivation of Algorithms*. Prentice Hall, 1990.

[Kin69] James Cornelius King. *A Program Verifier*. Ph.D. thesis, Carnegie-Mellon University, 1969.

[Knu02] Donald Knuth. All questions answered—University of Oslo, 30 August 2002. *TUGboat*, 23(3–4):249–261, 2002.

[Kya06] Marcel Kyas. *Distributing Creol*. Presented at the Nordic Workshop on Programming Theory, 2006.

[Lam08] Leslie Lamport. *The Writings of Leslie Lamport*. Available at `http://www.research.microsoft.com/users/lamport/pubs/pubs.html`, 2008.

[LP99] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, 21(3):502–526, 1999.

[MC81] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, 1999.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[NPN99] Tobias Nipkow and Leonor Prensa Nieto. Owicki/Gries in Isabelle/HOL. *Fundamental Approaches to Software Engineering, Second International Conference*, LNCS 1577, 188–203, 1999.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS 2283, Springer-Verlag, 2002.

[Ofs05] Morten Ofstad. *Dynamic Updates in the Creol Framework*. Master's thesis, University of Oslo, 2005.

[Ölv07] Peter Csaba Ölveczky. *Formal Modeling and Analysis of Distributed Systems in Maude*. Lecture notes, University of Oslo, Department of Informatics, 2007.

[ÖM04] Peter Csaba Ölveczky and José Meseguer. Specification and analysis of real-time systems using Real-Time Maude. *Fundamental Approaches to Software Engineering—7th International Conference, FASE 2004*, LNCS 2984, 354–358, Springer-Verlag, 2004.

[Owe93] Olaf Owe. Partial logics reconsidered: A conservative approach. *Formal Aspects of Computing*, 5(3):208–223, 1993.

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004.

[PMA08] Precise Modeling and Analysis, Department of Informatics, University of Oslo. *CREOL Home Page*. `http://www.ifi.uio.no/~creol/`, 2008.

[Shu01] Johann M. Schumann. *Automated Theorem Proving in Software Engineering*. Springer-Verlag, 2001.

[SM07] Ralf Sasse and José Meseguer. Java+ITP: A verification tool based on Hoare logic and algebraic semantics. *Electronic Notes in Theoretical Computer Science*, 176(4):29–46, 2007.

[Tur49] Alan M. Turing. *Checking a Large Routine*. Paper for the EDSAC Inaugural Conference, 1949.

[YJO06] Ingrid Chieh Yu, Einar Broch Johnsen, and Olaf Owe. Type-safe runtime class upgrades in Creol. *Proceedings of the 8th International Conference on Formal Methods for Open Object-Based Distributed Systems*, 202–217, LNCS 4037, Springer-Verlag, 2006.