Formal Semantics and Proof System for SCOOP *

Arnaud Bailly

ETH Zürich

Abstract

We present a formal semantics for the Simple Concurrent Object-Oriented Programming (SCOOP) extension to Eiffel. Our objective is to clarify the foundations of this extension, which has undergone only informal descriptions until now. The semantics is operational; it uses an imperative calculus of objects to represent configurations of SCOOP objects. The operational view provides a very intuitive way to understand how such configurations can evolve in time. We then present a proof system, that one can also view as an axiomatic semantics, proposing a set of proof rules for the inference of safety properties. We prove the soundness and completeness of the proof system regarding the operational semantics. Both semantics represent, in different stylistic ways, the meaning of SCOOP programs.

Keywords: Eiffel, Concurrency, SCOOP, Operational Semantics, Hoare logic.

*Research partially supported by the Hasler Foundation

1 Introduction

The advent of object-oriented programming in the eighties showed that concepts such as *interface*, *class*, and *inheritance* lead to pieces of software that are easier to write, maintain, and compose. This success in writing sequential applications has lead researchers to try to apply the same concepts to concurrent and distributed programming with the goal of benefiting from the same properties. With no obvious solution emerging and in spite of the fact that all researchers targeted this goal, many primitives have been pushed forward along the years, going sometimes in opposite directions. This has resulted in a continuous stream of publications, starting as early as the mid-eighties. Those proposals are usually unrelated or poorly related with to one another. In our view, a clear comparison of such models and primitives is essential, and it can be achieved only by using formal semantics and proof methods.

In this paper we provide such formal elements for one of those proposals: Bertrand Meyer's Eiffel-based SCOOP (Simple Concurrent Object-Oriented Programming). It was introduced as a generalization of the design-by-contract methodology to concurrent object-oriented programs. Since its first publication in 1993 SCOOP has, to our knowledge, undergone no formal study; however, it has been (and still is) used in several real-life developments, including one at the ETH Zürich. The formal study presented here allows us to try a comparison with other languages, notably Java and Polyphonic C#. We provide *two* semantics for SCOOP, and prove that they are strongly related by soundness and completeness theorems. We first provide a small-step operational semantics that describes how a configuration of concurrent objects written in the language can evolve. This semantics is close to intuition, and can serve as a reference for future implementations of compilers for the language. We then provide an axiomatic semantics, or *proof system*, that shows how to prove *safety properties* of programs; programmers naturally perform similar reasoning when writing a program. We advocate that it allows better comparison with other languages regarding the ease of use: the easier the proofs are, the easier it is to write correct programs. Whether pragmatic issues such as maintenance and reusability are indeed better addressed by those languages can then be better studied starting from the comparison of their respective formal semantics.

The contributions of this article are twofold. First, it provides foundations for the semantics of the SCOOP concurrent extension to Eiffel, and allows thorough comparison to other proposals such as Java with multithreading. Second, it provides the first sound and complete proof system for asynchronously communicating concurrent objects. We aim at a thorough comparison of concurrent object-oriented languages with SCOOP, based on pragmatic arguments but supported by a study of their formal (axiomatic) semantics. Previous such comparisons did not feature formal support, and were either lead by methodological purposes or by a looser comparison of primitives. McHale [McH94] studied the relationship existing among different synchronization primitives for objects, using some (limited) formal support. Unfortunately, other studies based on process algebras do not bring much insights about such pragmatic issues like *how programs should be conceived*. This idea of using proof rules have already been used for classifying monitors [BFC95]. Passing through, we give rules for other object-oriented languages, and have a glance

at issues not formally treated here, such as inheritance and refinement. The given comparison is partial, being based on the verification of *safety* properties only.

The organization of the paper is the following. First, we provide a brief overview of proposals for concurrent object-oriented programming, and introduce SCOOP. Then we provide a small-step operational semantics for SCOOP in Section 3. In Section 4 we show how to generalize Floyd's method for proofs of invariants to SCOOP. In Section 5 we try to compare the proposed proof system and the SCOOP programming primitive with other approaches, and briefly discuss some other pragmatic issues.

2 SCOOP

2.1 Object-Orientation and Concurrency

We first give an overview of the state-of-the-art in object-oriented concurrent programming. To go more in depth than we will here, good starting points are [Phi00] and [BGL98]. To combine object-orientation and concurrency, two decisions have to be made:

- how to let programs introduce concurrency in the computation (that is create new tasks), and
- how to let existing tasks interact with one another.

To the first point above three solutions have been proposed. The first one consists in providing no link whatsoever between object and tasks. Tasks are therefore created using `fork`-like primitives. The second possibility is to link task creation with object creation. In this case a task is created upon the creation of an object; the fact that a task should be created or not usually depends on the class of the created object. Programming in C++ under Unix would follow the first appproach, while programming in Java with threads would follow the second. For both cases task creation results in the state of the existing objects being shared by the tasks in case they are threads, or the state of existing objects being copied in the case where tasks are full-fledged processes. The third possibility is not to allow any task creation in the program, and to have it specified separately. The number of tasks is then is then constant along the computation as in [Sek02].

Two ways are also possible to let tasks interact. In shared-variable concurrency, one object can be shared by many threads, that may each execute one of the object's methods (resulting in *intra-object concurrency*). Method call between objects takes place within the thread of the caller object and is thus identical to "traditional" functional calls as present in sequential programming. Many solutions are then employed to limit concurrent execution within one object: execution can be made sequential using monitors as in Ada (protected objects) or by using looser annotations as in Java with synchronized methods.

In message-passing concurrency (found in distributed environments), the state of each object can be modified by only one task during the whole execution of the program. Sending and receiving messages can be done using special instructions

placed amidst the functional code of the objects, or can be tied with object method calls. Indeed, in the latter case when two objects handled by different tasks call each other's methods, remote communication arises naturally, that should be implemented using message-passing. The latter solution has therefore been largely adopted, for example in Java/RMI, Corba or COM, avoiding the redundancy of having two different primitives for remote communication.

In message-passing concurrency based on invocation of objects, further distinctions need to be made. Indeed, an object can be seen as representing a service that may or may not be continuously available, that is may or may not exhibit a *state*. For example, a typical *stateless* protocol would be http, while a typical *stateful* protocol would be ftp. Statelessness implies *input-enabledness*: messages addressed to a given object can be treated by this object in any order, and usually a FCFS policy will be followed. For objects representing stateless protocols, a mechanism for task creation is often coupled with method invocation in order to increase server availability, through intra-object concurrency. This is the case in Java/RMI where a new thread is created upon every method invocation. On the other hand stateful protocols usually prohibit intra-object concurrency and requests are handled in a sequence by the task in which the supplier object is executing. Since the supplier object requires that incoming requests should be treated in a specific order, the FCFS policy is abandoned, and the handling of incoming requests is delayed until the supplier object is in an appropriate state. What exactly is the required order can be found in the code of the class of the supplier object, under the form of an automaton. One can use:

- method guards (a request can be handled only when its guard is true),
- delay queues (a queue can be opened or closed using programming primitives, and only requests found in open queues can be treated),
- live routines (the routine contains `accept` statements which describe at a given time what requests can be treated),
- path expressions (a regular expression describes the states and transitions), etc.

When several requests that can be handled are present at the same time, either non-deterministic choice is made or a FCFS policy is used. An informal but detailed comparison of different synchronization primitives can be found in [McH94].

2.2 The SCOOP Model and Syntax

At the time of its design, the essential goals for SCOOP were to be "compatible" with design-by-contract as syntactically encountered in Eiffel on the one hand, and to try to propose a simple way of doing concurrent programming on the other hand. To reach compatibility and simplicity only one keyword, **separate**, has been introduced in the language. This keyword may qualify entities and classes, and its meaning is that a new task (called *processor* in Bertrand Meyer's terminology) should be created along with each instance of a separate class; the created task is dedicated to handling

requests directed to this object or to any instance of a non-separate class that would be created by it in the future. Hence, the use of the keyword **separate** allows to introduce concurrency in the computation. Intra-object concurrency is prohibited since each object is assigned to exactly one processor and processors are sequential. Synchronization among processors is handled through object routine call (routine is the Eiffel term for method), without any additional primitive.

Routines of SCOOP objects come equipped with Eiffel-style contracts, written as hoare-triple-like pre-conditions and post-conditions. Whenever a routine of an object is invoked while its pre-condition is false, it means a bug has been found in the program, and an exception should be raised (that will be so if the appropriate option of the compiler is turned on). In other words, supplier objects exhibit stateful protocols. To retain the contractual nature of assertions, the treatment of incoming separate calls is not delayed in SCOOP (like with method guards) but, like calls issued by non-separate objects, they are treated in any order and may therefore yield an exception.

However, contrary to the sequential case, the blame can not be put as easily on the client. Indeed, due to non-determinism in the relative speed of execution of the different clients of a given supplier, one client object can not predict the state in which its call will find the supplier object when applied. Before performing such calls, a client object therefore needs a way to synchronize with the suppliers it wants to access. This can be done in SCOOP using Conditional Critical Regions (CCR).

A CCR gathers a set of objects $a_1, \cdots a_k$ (named through instance variables of the client object), a condition, and a body under the syntax:

$$\texttt{with } a_1, \cdots a_k \texttt{ when } Cond \texttt{ do } Body \texttt{ end}$$

that states that the given objects should be locked for exclusive access by the current client when the condition is true, which will trigger the execution of the body. When the body terminates, the objects are released. An object has to lock a (set of) supplier(s) before accessing it (them). This implies that a client provoking an exception can be deemed liable for it.

In SCOOP, CCR syntax disappears, replaced by the syntax used for routine definition[1]:

$$m(a_1 : A_1, \cdots, a_k : A_k) \texttt{ is require } Cond \texttt{ do } Body \texttt{ ensure } Cond \texttt{ end}$$

This syntax strikingly resembles the one seen above for CCR. Obviously, the sets of objects and the body of the CCR are both present in the routine definition's syntax. In Eiffel, pre-conditions and post-conditions follow the `require` and `ensure` keywords. Adding further syntax can be avoided by distinguishing the parts of the `require` condition that bear on arguments which are tagged as `separate`. Those will be called *separate pre-conditions*, and they will become *wait conditions*, taking the place of the `when` $Cond$ in the CCR.

Within the body of a CCR (*i.e.* the body of a routine with one or more separate arguments), a semantic distinction between *queries* and *commands* is made. A query

[1] in the article we do not stick to SCOOP's precise syntax and formats, replacing it by a convenient, free-styled, Algol-like syntax.

is a routine that yields a result, whereas a command does not. Hence, the SCOOP semantics enforces that commands are evaluated *synchronously* (the client suspends its execution waiting for the result), whereas queries are evaluated *asynchronously* (the client never blocks). This should increase the degree of parallelism of the application.

There are further rules on the transmission of separate objects that have to be enforced in order to retain compatibility with Eiffel's type system. We will not detail them as we will use a slightly different program presentation for which we do not have to apply such rules, but further details can be found in .

3 An Operational Semantics For SCOOP

We first give a syntactic representation for classes, from which we will extract the syntax for routine definitions that will be part of object representation used to provide an operational semantics to the language.

3.1 Abstract Syntax

The abstract syntax that we use has all essential features of SCOOP programs. By *essential* features we mean conditional critical regions, the associated locking mechanism, the synchronous-asynchronous call distinction for queries-commands, and the object-oriented features that include class-based structuring, class instantiation primitive, and reference passing. We also introduce conditional and repetition structures of control flow, although in a syntax different from Eiffel. We restrict inheritance, which is not featured in our study, at least for the formal part. We make every class inherit from a class ANY which has no attributes and no routines. No other inheritance relationship among classes is allowed.

$$
\begin{array}{rcl}
exp & ::= & a \mid \texttt{Current} \mid \texttt{Void} \qquad exp_{ref} ::= exp \mid a.b \\
stm & ::= & a := exp \mid a := \texttt{create}\ A \mid a := b.m(\tilde{c}) \mid b.m(\tilde{c}) \mid \\
 & & \texttt{with}\ a_1, \cdots a_k\ \texttt{when}\ exp_{ref}\ \texttt{do}\ stm\ \texttt{end} \mid \\
 & & \texttt{if}\ exp\ \texttt{then}\ stm\ \texttt{else}\ stm\ \texttt{end} \mid \\
 & & \texttt{while}\ exp\ \texttt{do}\ stm\ \texttt{end} \\
 & & stm; stm \\
rout & ::= & m(a_1 : A_1, \cdots, a_k : A_k)\ \texttt{is do}\ stm; \texttt{return}\ exp\ \texttt{end} \mid \\
 & & m(a_1 : A_1, \cdots, a_k : A_k)\ \texttt{is do}\ stm; \texttt{return end} \\
cls & ::= & \texttt{class}\ A\ \texttt{is}\ rout_1, \cdots, rout_k\ \texttt{end}
\end{array}
$$

We consider the above presentation to be self-explanatory. As mentioned earlier, we replace the change in semantics for routine definition originally featured in SCOOP by the explicit use of conditional critical regions. This has no impact on the relevance of our study since we largely disregard inheritance, and any plain SCOOP program that does not make use of inheritance can be translated in an equivalent program using the present syntax. This shall be done by merely in-lining the body of routine definitions inside conditional critical regions.

To ease reasonings and definitions, we do not allow local (routine-bound) variables other than formal parameters. We also disallow assignments to such variables.

Furthermore, qualified references (of the form $x.a$) are only allowed to appear in the wait condition of a CCR: the power of observation that can be exercised through such a construction is equivalent to perform a routine invocation, and should be only allowed on locked objects. In that case, we require that all the qualified references appearing in the wait condition only access objects that are locked in the same CCR; this can be statically checked in a simple way. In expressions different from wait conditions, access to the attributes of an object can be performed if access routines are provided by the class of this object. Those restrictions do not limit the expressiveness of the model because any program not respecting them can be translated into an equivalent program that does. We finally shall make the following assumption: within a conditional critical region, no synchronous call may appear after the first asynchronous call. This implies that all asynchronous calls are accomplished in one motion, at the end of the execution of the CCR. This does limit the power of expression of the model because certain programs can not be expressed in this form without modifying their semantics. Applying this restriction allows to make a clear distinction between synchronous and asynchronous calls, both in the operational semantics and in the axiomatic semantics. It shows how both kinds of communication can be dealt with and enlightens their differences. To deal with the full language, synchronous calls can be translated in a particular form of asynchronous calls that return a value using a 2-way handshake protocol.

As we did in the syntax above, we shall use the following names and their decorated variants in the remaining of the paper:

- A, B, C are class names,
- a, b, c are attributes of an object or formal parameters of a routine,
- m, n, o are routine names and
- i, j, k are integer indices.

3.2 Dynamic Semantics

We will use several sets in order to give a semantics to the terms of our language. First we assume denumerable sets of names for each sort of name mentioned above, that is $Class$ for classes where $ANY \notin Class$, $Attr$ for attributes, $Params$ for parameters, and $Rout$ for routine names. We will note $Attr^A$ the set of attributes of a class A, and in the same way $Rout^A$, and $Params^{m,A}$ for a routine m of a class A. We will assume these domains to be pairwise disjoint. We will note O^A the denumerable set of object identities for a given class A, and O^A_{Void} for $O^A \cup Void$. We define $O^{ANY} \triangleq \bigcup_{A \in Class} O^A$ and $O^{ANY}_{Void} \triangleq O^{ANY} \cup \{Void\}$. A type is either a class name, an integer, a boolean, or any combination of such that uses only list and Cartesian product operators. It will be defined syntactically by

$$Type ::= Class \mid Int \mid Bool \mid \texttt{list}\ Type \mid Type \times Type\ .$$

The values Val^{Type} is the union of values possible for the given type, being O^A for a class, $\{tt, ff\}$ for a boolean, the integers for Int, a finite list of values of

$Type$ for `list` $Type$, and the set of couples of elements of type $Type_1$ and $Type_2$ for $Type_1 \times Type_2$. We shall use $=$ for comparison of references, booleans, and integers. We shall note tuples of arity k like $\langle a_1, \cdots, a_k \rangle$. Typical lists we be named l, potentially with decorations. The empty list will be noted $\star$, list construction $e \oplus l$ will place e at the head of the list (resp. $l \oplus e$ will place e at the end), while concatenation of lists l_1 and l_2 will be noted $l_1 \cdot l_2$. List equality will be noted $l_1 \doteq l_2$, meaning that l_1 and l_2 have the same elements in the same order. Taking the i^{th} element of a list l will be noted $l@i$. We further assume that each datatype has a set of well-defined operations which typical member will be noted `f` and its interpretation f.

We give a rewriting semantics based on global configurations of objects. This semantics is non-compositional, meaning that it does not rely on structural induction over terms for its definition. This facilitates the establishment of soundness and completeness for the proof system presented in Section 4.

We take an object $\langle \alpha, stm, \sigma, \tau, \lambda, \iota \rangle$ to be defined by the following characteristics:

- an identity or object reference, ranging over the elements of the set O with typical elements α, β, γ,
- the next statement to execute stm with a special statement I to denote *idle* objects,
- a partial function σ that gives the value of attributes of the object,
- a partial function τ that gives the value of the parameters of the current routine executed by the object,
- the *locking state* λ of the object that can be $\perp$ when the object is not locked, or $\beta^{\tilde{\alpha}}$ when the current object is locked by object β and $\tilde{\alpha} \triangleq \alpha_1, \cdots, \alpha_k$ are locked by the same CCR of β and $\alpha \in \tilde{\alpha}$,
- a queue of incoming asynchronous routine calls ι that can be $\star$ (for the empty queue) and contain special value $\diamond$, besides elements of the form $m(\tilde{\alpha})$ that denote the call to routine m with arguments $\tilde{\alpha}$.

We will typically note M, N configurations of such objects put in parallel (which will be written using the traditional "pipe" operator "|"). We will note $M \equiv N$ for the syntactic equality of M and N.

From the text of the program we assume the definition of a function $class$ that yields, for any existing object, the name of its class. For any class A, the function $rout(A)$ will, similarly, yield the definitions of the routines of A. We will not define these functions formally, since their precise definition is tedious but comprises no intrinsic difficulty.

We now define the evaluation function for expressions with references. Such evaluation is done in the context of an object α, implying the existence of a function σ from instance attributes to values, and of a function τ from routine parameters to values. The functions $\sigma_1, \cdots, \sigma_k$ and $\tau_1, \cdots, \tau_k$ of the locked objects $a_1, \cdots, a_k$ should also be passed as evaluation context, but we allow them to be empty if the

expression is free from qualified references. We will denote the value of expression exp_{ref} by $[\![exp_{ref}]\!]_{\mathcal{E}}^{\sigma,\tau,\tilde{\sigma},\tilde{\tau}}$ where $\mathcal{E}$ stands for *expression*. Its definition is given on Figure 1, where $Current$ is the name of the current object, $Void$ is a reference to any non-existing object, and `f` is the name of a statically-defined function. We shall, in the latter case, assume that `f` is always applied to a correct tuple of parameters (in number and type).

$$
\begin{array}{rcl}
[\![a_i.a]\!]_{\mathcal{E}}^{\sigma,\tau,\tilde{\sigma},\tilde{\tau}} & \triangleq & [\![a]\!]_{\mathcal{E}}^{\sigma_i,\tau_i} \\
[\![a]\!]_{\mathcal{E}}^{\sigma,\tau,\tilde{\sigma},\tilde{\tau}} & \triangleq & \sigma(a) \text{ if } a \in dom(\sigma) \\
[\![a]\!]_{\mathcal{E}}^{\sigma,\tau,\tilde{\sigma},\tilde{\tau}} & \triangleq & \tau(a) \text{ if } a \in dom(\tau) \\
[\![Current]\!]_{\mathcal{E}}^{\sigma,\tau,\tilde{\sigma},\tilde{\tau}} & \triangleq & \sigma(Current) \\
[\![Void]\!]_{\mathcal{E}}^{\sigma,\tau,\tilde{\sigma},\tilde{\tau}} & \triangleq & Void \\
[\![\mathtt{f}(exp_1,\cdots,exp_k)]\!]_{\mathcal{E}}^{\sigma,\tau,\tilde{\sigma},\tilde{\tau}} & \triangleq & f([\![exp_1]\!]_{\mathcal{E}}^{\sigma,\tau,\tilde{\sigma},\tilde{\tau}},\cdots,[\![exp_k]\!]_{\mathcal{E}}^{\sigma,\tau,\tilde{\sigma},\tilde{\tau}})
\end{array}
$$

Figure 1: Semantics of Expressions

The semantic rules are given in Figures 2 and 3. There are 10 axioms and one inference rule. We now give some insights on individual rules, starting with the ones of Figure 2. The only rule PAR is introduced to allow commutation of configurations along the parallel operator; this rule follows the chemical style by Berry and Gonthier, that has been used later by Milner for the π-calculus.

The first axiom ASS addresses assignment: the attribute a of object α is modified, resulting in an update of store σ where the old value for a is replaced by the value of e computed in the context of the attributes and local variables of α. The axiom CREATE introduces a new object of class A in the configuration and assigns the new reference to attribute a of object α. The created object is idle; its store contains initial values for all the fields, and the attribute $Current$ takes the new (self) reference (we use $\sigma\{a \mapsto \alpha\}$ for the partial function identical to σ except on x for which α is returned, with the domain of σ extended to a if necessary). Since the new object does not execute any routine when it is created, no local variables are defined for it, which is noted $\tau^{\emptyset}$. The object is not locked, and its queue of asynchronous calls is empty.

The LOCK and UNLOCK axioms allow one object (called β in the axioms) to lock, and subsequently unlock, a set of other objects (called $\alpha_1,\cdots,\alpha_k$ in the axioms). Originally, the objects in $\tilde{\alpha}$ (a shortcut notation for $\alpha_1,\cdots,\alpha_k$) must be idle, unlocked ($\lambda = \bot$) and their queues must be empty ($\iota \doteq \star$). The semantics it to lock them atomically if the wait condition is true. The special statement `unlock` is then placed immediately after the last instruction of the body. The statement `unlock` may be used later for unlocking objects $\tilde{\alpha}$. Please notice that, although `unlock` $\tilde{\alpha}$ is by nature a meta-syntactic operator, we could have avoided to use it by putting the needed information in the state of the object; we found the structure of the state already complicated enough not to do so. The locking operation imposes $\lambda = \beta^{\tilde{\alpha}}$ so that each object in $\tilde{\alpha}$ knows the set of objects locked with it. The only effect of unlocking is to put the *termination mark* $\diamond$ in all the queues of the locked

$$\frac{M \mid M' \rightarrow M''}{M' \mid M \rightarrow M''}\text{PAR}$$

$$\frac{}{M \mid \langle \alpha, a := e; stm, \sigma, \tau, \lambda, \iota \rangle \rightarrow M \mid \langle \alpha, stm, \sigma\{a \mapsto [\![e]\!]^{\sigma,\tau}_{\mathcal{E}}\}, \tau, \lambda, \iota \rangle}\text{ASS}$$

$$\frac{}{\begin{array}{c} M \mid \langle \alpha, a := \texttt{create}\ A; stm, \sigma, \tau, \lambda, \iota \rangle \\ \downarrow \\ M \mid \langle \alpha, stm, \sigma, \tau, \lambda, \iota \rangle \mid \langle \beta, I, \sigma^{init}_{A}\{Current \mapsto \beta\}, \tau^{\emptyset}, \bot, \star \rangle \end{array}}\text{CREATE}$$

$$\frac{[\![a_1]\!] = \alpha_1, \cdots, [\![a_k]\!] = \alpha_k \qquad [\![exp_{ref}]\!]^{\sigma,\tau,\tilde{\sigma},\tilde{\tau}}_{\mathcal{E}} = tt}{\begin{array}{c} M \mid \langle \beta, \texttt{with}\ a_1, \cdots a_k\ \texttt{when}\ exp_{ref}\ \texttt{do}\ stm\ \texttt{end}\ ; stm', \sigma, \tau, \lambda, \iota \rangle \mid \\ \langle \alpha_1, I, \sigma_1, \tau_1, \bot, \star \rangle \mid \cdots \mid \langle \alpha_k, I, \sigma_k, \tau_k, \bot, \star \rangle \\ \downarrow \\ M \mid \langle \beta, stm; \texttt{unlock}\ \tilde{\alpha}; stm', \tau, \lambda, \iota \rangle \mid \\ \langle \alpha_1, I, \sigma_1, \tau_1, \beta^{\tilde{\alpha}}, \star \rangle \mid \cdots \mid \langle \alpha_k, I, \sigma_k, \tau_k, \beta^{\tilde{\alpha}}, \star \rangle \end{array}}\text{LOCK}$$

$$\frac{\alpha_1, \cdots, \alpha_k \doteq \tilde{\alpha}}{\begin{array}{c} M \mid \langle \beta, \texttt{unlock}\ \tilde{\alpha}; stm, \sigma, \tau, \lambda, \iota \rangle \mid \\ \langle \alpha_1, I, \sigma_1, \tau_1, \beta^{\tilde{\alpha}}, \iota \rangle \mid \cdots \mid \langle \alpha_k, I, \sigma_k, \tau_k, \beta^{\tilde{\alpha}}, \iota \rangle \\ \downarrow \\ M \mid \langle \beta, stm, \sigma, \tau, \lambda, \iota \rangle \mid \\ \langle \alpha_1, I, \sigma_1, \tau_1, \beta^{\tilde{\alpha}}, \iota \oplus \diamond \rangle \mid \cdots \mid \langle \alpha_k, I, \sigma_k, \tau_k, \beta^{\tilde{\alpha}}, \iota \oplus \diamond \rangle \end{array}}\text{UNLOCK}$$

$$\frac{\alpha_1, \ldots, \alpha_k \doteq \tilde{\alpha}}{\begin{array}{c} M \mid \langle \alpha_1, I, \sigma_1, \tau_1, \beta^{\tilde{\alpha}}, \diamond \rangle \mid \cdots \mid \langle \alpha_k, I, \sigma_k, \tau_k, \beta^{\tilde{\alpha}}, \diamond \rangle \\ \downarrow \\ M \mid \langle \alpha_1, I, \sigma_1, \tau_1, \bot, \star \rangle \mid \cdots \mid \langle \alpha_k, I, \sigma_k, \tau_k, \bot, \star \rangle \end{array}}\text{RETURN}_{lock}$$

Figure 2: Operational Semantics: Assignment, Locking and Creation Rules

objects. Indeed, unlocking can not be done when the locker object terminates the execution of the conditional critical region, since asynchronous calls may still have to be executed by the locked objects at that time.

When all the locked objects have become idle and have treated all the incoming calls in their respective queues, they can synchronize through axiom RETURN$_{lock}$ to get free from the lock held on them. This has for condition that all objects in $\tilde{\alpha}$ have a termination sign in their queues (the caller has unlocked them).

$$\frac{\begin{array}{l} class(\beta) = B \qquad [\![b]\!]_{\mathcal{E}}^{\sigma_\alpha,\tau_\alpha} = \beta \qquad \tilde{\gamma} \doteq [\![\tilde{c}]\!]_{\mathcal{E}}^{\sigma_\alpha,\tau_\alpha} \\ \{m(a_1 : A_1, \cdots, a_k : A_k) \texttt{ is do } stm; \texttt{return } exp \texttt{ end}\} \in rout(B) \end{array}}{\begin{array}{c} M \mid \langle \alpha, a := b.m(\tilde{c}); stm_\alpha, \sigma_\alpha, \tau_\alpha, \lambda_\alpha, \iota_\alpha\rangle \mid \langle \beta, I, \sigma_\beta, \tau_\beta, \alpha^{\tilde{\beta}}, \star\rangle \\ \downarrow \\ M \mid \langle \alpha, \texttt{receive}_\beta\ a; stm_\alpha, \sigma_\alpha, \tau_\alpha, \lambda_\alpha, \iota_\alpha\rangle \mid \\ \langle \beta, stm; \texttt{return } exp, \sigma_\beta, \tau_\beta\{\tilde{a} \mapsto \tilde{\gamma}\}, \alpha^{\tilde{\beta}}, \star\rangle \end{array}} \text{CALL}_{sync}$$

$$\frac{}{\begin{array}{c} M \mid \langle \alpha, \texttt{receive}_\beta\ a; stm, \sigma_\alpha, \tau_\alpha, \lambda_\alpha, \iota_\alpha\rangle \mid \langle \beta, \texttt{return } exp, \sigma_\beta, \tau_\beta, \alpha^{\tilde{\beta}}, \iota\rangle \\ \downarrow \\ M \mid \langle \alpha, stm, \sigma_\alpha, \tau_\alpha\{a \mapsto [\![exp]\!]_{\mathcal{E}}^{\sigma_\beta,\tau_\beta}\}, \lambda_\alpha, \iota_\alpha\rangle \mid \langle \beta, I, \sigma_\beta, \tau^\emptyset, \alpha^{\tilde{\beta}}, \star\rangle \end{array}} \text{RETURN}_{sync}$$

$$\frac{\begin{array}{l} class(\beta) = B \qquad [\![b]\!]_{\mathcal{E}}^{\sigma_\alpha,\tau_\alpha} = \beta \qquad \tilde{\gamma} \doteq [\![\tilde{c}]\!]_{\mathcal{E}}^{\sigma_\alpha,\tau_\alpha} \\ \{m(a_1 : A_1, \cdots, a_k : A_k) \texttt{ is do } stm; \texttt{return end}\} \in rout(B) \end{array}}{\begin{array}{c} M \mid \langle \alpha, b.m(\tilde{c}); stm_\alpha, \sigma_\alpha, \tau_\alpha, \lambda_\alpha, \iota_\alpha\rangle \mid \langle \beta, stm_\beta, \sigma_\beta, \tau_\beta, \alpha^{\tilde{\beta}}, \iota_\beta\rangle \\ \downarrow \\ M \mid \langle \alpha, stm_\alpha, \sigma_\alpha, \tau_\alpha, \lambda_\alpha, \iota_\alpha\rangle \mid \langle \beta, stm_\beta, \sigma_\beta, \tau_\beta, \alpha^{\tilde{\beta}}, \iota_\beta \cdot m(\tilde{\gamma})\rangle \end{array}} \text{CALL}_{async}$$

$$\frac{\begin{array}{l} class(\beta) = B \\ \{m(a_1 : A_1, \cdots, a_k : A_k) \texttt{ is do } stm; \texttt{return end}\} \in rout(B) \end{array}}{M \mid \langle \alpha, I, \sigma, \tau, \lambda, m(\tilde{\gamma}) \cdot \iota\rangle \rightarrow M \mid \langle \alpha, stm; \texttt{return}, \sigma, \tau\{\tilde{a} \mapsto \tilde{\gamma}\}, \lambda, \iota\rangle} \text{ACCEPT}_{async}$$

$$\frac{}{M \mid \langle \alpha, \texttt{return}, \sigma, \tau, \lambda, \iota\rangle \rightarrow M \mid \langle \alpha, I, \sigma, \tau^\emptyset, \lambda, \iota\rangle} \text{RETURN}_{async}$$

Figure 3: Operational Semantics: Synchronous and Asynchronous Call Rules

Rules for synchronous and asynchronous calls are presented on Figure 3. Two interactions can be observed for synchronous interactions: the beginning of the call, and its end when a value is returned. Triggering the call (axiom CALL$_{sync}$) implies that the supplier is idle and the client is waiting to execute this call; the queues of the supplier should be empty since no asynchronous call may be performed before a synchronous call (according to the assumption above). Triggering the call

results in the execution of the statement stm of the wanted routine m and the assignment of the formal parameters $\tilde{a}$ with the value of the effective arguments. For the client, a special instruction `receive` is inserted as the next statement to execute in order to get the returned value. This `receive` can be executed only through axiom RETURN$_{sync}$, reflecting the synchronous nature of the call. Then the variable a is assigned with the result of the evaluation of the expression.

Asynchronous calls have three steps of observation: their emission, their delivery to the supplier, and the termination of the execution on the supplier side. The call consists, in axiom CALL$_{async}$, to place the call in the queue of the supplier object, provided that this object is locked by the client. The effective arguments of the call are evaluated in the context of the client object. Note that the supplier does not have to be idle for this axiom to be applicable. The call can later be accepted by the supplier (axiom ACCEPT$_{async}$), if this supplier is idle and the call is in first position in the queue. The called routine then starts executing. This routine ends by a plain `return` that can be eliminated by the supplier object independently of its environment (no synchronization needed) using axiom RETURN$_{async}$.

The initial state of a computation comprises an instance of a statically determined *root* class, featuring a statically determined creation procedure m. To simplify the proof system presented in the next section, we will further assume that no other instance of this root class may be later created by the program. The root object will be deemed to be locked by itself and execute the body of its creation procedure; it can be written formally $\langle \alpha, stm, \sigma_A^{init}\{Current \mapsto \alpha\}, \tau^{init}, \alpha^\alpha, \star \rangle$, stm being the body of routine m, σ^{init} being defined as above, and τ_{init} containing the effective arguments to the creation procedure. The object is deemed to be locked by itself, in a "virtual" conditional critical region executed by itself and locking no other object but itself. Other initial configurations could have been considered but we chose this one because it facilitates the formulation of the proof system.

Definition 3.1 (Reachability).
A configuration M' is said to be reachable *from a configuration M if and only if $M \equiv M'$ or $M \rightarrow M'$ or there exists $M_1, \cdots, M_k$ for some $k \geq 1$ such that*

$$M \rightarrow M_1 \rightarrow \cdots \rightarrow M_k \rightarrow M' \ .$$

In such a case we will note $M \rightarrow^* M'$, for the reflective and transitive closure of $\rightarrow$ representing reachability.

4 A Proof System For SCOOP

We present a proof system, based on first-order logic, that allows to prove safety properties of SCOOP programs. Any safety property can be expressed as a *global invariant* that should be maintained by the program under scrutiny. To prove a global invariant we employ Ashcroft's method based on inductive assertions [Ash75], that is itself an adaptation of Floyd's method [Flo67] to concurrent programs. The method consists in showing that

- the global invariant is true initially,

- no (atomic) computation step can invalidate that invariant.

Stylistically, the proof system is rephrased as a two-level deductive system, following [OG76] and [AFdR80]. The rephrasing consists in separating:

- *local reasoning* about statements at the object level, which involves making assumptions about the global environment of execution of those statements, from
- *global reasoning* about the different assumptions used at the local level, which should pass a so-called *cooperation test*, meaning that they should not contradict one another.

At the local level, annotations borne by a program are often called *proof outlines*. We borrowed the name of the global test for message-passing processes from [AFdR80]; the corresponding test for shared-variable concurrency is called *non-interference* [OG76]. Formalizing the system as the verification of a global invariant has the advantage compared to a Hoare logic that it can be applied also to *reactive* programs, that are programs that do not terminate.

This results in a simpler proof system, because reasoning on synchronous calls can be done without making any assumption about the contents of the queues of the target objects (they are all empty). Extending the proof system to take care of intertwined synchronous and asynchronous calls would not be difficult, but it would make writing actual proofs more difficult.

We also assume that there is exactly one object per processor; in other words all objects are *separate* from one another. This allows to avoid distinguishing local from separate calls, all calls being separate. We finally disallow variables referring to currently locked objects to be modified.

The general form of the proof system, assertion language and notations are borrowed from [AdBSdR03, dB02, AdB94]. This has, among others, the virtue of allowing easier comparison with those proof systems.

4.1 An Example of a Two-Tiered CSP Proof (Tutorial)

In order to illustrate and motivate the proof method outlined above, we give a very short example of a proof of CSP processes following [AFdR80]. Please consider first the usual proof rule for assignment in a sequential program:

$$\{p[e/x]\}\ x := e\ \{p\}$$

states that, for the predicate p to be true (immediately) after the assignment, then the predicate that is obtained from p by replacing all the occurrences of x by e has to be true (immediately) before the assignment. Then $p[e/x]$ is called the *weakest precondition* for predicate p. To prove that assignment preserves a global invariant GI, one only has to replace p with GI.

To cope with parallelism, one may consider entire configurations of parallel processes (their Cartesian product) and show that, for any possible interleaving of the actions of such configuration, the invariant is maintained. This results in an exponential blowup of the number of checks to accomplish [dRdBH$^+$01, Chapter 3].

A solution to subdue this complexity is to use the structure of the processes; this results in a *two-tiered* proof system. This is possible for message-passing processes under the condition that assignments appearing in individual processes are restricted so that they can not invalidate the global invariant (*i.e.* only variables *not* appearing in the invariant can be assigned to). Each process can then be checked individually against a so-called *proof outline* made of *local assertions*, giving the low-tier of the proof system. Concurrency does not play a role in this lower tier, the consistency check only guarantees that the proof outline reflects correctly the local actions of the process, respecting for example the weakest precondition rule for assignment. In the high tier, one must prove that any interaction among processes may not invalidate the global invariant, in which case they are said to *cooperate*.

The cooperation test can be seen as a generalization of the weakest precondition rule to parallel processes. We name processes put in parallel (with the "||" operator) P_1, P_2, $\cdots$ from left to right. So, the following program, allows the process P_1 (on the left) to send the value of expression e to P_2 (on the right), which P_2 agrees to receive and to keep in variable x:

$$P_2!e \;||\; P_1?x$$

Executing this program would have an effect similar to the "sequential" assignment above, and one would like to apply the same rule:

$$\{p[e/x]\}\ P_2!e \;||\; P_1?x\ \{p\}$$

which is correct.

In its full version, the cooperation test states that, for any communication statements $P_2!e$ and $P_1?x$ that may actually be executed conjointly (they are said to *match*), then their proof outlines $\{p\}\ P_2!e\ \{q\}$ and $\{p'\}\ P_1?x\ \{q'\}$ must be such that if $GI \wedge p \wedge p'$ is true immediately before the communication takes place then $(GI \wedge q \wedge q')[e/x]$ has to be true immediately after. This relies on the assumption that, like assignment, communication happens atomically.

However, this is not sufficient in general and we just need a slightly less trivial example to make the complexity appear:

$$((P_2!2; P_2?x) \sqcap (P_2?x; P_2!1)) \;||\; ((P_1?y; P_1!1) \sqcap (P1!2; P_1?y))$$

where we use completely parenthesized notation, semicolon for the sequence, and $\sqcap$ for the nondeterministic (external) choice.

Suppose indeed that all variables are initially set to 0, and that the invariant we want to prove is $GI \stackrel{\Delta}{=} \neg(x = 2 \wedge y = 2)$. This invariant is obviously true on the example above, because the execution of communication assigning 2 to y excludes the execution of communication assigning 0 to x, and conversely the execution of the second communication implies that the first has not been executed. However, to pass the cooperation test, one would have to consider all couples of statements of the form $P_2!2 \;||\; P_1?y$, and show that their respective executions preserve the invariant (which we would not be able to do in the present case because some of them do not). The necessary distinction is that of *semantic matching* of communication instructions, as opposed to *syntactic matching*.

It is shown in [AFdR80] that any semantic matching or mismatching in parallel processes can always be determined using the cooperation test by merely introducing:

- *bracketed sections* enclosing communication statements (followed by one or many assignments) like $\langle P_1!x; x := 1\rangle$ that ensure the *atomicity* of the execution of such communication and assignments, and
- *auxiliary variables* that are *observation* variables and can not influence the execution of the program; each auxiliary variable may be modified only inside the bracketed sections of *one* of the processes.

Then, quite surprising reasoning rules are allowed in proof outlines:

$$\{p\}\ P_2!e\ \{q\} \text{ and } \{p'\}\ P_1?y\ \{q'\}$$

for any P_1, P_2, y, and e. Those rules state that, in the case of emission or reception of a message, *any* post-condition can be assumed to be true locally. In the case of reception this is obvious that the received value can not be determined locally, and one has to make an assumption about it, resulting in a post-condition of an unknown form. For the emission, such "miraculous" post-condition is allowed because one may not actually know whether the emission is going to be executed (*i.e.* has a semantical match) or not. In both cases, assignments to auxiliary variables is sufficient to determine semantic matching.

The cooperation test, for any pair of syntactically matching communication statements with proof outlines $\{p\}\ \langle P_2!e; i := e_i\rangle\ \{q\}$ and $\{p'\}\ \langle P_1?x; j := e_j\rangle\ \{q'\}$, then becomes:

$$(GI \wedge p \wedge p') \Rightarrow (GI \wedge q \wedge q')[e, e_i, e_j/x, i, j]$$

Applied to the example above, we can introduce auxiliary variables i and j, and bracket the processes as follows:

$$\begin{array}{l}\{i=0\}\langle P_2!2; i := i+1\rangle; \{i=1\}\langle P_2?x; i := i+1\rangle\{x=1 \wedge i=2\} \\ \qquad\sqcap \\ \{i=0\}(\langle P_2?x; i := i+1\rangle; \{x=2 \wedge i=1\}\langle P_2!1; i := i+1\rangle\{x=2 \wedge i=2\}\end{array}$$

and

$$\begin{array}{l}\{j=0\}\langle P_1?y; j := j+1\rangle; \{y=2 \wedge j=1\}\langle P_1!1; j := j+1\rangle\{y=2 \wedge j=2\} \\ \qquad\sqcap \\ \{j=0\}\langle P_1!2; j := j+1\rangle; \{j=1\}\langle P_1?y; j := j+1\rangle\{y=1 \wedge j=2\}\end{array}$$

Strengthening the global invariant $Gi \triangleq (i=j) \wedge \neg(x=2 \wedge y=2)$ we can show cooperation, for example between

$$\{j=0\}\langle P_1?y; j := j+1\rangle\{y=2 \wedge j=1\}$$

and

$$\{x=2 \wedge i=1\}\langle P_2!1; i := i+1\rangle\{x=2 \wedge i=2\}$$

we have

$$(j=0 \wedge x=0 \wedge i=1 \wedge (i=j) \wedge \neg(x=2 \wedge y=2)) \Leftrightarrow \mathit{ff}$$

by contradiction on the equality of i and j, which implies anything including

$$(y = 2 \wedge j = 1 \wedge x = 2 \wedge i = 2 \wedge GI)[1, i+1, j+1/y, i, j]$$

and thus shows cooperation. Hence, passing the cooperation test should be trivial for semantically mismatching communication statements, because assignments to auxiliary variables and their comparison in the global invariant should make preconditions false.

Once the invariant has been proved using the program completed with auxiliary variables, an additional rule can be used to suppress those auxiliary variables, and hence obtain a proof for the original program. This rule is sound since the auxiliary variables do not modify the behavior of the program, but only observe it.

4.2 The Assertion Language

We use, as in [AdBSdR03, dB02, AdB94], both a *local* and a *global* assertion language. Logical variables shall range over a denumerable set V with typical elements x, y, z, and their decorated variants.

The local assertion language allows to state properties at the object level by evaluating and comparing the values of attributes and local variables of a given object assumed as context. The identity of this object can be accessed through the attribute $Current$. In order to enforce locality, qualified references to other objects are disallowed: for example, taking the value of attribute b of an attribute a, usually written $a.b$, is forbidden. In local assertions any name a has to be the name of an attribute of the current object. Integers and booleans can then be manipulated freely, while the only allowed operation on references is comparison. Quantified logical variables are given a type T, which in the case of local assertions is restricted to integers, booleans, and tuples built on those types.

At the global level there is no object known as $Current$ for a context, and all references should be qualified, using some logical variable x^T that will range over the set of *existing* objects of type T, not including $Void$. A qualified reference can then only refer to an attribute (and not to a local variable). Please note that in general we shall only mention the type of a variable when necessary, leaving it otherwise determined by the context.

$exp_l ::= x \mid a \mid Current \mid Void \mid \mathtt{f}(exp_l, \cdots, exp_l)$	$e \in LExp$	local expressions
$ass_l ::= exp_l \mid \neg ass_l \mid ass_l \wedge ass_l \mid \exists x^T : ass_l$	$p \in LAss$	local assertions
$exp_g ::= x \mid Void \mid exp_g.a \mid \mathtt{f}(exp_g, \cdots, exp_g)$	$E \in GExp$	global expressions
$ass_g ::= exp_g \mid \neg ass_g \mid ass_g \wedge ass_g \mid \exists x^T : ass_g$	$P \in GAss$	global assertions

Table 1: Syntax of Assertions

Other logical connectives and constants tt and ff can be defined as usual using the ones provided here. The scope of quantification will be considered to extend as much as possible to the right.

The semantics of assertions is defined in Table 2. The semantics of local assertions $[\![\ ]\!]_{\mathcal{L}}$ is given assuming a context that encloses the values of logical variables ω, the

values of the attributes of the current object σ. The values of the parameters of the currently executed call τ are not needed, since they can not be referred to by assertions. The semantic function for global assertions $[\![\ ]\!]_{\mathcal{G}}$ takes as arguments a function ω giving the current value of each logical variables, and the function $\tilde{\sigma}$ giving the values of the attributes of all currently existing objects. We formally define $\tilde{\sigma}$, based on a list $\sigma_1, \cdots, \sigma_k$ of object states, as the function such that for any $\alpha_i \in \alpha_1, \cdots, \alpha_k$ we have $\tilde{\sigma}(\alpha_i) \triangleq \sigma_i$. In other words $\tilde{\sigma}$ gives the set of existing objects and the values of their attributes.

$$
\begin{array}{rcl}
[\![x]\!]^{\omega,\sigma}_{\mathcal{L}} & \triangleq & \omega(x) \\
[\![a]\!]^{\omega,\sigma}_{\mathcal{L}} & \triangleq & \sigma(a) \text{ if } a \in dom(\sigma) \\
[\![Current]\!]^{\omega,\sigma}_{\mathcal{L}} & \triangleq & \sigma(Current) \\
[\![\mathtt{f}(e_1, \cdots, e_k)]\!]^{\omega,\sigma}_{\mathcal{L}} & \triangleq & f([\![e_1]\!]^{\omega,\sigma}_{\mathcal{L}}, \cdots, [\![e_k]\!]^{\omega,\sigma}_{\mathcal{L}}) \\
[\![\neg p]\!]^{\omega,\sigma}_{\mathcal{L}} & \triangleq & \neg [\![p]\!]^{\omega,\sigma}_{\mathcal{L}} \\
[\![p_1 \wedge p_2]\!]^{\omega,\sigma}_{\mathcal{L}} & \triangleq & [\![p_1]\!]^{\omega,\sigma}_{\mathcal{L}} \wedge [\![p]\!]^{\omega,\sigma}_{\mathcal{L}} \\
[\![\exists x^T : p]\!]^{\omega,\sigma}_{\mathcal{L}} & \triangleq & [\![p]\!]^{\omega[x \mapsto v],\sigma}_{\mathcal{L}} = tt \text{ for some } v \in Val^T
\end{array}
$$

$$
\begin{array}{rcl}
[\![x]\!]^{\omega,\tilde{\sigma}}_{\mathcal{G}} & \triangleq & \omega(x) \\
[\![E.a]\!]^{\omega,\tilde{\sigma}}_{\mathcal{G}} & \triangleq & \sigma([\![E]\!]^{\omega,\tilde{\sigma}}_{\mathcal{G}})(a) \\
[\![\mathtt{f}(E_1, \cdots, E_k)]\!]^{\omega,\tilde{\sigma}}_{\mathcal{G}} & \triangleq & f([\![E_1]\!]^{\omega,\tilde{\sigma}}_{\mathcal{G}}, \cdots, [\![E_k]\!]^{\omega,\tilde{\sigma}}_{\mathcal{G}}) \\
[\![\neg P]\!]^{\omega,\tilde{\sigma}}_{\mathcal{G}} & \triangleq & \neg [\![P]\!]^{\omega,\tilde{\sigma}}_{\mathcal{G}} \\
[\![P_1 \wedge P_2]\!]^{\omega,\tilde{\sigma}}_{\mathcal{G}} & \triangleq & [\![P_1]\!]^{\omega,\tilde{\sigma}}_{\mathcal{G}} \wedge [\![P]\!]^{\omega,\tilde{\sigma}}_{\mathcal{G}} \\
[\![\exists x^T : P]\!]^{\omega,\tilde{\sigma}}_{\mathcal{G}} & \triangleq & [\![P]\!]^{\omega[x \mapsto v],\tilde{\sigma}}_{\mathcal{G}} = tt \text{ for some } v \in Val^T
\end{array}
$$

Table 2: Semantics of Local and Global Assertions

We will write $\omega, \sigma \models_{\mathcal{L}} p$ if and only if $[\![p]\!]^{\omega,\sigma}_{\mathcal{L}} = tt$, and $\models_{\mathcal{L}} p$ if and only if $[\![p]\!]^{\omega,\sigma}_{\mathcal{L}} = tt$ for all ω, σ, and τ. In the same way, we will write $\omega, \tilde{\sigma} \models_{\mathcal{G}} P$ for $[\![P]\!]^{\omega,\tilde{\sigma}}_{\mathcal{G}} = tt$, and $\models_{\mathcal{G}} P$ when $[\![P]\!]^{\omega,\tilde{\sigma}}_{\mathcal{G}} = tt$ for all ω and $\tilde{\sigma}$.

4.3 The Proof Rules

4.3.1 Needed Adaptations

We now adapt the method outlined in Section 4.1 to deal with SCOOP programs. The axiomatic semantic rules tend to reflect the operational semantic rules of Section 3.2 dealing with the same construction. Compared to the axiomatic semantics for CSP outlined in Section 4.1, the essential differences that we have to cope with are:

- *object references* can be transmitted (introducing the *aliasing* phenomenon),
- the *locking mechanism* is obtained through conditional critical regions,
- *routine calls* are used to communicate (not simple message passing),

- *asynchronous calls* are available.

Proof rules for asynchronous message passing and routine calls are given in [SS84], while aliasing and reference passing are treated in [AdB94]. Both rely on earlier contributions, such as [Mor82] and [GdRR82].

In [GdRR82, SS84], the cooperation test is given a slightly different form in order to handle routine definitions and calls. To track the contents of communication channels in [SS84], auxiliary variables are introduced and their values updated by ad-hoc instructions added to the program to be verified. We combine both approaches (as done for example in [AdBSdR03]), with main challenge being tracking of lock operations.

4.3.2 Tracking Down the Locking Structure

We do not claim to bring any outstanding technical novelty in this article; the key difficulty we had to solve is the construction of the cooperation test dealing with the locking mechanism. Such a test is not present in previous proof systems, since this construction is not used by any other object-oriented concurrent language that we know of (although CCRs are mentioned in [Phi00] as a way of synchronizing objects, no language that uses them is referred to).

Assertions found in proofs of SCOOP programs need a certain power of observation in order to be sound and allow one to prove all true safety properties of a given program. More precisely, which objects are locked at any given point in time, and in which conditions they have been locked (by which object, CCR, etc.) is of prime relevance: establishing the structure of the tree of locked objects is a typical safety property. Much of the effort in devising the following proof system has been made to give a way to encode such locking structures in the logic. Also, the contents of the queues used to keep asynchronous calls before they are handled by the supplier object have to be observable. This is much more simpler to encode than the locking-related structure. To give such observation power to the specifier, we introduce auxiliary variables and enforce their updates through the rules of the proof system.

To describe the locking structure, we need to find out for each object:

- by which other object it has been locked (the locking object is the client),
- by what CCR in its client the object has been locked,
- what the other objects locked with the current object in the same CCR are.

The locking structure resembles the one depicted in Figure 4. This figure represents the snapshot of a computation among objects numbered 1,2,... 10, which are all separate from one another. In this figure objects 2, 3 and 4 are locked by the same CCR executed by object 1. The body of this CCR has synchronously called routine `m` on object 3, which is now executing. This routine executes the body of a CCR that had locked objects 6 and 7, and called query `n` object 6, which is now executing. The body of the routine executed by object 6 has then locked 7, 8 and 9, and asynchronously called routines $\mathtt{o}_1$ and $\mathtt{o}_2$ respectively on 7 and 8, which are now

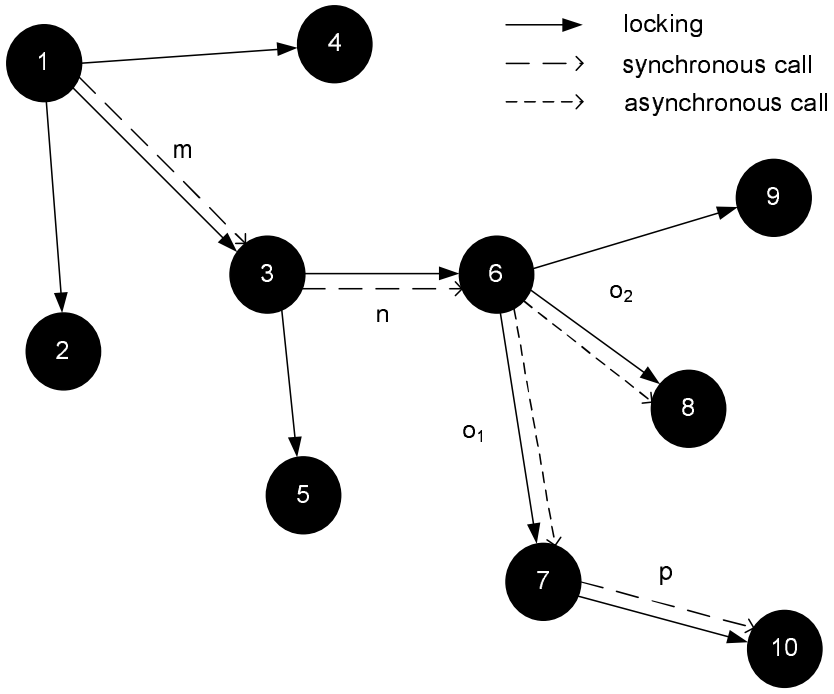

Figure 4: An Example Locking Structure Among SCOOP Objects

executing. The call on 7 has locked object 10 and object 10 is now synchronously executing routine `p` invoked by 7.

The scenario above emphasizes the fact that an object has been locked by another one as a consequence of either a synchronous or an asynchronous call. This knowledge is indeed needed because synchrony and asynchrony have different implications on the release of objects. For example, in Figure 4 objects 1, 2, 3, 4 and 5 will be locked until the routine executed by object 6 has finished. On the other hand, since object 6 has only made two asynchronous calls, it does not require them to terminate in order to terminate the execution of `n`. Hence, we do not know whether objects 1 to 6 will effectively be locked during the whole execution of `p`, and probably they will not. So during the execution of `p`, only objects 7, 8, 9 and 10 are locked for sure; the others objects may be locked or not locked. This is the key characteristics of reasoning about SCOOP programs.

4.3.3 Augmentation with Auxiliary Attributes

We now introduce the auxiliary attributes that we need to describe configurations of objects, reflecting the semantics of locking and asynchronous calls. We first assume that each CCR in the analyzed program is given a unique identifier that we consider to be an integer (starting from 1). Each object has an attribute *lockedBy* that is a triple $\langle x, i, j \rangle$ indicating that the current object is locked by the j^{th} execution of CCR i by object x, or special value $\perp$ if the object is not locked. Each object x also maintains a list *locked* of object that are currently locked by it. Attribute *locked* needs to be a list because of asynchronous calls: an object may finish the execution of a CCR and start executing another one while asynchronous calls are

still being executed in the context of the first CCR. The particular case where the two CCRs are the same (in a loop for example) motivates the need for another number (j above) to distinguish different CCR executions. We will actually take the value j from an array *ccr* (represented as a list) that will contain the current execution number of CCR i in its i^{th} cell, noted $ccr@i$. Also, attribute *locked* will actually be a heterogeneous list, since it does not contain only the references to the locked objects: its first element is a tuple, its second and third elements are the numbers associated with the executed CCR, and those are followed by the list of identities of the locked objects. The tuple contains the values of the *lockedBy* attribute of the current object at the time of the execution of the CCR, completed by a boolean μ that indicates whether the routine currently executed by the locking object is synchronous ($\mu = tt$) or asynchronous ($\mu = ff$). This will also be denoted on each object for the currently executed method by an auxiliary boolean attribute named *smode*, which possible values will retain the same interpretation as the ones for μ. We will take the root object to be locked by itself, with fictitious number 0 for both CCR number and execution index. The program to verify is augmented so that *smode* and array *ccr* are updated when necessary, at the beginning of CCR i for $ccr@i$, and upon the start of the execution of each routine for *smode*.

To reflect the effects of asynchronous routine calls, each object y is extended with an auxiliary attribute *queue* that gives, for each object x_i currently locked by y, the sequence (that is a list, noted $y.queue(x_i)$) of asynchronous calls that have been emitted during the execution of the current CCR. Each call of routine m with arguments $\tilde{a}$ will be simply noted by $m(\tilde{a})$. Since *queue* is like a partial function from objects to lists, we will use functional update like $\{x_i \mapsto e\}$.

Each object will also have an attribute *in* that will give the sequence of asynchronous calls that have been treated. The call that is currently treated, if any, will be marked in the list as $m(\tilde{a})^\dagger$. We will consider $m(\tilde{a})^\dagger$ to be strictly different from $m(\tilde{a})$. This will allow to observe the sequentiality of the treatment of asynchronous routine calls. Regarding queues and their manipulation, we will treat them as lists, using the operators introduced in Section 3.2.

In order to cope with asynchronous routine invocations, we also need to introduce *auxiliary parameters* in routine declaration. The value of such parameters is prevented, as for auxiliary attributes, to influence the flow of control of a program. The need for auxiliary parameters in the reasoning arises in one precise case: when a supplier object starts processing an asynchronous routine invocation that it found in its queue, the state of the supplier and the state of the client are not known. This problem is not present in programs that use only synchronous routine invocation or asynchronous message-passing with CSP-like primitives. In those two latter cases, the state of at least one of the communicating peers is known when the routine starts executing. Here, as it is seen in the proof rule handling asynchronous routine invocation in Section 4.3.6, at this step only the value of the arguments passed to the routine can be used to reason about the correctness of the invocation. Introducing additional parameters for routines may therefore be necessary. Nothing will distinguish auxiliary parameters from ordinary parameters, except that auxiliary parameters may only be used in the right member of assignments to auxiliary attributes in bracketed sections. Suppressing auxiliary attributes and parameters will

be not modify the behavior of the program.

We will consider names of other auxiliary attributes to range over u, v, and w with the usual decorations.

4.3.4 Bracketed Sections

In order to accurately observe process interactions, we need to update the auxiliary variables atomically with respect to the occurrence of the phenomenon they are supposed to observe. This is made possible, as in other proof systems, by placing potentially interacting instructions into bracketed sections that may contain additional assignments to auxiliary attributes. Should be bracketed:

- synchronous routine invocations (like `<a := b.m(c)>`),
- asynchronous routine invocations (like `<a.b(c); u := 1>`),
- the first and last instruction of the body of conditional critical regions (like `<with ... do u := 0> ... <v := 1 end>`),
- the first and last instruction of the body of a routine (like `<m (x) is do u := 0> ... <v := 1 return> end`),
- object creation instructions (like `<a := new C; u := 1>`).

For a synchronous routine invocation, we allow no observation assignment to take place. To observe such routine invocation from the client point-of-view is not needed, any observation that may be needed when proving some property can be introduced in the code of the supplier object (allowing additional observation on the client side would not be harmful, but unnecessarily complicate the proof system). Asynchronous routine invocation can be observed on the client side by assigning an auxiliary variable (`u := 1`). In such a case, the emission of the message will be observed. Contrarily to the case of synchronous routine invocations, some properties that may not be proved unless such observations are allowed. For a CCR we consider the locking of many objects and the evaluation of the wait condition to be atomic, allowing their observation through the given assignment (`u := 0`). The execution of the `end` statement corresponds to letting the locked objects free from the lock (their effective release can happen at any time afterward). For the body of a routine, the bracketed section at the beginning corresponds to observing a fictitious instruction binding the values of the effective arguments to the names of the formal parameters. The bracketed `return` allows the observation of the final state of the object at the time of the return. It is always the final state that is observed, since the return instruction may only evaluate an expression (that by definition do no produce side effects). Finally, the bracketing of object creations allow the observation to take place immediately after the creation of the new object.

4.3.5 The Local Proof System

The local proof system describes how proof outlines can be formed. To deal with the usual algol-like primitives we employ the weakest-precondition rules given by Hoare

[Hoa69]. For the other primitives, that are CCR, method call and method definition, we give ad-hoc rules. Those rules are quite simple, retaining the "miraculous" feel of the ones used for CSP [AFdR80, SS84].

For routine invocations for example,

$$\frac{}{\{p\} < a.m(\tilde{b}) > \{q\}}$$

the rule states that anything can be assumed to be true whenever the instruction executes. This reflects the fact that proof outlines built using the local proof systems only allow one to make assumptions that will eventually be verified or invalidated by the global cooperation test. We have similar rules for the other interacting instructions given on Figure 5.

$$\frac{}{\{p\} < c := a.m(\tilde{b}) > \{q\}} \qquad \frac{}{\{p\} < a.m(\tilde{b}) > \{q\}}$$

$$\frac{}{\{p\} < \texttt{with}\ a_1, \cdots a_k\ \texttt{when}\ exp_{ref}\ \texttt{do}\ > \{q\}} \qquad \frac{}{\{p\} < \texttt{end} > \{q\}}$$

$$\frac{}{\{p\} < m(a_1 : A_1, \cdots, a_k : A_k)\ \texttt{is do}\ > \{q\}} \quad \frac{}{\{p\} < \texttt{return}\ exp > \{q\}}$$

$$\frac{}{\{p\} < \texttt{return} > \{q\}}$$

Figure 5: Ad-hoc Rules of the local Proof System

The formation of more complex bracketed sections is allowed by introducing assignments to auxiliary attributes, as long as the syntactic constraints described in the previous section are respected. In such a case, the weakest precondition rule applies, as for example in:

$$\frac{\begin{array}{c}\{q[e/x]\}u := exp\{q\} \\ \{p\} < \texttt{with}\ a_1, \cdots a_k\ \texttt{when}\ exp_{ref}\ \texttt{do}\ > \{q[e/x]\}\end{array}}{\{p\} < \texttt{with}\ a_1, \cdots a_k\ \texttt{when}\ exp_{ref}\ \texttt{do}\ u := exp > \{q\}}$$

Although we do not make any syntactic distinction among the different `end` statements (for CCR, routines, and classes), we will from here assume that only the `end` statements of conditional critical regions are bracketed.

4.3.6 The Global Proof System: Cooperation Test

Before we can write the cooperation test, we need a way of translating local assertions and expressions including qualified references into global assertions. This is easily

done, given a logical variable, by merely prefixing routine and attribute accesses with the given name and replacing *Current* by the same name:

$$\begin{array}{rcl@{\qquad}rcl} y{\downarrow}x & \triangleq & y & Void{\downarrow}x & \triangleq & Void \\ a{\downarrow}x & \triangleq & x.a & (\neg p){\downarrow}x & \triangleq & \neg(p{\downarrow}x) \\ a.b{\downarrow}x & \triangleq & x.a.b & (p_1 \wedge p_2){\downarrow}x & \triangleq & (p_1{\downarrow}x) \wedge (p_2{\downarrow}x) \\ Current{\downarrow}x & \triangleq & x & (\exists y : p){\downarrow}x & \triangleq & \exists y : p{\downarrow}x \end{array}$$

Aliasing should be dealt with in global expressions and assertions: we use the solution based on substitution of aliases pioneered by Morris [Mor82] and later used also by America and de Boer [AdB94]. To deal with new aliases that may result from an assignment $x.a := e$, one may simply substitute the potentially aliased expressions $e'.a$ with a test: if the expression e' is an alias of x, then e is substituted to $e'.a$; if not nothing is changed. This results in:

$$e'.a[e/x.a] \triangleq \text{if } e'[e/x.a] = x \text{ then } e \text{ else } e'[e/x.a].a \text{ end}$$

which can be easily translated into the global assertion language. That substitution preserves constants and commutes with other operators (substituting in an operator applied to operands is substituting in the operands followed by application of the operator).

The rules of the cooperation test below are given that allow only one assignment to an auxiliary variable. This simplifies the writing of the rules but it is not a formal prerequisite for soundness or completeness of the proof system. One may freely use the obvious generalization of the system that allow to the assignment of many auxiliary variables in the same bracketed section.

We first introduce the cooperation tests for synchronous and asynchronous routine calls. They are both quite simple and they faithfully reflect the operation semantics of the same constructions in Section 3, which are respectively CALL_{sync} and RETURN_{sync} for the synchronous call, and CALL_{sync}, RETURN_{sync} and ACCEPT_{async} for the asynchronous call. Those tests involve an extra invariant GI_{ccr} that depends on the conditional critical regions present in the program. They will be defined later in this section.

The cooperation test for synchronous routine call is similar to the one for CSP seen before. However, the difference between syntactic presentation of routines and message receptions implies that the state of the supplier object is not known precisely at the time of the routine call, whereas it is known in CSP (it respects p for a reception instruction $P?x$ with proof outline $\{p\}P?x\{q\}$). In the case of a routine, one could suggest to use an invariant for the supplier object. Such an invariant would have to be true between any two routine calls to that object. However, by always being true before a call, it would be of no use to distinguish semantically matching from semantically mismatching calls. Therefore, we do not use such an invariant. When a call is made, we consider two objects denoted by new variables x and y with non-void values, and enforce that y be the supplier for the call of x (by $b{\downarrow}x = y$), and that the antecedent for the call in the proof outline be true ($p_A{\downarrow}x$). In such a case, the execution of the calling instruction should maintain the invariant

I_{sync} stating that y is locked by x ($y.lockedBy = \langle x, i, j\rangle$), and that there are no pending asynchronous routine calls ($\forall y' : y' \notin dom(x.queue) \vee x.queue(y') \doteq \star$). Immediately after the computation step, the global invariant should be true and so must be the assertion p_B placed at the beginning of the routine, in an environment where the arguments have replaced the formal parameters and the supplier object is now active ($\neg y.idle$) while its *smode* attribute is set to true. Please note that in such a rule, free logical variables are implicitly universally quantified; it follows from the definition of satisfaction in Section 4.2.

Definition 4.1 (Cooperation Test: Synchronous Routine Call).
Let $b_1, \ldots, b_k$ be attributes of respective types $B_1, \cdots, B_k$ in a class A of client objects. In a CCR locking those attributes, with $\tilde{b'} \subseteq \tilde{b}$ and $b \in \tilde{b}$, for a routine call

$$\{p_A\}c := b.m(\tilde{b'})\{q_A\}$$

that syntactically matches routine definition

$$\langle m(\tilde{b'} : \tilde{B'})\ \texttt{is do}\ u := exp\rangle\{p_B\}S\{q_B\}\langle v := exp'; \texttt{return}\ exp''\rangle$$

in a class B of supplier objects. The proof outlines of A and B cooperate if for some new logical variables x and y of types A and B we have

$\models_g \ GI \wedge GI_{ccr} \wedge I_{sync} \wedge x \neq Void \wedge y \neq Void \wedge p_A{\downarrow}x \wedge b{\downarrow}x = y$
$\Rightarrow \ (GI \wedge GI_{ccr} \wedge I_{sync} \wedge p_B{\downarrow}y)[exp{\downarrow}y, \#t/y.u, y.smode][\tilde{b'}{\downarrow}x/y.\tilde{b'}]$

and

$\models_g \ GI \wedge GI_{ccr} \wedge I_{sync} \wedge x \neq Void \wedge y \neq Void \wedge q_B{\downarrow}y \wedge b{\downarrow}x = y$
$\Rightarrow \ (GI \wedge GI_{ccr} \wedge I_{sync} \wedge q_A{\downarrow}x)[exp'{\downarrow}y, \#f, exp''{\downarrow}y/y.v, y.smode, x.c]$

for $I_{sync} \triangleq y.lockedBy = (x, i, j) \wedge (\forall y' : y' \notin dom(x.queue) \vee x.queue(y') \doteq \star)$.

A synchronous routine call maintains both the *lockedBy* and the *queue* attributes of the client object. Asynchronous routine calls modify the queues of the supplier objects as expected. Hence only the value of the *lockedBy* auxiliary attribute is kept invariant, as reflected by assertion I_{async} used in the rule. In the first step, the asynchronous emission of the message, the invariant should be maintained while the *queue* attribute of the sender is updated with the name of the routine to be called and the values of the effective arguments. Since the call is asynchronous, the post-condition q_A of the calling instruction should be also true. The sent message may then be handled by the supplier y if it progresses to the head of the queue of incoming messages. This assertion is verified in the second clause by taking the sequence of sent messages and retrieving from it the sequence of already received messages. The head message is put in $y.in$ with a special tag †. This allows one to prove the correctness of the behavior of the called routine while assuming that messages are treated one at a time (there is no intra-object concurrency in SCOOP). This is enforced because $y.in$ will never be a proper prefix of $x.queue(y)$ when $y.in$ contains a tagged message, preventing further deliveries before the end of the execution of the routine. The correctness of this termination is given by the third satisfaction condition. Termination in this case only amounts to taking the tag off the message in the $y.in$ sequence, and modifying the auxiliary variable updated in the bracketed section.

Definition 4.2 (Cooperation Test: Asynchronous Routine Call).
Let $b_1, \ldots, b_k$ be attributes of respective types $B_1, \cdots, B_k$ in a class A of client objects. In a CCR locking those attributes, with $\tilde{b'} \subseteq \tilde{b}$ and $b \in \tilde{b}$, for a routine call

$$\{p_A\} b.m(\tilde{b'}) \{q_A\}$$

that syntactically matches routine definition

$$\langle m(\tilde{b'} : \tilde{B'}) \texttt{ is do } u := exp \rangle \{p_B\} S \{q_B\} \langle v := exp'; \texttt{return} \rangle$$

in a class B of supplier objects. The proof outlines of A and B cooperate if for some new logical variables x and y of types A and B we have

$\models_g \quad GI \wedge I_{async} \wedge x \neq Void \wedge y \neq Void \wedge p_A \downarrow x \wedge b \downarrow x = y$
$\Rightarrow \quad (GI \wedge I_{async} \wedge q_A \downarrow x)[x.queue\{y \mapsto x.queue(y) \oplus m(\tilde{b'} \downarrow x)\}/x.queue]$
and
$\models_g \quad GI \wedge I_{async} \wedge x \neq Void \wedge y \neq Void \wedge x.queue(y) \doteq (y.in \oplus m(\tilde{z})) \cdot l$
$\Rightarrow \quad (GI \wedge I_{async} \wedge p_B \downarrow y)[exp \downarrow y, y.in \oplus m(\tilde{z})^\dagger / y.u, y.in][\tilde{z}/y.\tilde{z}]$
and
$\models_g \quad GI \wedge I_{async} \wedge x \neq Void \wedge y \neq Void \wedge q_B \downarrow y \wedge y.in = l \oplus m(\tilde{z})^\dagger$
$\Rightarrow \quad (GI \wedge I_{async})[exp' \downarrow y, l \oplus m(\tilde{z})/y.v, y.in]$

where $I_{async} \stackrel{\Delta}{=} y.lockedBy = \langle x, i, j \rangle \wedge GI_{ccr}$.

We now present the obligations given by the cooperation test for conditional critical regions. There are three of them. Each of them corresponds to the observation of one atomic computation step that has an operational semantics correspondence on Figure 2: locking the objects corresponds to axiom LOCK, unlocking the objects (by placing the termination mark $\diamond$ in their message queues) corresponds to axiom UNLOCK, and effectively liberating the objects from the lock corresponds to axiom RETURN_{lock}.

The first condition of Definition 4.3 requires that for a set of existing objects that are not locked (denoted using the special value $\perp$ for *lockedBy*), when the wait condition is true and the pre-condition for the proof outline of the CCR is respected, then the global invariant should be true along with the intermediate assertion p_2 in the modified environment. That modified environment consists in several assignments to variables. The special substitution operator is used to deal with aliasing. Are updated: the attribute u of current object x (assigned in the bracketed section), the list *locked* of x to which is added an item giving all necessary information about the present lock (the attribute *lockedBy* of x along with the calling mode of the current routine, number of the CCR i and its current index $ccr@i$, and the identities of the locked objects).

Definition 4.3 (Cooperation Test: CCR).
In a class A with attributes $b_1, \cdots, b_k$ belonging to classes $B_1, \cdots, B_k$, then for a CCR numbered i_{ccr} written

$\{p_1\} \quad \langle \texttt{with } b_1, \cdots, b_k \texttt{ when } exp \texttt{ do } u := exp' \rangle$
$\{p_2\} \quad stm$
$\{p_3\} \quad \langle v := exp'' \texttt{end} \rangle$
$\{p_4\}$

the proof outlines of A and $B_1, \cdots, B_k$ cooperate if for some new logical variables $x, y_1, \ldots, y_k$ of respective types $A, B_1, \cdots, B_k$ we have:

$$
\begin{array}{ll}
\models_g & GI \wedge x \neq Void \wedge y_1 \neq Void \wedge \ldots \wedge y_k \neq Void \wedge x.b_1 = y_1 \wedge \ldots \wedge x.b_k = y_k \wedge \\
& y_1.lockedBy = \bot \wedge \ldots \wedge y_k.lockedBy = \bot \wedge exp{\downarrow}x \wedge p_1{\downarrow}x \wedge GI_{ccr} \\
\Rightarrow & (GI \wedge GI_{ccr} \wedge p_2{\downarrow}x)[\ x.locked \oplus \langle\langle x.smode, x.lockedBy\rangle, i_{ccr}, x.ccr@i, y_1, \ldots, y_k\rangle, \\
& \qquad\qquad \langle x, i_{ccr}, x.ccr@i_{ccr}\rangle, \ldots, \langle x, i_{ccr}, x.ccr@i_{ccr}\rangle, exp'{\downarrow}x\ / \\
& \qquad\qquad x.locked, y_1.lockedBy, \ldots, y_k.lockedBy, x.u]
\end{array}
$$

and

$$
\begin{array}{ll}
\models_g & GI \wedge x \neq Void \wedge p_3{\downarrow}x \wedge \langle\langle \mu, x', i', j'\rangle, i_{ccr}, j, y_1, \ldots, y_k\rangle \in x.locked \wedge I_{ccr} \\
\Rightarrow & (GI \wedge I_{ccr} \wedge p_4{\downarrow}x)[\ exp'{\downarrow}x, x.ccr@i_{ccr} + 1, \\
& \qquad\qquad x.queue\{y_1 \mapsto x.queue(y_1) \oplus \diamond, \ldots, y_k \mapsto x.queue(y_k) \oplus \diamond\}\ / \\
& \qquad\qquad x.v, x.ccr@i_{ccr}, x.queue]
\end{array}
$$

and

$$
\begin{array}{ll}
\models_g & GI \wedge x \neq Void \wedge x.locked \doteq l_1 \cdot (\langle\langle \mu, x', i', j'\rangle, i_{ccr}, j, y_1, \ldots, y_k\rangle \oplus l_2) \wedge \\
& x.queue(y_1) \doteq y_1.in \oplus \diamond \wedge \ldots \wedge x.queue(y_k) \doteq y_k.in \oplus \diamond \wedge GI_{ccr} \\
\Rightarrow & (GI \wedge GI_{ccr})[\ l_1 \cdot l_2, x.queue\{y_1 \mapsto \star, \ldots, y_k \mapsto \star\}, \star, \ldots, \star, \bot, \cdots, \bot\ / \\
& \qquad\qquad x.locked, x.queue, y_1.in, \ldots, y_k.in, y_1.lockedBy, \cdots, y_k.lockedBy]
\end{array}
$$

where $I_{ccr} \stackrel{\Delta}{=} y_1.lockedBy = \langle x, i_{ccr}, j\rangle \wedge \cdots \wedge y_k.lockedBy = \langle x, i_{ccr}, j\rangle \wedge GI_{ccr}$.

The second condition to fulfill concerns the execution of the statement releasing the lock: the `end` statement. Using assertion p_3, one should prove that putting the termination sign $\diamond$ in the queues of the locked objects will maintain the global invariant and will not change the set of locked objects. Please note that the precise state of the locked object is not known, since this can happen at any moment after the execution of all the asynchronous calls present in the body of the CCR.

The third condition corresponds to the effective release of the objects. There is one major difference with the operational semantics presented in Section 3.2: here, the state of the client object is modified when performing the action. This can be explained by the fact that we need some auxiliary attribute (*locked*) to be placed on the client object, and not on the supplier objects. After the execution all the queues of the locked objects are empty, and both *queue* and *in* attributes are reset accordingly.

In the conditions above, we used an invariant GI_{ccr} which definition follows.

Definition 4.4 (CCR Global Invariant).

The CCR global invariant GI_{ccr} implies, for each CCR number i in a class A locking attributes $a_1, \cdots, a_n$ of type $A_1, \cdots A_n$, that for x of type A and $x_1, \cdots, x_n$ of type $A_1, \cdots A_n$ we have

$$
\begin{array}{l}
x \neq Void \\
\Rightarrow \left(\begin{array}{l}
\left(\begin{array}{l}
(x_1 \neq Void \wedge x_1.lockedBy = \langle x, i_{ccr}, j \rangle) \Rightarrow \exists x_2, \ldots, x_k : \\
\left(\begin{array}{l}
x_2.lockedBy = \langle x, i_{ccr}, j \rangle \wedge \ldots \wedge x_k.lockedBy = \langle x, i_{ccr}, j \rangle \wedge \\
\exists x', \mu, i', j' : (\langle \langle x', \mu, i', j' \rangle, i_{ccr}, j, x_1, \ldots, x_k \rangle \in x.locked \wedge \\
\qquad \mu \Rightarrow x.lockedBy = \langle x', i', j' \rangle)
\end{array} \right) \\
\quad \wedge \ldots \wedge \\
(x_k \neq Void \wedge x_k.lockedBy = \langle x, i_{ccr}, j \rangle) \Rightarrow \exists x_1, \ldots, x_{k-1} : \\
\left(\begin{array}{l}
x_1.lockedBy = \langle x, i_{ccr}, j \rangle \wedge \ldots \wedge x_{k-1}.lockedBy = \langle x, i_{ccr}, j \rangle \wedge \\
\exists x', \mu, i', j' : (\langle \langle x', \mu, i', j' \rangle, i_{ccr}, j, x_1, \ldots, x_k \rangle \in x.locked \wedge \\
\qquad (\mu \Rightarrow x.lockedBy = \langle x', i', j' \rangle))
\end{array} \right)
\end{array} \right) \\
\wedge \\
(\exists x', \mu, j, i', j', x_1, \ldots, x_k : \langle \langle x', \mu, i', j' \rangle, i_{ccr}, j, x_1, \ldots, x_k \rangle \in x.locked \Rightarrow \\
(x_1.lockedBy = \langle x, i_{ccr}, j \rangle \wedge \cdots \wedge x_k.lockedBy = \langle x, i_{ccr}, j \rangle))
\end{array} \right)
\end{array}
$$

Without GI_{ccr}, the proof system would be neither sound nor complete (see Section 4.5). It can be seen as enforcing the consistency of the variables that describe the effects of locking and liberation of objects. It allows one to infer, from the state of a given object, the complete locking structure that has yielded this object to be locked itself. On the other hand, it impeaches the global invariant to imply any false assertion about this structure. For example, on Figure 4, when routine is called `n` on object 6 one should be able to deduce that objects 1 to 5 are locked, with the precise relationship depicted on the figure. Similarly for object 10, that upon executing `p` should be able to infer that object 7 has locked it, but nothing else about the other objects, since the routine of 7 calling 10 has itself been called asynchronously and the objects "higher" in the tree can therefore be unlocked without giving any further notice to 7 or 10.

This obligation can also be seen as imposing the consistency on the values of attributes *locked* and *lockedBy*. It says that, knowing that some object x_i is locked by some x using CCR number i with index j, one should be able to find the set of objects locked during the execution of the very same CCR. The knowledge of such set propagates recursively up the tree, from the clause $\mu \Rightarrow x.lockedBy = \langle x', i', j' \rangle$.

We now address the issue of object creation. The essential problem in the formulation of the corresponding rule is that all the quantifications have to range over the whole set of existing objects, including the newly created object. The assertions that bear on the global state *before* the creation of that object must therefore be adapted so that the new object is excluded from the range of their occurring quantifications. Hence we define $P \backslash l$ that replaces quantifications $\exists x : P'$ occurring in P by $\exists x : (\exists l_1, l_2 : l \doteq (l_1 \cdot (x \oplus l_2))) \wedge P' \backslash l$.

Definition 4.5 (Cooperation Test: Object Creation). *For a new statement $\{p\}\langle b := new\ B; u := exp \rangle\{q\}$ in a routine of a class A then the proof outline cooperates if there exists a new variable x of type A such that*

$$
\begin{array}{rl}
\models_g & \exists y^B, z^B, l^{\texttt{list}\ ANY} : x \neq Void \wedge y \neq Void \wedge Init(y) \wedge y \notin l \wedge \\
& (p \downarrow x)[z/x.a] \wedge (\forall y_1^B : y_1 \in l \vee y_1 = y) \wedge (GI \wedge GI_{ccr}) \backslash x \\
\Rightarrow & (GI \wedge GI_{ccr} \wedge q \downarrow x)[exp \downarrow x / x.u]
\end{array}
$$

where the definition $Init(x)$ is syntactical, requiring for any new object x of class C

that its attributes are set to ff, 0, $\star$ and Void in that order for booleans, integers, lists, and references to object. This can be generalized to record types in the usual way.

The initial correctness property states that there is only one instance of the root class (the so-called root object) in the initial configuration and no instance of any other class exists, and that the global invariant is satisfied. The final correctness condition says that the invariant is still respected when the creation routine of the root object terminates. The root object then becomes idle and available for incoming routine calls. It is required to do a separate check in those cases because there is, by definition, no explicit call to the creation procedure in the program itself, and other checks imposed by the cooperation test will not encompass this case. The creation method is assumed to be called by a virtual CCR numbered 0. To simplify reasoning, verification conditions check the combined effects of both locking (and unlocking) of the root object and call (and termination of the call) of the creation routine. Hence, the conditions require that the global and CCR invariants have to be true in an environment where both *locked* and *lockedBy* auxiliary attributes of the root object are modified compared to the previous configuration.

Definition 4.6 (Initial and Final Correctness). *A global invariant is* initially correct *for a root class* `A` *featuring creation procedure*

$$\langle make(\tilde{a} : \tilde{T})\ \texttt{is do}\ u := exp\rangle\{p\}S\{q\}\langle v := exp'; \texttt{return}\rangle$$

with T either a boolean, an integer, or a type constructed using them, if for a new logical variable x of class `A` *then*

$$\begin{array}{ll} \models_g & x \neq Void \wedge Init(x) \wedge (\forall y^{ANY} : x = y) \\ \Rightarrow & (GI \wedge GI_{ccr} \wedge p{\downarrow}x)[\langle x, 0, 0\rangle, \langle\langle x, \#, 0, 0\rangle, 0, 0, x\rangle, exp{\downarrow}x/x.lockedBy, x.locked, x.u] \end{array}$$

and

$$\begin{array}{l} \models_g x \neq Void \wedge GI \wedge GI_{ccr} \wedge x.locked \doteq l_1 \cdot (\langle\langle \mu, x, 0, 0\rangle, 0, 0, x\rangle \oplus l_2) \wedge q{\downarrow}x \wedge \\ \quad x.lockedBy = \langle x, 0, 0\rangle \\ \Rightarrow (GI \wedge GI_{ccr})[exp'{\downarrow}x, ff, l_1 \cdot l_2, \perp / x.v, x.smode, x.locked, x.lockedBy] \end{array}$$

4.4 An Example

We present a very simple example. The program of Figure 6 is written using an Eiffel/SCOOP-like syntax; the translation into the Algol-like syntax we used in the proof system is straightforward. The program consists in two classes `A` and `B`, `A` being the root class. When an instance α of `A` is created using routine `make`, it creates two instances β and γ of `B` and calls routine `m`, respectively `n`, on them. Sicne these calls are asynchronous, α can become idle, while `m` and `n` try to lock it back and call its method `set_v`.

One may prove that routine `set_v` of α is never executed when called in routine `m` by object β because the wait condition of routine `m` requiring $\alpha.v = 4$ will never be true. Since this routine invocation is the only one that may set attribute `v` of `a`

to 3, we obtain a provisory and minimalistic version of the global invariant to prove, written $GI' \triangleq \forall x^A : x.v \neq 3$. This is true, intuitively because there is no other call to `a` that can set `v` to 4. On the example there is no need to reason on the identity of objects, because only one instance of `A` is created and, although two instances of `B` are created, the proof is the same as if there were only one instance. However we claim that this example, although simplistic in nature, shows the essence of the use of the cooperation test in our proof system, by reasoning with the cooperation test for CCR and using the wait condition to prove the impossibility of execution of a call.

We will number the conditional critical regions appearing in the program as follows: 1 for the CCR in routine `make` of class `A`, 2 and 3 for the ones appearing in routines `m` and `n` of class `B`.

```
class A
create make
feature {NONE}
    b1, b2: separate B
    v : INTEGER
feature
    make is
    do
        create b1.make(Current)
        create b2.make(Current)
        with b1, b2 when true
        do -- CCR number 1
            b1.m
            b2.n
        end
    end

    set_v (i : INTEGER) is
    do
        v := i
    end
end -- class A
```

```
class B
create make
feature
    a : separate A
    make (p: separate A) is
    do
        a := p
    end

    m is do
        with a when a.v = 4
        do -- CCR number 2
            a.set_v(3)
        end
    end

    n is
    do
        with a when true
        do -- CCR number 3
            a.set_v(2)
        end
    end
end -- class B
```

Figure 6: A Simple Example Program

To be able to use the cooperation test, we need to produce a proof outline of the program. As attribute `v` of class `A` appears in the invariant, we make it an auxiliary attribute and allow updating its value only inside bracketed sections. Those bracketed sections are placed as on Figure 7. In the proof outline, all object creations, conditional critical regions, routine invocations, and routine definitions are bracketed. We do not include the declarations of the pre-defined auxiliary variables

that would be added to any class, as described in Section 4.3.3.

The proof outline is almost trivial, containing mainly `true` assertions. The only assertions that are not trivially true are the ones in routine `m` of class `B`. Indeed the only way, for the call to `set_v` within the body of `m`, to pass the cooperation test is to prove that it will never be executed. We need to make this fact explicit locally in the proof outline of `B`. For this we use an auxiliary variable `w`, that we set to true when the execution of the body of the CCR begins. The link between `w` and the value of `a.v` for an instance of class `B` is easy to establish:

$$\forall x^A, y^B : (y.a = x \wedge y.w) \Rightarrow x.v = 4$$

which we can deem to be trivially met by introducing the conjunct

$$\forall x^A : x.v \neq 4$$

in the global invariant. We can then also conjunct with it $\forall y^B : \neg y.w$ in order to let the cooperation test for the call to `set_v` in `m` become trivial. But the essential fact that we need to prove in order to establish that `v` is never set to 3 is that a message `set_v` containing 3 or 4 is never sent to an instance of class `A`; that `v` is different from 3 and 4 can actually be inferred easily from this fact. This can be written

$$\forall x^A, y^B, i, l, l' : y.queue(x) = (l \oplus set\_v(i)) \cdot l' \Rightarrow (i \neq 3 \wedge i \neq 4)$$

and we obtain the global invariant

$$\begin{array}{rcl} GI & \triangleq & \forall x^A : x.v \neq 3 \wedge x.v \neq 4 \wedge \\ & & (\forall y^B, i, l, l' : \neg y.w \wedge (y.queue(x) = (l \oplus set\_v(i)) \cdot l' \Rightarrow (i \neq 3 \wedge i \neq 4))) \end{array}$$

Initial correctness is trivially true: there is only one instance of class A, which attribute `v` has value 0, thus implying GI. Similarly, final correctness is easily verified, since the bracketed assignments in routine `make` of class `A` do not change the value of `v`, while GI_{ccr} remains true because attributes *locked* and *lockedBy* are modified consistently with one another.

The cooperation test for the routine calls and conditional critical regions appearing in class A, passing the cooperation test is easy. For the bracketed sections in routine `make` this result is indeed immediate, since bracketed sections there modify neither the auxiliary variable `v` nor the output queue `queue` of an object of class `B`, and thus can not change the validity of GI. Routine `set_v` itself does not contain any routine call or CCR, so the cooperation test is passed.

We now turn to bracketed sections of routines `m` and `n` of class `B`. They are more interesting than the ones in class `A` because they include conditional critical regions locking an instance of `A`, and calls to routine `set_v` on such instances.

The cooperation test for the CCR appearing in routine `m` is passed because the supplier object attached to attribute `a` of the client will never let the wait condition of the CCR become true. Using new variables z_c of class B and z_s of class A, the first condition to fulfill is:

$$\begin{array}{rl} \models_g & GI \wedge z_c \neq Void \wedge z_s \neq Void \wedge z_c.a = z_s \wedge z_s.lockedBy = \perp \wedge z_c.a.v = 4 \wedge GI_{ccr} \\ \Rightarrow & (GI \wedge z_c.w \wedge GI_{ccr})[\ \#t, z_c.locked \oplus \langle\langle z_c.smode, z_c.lockedBy\rangle, 2, z_c.ccr@2, z_s\rangle, \\ & \qquad \langle z_c, 2, z_c.ccr@2\rangle / z_c.w, z_c.locked, z_s.lockedBy] \end{array}$$

```
class A
create make
feature {NONE}
   b1, b2: separate B
   v : INTEGER
   -- list of pre-defined
   -- auxiliary variables
feature
  <make is
   do>
      {true}
     <create b.make(Current)>
      {true}
     <create c.make(Current)>
      {true}
     <with b, c when true
      do >
         {true}
        <b.m>
         {true}
        <c.n>
         {true}
     <end>
      {true}
     <return>
   end

  <set_v (k : INTEGER) is
   do
      v := k>
      {v /= 3}
     <return>
   end
end -- class A
```

```
class B
create make
feature
   a : separate A
   -- list of pre-defined
   -- auxiliary variables
   w : BOOLEAN
  <make (p: separate A) is
   do>
      {true}
      a := p
      {true}
   end

  <m is do>
      {true}
     <with a when a.v = 4
      do
         w := true>
         {w}
        <a.set_v(3)>
         {w}
     <end>
      {true}
     <return>
   end

  <n is do >
      {true}
     <with a when true
      do
         {true}
        <a.set_v(2)>
         {true}
     <end>
     {true}
     <return>
   end
end -- class B
```

Figure 7: Proof Outline for the Example Program

$$
\begin{aligned}
GI_{ccr} \triangleq\ & \left(\begin{array}{l}
\forall x^A, j, y^B : y.lockedBy = \langle x, 1, j\rangle \Rightarrow \\
\left(\begin{array}{l}
\exists z^B, x'^{ANY}, \mu, i', j' : z.lockedBy = \langle x, 1, j\rangle \wedge \\
\langle\langle x', \mu, i', j'\rangle, 1, j, y, z\rangle \in x.locked \wedge (\mu \Rightarrow x.lockedBy = \langle x', i', j'\rangle)
\end{array}\right) \\
\wedge \\
\forall x^A, j : (\exists x', \mu, j, i', j', y^B, z^B : \langle\langle x', \mu, i', j'\rangle, 1, j, y, z\rangle \in x.locked \Rightarrow \\
(y.lockedBy = \langle x, 1, j\rangle \wedge z.lockedBy = \langle x, 1, j\rangle)
\end{array}\right) \\
\wedge\ & \left(\begin{array}{l}
\forall y^B, j, x^A : x.lockedBy = \langle y, 2, j\rangle \Rightarrow \\
\left(\begin{array}{l}
\exists x'^{ANY}, \mu, i', j' : \langle\langle x', \mu, i', j'\rangle, 2, j, x\rangle \in y.locked \wedge \\
(\mu \Rightarrow y.lockedBy = \langle x', i', j'\rangle)
\end{array}\right) \\
\wedge \\
\forall y^B, j : (\exists x', \mu, j, i', j', x^A : \langle\langle x', \mu, i', j'\rangle, 2, j, x\rangle \in y.locked) \Rightarrow \\
\big(x.lockedBy = \langle y, 2, j\rangle\big)
\end{array}\right) \\
\wedge\ & \left(\begin{array}{l}
\forall z^B, j, x^A : x.lockedBy = \langle z, 3, j\rangle \Rightarrow \\
\left(\begin{array}{l}
\exists x'^{ANY}, \mu, i', j' : \langle\langle x', \mu, i', j'\rangle, 3, j, x\rangle \in z.locked \wedge \\
(\mu \Rightarrow z.lockedBy = \langle x', i', j'\rangle)
\end{array}\right) \\
\wedge \\
\forall z^B, j : (\exists x', \mu, j, i', j', x^A : \langle\langle x', \mu, i', j'\rangle, 3, j, x\rangle \in z.locked) \Rightarrow \\
\big(x.lockedBy = \langle z, 3, j\rangle\big)
\end{array}\right)
\end{aligned}
$$

Figure 8: The Global CCR Invariant GI_{ccr} for the Example

This condition is trivially true since $GI \Rightarrow \forall x^A : x.v \neq 4$ and thus $z_c.a.v \neq 4$. The second condition is verified for similar reasons:

$$\begin{array}{ll} \models_g & GI \wedge z_c \neq Void \wedge z_c.w \wedge \langle\langle \mu, x'_c, i', j' \rangle, 2, j, z_s \rangle \in z_c.locked \wedge I_{ccr} \\ \Rightarrow & (GI \wedge I_{ccr})[z_c.ccr@i + 1, z_c.queue\{z_s \mapsto z_c.queue(z_s) \oplus \diamond\} \ / z_c.ccr@i, z_c.queue] \end{array}$$

because $GI \Rightarrow \forall y^B : \neg y.w$ and thus $\neg z_c.w$. The third condition, written

$$\begin{array}{l} \models_g GI \wedge GI_{ccr} \wedge z_c \neq Void \wedge z_c.locked \doteq l_1 \cdot (\langle\langle \mu, x'_c, i', j' \rangle, 2, j, z_s \rangle \oplus l_2) \wedge \\ \quad z_c.queue(z_s) \doteq z_s.in \oplus \diamond \\ \Rightarrow (GI \wedge GI_{ccr})[\ l_1 \cdot l_2, x_c.queue\{z_s \mapsto \star\}, \star, \bot \ / \\ \qquad\qquad z_c.locked, z_c.queue, z_s.in, z_s.lockedBy] \end{array}$$

modifies the queue of variable x_c of type `B` by setting it to the empty queue. It thus makes the corresponding part of the invariant

$$\forall x^A, y^B, i, l, l' : y.queue(x) = (l \oplus set\_v(i)) \cdot l' \Rightarrow (i \neq 3 \wedge i \neq 4)$$

trivially true, while the rest of the GI remains unchanged. The proof that GI_{ccr} remains invariant in the new environment comes from the fact that the `locked` and `lockedBy` auxiliary attributes of the locker and locked objects are modified consistently (and atomically).

The emission of the call to `set_v` in method `m` is a semantic mismatch since we just showed that the CCR enclosing it will not execute its body. However, since there is a syntactical match between this call and routine `set_v`, we need to prove the success of the cooperation test. Since the call is asynchronous, there are three steps to verify: the sending of the message, the beginning of the treatment of this message by the supplier object, and the end of the treatment.

The cooperation test for the emission of the message for two variables z_s and z_c of classes `A` (resp. `B`) is expressed as:

$$\begin{array}{ll} \models_g & GI \wedge I_{async} \wedge z_c \neq Void \wedge z_s \neq Void \wedge z_c.w \wedge z_c.a = z_s \\ \Rightarrow & (GI \wedge I_{async} \wedge z_c.w)[z_c.queue\{z_s \mapsto z_c.queue(z_s) \oplus set\_v(3)\}/z_c.queue] \end{array}$$

and it is true because $GI \Rightarrow \forall y^B : \neg y.w$.

To prove cooperation of the delivery of the message to the supplier object, we can not rely on the proof outline of the client as we did until now. Indeed, the proof outline of the caller does not appear in this case of the cooperation test; the only element known at the time of the delivery is that there is a message in the sending queue of the client with certain effective arguments. Therefore, the global invariant thus has to contain information about those arguments in order to establish that the call is semantically mismatching:

$$\begin{array}{ll} \models_g & GI \wedge I_{async} \wedge z_c \neq Void \wedge z_s \neq Void \wedge z_c.queue(z_s) \doteq (z_s.in \oplus set\_v(j)) \cdot l \\ \Rightarrow & (GI \wedge I_{async} \wedge z_s.v \neq 3)[z_s.k, z_s.in \oplus set\_v(j)^\dagger / z_s.v, z_s.in][j/z_s.k] \end{array}$$

for which the sub-formula on the right-hand side of the implication expands to:

$$\begin{pmatrix} z_s.v \neq 3 \wedge I_{async} \wedge \forall x^A : x.v \neq 3 \wedge x.v \neq 4 \ \wedge (\forall y^B, i, l, l' : \\ \neg y.w \wedge (y.queue(x) = (l \oplus set\_v(i)) \cdot l' \Rightarrow (i \neq 3 \wedge i \neq 4))) \end{pmatrix} [\ldots][j/z_s.k]$$

on which we can apply substitution, obtaining:

$$\left(\begin{array}{c}(\text{if } z_s = z_s \text{ then } z_s.k \text{ else } z_s.v) \neq 3 \wedge I_{async} \wedge \\ \forall x^A : (\text{if } x = z_s \text{ then } z_s.k \text{ else } x.v) \neq 3 \wedge (\text{if } x = z_s \text{ then } z_s.k \text{ else } x.v) \neq 4) \wedge \\ (\forall y^B, i, l, l' : \neg y.w \wedge (y.queue(x) = (l \oplus set\_v(i)) \cdot l' \Rightarrow (i \neq 3 \wedge i \neq 4)))\end{array}\right)[j/z_s.k]$$

and again after simplification:

$$\left(\begin{array}{c}j \neq 3 \wedge I_{async} \wedge \\ \forall x^A : (\text{if } x = z_s \text{ then } j \text{ else } x.v) \neq 3 \wedge (\text{if } x = z_s \text{ then } j \text{ else } x.v) \neq 4 \wedge \\ (\forall y^B, i, l, l' : \neg y.w \wedge (y.queue(x) = (l \oplus set\_v(i)) \cdot l' \Rightarrow (i \neq 3 \wedge i \neq 4)))\end{array}\right)$$

which by case splitting on the possible value of x, is equivalent to:

$$\left(\begin{array}{c}j \neq 3 \wedge j \neq 4 \wedge I_{async} \wedge \\ \forall x^A : (x \neq z_s \Rightarrow (x.v \neq 3 \wedge x.v \neq 4)) \wedge (\forall y^B, i, l, l' : \\ \neg y.w \wedge (y.queue(x) = (l \oplus set\_v(i)) \cdot l' \Rightarrow (i \neq 3 \wedge i \neq 4)))\end{array}\right)$$

Through this development we exemplified the work of the substitution operators. The above assertion is easily implied by the antecedent of the rule that expands to:

$$\begin{array}{l}I_{async} \wedge z_c \neq Void \wedge z_s \neq Void \wedge z_c.queue(z_s) \doteq (z_s.in \oplus set\_v(j)) \cdot l \\ \forall x^A : x.v \neq 3 \wedge x.v \neq 4 \wedge \\ (\forall y^B, i, l, l' : \neg y.w \wedge (y.queue(x) = (l \oplus set\_v(i)) \cdot l' \Rightarrow (i \neq 3 \wedge i \neq 4)))\end{array}$$

The essential point here is that the value j passed as argument to the `set_v` call must be different from 3 and 4 because any instance y of class `B`, including z_c, may contain such calls only with values different from 3 and 4, and thus

$$\forall x^A, y^B, i, l, l' : y.queue(x) = (l \oplus set\_v(i)) \cdot l' \Rightarrow (i \neq 3 \wedge i \neq 4)$$

implies for any z_c and z_s of class `A` and `B` with $\forall j, l : z_c.queue(z_s) \doteq (z_s.in \oplus set\_v(j)) \cdot l$ that $j \neq 3 \wedge j \neq 4$.

the cooperation test for the end of the execution of the routine `set_v` is proved with the help of the proof outline of the supplier object, stating that $z_s.v \neq 3$:

$$\begin{array}{ll}\models_g & GI \wedge I_{async} \wedge z_c \neq Void \wedge z_s \neq Void \wedge z_s.v \neq 3 \wedge z_s.in = l \oplus set\_v(j)^\dagger \\ \Rightarrow & (GI \wedge I_{async})[l \oplus set\_v(j)/z_s.in]\end{array}$$

because GI implies again that any object of class `A` has a value for `v` different from 3, and thus the implication is trivially true.

The checks to be done for the CCR and routine call in routine `n` demand slightly more complicated reasoning since there is semantical match with the routine `set_v`, but they nonetheless preserve the global invariant and the CCR invariant.

4.5 Soundness and Completeness

A proof system is *sound* if any assertion that can be proved by correct application of the deduction rules on a given program implies that this assertion is true in all

reachable states of that program. A proof system is (*relatively*) *complete* if every assertion that is true in all reachable states of a program has a proof that can be written by correctly applying the rules of the system. One speaks about *relative* completeness because such completeness proof can only be done relatively to an underlying logic that is used to encode (at least) the datatypes representing the basic computation structures of the language. Such a logic is usually at least as strong as Peano arithmetic; hence full completeness would hence not be provable.

Definition 4.7 (Satisfaction). *Let φ be a proof outline of a program prog. φ yields for each instruction stm of the program (including bracketed sections) a precondition $pre(stm)$ and a postcondition $post(stm)$. We say that prog* satisfies *the global invariant GI under the proof outline φ, noted $prog \models_\varphi GI$ iff for any configuration*

$$M \equiv \langle \alpha_1, stm_1, \sigma_1, \tau_1, \lambda_1, \iota_1 \rangle \mid \ldots \mid \langle \alpha_k, stm_k, \sigma_k, \tau_k, \lambda_k, \iota_k \rangle$$

reachable from the initial state of the execution of prog, for the environment $\tilde{\sigma}$ deduced from M and for a logical environment ω referring to objects only in $\tilde{\alpha}$, the following holds:

- $\omega, \sigma_i \models_{\mathcal{L}} pre(stm_i)$ *for any* $\langle \alpha_i, stm_i, \sigma_i, \tau_i, \lambda_i, \iota_i \rangle$ *with* $1 \leq i \leq k$, *and*
- $\omega, \tilde{\sigma} \models_{\mathcal{G}} GI$.

Definition 4.8 (Provability).
We say that GI is provable *for prog using proof outline φ, noted $prog \vdash_\varphi GI$ if the proof outlines of classes in φ cooperate to maintain GI invariant.*

Theorem 4.1 (Soundness of the Proof System).
If $prog \vdash_\varphi GI$ then $prog \models_\varphi GI$.

Definition 4.9 (Suppression of Auxiliary Attributes). *For two programs prog and $prog'$ we note $prog \rightsquigarrow prog'$ the fact that $prog'$ is obtained from prog by suppressing the declaration and all references to one or more auxiliary attributes.*

Theorem 4.2 (Completeness of the Proof System).
If $prog \models_\varphi GI$ then there exists a program $prog'$, a proof outline φ' associated with it, and an invariant GI' such that $prog \rightsquigarrow prog'$, $GI' \Rightarrow GI$, and $prog' \vdash_{\varphi'} GI'$.

The soundness and completeness proofs of our system are fairly standard. The soundness proof involves a routine induction on the length of the computation, proving that for any operational semantic rule that may be applied, the verification conditions enforce the respect of the global invariant. Even though the structure of the proof is the same as the ones usually met in the literature, we notice some aspects that differentiate our proof from the one proposed in [AdBSdR03] for Java. Indeed, some mechanisms that have global effects are inherently asynchronous in SCOOP and they do not correspond to any statement in the object whose state they affect. This is for example the case for the release of the locks, that changes the *locked* attribute of the client object without explicitly synchronizing with it at the time of the release. This forbids to synchronously reflect the behavior of all the mechanisms in the language using auxiliary variables, as it is done in [AdBSdR03]

for locking and wait/notify mechanisms. Thus, in our setting, there is no equivalent to lemmas 6 and 7 in [AdBSdR03]. Instead we have to use the global invariant GI_{ccr} in order to achieve a similar effect in such a proof. The second part of the proof is to show that one may take away the auxiliary variables and assignments without modifying the reachable states of the program.

The same issues related to asynchrony and other specificities of SCOOP force us to introduce some ad-hoc techniques in the completeness proof. The general principle of the proof is to annotate the program with the *strongest possible* assertions, uniquely identifying the way each statement in the program can be reached after the execution of another reachable statement. Such *reachability predicates* can then be used to show that whenever a statement is reachable if its class is considered individually, then it will be reachable in a global environment given by all the other classes in the program. Such a proof is called a *merging lemma*, and implies showing that the proof outlines and global invariant described by the reachability predicate actually cooperate.

5 Pragmatic analysis

In this section we provide a *short* and *far from complete* study on the SCOOP programming pragmatics. All the considerations we make here are based on the experience gained when devising the operational semantics and the proof system presented in the paper. An important point is that the given proof system does not consider liveness properties other than deadlock. We anticipate that some points made in this section will probably be slightly modified when liveness is taken into account in further studies.

We started from the hypothesis that, in the process of writing a (SCOOP) program, the programmer performs a reasoning process that is actually very similar to the one needed to write a formal proof of correctness for the same program. The main difference is that the programmer does not write all the details as assertions and deduction rules, and is therefore not sure that the program he or she writes is correct. Taking this hypothesis as a postulate, we claim that programming in SCOOP is not more simple than in any other language featuring concurrency-related primitives. Placing all the languages at the same level may seem to be short-sighted, but we hereby claim that the complexity is not so much in the language, but in the programming activity itself. Several elements come in support to that statement.

The first one is that *no rule such as the one suggested in [Mey97, p. 1024] can actually exist to reason about SCOOP programs.* This rule was written:

$$\frac{\{Inv \wedge \bigwedge_{p \in pre(r)} p\} Body(r) \{Inv \wedge \bigwedge_{q \in post(r)} q\}}{\{Inv \wedge \bigwedge_{p \in NonSep\_pre(r)} p'\} Call(r) \{Inv \wedge \bigwedge_{q \in NonSep\_post(r)} q'\}}$$

where p' and q' are like p and q but the formal parameters for the call have been replaced by the effective arguments, and $NonSep\_pre(r)$ is the set of pre-conditions for the routine r that are not wait conditions (similarly for $NonSep\_post(r)$).

Such a rule can not exist for the simple reason that, as shown in the proof system, preconditions and postconditions of routines are not powerful enough to help significantly when reasoning about concurrent programs (they do not even appear in the rules for method call seen in Section 4.3.6). Indeed, one has to deal with *interferences*, that are (non-deterministic) interleavings of atomic operations that may influence in a drastic way the future of the execution of a program. In SCOOP, objects may not interfere through shared variables but they may interfere by (asynchronously) sending messages to each other. In all the existing proof systems one may find in the literature [dRdBH$^+$01], reasoning about interferences is accomplished by the means of an invariant describing the relations existing among the different concurrent processes that execute in parallel. In order to propose a compositional, syntax-driven rule like the one mentioned above, one has to provide specifications that precisely represent the potential interferences that a given routine may produce during its execution; this specification will take the form of a local invariant (for the processor executing the method) *that should be checked after each instruction*. Such specification would therefore be completely different, and much more complex, than the Hoare-style assertions the rule above proposes to use.

If one had at hand the rule suggested in [Mey97], reasoning about SCOOP programs would indeed be much simpler. However, as shown by our example proof, devising a global invariant and the related proof outlines for classes requires to foresee potentially harmful interferences, and then to determine whether they will effectively occur or not. This, in turn, implies the need to characterize deadlocks in order to avoid exploring useless branches of the proof tree that would lead to the conclusion that the wanted property is violated.

We may doubt that one could qualify such reasonings as "simple". They are similar to what should be done in order to prove Java programs, for example. To see this, it is enough to make a coarse-grain comparison between our proof system and the one presented in [AdBSdR03]. Essentially, the proof system for Java requires to provide a *class invariant* that must be true after the execution of every instruction by an instance of this class. This is needed because Java allows intra-object concurrency, and thus the methods of an object executing at the same time in different threads may interfere through variable-sharing. Class invariants are in this case similar to the invariants introduced by Owicki and Gries [OG76] in their proof system to perform the interference-freedom test. SCOOP does not allow intra-object concurrency, and we hence do not need the interference-freedom test. In a way, intra-object concurrency is replaced by asynchronous message-passing which is not needed in Java. The additional complexity is then evident in the proof system for SCOOP: an asynchronous call requires three conditions to pass the cooperation test, while a synchronous test requires only two. Furthermore, the release of the locks by a CCR is also asynchronous in SCOOP. Asynchrony is difficult to reason about because it is asymmetric: the state of at least one of the interacting parties that was at the origin of the interaction is not known when this interaction occurs. It means that this interaction is under the control of no particular object, and it may be delayed by the execution machine as much as wanted. The number of interleavings one has to check is therefore higher: the interaction can occur at any time after its original triggering. Whether or not one of the two solutions (Java versus

SCOOP) yields actually more difficult proofs is hard to decide. Lacking a better solution, we now postulate that the difficulty in this case is similar.

Evaluating the tradeoffs between the ease of programming and the difficulty in reasoning becomes the essence of the game. At first glance, the proof system for Java given in [AdBSdR03] manages to provide some form of separation between the due interference-freedom and cooperation tests. This is done by imposing syntactic constraints on the programs to be proved. The primitive for synchronized blocks is also prevented from appearing, at the benefit of synchronized methods. The synchronization mechanism of CCR in SCOOP can precisely be seen as an extension of Java's synchronized blocks: in Java only one object `o` can be locked using the primitive "`synchronized(o){ ... }`" while in SCOOP, many objects can be locked *atomically*. Reasoning about the SCOOP CCR primitive in a modular fashion is then quite difficult because locking and unlocking objects typically involve the environment of the objects under scrutiny. For instance, in the example shown on Figure 4, object 6 should be able to infer that objects 1, 2, 3, 4 and 5 are locked, while it may even not have a reference to them. This, together with the fact that lock release is asynchronous, is the rationale for the presence of GI_{ccr} in the proof system. By contrast, nothing similar is needed for reasoning about Java classes in [AdBSdR03]. Recovering a form of modularity would require the to use of more elaborate specifications that involve universal quantification ranging on the objects of the environment, in order to determine which ones are locked and which ones are not.

Following this course of reflexion, one may then wonder about the pragmatic use and motivation of the CCR primitive in SCOOP. Providing a definite answer is not on the agenda of the present paper. Nevertheless, some elements can be given. The wait condition included in the primitive is not needed at a theoretical level: any program that uses wait conditions can be translated into one that does not use any. The translation replaces the CCR by a `while` loop locking the supplier objects, evaluating the wait condition, and executing the body of the CCR if the condition is true. Otherwise, the objects are released and a new iteration of the loop can take place. The essential motivation to keep the CCR structure is to provide a better management of inheritance through the integration with (enclosing) routine interfaces. However, introducing inheritance does not hamper the above translation procedure to be applicable and effective, while the synchronization code is not completely separated from the functional code in the original SCOOP proposal, resulting in inheritance anomalies. This can be seen for example in the "Santa Claus" example appearing in [Hum04].

In a general way, the CCR primitive in SCOOP can also be viewed in the light of the opposition of concurrency and distributed programming. Programming with a CCR-like primitive seems "natural" in a concurrent environment, where processes are tightly coupled and one may require to have an atomic view on the state of many objects. In that case, the ability to lock many objects at once seems useful, being for example at the gist of the simple solution provided to the dining philosophers problem. In distributed, message-passing systems, the observations will be quite different. Indeed, the atomicity of locking implies that a consensus must be realized among objects included in a set that contains at least the locked objects.

Such a consensus has lower bounds in terms of the number of messages that must be exchanged to reach it, that will make a highly efficient implementation impossible. Furthermore, certain problems (such as mutual exclusion and consensus) are known to be insolvable in systems that are purely symmetric. The CCR construction allows the programmer to provide a fully symmetric solution, as in the dining philosophers example. This implies that the underlying execution platform (or compiler) would have to use an algorithm that would decide *how* to introduce some asymmetry. This would be needed to avoid deadlock, among other things. Programmers additionally tend not to rely on such possibility of automatically deciding how to introduce asymmetry in the system, and introduce it themselves. The CCR mechanism locking many objects would probably be forlorn then, as shown for example in alternative solutions to the dining philosophers [Hum04]. A final reason for which programmers would not use the full power of the CCR primitive in the distributed case is that atomic consistency is too strong to achieve in a distributed environment to be efficient, and thus programmers usually choose looser models such as sequential (or even causal) consistency. An illustration of this, among others, is the implementation of the distance vector routing by Emmanuel Python [Pyt04].

The final discussion we shall hold concerns the use of SCOOP for applying the Design-By-Contract methodology in a concurrent setting. Whatever criticisms one may have regarding the programming primitives, SCOOP shall be successful if a programmer may find bugs more easily than when using other programming languages. We claim that the present study is not sufficient to answer this question which requires rather deep exploration of methodological issues. The Design-By-Contract method applied to Eiffel programs leads to an organized form of testing and documentation of the code. It happens that the contractual specifications, appearing in Eiffel in the form of Hoare triples, correspond exactly to the ones needed for compositional reasoning. This is not the case in SCOOP. The specifications, for which we may exhibit a syntactic representation later by devising a compositional proof system, will be very different from the Eiffel-style contracts. They will be expressed in first-order logic, and because of the need for performing non-local reasoning to determine when an object is locked, they will require the use of universal quantification. Furthermore, even in other cases, the compositional specifications of such objects would not be written in an executable form, and thus not suitable for testing. The question is therefore open: can we find a form of contract-like specification that would allow us to document classes and perform unit testing in a guided way so that bugs are more easily found in SCOOP classes than *e.g.* in Java classes? This again requires further investigation, that should follow soon. The question of changing programming primitives if necessary should also be present in order to achieve this goal. This would indeed result in an adaptation of the Design-By-Contract methodology to handle concurrent programs.

6 Conclusion

We have presented two formal semantics for SCOOP that we think form an interesting support to understand and develop the language. Several aspects have however

been taken out from those semantics in order to make the study self-contained. Among them is inheritance, and the use of routines as conditional critical regions, locking the objects that are given to it as arguments and featuring a wait condition. The duel mechanism described in [Mey97] and exceptions are also not represented. This was motivated by the goal to avoid interferences of those phenomena (which would have had a very strong impact) with the proof methodology. We intend to cope with some of those aspects in future works, especially with inheritance.

References

[AdB94] Pierre America and Frank de Boer. Reasoning about dynamically evolving process structures. *Formal Aspects of Computing*, 6(3):269–316, 1994.

[AdBSdR03] Erika Abraham, Frank S. de Boer, Martin Steffen, and Willem-Paul de Roever. A hoare logic for monitors in java. Technical Report TR-ST-03-1, Christian-Albrechts-Universität Zu Kiel, April 20, 2003.

[AFdR80] Krzysztof R. Apt, Nissim Francez, and Willem P. de Roever. A proof system for communicating sequential processes. *ACM Trans. Program. Lang. Syst.*, 2(3):359–385, 1980.

[Ash75] E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10:110–135, 1975.

[BFC95] Peter A. Buhr, Michel Fortier, and Michael H. Coffin. Monitor classification. *ACM Comput. Surv.*, 27(1):63–107, 1995.

[BGL98] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, September 1998.

[dB02] Frank S. de Boer. A hoare logic for dynamic networks of asynchronously communicating deterministic processes. *Theoretical Computer Science*, 274(1):3–41, 2002.

[dRdBH+01] Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods.* Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, November 2001.

[Flo67] R. W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Mathematical aspects of computer science: Proc. American Mathematics Soc. symposia*, volume 19, pages 19–31, Providence RI, 1967. American Mathematical Society.

[GdRR82] Rob Gerth, Willem P. de Roever, and Marly Roncken. Procedures and concurrency: A study in proof. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 132–163. Springer-Verlag, 1982.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[Hum04] Matthias Humbert. Comparative analysis of examples in scoop. Technical report, ETHZ, 2004.

[McH94] Ciaran McHale. *Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance.* PhD thesis, Department of Computer Science, Trinity College, Dublin 2, Ireland, October 1994.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Ed.* Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1997.

[Mor82] Joseph Morris. A general axiom of assignment. In M. Broy and G.Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 25–51. Reidel, 1982.

[OG76] Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976. Papers from the Fifth ACM Symposium on Operating Systems Principles (Univ. Texas, Austin, Tex., 1975).

[Phi00] Michael Philippsen. A survey of concurrent object-oriented languages. *Concurrency: Practice and Experience*, 12(10):917–980, August 2000.

[Pyt04] Emmanuel Python. Distance vector routing using scoop. Technical report, ETHZ, 2004.

[Sek02] Emil Sekerinski. Concurrent object-oriented programs: From specification to code. In *FMCO*, volume 2852 of *LNCS*, pages 403–423. Springer-Verlag, 2002.

[SS84] Richard D. Schlichting and Fred B. Schneider. Using message passing for distributed programming: proof rules and disciplines. *ACM Trans. Program. Lang. Syst.*, 6(3):402–431, 1984.