Universitetet i Oslo
Institutt for informatikk

Creol: A Type-Safe Object-Oriented Model for Distributed Concurrent Systems

Einar B. Johnsen,
Olaf Owe, and
Ingrid Chieh Yu

Research report 327
ISBN 82-7368-281-1

June 2005,
Revised May 2006

To appear in Theoretical Computer Science.

Creol: A Type-Safe Object-Oriented Model for Distributed Concurrent Systems

Einar Broch Johnsen *, Olaf Owe, and Ingrid Chieh Yu

Department of Informatics, University of Oslo, N-0316 Oslo, Norway

Abstract

Object-oriented distributed computing is becoming increasingly important for critical infrastructure in society. In standard object-oriented models, objects synchronize on method calls. These models may be criticized in the distributed setting for their tight coupling of communication and synchronization; network delays and instabilities may locally result in much waiting and even deadlock. The Creol model targets distributed objects by a looser coupling of method calls and synchronization. Asynchronous method calls and high-level local control structures allow local computation to adapt to network instability. Object variables are typed by interfaces, so communication with remote objects is independent from their implementation. The inheritance and subtyping relations are distinct in Creol. Interfaces form a subtype hierarchy, whereas multiple inheritance is used for code reuse at the class level. This paper presents the Creol syntax, operational semantics, and type system. It is shown that runtime type errors do not occur for well-typed programs.

Key words: distributed object-oriented systems, type and effect system, type soundness

1 Introduction

The importance of distributed computing is increasing in society with the emergence of applications for electronic banking, electronic police, medical journaling systems, electronic government, etc. All these applications are critical in the sense that system breakdown may have disastrous consequences. Furthermore, as these distributed applications are nonterminating, they are

* Corresponding author.
Email addresses: einarj@ifi.uio.no (Einar Broch Johnsen), olaf@ifi.uio.no (Olaf Owe), ingridcy@ifi.uio.no (Ingrid Chieh Yu).

therefore best understood in terms of non-functional or structural properties. In order to reason about the non-functional properties of distributed applications, high-level formal models are needed.

It is often claimed that object orientation and distributed systems form a natural match. Object orientation is the leading paradigm for open distributed systems, recommended by the RM-ODP [42]. However, standard object-oriented models do not address the specific challenges of distributed computation. In particular, object interaction by means of (remote) method calls is usually synchronous. In a distributed setting, synchronous communication gives rise to undesired and uncontrolled waiting, and possibly deadlock. In addition, separating execution threads from distributed objects breaks the modularity and encapsulation of object orientation, and leads to a very low-level style of programming. Consequently, we believe that distribution should not be transparent to the programmer as in the RPC model, rather communication in the distributed setting should be explicitly asynchronous. Asynchronous message passing gives better control and efficiency, but does not provide the structure and discipline inherent in method declarations and calls. In particular, it is unclear how to combine message passing with standard notions of inheritance. It is unsettled how asynchronous communication and object orientation should be combined. Intuitive high-level programming constructs are needed to unite object orientation and distribution in a natural way. We propose a model of distributed systems based on concurrent objects communicating by asynchronous method calls.

This paper presents the high-level object-oriented modeling language Creol [46–49], which addresses distributed systems. The language is based on concurrent objects typed by behavioral interfaces, communication by asynchronous method calls, and so-called processor release points. Processor release points support a notion of non-blocking method calls, and allow objects to dynamically change between active and reactive behavior. The model integrates asynchronous communication and multiple inheritance, including method overloading and redefinition. In order to allow flexible reuse of behavior as well as of code, behavior is declared in interfaces while code is declared in classes. Both interfaces and classes are structured by multiple inheritance, but inheritance of code is separated from inheritance of behavior. Consequently, the implementation code of a class may be reused without inheriting the external behavior of the class. Creol has an operational semantics defined in rewriting logic [57], which is executable with Maude [20] and provides an interpreter and analysis platform for system models.

This paper extends previous work on Creol by introducing a nominal type system for Creol programs. It is shown that the execution in objects typed by behavioral interfaces and communicating by means of asynchronous method calls, is type-safe. In particular, method binding always succeeds for well-

typed programs. Behavioral interfaces make all external method calls virtually bound. The typing of mutually dependent interfaces is controlled by a notion of *contract*. Furthermore, it is shown that the language extended with high-level constructs for local control, allowing objects to better adapt to the external nondeterminism of the distributed environment at runtime, remains type-safe. Finally, the full Creol language is considered, with multiple inheritance at the class level and a *pruned binding strategy* for late bound internal method calls. It is shown that executing programs in the full language is type-safe.

Paper overview. The rest of this paper is structured as follows. Section 2 presents behavioral interfaces used to type object variables. Section 3 presents an executable language with asynchronous method calls, its type system, and its operational semantics. The language, type system, and operational semantics are extended in Section 4 with local control structures, and in Section 5 with multiple inheritance and the pruned binding strategy. Section 6 discusses related work and Section 7 concludes the paper.

2 Behavioral Object Interfaces

In object-oriented viewpoint modeling [37,75,78], an object may assume different roles or views, depending on the context of interaction. These roles may be captured by specifications of certain parts of the externally observable behavior of objects. The specification of a role will naturally include both syntactic and semantic information about objects. A *behavioral interface* consists of a set of method names with signatures, and semantic constraints on the use of these methods. Interface inheritance is restricted to a form of behavioral subtyping. An interface may inherit several interfaces, in which case it is extended with their syntactic and semantic requirements.

Object variables (references) are typed by behavioral interfaces. Object variables typed by different interfaces may refer to the same object identifier, corresponding to the different roles the object may assume in different contexts. An object *supports* an interface I if it complies with the role specified in I, in which case it may be referred to by an object variable typed by I. A class *implements* an interface if its object instances support the behavior described by the interface. A class may implement several interfaces. Objects of different classes may support the same interface, corresponding to different implementations of the same behavior. Reasoning control is ensured by substitutability at the level of interfaces: *an object supporting an interface I may be replaced by another object supporting I or a subinterface of I* in a context depending on I, although the latter object may be of another class. This substitutability is reflected in the executable language by the fact that virtual (or late) binding applies to all external method calls, as the runtime class of the called object

is not statically known.

For active objects we may want to restrict access to the methods provided in an interface, to calling objects of a particular interface. This way, the active object may invoke methods of the caller and not only complete invocations of its own methods. Thus callback is supported in the run of a protocol between distributed objects. For this purpose, an interface has a semantic constraint in the form of a so-called *cointerface* [44,45]. The communication environment of an object, as considered through the interface, is restricted to external objects supporting the given cointerface. For some objects no such knowledge is required. In this case the cointerface is *Any*, the superinterface of all interfaces. *Mutual dependency* is specified if two interfaces have each other as cointerface.

2.1 Syntax

A syntax for behavioral interfaces is now introduced. Let Mtd denote the set of method names, v a program variable, and T a type. The type T may be either an interface or a data type.

Definition 1 A *method* is represented by a term

$$method\,(Name, Co, Inpar, Outpar, Body),$$

where $Name \in \mathsf{Mtd}$ is a method name, Co is an interface, $Inpar$ and $Outpar$ are lists of parameter declarations of the form $v : T$, and $Body$ is a pair $\langle Var, Code\rangle$ consisting of a list $Vdecl$ of variable declarations (with initial expressions) and a list $Code$ of program statements. If Mtd is a set of methods, denote the subset of Mtd with methods of a given name by

$$Mtd(Name) = \{method\,(Name, Co, Inpar, Outpar, Body) \in Mtd\}.$$

Let $\mathcal{M}$ denote the set of method terms, and $\tau_{\mathcal{M}}$ the set of method names with typical element m. For convenience, the elements of a method tuple may be accessed by dot notation. The symbol ϵ denotes the empty sequence (or list). To conveniently organize object viewpoints, interfaces are structured in an inheritance hierarchy.

Definition 2 An *interface* is represented by a term

$$interface\,(Inh, Mtd)$$

of type $\mathcal{I}$, where Inh is a list of interfaces, defining inheritance, and Mtd is a set of methods such that $m.Body = \langle \epsilon, \epsilon\rangle$ for all $m \in Mtd$.

Let $\tau_{\mathcal{I}}$ denote the set of interface names, with typical elements I and J. Names are bound to interface terms in the typing environment. If I inherits J, the

$$
\begin{array}{lll}
IL & ::= & [\textbf{interface}\ I\ [\textbf{inherits}\ [I]^{+}_{,}]^{?}\ \textbf{begin}\ [\textbf{with}\ I\ \mathit{Msig}^{*}]^{?}\ \textbf{end}]^{*} \\
\mathit{Msig} & ::= & \textbf{op}\ m\ ([\textbf{in}\ \mathit{Param}]^{?}\ [\textbf{out}\ \mathit{Param}]^{?}) \\
\mathit{Param} & ::= & [v : T]^{+}_{,}
\end{array}
$$

Figure 1. A syntax for the abstract representation of interface specifications. Square brackets are used as meta parenthesis, with superscript $^{?}$ for optional parts, superscript * for repetition zero or more times, whereas $[\ldots]^{+}_{d}$ denotes repetition one or more times with d as delimiter. E denotes a list of expressions.

methods declared in both I and J must be available in any class that implements I. Dot notation may be used to refer to the different elements of an interface; e.g., $\mathit{interface}(\mathit{Is}, M).\mathit{Mtd} = M$. The name $\mathit{Any} \in \tau_{\mathcal{I}}$ is reserved for $\mathit{interface}(\epsilon, \emptyset)$, and the name $\varsigma \in \tau_{\mathcal{I}}$ is reserved for type checking purposes. An abstract representation of an interface may be given following the syntax of Figure 1. In the abstract representation all methods of an interface have the same cointerface, declared in a **with**-clause, encouraging an aspect-oriented specification style [50].

Even if two interfaces have the same set of methods, it may be undesirable to (accidentally) identify them. Consequently, we use a nominal subtype relation [67]. An interface is a subtype of its inherited interfaces. The subtype relation may also be explicitly extended by subtype declarations. The extension of this notion of interface with semantic constraints on the observable communication history and the refinement of such interfaces is studied in [44, 45].

2.2 Example

In order to illustrate the interface notion and pave the way for future examples, we consider the interfaces of a node in a peer-to-peer file sharing network. A *Client* interface captures the client end of the node, available to any user of the system. It offers methods to list all files available in the network and to request the download of a given file from a given server. A *Server* interface offers a method for obtaining a list of files available from the node and a method for downloading packets, i.e., parts of a target file. The *Server* interface is only available to other servers in the network. Due to the cointerface, type soundness [67] will guarantee that any caller of a server request understands the *enquire* and *getPacket* methods. The two interfaces may be inherited by a third interface *Peer* which describes nodes that are able to act according to both the client role and the server role. In the *Peer* interface, the cointerface requirement of each superinterface restricts the use of the methods inherited from that superinterface. For simplicity method signatures are omitted here, these are discussed in Section 3.3.

```
interface Client
begin
 with Any
  op availFiles
  op reqFile
end
```

```
interface Server
begin
 with Server
  op enquire
  op getLength
  op getPacket
end
```

```
interface Peer
 inherits Client, Server
begin
end
```

3 Object Interaction by Asynchronous Method Calls

Inter-process communication is becoming increasingly important with the development of distributed computing, both over the Internet and over local networks. While object orientation is the leading framework for distributed and concurrent systems, standard models of object interaction seem less appropriate for distributed concurrent objects. To motivate Creol's asynchronous method calls, we give a brief review of the basic interaction models for concurrent processes with respect to distributed interaction.

The three basic interaction models for concurrent processes are shared variables, remote method calls, and message passing [6]. Shared memory models do not generalize well to distributed environments, so shared variables are discarded as inappropriate to capture object interaction in the distributed setting. With the *remote method invocation* (RMI) model, an object is activated by a method call. The thread of control is transferred with the call so there is a master-slave relationship between the caller and the callee. Caller activity is blocked until the return values from the method call have been received. A similar approach is taken with the execution threads of; e.g., Hybrid [64] and Java [36], where concurrency is achieved through multithreading. The interference problem related to shared variables reemerges when threads operate concurrently in the same object, which happens with non-synchronized methods in Java. Reasoning about programs in this setting is a highly complex matter [2, 18]: Safety is by convention rather than by language design [11]. Verification considerations therefore suggest that all methods should be synchronized, which is the approach taken in, e.g., Hybrid. However, when the language is restricted to synchronized methods, an object making a remote method call must *wait* for the return of the call before it can proceed with its activity. Consequently, any other activity in the object is prohibited while waiting. In a distributed setting this limitation is severe; delays and instabilities may cause much unnecessary waiting. A nonterminating method will even block the evaluation of other method activations, which makes it difficult to combine active and passive behavior in the same object.

In contrast to remote method calls, message passing is a communication form

without any transfer of control between concurrent objects. A method call can here be modeled by an invocation and a reply message. Message passing may be synchronous, as in Ada's Rendezvous mechanism, in which case both the sender and receiver process must be ready before communication can occur. Hence, objects synchronize on message transmission. Remote method invocations may be captured in this model if the calling object blocks between the two synchronized messages representing the call [6]. If the calling object is allowed to proceed for a while before resynchronizing on the reply message we obtain a different model of method calls which from the caller perspective resembles *future variables* [80] (or eager invocation [28]). For distributed systems, even such synchronization must necessarily result in much waiting.

Message passing may also be asynchronous. In the asynchronous setting message emission is always possible, regardless of when the receiver accepts a message. Communication by asynchronous message passing is well-known from, e.g., the Actor model [3,4]. Languages with notions of future variables are usually based on asynchronous message passing. In this case, the caller's activity is synchronized with the arrival of the reply message rather than with its emission, and the activities of the caller and the callee need not directly synchronize [8,16,23,43,79,80]. This approach seems well-suited to model communication in distributed environments, reflecting the fact that communication in a network is not instantaneous. Asynchronous message passing, without synchronization and transfer of the thread of control, avoids unnecessary waiting in the distributed setting by providing better control and efficiency. Generative communication in, e.g., Linda [17] and Klaim [9] is an approach between shared variables and asynchronous message passing, where messages without an explicit destination address are shared on a possibly distributed blackboard. However, method calls imply an ordering on communication not easily captured in thse models. Actors do not distinguish replies from invocations, so capturing method calls with Actors quickly becomes unwieldy [3]. Asynchronous message passing does not provide the structure and discipline inherent in method calls. The integration of the message concept in the object-oriented setting is unsettled, especially with respect to inheritance and redefinition. A satisfactory notion of method call for the distributed setting should be asynchronous, combining the advantages of asynchronous message passing with the structuring mechanism provided by the method concept. Such a notion is proposed in Creol's communication model.

3.1 Syntax

A simple language for concurrent objects is now presented, which combines so-called *processor release points* and *asynchronous method calls*. Processor release points influence the internal control flow in objects. This reduces time

spent waiting for replies to method calls in the distributed setting and allows objects to dynamically change between active and reactive behavior.

At the imperative level, attributes and method declarations are organized in classes. Classes may have parameters [25] which can be data values or objects. Class parameters are similar to constructor parameters in Java [12], except that they form part of the state, making the assignment of parameter values to object attributes redundant. Objects are dynamically created instances of classes, their persistent state consists of declared class parameters and attributes. The state of an object is encapsulated and can only be accessed via the object's methods. Among the declared methods, we distinguish the method *run*, which is given a special treatment operationally. After initialization the *run* method, if provided, is started. Apart from *run*, declared methods may be invoked internally and by other objects supporting the appropriate interfaces. When called from other objects, these methods reflect reactive (or passive) behavior in the object, whereas *run* initiates active behavior. Methods need not terminate and all method activations may be temporarily *suspended*. The activation of a method results in a *process* executed in a Creol object. In fact, execution in a Creol object is organized around an unordered queue of processes competing for the object's processor.

In order to focus on the communication aspects of concurrent objects, we assume given a functional language for defining local data structures by means of data types and functions performing local computations on terms of such data types. Data types are built from basic data types by type constructors.

Definition 3 Let τ_B be a set of basic data types and $\tau_{\mathcal{I}}$ a set of interface names, such that $\tau_B \cap \tau_{\mathcal{I}} = \emptyset$. Let τ denote the set of all types including the basic and interface types; i.e., $\tau_B \subseteq \tau$ and $\tau_{\mathcal{I}} \subseteq \tau$.

Let T_B and T be typical elements of τ_B and τ. The nominal subtype relation $\preceq$ is a reflexive partial ordering on types, including interfaces. We let $\bot$ represent an undefined (and illegal) type; thus for any type T we have $\neg(\bot \preceq T)$ and $\neg(T \preceq \bot)$. We denote by Data the supertype of both data and interface types. Apart from Data, a data type may only be a subtype of a data type and an interface only of an interface. Every interface is a subtype of *Any*, except ς which is only related to itself (i.e., $\varsigma \preceq \varsigma$). Nominal constraints restrict a structural subtype relation which ensures substitutability: if $T \preceq T'$ then any value of T may masquerade as a value of T' [12,54]. For product types R and R', $R \preceq R'$ is the point-wise extension of the subtype relation; i.e., R and R' have the same length l and $T_i \preceq T'_i$ for every i $(0 \leq i \leq l)$ and types T_i and T'_i in position i in R and R', respectively. To explain the typing and binding of methods, $\preceq$ is extended to function spaces $A \to B$, where A and B are (possibly empty) product types:

$$A \to B \preceq A' \to B' \Leftrightarrow A' \preceq A \wedge B \preceq B'$$

For types U and V, the intersection $T = U \cap V$ is such that $T \preceq U$, $T \preceq V$, and $T' \preceq T$ for all T' such that $T' \preceq U$ and $T' \preceq V$. (If no such T exists, $U \cap V = \bot$.) For every type T, we let d_T denote the *default value* of T (e.g., d_I is called *null* for $I \in \tau_{\mathcal{I}}$, d_{Nat} may be *zero*, etc).Type schemes such as parameterized data types may be applied to types in τ to form new types in τ. It is assumed in the examples of the sequel that τ_B includes standard types such as the Booleans Bool, the natural numbers Nat, and the strings Str, and that the type schemes include Set$[T]$ and List$[T]$. Expressions without side effects are given by a functional language $\mathcal{F}$ defined as follows:

Definition 4 Let $\mathcal{F}$ be a type sound functional language which consists of expressions $e \in$ Expr constructed from

- constants or variables of the types in τ_B,
- variables of the types in $\tau_{\mathcal{I}}$, and
- functions defined over terms of the types of τ.

In particular, ObjExpr and BoolExpr are subsets of Expr typed by interfaces and Booleans, respectively. There are no constructors or field access functions for terms of interface types, but object references may be compared by equality.

Assume given a typing environment $\Gamma_{\mathcal{F}}$ which provides the type information for the constants and functions in $\mathcal{F}$, and let Γ extend $\Gamma_{\mathcal{F}}$ with type information for variables. If d is a variable, constant, or function in $\mathcal{F}$, then $\Gamma(d)$ denotes the type of d in Γ. In particular, $\Gamma(d_T) = T$. For $e \in$ Expr, $\Gamma \vdash_{\mathrm{F}} e : T$ denotes that e is type-correct and has type T in Γ (i.e., $T \neq \bot$). If e is type-correct in Γ and v is a variable occurring in e, then $\Gamma(v) \neq \bot$. Let Var be the type of variable names. Variable names are bound to actual values in a state.

Definition 5 Let σ : Var $\mapsto$ Data be a *state* with domain $\{v_1, \ldots, v_n\}$. If $\Gamma(\sigma(v_i)) \preceq \Gamma(v_i)$ for every i $(1 \leq i \leq n)$, then the state σ is *well-typed*.

The *evaluation* of an expression $e \in$ Expr, relative to a state σ, is denoted $\mathit{eval}(e, \sigma)$. It is assumed in Definition 4 that $\mathcal{F}$ is *type sound* [67]: well-typed expressions remain well-typed during evaluation. Technically, the type soundness of $\mathcal{F}$ is given as follows: Let $\vec{v}$ be the variables in an expression $e \in$ Expr and assume that σ is a well-typed state defined for all $v \in \vec{v}$. If $\Gamma \vdash_{\mathrm{F}} e : T$ then $\Gamma \vdash_{\mathrm{F}} \mathit{eval}(e, \sigma) : T'$ such that $T' \preceq T$.

The assumption of type soundness for $\mathcal{F}$ leads to a restricted use of partial functions. For instance, one may adapt the order-sorted approach [34] where partial functions are allowed by identifying the subdomains in which they give defined values, and require that each application of a partial function is defined. In this approach, the head of an empty list would not be type-correct, but the head of a list suffixed by an element would be type-correct.

$$
\begin{array}{lll}
CL & ::= & [\textbf{class}\ C\ [(\mathit{Param})]^?\ [\textbf{contracts}\ [I]^+_,]^?\ [\textbf{implements}\ [I]^+_,]^? \\
 & & \quad [\textbf{inherits}\ [C[(\mathrm{E})]^?]^+_,]^?\ \textbf{begin}\ [\textbf{var}\ \mathit{Vdecl}]^?\ [[\textbf{with}\ I]^?\ \mathit{Mdecls}]^*\ \textbf{end}]^* \\
\mathit{Vdecl} & ::= & [v : T[= e]^?]^+_, \\
\mathit{Mdecls} & ::= & [\mathit{Msig} == [\textbf{var}\ \mathit{Vdecl};]^?\ \mathrm{S}]^+
\end{array}
$$

Figure 2. An syntax outline for the abstract representation of classes, excluding expressions e, expression lists E, and statement lists S (which are defined in Figure 3).

Classes and objects. An object-oriented language is now constructed, extending the functional language $\mathcal{F}$. Classes are defined in a traditional way, including declarations of persistent state variables and method definitions.

Definition 6 A *class* is represented by a term

$$\mathit{class}\,(\mathit{Param}, \mathit{Impl}, \mathit{Contract}, \mathit{Inh}, \mathit{Var}, \mathit{Mtd}),$$

where *Param* is a list of typed program variables, *Contract* and *Impl* are lists of interface names, *Inh* is a list of instantiated class names, defining class inheritance, *Var* is a list of typed program variables with initial expressions, and *Mtd* is a set of methods.

Each method is equipped with an element *Co* specifying the cointerface associated with the method (Definition 1). For purely internal methods, the cointerface element contains the special name ς. Notice that *Impl* represents interfaces implemented by the class, whereas *Contract* represents interfaces implemented by the class and all subclasses. Thus *Contract* claims are inherited by subclasses, but *Impl* claims are not. A class C is said to contract an interface I if a subinterface of I appears in the *Contract* clause of C or of a superclass of C. The typing of remote method calls in a class C relies on the fact that the calling object supports the contracted interfaces of C, and these are used to check the cointerface requirements of the calls.

Let $\mathcal{C}$ denote the set of class terms, and $\tau_{\mathcal{C}}$ the set of class names with typical element C. In the typing environment, class names are bound to class terms. For convenience, dot notation is used to denote the different elements of a class; e.g., *Cl.Var* denotes the variable list of a class *Cl*. An abstract representation of a class may be given following the syntax of Figure 2. Variable declarations are defined as a sequence *Vdecl* of statements $v : T$ or $v : T = e$, where v is the name of the attribute, T its type, and e an optional expression providing an initial value for v. This expression may depend on the actual values of the class parameters. A statement $v : T$ without an initial expression is initialized to the default value d_T. Overloading of methods is allowed. The pseudo-variable *self* is used for self reference in the language; its value cannot be modified. Issues related to inheritance are considered in Section 5; until then, we consider classes without explicit inheritance.

Syntactic categories.		*Definitions.*
g in Guard	v in Var	$g ::= \mathit{wait} \mid b \mid t? \mid g_1 \wedge g_2 \mid g_1 \vee g_2$
t in Label	s in Stm	$r ::= x.m \mid m$
m in Mtd	r in MtdCall	$\mathrm{S} ::= s \mid s; \mathrm{S}$
e in Expr	b in BoolExpr	$s ::= \mathbf{skip} \mid (\mathrm{S}) \mid \mathrm{V} := \mathrm{E} \mid v := \mathbf{new}\ C(\mathrm{E})$
x in ObjExpr		$\mid r(\mathrm{E}; \mathrm{V}) \mid !r(\mathrm{E}) \mid t!r(\mathrm{E}) \mid t?(\mathrm{V}) \mid \mathbf{await}\ g$

Figure 3. An outline of the language syntax for program statements, with typical terms for each syntactic category. Capitalized terms such as V, S, and E denote lists, sets, or multisets of the given syntactic categories, depending on the context.

An object offers methods to its environment, specified through a number of interfaces. All interaction between objects happens through method calls. In the asynchronous setting method calls can always be emitted, because the receiving object cannot block communication. *Method overtaking* is allowed: if methods offered by an object are invoked in one order, the object may evaluate the corresponding method activations in another order. A method activation is, roughly speaking, a list S of program statements evaluated in the context of a state. Due to the possible interleavings of different method executions, the values of an object's program variables are not entirely controlled by a method activation which suspends itself before completion. However, a method may have local variables supplementing the object attributes. In particular, the values of formal parameters are stored locally, but other local variables may also be created. Among the local variables of a method, certain variables are used to organize inter-object communication; there is read-only access to *caller* and *label*. Assignment to local and object variables is expressed as $\mathrm{V} := \mathrm{E}$ for (the same number of) program variables V and expressions E. In the object creation statement $v := C(\mathrm{E})$, v must be a variable declared of an interface implemented by C and E are actual values for the class parameters. We refer to $C(\mathrm{E})$ as the *instantiated class name*. The syntax for program statements is given in Figure 3.

Let Label denote the type of method call identifiers, partially ordered by $<$ and with least element 1, and let the operation $\mathit{next} : \mathsf{Label} \to \mathsf{Label}$ be such that $\forall x \in \mathsf{Label}\,.\, x < \mathit{next}(x)$. A method is *asynchronously* invoked with the statement $t!x.m(\mathrm{E})$, where $t \in \mathsf{Label}$ provides a locally unique reference to the call, x is an object expression, m a method name, and E an expression list with the actual in-parameters supplied to the method. The call is *internal* when x is omitted, otherwise the call is *external*. A method with ς as cointerface may only be called internally. Labels are used to identify replies and may be omitted if a reply is not explicitly requested. As no synchronization is involved, process execution can proceed after calling a method until the return value from the method is actually needed by the process.

To fetch the return values from a call, say in a variable list V, we may ask for the reply to our call: $t?(\mathrm{V})$. This statement treats V as a list of future

variables. If the reply to the call has arrived, return values may be assigned to V and the execution continues without delay. If the reply has not arrived, process execution is *blocked* at this statement. In order to avoid blocking in the asynchronous case, processor release points are introduced by means of reply guards. In this case, process execution is *suspended* rather than blocked.

Any method may be invoked in a synchronous as well as an asynchronous manner. *Synchronous* (RMI) method calls are given the syntax $x.m(\mathrm{E};\mathrm{V})$, which is defined by $t!x.m(\mathrm{E});\, t?(\mathrm{V})$ for some fresh label t, *immediately* blocking the processor while waiting for the reply. This way the call is perceived as synchronous by the caller, although the interaction with the callee is in fact asynchronous. The callee does not distinguish synchronous and asynchronous invocations of its methods. It is clear that in order to reply to local calls, the calling method must eventually suspend its own execution. A local call may be either internal, or external if the callee is equal to *self*. Therefore, the reply statement $t?(\mathrm{V})$ enables execution of the call identified by t when this call is local. The language does not otherwise support monitor reentrance; mutual or cyclic synchronous calls between objects may therefore lead to deadlock.

Potential suspension is a expressed through *processor release points*, a basic programming construct in the language, using guard statements [29]. In Creol, guards influence the control flow between processes inside concurrent objects. A guard g is used to explicitly declare a potential release point for the object's processor with the statement **await** g. Guard statements can be nested within a method body, corresponding to a series of potential suspension points. Let S_1 and S_2 denote statement lists; in $\mathrm{S}_1;\, \mathbf{await}\ g;\, \mathrm{S}_2$ the guard g corresponds to an inner release point. A guard statement is *enabled* if its guard evaluates to true. When an inner guard which is not enabled is encountered during process execution, the process is suspended and the processor released. The *wait* guard is a construct for explicit release of the processor. The reply guard $t?$ is enabled if the reply to the method invocation with label t has arrived. Guards may be composed: $g_1 \wedge g_2$ is enabled if both g_1 and g_2 are enabled. The evaluation of guard statements is atomic. After process suspension, the object's suspended processes compete for the free processor: *any* suspended and enabled process may be selected for execution. For convenience, we introduce the following abbreviations:

await $t?(\mathrm{V})$ = **await** $t?\,;\, t?(\mathrm{V})$
await $r(\mathrm{E};\mathrm{V})$ = $t!r(\mathrm{E})\,;$ **await** $t?(\mathrm{V})$, where t is a fresh label.

Using reply guards, the object processor need not block while waiting for replies. This approach is more flexible than future variables: suspended processes or new method activations may be evaluated while waiting for a reply. If the called object does not eventually reply, deadlock is avoided in the sense that other activity in the object is possible although the process itself will a

priori remain suspended. However, when a reply arrives, the *continuation* of the original process must compete with other enabled suspended processes.

3.2 Virtual Binding

Due to the interface typing of object variables, the actual class of the receiver of an external call is not statically known. Consequently, external calls are virtually bound.

Let the function *Sig* give the signature of a method, defined by $Sig(m) = type(m.Inpar) \rightarrow type(m.Outpar)$ where *type* returns the product of the types in a parameter declaration. The static analysis of a synchronous internal call $m(\text{E}; \text{V})$ assigns unique types to the in- and out-parameter depending on the textual context, say that the parameters are textually declared as $\text{E} : T_\text{E}$ and $\text{V} : T_\text{V}$. The call is *type-correct* if there is a method declaration $m : T_1 \rightarrow T_2$ in the class C such that $T_1 \rightarrow T_2 \preceq T_\text{E} \rightarrow T_\text{V}$. A synchronous external call $o.m(\text{E}; \text{V})$ to an object o of interface I is type-correct if it can be bound to a method declaration in I in a similar way. The static analysis of a class verifies that it implements the methods declared in its interfaces. Assuming that any object variable typed by an interface I points to an instance of a class implementing I, method binding will succeed regardless of the actual class of the object. At runtime, the class of the object is dynamically identified and the method is virtually bound. Remark that if the method is overloaded; i.e., there are several methods with the same name in the class, the types of the actual parameter values, including the out-parameter, and the actual cointerface are used to correctly bind the call (see Section 5).

Asynchronous calls may be bound in the same way, provided that the type of the actual parameter values and cointerface can be determined. In the operational semantics of Creol, it is assumed that this type information is included at compile-time in both synchronous and asynchronous method invocations.

3.3 Example

A peer-to-peer file sharing system consists of nodes distributed across a network. Peers are equal: each node plays both the role of a server and of a client. In the network, nodes may appear and disappear dynamically. As a client, a node requests a file from a server in the network and downloads it as a series of packet transmissions until the file download is complete. The connection to the server may be blocked, in which case the download automatically resumes if the connection is reestablished. A client may run several downloads concurrently, at different speeds. We assume that every node in the network

```
class Node (db:DB)
  implements Peer
begin
with Server
 op enquire(out files:List[Str]) == await db.listFiles(ϵ;files)
 op getLength(in fId:Str out lth:Nat) == await db.getLength(fId;lth)
 op getPacket(in fId:Str; pNbr:Nat out packet:List[Data])
   == var f:List[Data];await db.getFile(fId;f); packet:=f[pNbr]
with Any
 op availFiles (in sList:List[Server] out files:List[Server×Str])
  == var l1:Label; l2:Label; fList:List[Str];
  if (sList = empty) then files:= empty
  else l1!hd(sList).enquire(); l2!this.availFiles(tl(sList));
    await l1? ∧ l2?; l1?(fList); l2?(files); files:=((hd(sList),fList); files) fi
 op reqFile(in sId:Server out fId:Str) ==
  var file:List[Data]; packet:List[Data]; lth:Nat;
   await sId.getLength(fId;lth); while (lth > 0) do
    await sId.getPacket(fId, lth; packet); file:=(packet;file); lth:=lth - 1 od;
  !db.storeFile(fId, file)
end
```

Figure 4. A class capturing nodes in a peer-to-peer network.

has an associated database with shared files. Downloaded files are stored in the database, which implements the interface *DB* and is not modeled here:

```
interface DB
begin
with Server
 op getFile(in fId:Str out file:List[List[Data]])
 op getLength(in fId:Str out length:Nat)
 op storeFile(in fId:Str; file:List[Data])
 op listFiles(out fList:List[Str])
end
```

Here, *getFile* returns a list of packets; i.e., a sequence of sequences of data, for transmission over the network, *getLength* returns the number of such sequences, *listFiles* returns the list of available files, and *storeFile* adds a file to the database, possibly overwriting an existing file.

Nodes in the peer-to-peer network which implement the *Peer* interface can be modeled by a class *Node*, given in Figure 4. *Node* objects can have several interleaved activities: several downloads may be processed simultaneously as well as uploads to other servers, etc. All method calls are asynchronous: If a server temporarily becomes unavailable, the transaction is suspended and may

resume at any time after the server becomes available again. Processor release points ensure that the processor will not be blocked and transactions with other servers not affected. In the class, the method *availFiles* returns a list of pairs where each pair contains a file identifier *fId* and the server identifier *sId* where *fId* may be found, *reqFile* the file associated with *fId*, *enquire* the list of files available from the server, and *getPacket* a particular packet in the transmission of a file. The list constructor is represented by semicolon. For $x : T$ and $\mathrm{S} : \mathsf{List}[T]$, we let $hd(x; \mathrm{S}) = x$, $tl(x; \mathrm{S}) = \mathrm{S}$, and $\mathrm{S}[i]$ denote the i'th element of S, provided $i \leq length(\mathrm{S})$.

3.4 Typing

The type analysis of statements and declarations is formalized by a deductive system for judgments of the form

$$\Gamma \vdash_i D \langle\, \Delta \,\rangle,$$

where Γ is the typing environment, $i \in \{\mathrm{S}, \mathrm{V}\}$ specifies the syntactic category (Stm or Var, respectively), D is a Creol construct (statement or declaration), and Δ is the *update* of the typing environment. The typing judgment means that D contains no type errors when checked with the environment Γ. The typing environment resulting from the type analysis of D becomes Γ overridden by Δ, denoted $\Gamma + \Delta$. The rule for sequential composition SEQ is captured by

$$\text{(SEQ)} \quad \frac{\Gamma \vdash_i D \langle\, \Delta \,\rangle \qquad \Gamma + \Delta \vdash_i D' \langle\, \Delta' \,\rangle}{\Gamma \vdash_i D; D' \langle\, \Delta + \Delta' \,\rangle}$$

where $+$ is an associative operator on mappings with the identity element $\emptyset$. We abbreviate $\Gamma \vdash_i D \langle\, \emptyset \,\rangle$ to $\Gamma \vdash_i D$.

For our purpose, the typing environment Γ is given as a *family of mappings*: $\Gamma_{\mathcal{F}}$ describes the constants and operators of $\mathcal{F}$, $\Gamma_{\mathcal{I}}$ the binding of interface names to interface terms, $\Gamma_{\mathcal{C}}$ the binding of class names to class terms, Γ_{V} the binding of program variable names to types, the mapping Γ_P is related to the binding of asynchronous internal and external method calls, and Γ_{Sig} stores derived actual signatures for asynchronous method invocations. Remark that $\Gamma_{\mathcal{I}}$ and $\Gamma_{\mathcal{C}}$ correspond to static tables. Declarations may only update Γ_{V} and program statements may not update Γ_{V}. Mapping families are now formally defined.

Definition 7 Let n be a name, d a declaration, $i \in \{\mathcal{I}, \mathcal{C}, V, P, Sig\}$ a mapping index, and $[n \overset{i}{\mapsto} d]$ the binding of n to d indexed by i. A *mapping family* Γ is built from the empty mapping family $\emptyset$ and indexed bindings by the constructor $+$. The mapping with index i is extracted from Γ as follows

$$\begin{aligned}\emptyset_i &= \epsilon \\ (\Gamma + [n \overset{i'}{\mapsto} d])_i &= \textbf{if } i = i' \textbf{ then } \Gamma_i + [n \overset{i}{\mapsto} d] \textbf{ else } \Gamma_i.\end{aligned}$$

For an indexed mapping Γ_i, mapping application is defined by

$$\begin{aligned}\epsilon(n) &= \bot \\ (\Gamma_i + [n \overset{i}{\mapsto} d])(n') &= \textbf{if } n = n' \textbf{ then } d \textbf{ else } \Gamma_i(n').\end{aligned}$$

3.4.1 Typing of Programs

A class or interface declaration binds a name to a class or interface term, respectively. Class and interface names need not be distinct. A program consists of interface and class declarations, represented by the mappings $\Gamma_{\mathcal{I}} : \tau_{\mathcal{I}} \to \mathcal{I}$ and $\Gamma_{\mathcal{C}} : \tau_{\mathcal{C}} \to \mathcal{C}$, and an initial object creation message **new** $C(\text{E})$. In a nominal type system, each interface and class of a program is type checked in the context of the mappings $\Gamma_{\mathcal{F}}$, $\Gamma_{\mathcal{I}}$, and $\Gamma_{\mathcal{C}}$.

$$\text{(PROG)}\quad \frac{\begin{array}{ll}\forall I \in \tau_{\mathcal{I}} \cdot \Gamma_{\mathcal{I}} \vdash \Gamma_{\mathcal{I}}(I) & \Gamma_{\mathcal{F}} + \Gamma_{\mathcal{C}} \vdash_{\text{S}} \textbf{ new } C(\text{E}) \\ \forall C \in \tau_{\mathcal{C}} \cdot \Gamma_{\mathcal{F}} + \Gamma_{\mathcal{I}} + \Gamma_{\mathcal{C}} + [\mathit{self} \overset{\text{V}}{\mapsto} C] \vdash \Gamma_{\mathcal{C}}(C) \langle\, \Delta^c \,\rangle & \end{array}}{\Gamma_{\mathcal{F}} \vdash \Gamma_{\mathcal{I}}, \Gamma_{\mathcal{C}}, \textbf{new } C(\text{E}) \langle\, \bigcup_{C \in \tau_{\mathcal{C}}} \Delta^c_{Sig} \,\rangle}$$

When type checking a class, *self* is bound to the class name. Type checking a program succeeds if all interfaces are well-formed, all classes are type-correct, and the initial object creation message is type-correct in the context of the program's class declarations. In order to focus on the type checking of classes, the method set *Mtd* of interface terms here includes both locally declared and inherited methods. The rule for type checking interface declarations may now be given as follows:

$$\text{(INTERFACE)}\quad \frac{\forall m \in \mathit{Mtd} \;\cdot\; \Gamma \vdash_{\text{V}} \; m.\mathit{Inpar}; m.\mathit{Outpar} \langle\, \Delta \,\rangle}{\Gamma \vdash \mathit{interface}\,(\mathit{Inh}, \mathit{Mtd})}$$

The type checking of classes is now considered in detail. The rule for type checking class declarations is given as follows:

$$\text{(CLASS)}\quad \frac{\begin{array}{ll}\Gamma \vdash_{\text{V}} \; \mathit{Param}; \mathit{Var} \langle\, \Delta \,\rangle & \forall m \in \mathit{Mtd} \;\cdot\; \Gamma + \Delta \vdash m \langle\, \Delta^m \,\rangle \\ \multicolumn{2}{l}{\forall I \in (\mathit{Impl}; \mathit{Contract}) \cdot \forall m' \in \Gamma_{\mathcal{I}}(I).\mathit{Mtd} \cdot \exists m \in \mathit{Mtd} \cdot} \\ \multicolumn{2}{l}{\quad m.\mathit{Name} = m'.\mathit{Name} \wedge \mathit{Sig}(m) \preceq \mathit{Sig}(m') \wedge m'.\mathit{Co} \preceq m.\mathit{Co}}\end{array}}{\Gamma \vdash \mathit{class}\,(\mathit{Param}, \mathit{Impl}, \mathit{Contract}, \epsilon, \mathit{Var}, \mathit{Mtd}) \langle\, \bigcup_{m \in \mathit{Mtd}} \Delta^m_{\mathit{Sig}} \,\rangle}$$

A class may implement a number of interfaces. For each interface, the class must provide methods with signatures that are correct with respect to the method signatures of the interface. The class' variable declarations are type

checked after extending the typing environment with the class parameters, because the variable declarations may include initial expressions that use these parameters. Before type checking the methods, the typing environment is extended with the declared parameters and variables of the class. Method bodies are type checked in the typing environment updated after type checking method parameters and local attributes, including *caller*. At this point Γ_P is empty, since no asynchronous method invocation has been encountered.

$$\text{(METHOD)}\quad \frac{\begin{array}{c}\Gamma \vdash_{\mathrm{V}} \ (\mathit{caller} : \mathit{Co}); \mathit{Inpar}; \mathit{Outpar}; \mathit{Body.Var} \ \langle\ \Delta\ \rangle \\ \Gamma + \Delta + \emptyset_P \vdash_{\mathrm{S}} \ \mathit{Body.Code} \ \langle\ \Delta'\ \rangle\end{array}}{\Gamma \vdash \mathit{method}\,(\mathit{Name}, \mathit{Co}, \mathit{Inpar}, \mathit{Outpar}, \mathit{Body}) \ \langle\ \Delta'_{\mathit{Sig}}\ \rangle}$$

In order to use self reference in expressions inside classes, we introduce qualified self references by the keyword **qua**, which uniquely controls the typing in case the class has many contracts.

$$\text{(SELF-REF)}\quad \frac{\exists I' \in (\Gamma_{\mathcal{C}}(\Gamma_{\mathrm{V}}(\mathit{self})).\mathit{Contract}; \mathsf{Any}) \ \cdot \ I' \preceq I}{\Gamma \vdash_{\mathrm{F}} \ \mathit{self}\ \mathbf{qua}\ I : I}$$

3.4.2 Typing of Parameter and Variable Declarations

A method may have local variable declarations preceding the program statements. Thus, both class and local variable declarations may extend the typing environment provided that these are not previously declared in the typing environment. The typing rules for variable and parameter declarations are given in Figure 5.

$$\text{(VAR-AX)}\ \ \Gamma \vdash_{\mathrm{V}} \ \epsilon \qquad\qquad \text{(PAR)}\ \frac{\Gamma_{\mathrm{V}}(v) = \bot \quad T \preceq \mathsf{Data} \quad v \neq \mathit{label}}{\Gamma \vdash_{\mathrm{V}} \ v : T \ \langle\ [v \overset{\mathrm{V}}{\mapsto} T]\ \rangle}$$

$$\text{(VAR)}\ \frac{\Gamma \vdash_{\mathrm{V}} \ v : T \ \langle\ \Delta\ \rangle \quad \Gamma \vdash_{\mathrm{F}} \ e : T' \quad T' \preceq T}{\Gamma \vdash_{\mathrm{V}} \ v : T = e \ \langle\ \Delta\ \rangle}$$

Figure 5. Typing of variable and parameter declarations.

3.4.3 Typing of Basic Statements

The typing of basic statements is given in Figure 6, as well as a statement for object creation. The typing system for $\mathcal{F}$ is used to type check expressions. The last premise of NEW ensures that the new object implements an interface which is a subtype of the declared interface of the variable v, extending rule NEW2 for initial object creation messages.

$$\text{(SKIP)}\quad \Gamma \vdash_{\mathrm{S}} \mathbf{skip}$$

$$\text{(EMPTY)}\quad \Gamma \vdash_{\mathrm{S}} \epsilon$$

$$\text{(ASSIGN)}\ \frac{\Gamma \vdash_{\mathrm{F}} \mathrm{E} : T' \qquad T' \preceq \Gamma_{\mathrm{V}}(\mathrm{V}) \qquad \Gamma_{\mathrm{V}}(\mathrm{V}) \neq \mathsf{Label} \qquad \mathrm{V} \neq \mathit{caller}}{\Gamma \vdash_{\mathrm{S}} \mathrm{V} := \mathrm{E}}$$

$$\text{(AWAIT-}b\text{)}\ \frac{\Gamma \vdash_{\mathrm{F}} b : \mathsf{Bool}}{\Gamma \vdash_{\mathrm{S}} \mathbf{await}\ b}$$

$$\text{(AWAIT-}\wedge\text{)}\ \frac{\Gamma \vdash_{\mathrm{S}} \mathbf{await}\ g_1 \qquad \Gamma \vdash_{\mathrm{S}} \mathbf{await}\ g_2}{\Gamma \vdash_{\mathrm{S}} \mathbf{await}\ g_1 \wedge g_2}$$

$$\text{(AWAIT-}t?\text{)}\ \frac{\Gamma_{\mathrm{V}}(t) = \mathsf{Label}}{\Gamma \vdash_{\mathrm{S}} \mathbf{await}\ t?}$$

$$\text{(NEW2)}\ \frac{\Gamma \vdash_{\mathrm{F}} \mathrm{E} : T \qquad T \preceq \mathit{type}(\Gamma_{\mathcal{C}}(C).\mathit{Param})}{\Gamma \vdash_{\mathrm{S}} \mathbf{new}\ C(\mathrm{E})}$$

$$\text{(NEW)}\ \frac{\Gamma \vdash_{\mathrm{S}} \mathbf{new}\ C(\mathrm{E}) \qquad \exists I \in (\Gamma_{\mathcal{C}}(C).\mathit{Impl}; \Gamma_{\mathcal{C}}(C).\mathit{Contract}) \cdot I \preceq \Gamma_{\mathrm{V}}(v)}{\Gamma \vdash_{\mathrm{S}} v := \mathbf{new}\ C(\mathrm{E})}$$

Figure 6. Typing of basic statements.

3.4.4 Typing of Asynchronous Calls

In order to successfully bind method calls, the types of the formal parameters and cointerface of the declared method must correspond to the types of the actual parameters and cointerface of the call. This is verified by a predicate *match*, defined as follows:

Definition 8 Let T and U be types, I an interface, m a method name, and M a set of methods. Define $\mathit{match} : \tau_{\mathcal{M}} \times \tau \times \tau_{\mathcal{I}} \times \mathsf{Set}[\mathcal{M}] \to \mathsf{Bool}$ by

$$\mathrm{match}(m, T \to U, I, \emptyset) = \mathsf{false}$$
$$\mathrm{match}(m, T \to U, I, \{m'\} \cup \mathrm{M}) = (m = m'.\mathit{Name} \wedge I \preceq m'.\mathit{Co} \wedge \mathit{Sig}(m') \preceq T \to U).$$

If a call has a match in an interface or class, we say that the call is covered:

Definition 9 An external method call is *covered* in an interface if there is a method declaration in the interface which may be type-correctly bound to the call, including the actual parameter types of output variables. An internal method call is *covered* in a class if there is a method declaration in the class, or a superclass, which may be type-correctly bound to the call, including the actual parameter types of output variables.

For synchronous calls, it is straightforward to check whether a call is covered, since the types of the actual in- and out-parameters can be derived directly from the textual invocation. In contrast, checking whether asynchronous calls are covered is more involved, since the type information provided by the textual invocation is not sufficient: the correspondence between in- and out-parameters is controlled by label values. The increased freedom in the language gained from using labels, requires a more sophisticated type analysis. In order to use the type system to derive signatures for asynchronous method

invocations, it is assumed in method bodies that every asynchronous invocation using a label t is uniquely indexed; e.g., $t!r(\text{E})$ is transformed by the parser into $t_i!r(\text{E})$ for a fresh index i. The mappings $\Gamma_P : \mathsf{Label} \to \mathsf{Set}[\mathsf{Nat}]$ and $\Gamma_{Sig} : \mathsf{Label}_{\mathsf{Nat}} \to \tau_{\mathcal{I}} \times \tau_{\mathcal{M}} \times \tau_{\mathcal{I}} \times \tau$ are used as an effect system [55,74] for type checking asynchronous calls. The mapping Γ_P maps labels to *sets* of indices, uniquely identifying the occurrences of calls corresponding to a label. (If Γ is a typing environment and no calls on a label t have been recorded in $\Gamma_P(t)$, we let $\Gamma_P(t) = \emptyset$.) The mapping Γ_{Sig} maps indexed labels to tuples containing the information needed to later refine the type analysis of the method calls associated with the indexed labels. For a label t and any $i \in \Gamma_P(t)$, t is the label associated with the i'th internal or external asynchronous *pending* call for which the types of the out-parameters have yet to be resolved. After the out-parameters have been resolved, $\Gamma_{Sig}(t_i)$ is updated with the refined actual signature of the call associated with the indexed label t_i. Hence, Γ_{Sig} provides the interface of the callee and the actual signature for the asynchronous calls in the code. Both mappings Γ_P and Γ_{Sig} are part of the typing environment used for type checking method bodies.

External and Internal Invocations and Replies

The typing rules for external and internal invocations and replies are given in Figure 7. As the types of the actual in- and out-parameters of every synchronous call can be derived immediately, the type system can directly decide if the call is type-correct. Asynchronous calls without labels can also be type checked directly, as the reply values cannot subsequently be requested. Consequently, the type of any formal out-parameter is type checked against the supertype Data. For an external asynchronous invocation with indexed label t_i, the type of the return values is yet unknown. Type checking with the exact type of the out-variables must be *postponed* until the corresponding reply statement for t is eventually analyzed. Therefore, i is added to the set of pending calls $\Gamma_P(t)$ and the invocation is recorded in Γ_{Sig} with a mapping from t_i to the callee's interface, the method name, the caller's interface, and a preliminary actual signature for the call. Some erroneous invocations may already be eliminated by type checking against this preliminary signature.

For internal calls, the method must have cointerface ς. For external calls $x.m$, the interface of x must offer a method m with a cointerface *contracted* by the current class (and thereby supported by the actual calling object). This implies that *remote calls to self* are allowed when the class itself contracts an interface allowed as cointerface for the method.

A reply is requested through a reply statement or a guard, if there are pending invocations with the same label. For a reply statement $t?(\text{V})$, the matching invocations must be type checked again as the types of the out-parameters V

$$\text{(EXT-SYNC)}\ \frac{\Gamma \vdash_{\mathrm{F}}\ e : I \qquad \Gamma \vdash_{\mathrm{F}}\ \mathrm{E} : T \qquad Co \in \overline{Contract} \wedge \mathrm{match}(m, T \to \Gamma_{\mathrm{V}}(\mathrm{V}), Co, \Gamma_{\mathcal{I}}(I).Mtd)}{\Gamma \vdash_{\mathrm{S}}\ e.m(\mathrm{E};\mathrm{V})}$$

$$\text{(EXT-ASYNC)}\ \frac{\Gamma \vdash_{\mathrm{F}}\ e : I \qquad \Gamma \vdash_{\mathrm{F}}\ \mathrm{E} : T \qquad Co \in \overline{Contract} \wedge \mathrm{match}(m, T \to \mathsf{Data}, Co, \Gamma_{\mathcal{I}}(I).Mtd)}{\Gamma \vdash_{\mathrm{S}}\ !e.m(\mathrm{E})}$$

$$\text{(EXT-ASYNC-L)}\ \frac{\Gamma \vdash_{\mathrm{F}}\ e : I \qquad \Gamma \vdash_{\mathrm{F}}\ \mathrm{E} : T \qquad \Gamma_{\mathrm{V}}(t) = \mathsf{Label} \qquad Co \in \overline{Contract} \wedge \mathrm{match}(m, T \to \mathsf{Data}, Co, \Gamma_{\mathcal{I}}(I).Mtd)}{\Gamma \vdash_{\mathrm{S}}\ t_i!e.m(\mathrm{E}) \ \langle\ [t \overset{P}{\mapsto} \{i\}] + [t_i \overset{Sig}{\mapsto} \langle I, m, Co, T \to \mathsf{Data}\rangle]\ \rangle}$$

$$\text{(INT-SYNC)}\ \frac{\mathrm{match}(m, T \to \Gamma_{\mathrm{V}}(\mathrm{V}), \varsigma, \overline{Mtd}) \qquad \Gamma \vdash_{\mathrm{F}}\ \mathrm{E} : T}{\Gamma \vdash_{\mathrm{S}}\ m(\mathrm{E};\mathrm{V})}$$

$$\text{(INT-ASYNC)}\ \frac{\mathrm{match}(m, T \to \mathsf{Data}, \varsigma, \overline{Mtd}) \qquad \Gamma \vdash_{\mathrm{F}}\ \mathrm{E} : T}{\Gamma \vdash_{\mathrm{S}}\ !m(\mathrm{E})}$$

$$\text{(INT-ASYNC-L)}\ \frac{\Gamma_{\mathrm{V}}(t) = \mathsf{Label} \quad \Gamma \vdash_{\mathrm{F}}\ \mathrm{E} : T \quad \mathrm{match}(m, T \to \mathsf{Data}, \varsigma, \overline{Mtd})}{\Gamma \vdash_{\mathrm{S}}\ t_i!m(\mathrm{E}) \ \langle\ [t \overset{P}{\mapsto} \{i\}] + [t_i \overset{Sig}{\mapsto} \langle \varsigma, m, \varsigma, T \to \mathsf{Data}\rangle]\ \rangle}$$

$$\text{(AWAIT-REPLY)}\ \frac{\Gamma_P(t) \neq \emptyset}{\Gamma \vdash_{\mathrm{S}}\ \mathbf{await}\ t?}$$

$$\text{(REPLY)}\ \frac{\begin{array}{c}\Gamma_P(t) \neq \emptyset \\ \forall i \in \Gamma_P(t)\ \cdot\ \Gamma_{Sig}(t_i) = \langle I, m, Co, T_1 \to T_2\rangle \wedge (T_2 \cap \Gamma_{\mathrm{V}}(\mathrm{V})) \neq \bot\ \wedge \\ (I = \varsigma \Rightarrow \mathrm{match}(m, T_1 \to (T_2 \cap \Gamma_{\mathrm{V}}(\mathrm{V})), Co, \overline{Mtd})) \wedge \\ (I \neq \varsigma \Rightarrow \mathrm{match}(m, T_1 \to (T_2 \cap \Gamma_{\mathrm{V}}(\mathrm{V})), Co, \Gamma_{\mathcal{I}}(I).Mtd))\end{array}}{\Gamma \vdash_{\mathrm{S}}\ t?(\mathrm{V}) \ \langle\ [t \overset{P}{\mapsto} \emptyset] + \bigcup_{i \in \Gamma_P(t)} [t_i \overset{Sig}{\mapsto} \langle I, m, Co, T_1 \to (T_2 \cap \Gamma_{\mathrm{V}}(\mathrm{V}))]\ \rangle}$$

Figure 7. Typing of external and internal invocation and reply statements. For an element F, $\overline{F}$ denotes $\Gamma_{\mathcal{C}}(\Gamma_{\mathrm{V}}(\mathit{self})).F$, i.e., the corresponding element in the current class term.

are now known. If the reply rule succeeds, all pending calls on t have been type checked against the actual type of V. This type checking depends on the interface of the callee for external calls and the class of *self* for internal calls. The pending calls on t are removed from Γ_P and the stored signatures for these calls in Γ_{Sig} are replaced by the refined signatures. Note that the type-correctness of pending calls without a corresponding reply statement is given directly by the typing rules for method invocations; i.e., the preliminary signature was sufficiently precise.

Example. Typing with labeled invocation and reply statements, where the binding depends on out-parameters, is illustrated by the following example.

interface A
begin with B
op m(**out** x:Bool)
end

interface B
begin with A
op m(**out** x:Nat)
end

interface AB
inherits A, B
begin end

class C **contracts** AB
begin
op run == **var** x:Nat; o:AB; t:Label; o:=**new** C; t!o.m(); **await** t?; t?(x)
with B **op** m(**out** x:Bool) == x:=true
with A **op** m(**out** x:Nat) == x:=0
end

Here type checking succeeds, binding the call to m in *run* to interface B. Let t_1 be the (first) invocation with label t. The result of the type analysis is that $\Gamma_{Sig}(t_1) = \langle AB, m, AB, \epsilon \rightarrow \mathsf{Nat} \rangle$. This information is passed on to the runtime system to ensure proper binding of the call. At runtime the call is then bound to the last declaration of m in C (see Section 3.5). In contrast, if the reply statement were removed from *run*, the result of the type analysis would be $\Gamma_{Sig}(t_1) = \langle AB, m, AB, \epsilon \rightarrow Data \rangle$ and the call may be bound to either m.

Some initial properties of the type system are now presented.

Lemma 1 *No type checked list of program variable declarations dereference undeclared program variables.*

Proof. The proof is by induction over the length of a type checked list *Var* of program variable declarations. If $Var = \epsilon$, no variables are dereferenced and the lemma holds. We now assume that no undeclared program variables are dereferenced in *Var* and show that this is also the case for $Var; v : T = e$ by considering the variables in e. Let $\Gamma' = \Gamma + \Delta$ such that $\Gamma \vdash_{\mathrm{V}} Var \langle \Delta \rangle$. Rule VAR requires that $\Gamma' \vdash_{\mathrm{F}} e : T'$. It follows from the type system for $\mathcal{F}$ that $T' \neq \bot$, so all variables in e have been declared in Γ'. ■

Lemma 2 *No type checked methods assign to or dereference undeclared program variables.*

Proof. We consider a method term *method*(*Name*, *Co*, *Inpar*, *Outpar*, *Body*) type checked in the context of some class. The in- and out-parameters *Inpar* and *Outpar*, *label*, and *caller* do not have initial expressions, so they do not dereference any program variables. The method body *Body* consists of a list of local variable declarations *Var* and a list of program statements *Code*. It follows from Lemma 1 that *Var* does not dereference undeclared program variables. The proof proceeds by induction over the length of *Code*. Recall that if $\Gamma \vdash_{\mathrm{F}} \mathrm{E} : T$ for some expression E, then $T' \neq \bot$, so for all variables v in E, $\Gamma(v) \neq \bot$ and v is declared in Γ.

If $Code = \epsilon$, the lemma trivially holds. For the induction step, we now assume that no undeclared program variables are assigned to or dereferenced in *Code* and show that this also holds for $Code; s$ by case analysis of s. Let $\Gamma = \Gamma' + \Delta$ such that $\Gamma' \vdash_{\mathrm{S}} Code \langle\, \Delta \,\rangle$.

- No program variables are assigned to or dereferenced by **skip**.
- For $\mathrm{V} := \mathrm{E}$, the ASSIGN rule asserts that $\Gamma \vdash_{\mathrm{F}} \mathrm{E} : T'$ and $T' \preceq \Gamma_{\mathrm{V}}(\mathrm{V})$. Since $T' \preceq \Gamma_{\mathrm{V}}(\mathrm{V})$, $\Gamma_{\mathrm{V}}(\mathrm{V}) \neq \bot$ and V is declared.
- For $x :=$ **new** $C(\mathrm{E})$, the NEW rule asserts that $\Gamma \vdash_{\mathrm{F}} \mathrm{E} : T$ and that there exists an interface $I \in \Gamma_{\mathcal{C}}(C).\mathit{Impl}; \Gamma_{\mathcal{C}}(C).\mathit{Contract}$ such that $I \preceq \Gamma_{\mathrm{V}}(x)$. Consequently $\Gamma_{\mathrm{V}}(x) \neq \bot$ and x is declared.
- For $x.m(\mathrm{E}; \mathrm{V})$, rule EXT-SYNC asserts that $\Gamma \vdash_{\mathrm{F}} x : I$, $\Gamma \vdash_{\mathrm{F}} \mathrm{E} : T$ and there is a cointerface $Co \in \Gamma_{\mathcal{C}}(\Gamma_{\mathrm{V}}(\mathit{self})).\mathit{Contract}$ such that $\mathit{match}(m, T \to \Gamma_{\mathrm{V}}(\mathrm{V}), Co, \Gamma_{\mathcal{I}}(I).\mathit{Mtd})$. The match is impossible unless $\Gamma_{\mathrm{V}}(\mathrm{V}) \neq \bot$.
- For $!x.m(\mathrm{E})$, rule EXT-ASYNC asserts that $\Gamma \vdash_{\mathrm{F}} x : I$ and $\Gamma \vdash_{\mathrm{F}} \mathrm{E} : T$.
- For $t!x.m(\mathrm{E})$, the additional condition $\Gamma_{\mathrm{V}}(t) = \mathsf{Label}$ of rule EXT-ASYNC-L asserts that also t has been declared.
- For $m(\mathrm{E}; \mathrm{V})$, rule INT-SYNC gives us the judgment $\Gamma \vdash_{\mathrm{F}} \mathrm{E} : T$ such that $\mathit{match}(m, T \to \Gamma_{\mathrm{V}}(\mathrm{V}), \varsigma, \Gamma_{\mathcal{C}}(\Gamma_{\mathrm{V}}(\mathit{self})).\mathit{Mtd})$. It follows that $\Gamma_{\mathrm{V}}(\mathrm{V}) \neq \bot$.
- For $!m(\mathrm{E})$, rule INT-ASYNC asserts that $\Gamma \vdash_{\mathrm{F}} \mathrm{E} : T$.
- For $t!m(\mathrm{E})$, the additional condition $\Gamma_{\mathrm{V}}(t) = \mathsf{Label}$ of rule INT-ASYNC-L asserts that also t has been declared.
- For $t?(\mathrm{V})$, rule REPLY asserts that $\Gamma_P(t) \neq \emptyset$, so there are pending calls on label t. Consequently there must be an invocation on t in *Code* which has been type checked by either EXT-ASYNC-L or INT-ASYNC-L. In both cases, the condition $\Gamma_{\mathrm{V}}(t) = \mathsf{Label}$ guarantees that t has been declared. The matches for the pending calls on t imply that $\Gamma_{\mathrm{V}}(\mathrm{V}) \neq \bot$.
- For **await** g, induction over the construction of g shows that all variables are declared. The base cases are handled by AWAIT-b and AWAIT-t?.
- $\mathrm{S}_1; \mathrm{S}_2$ follows by the induction hypothesis. ■

Lemma 3 *If a synchronous call* $e.m(\mathrm{E}; \mathrm{V})$ *or* $m(\mathrm{E}; \mathrm{V})$ *is type-correct, then the corresponding asynchronous invocation* $!e.m(\mathrm{E})$ *or* $!m(\mathrm{E})$ *is also type-correct.*

Proof. Assume that $e.m(\mathrm{E}; \mathrm{V})$ is type-correct. The rule EXT-SYNC asserts that there is an interface I, a cointerface $Co \in \Gamma_{\mathcal{C}}(\Gamma_{\mathrm{V}}(\mathit{self})).\mathit{Contract}$ and a type T such that $\Gamma \vdash_{\mathrm{F}} e : I$, $\Gamma \vdash_{\mathrm{F}} \mathrm{E} : T$, and $\mathrm{match}(m, T \to \Gamma_{\mathrm{V}}(\mathrm{V}), Co, \Gamma_{\mathcal{I}}(I).\mathit{Mtd})$. It follows from the match that $\Gamma_{\mathrm{V}}(\mathrm{V}) \neq \bot$, so we have $\Gamma_{\mathrm{V}}(\mathrm{V}) \preceq \mathsf{Data}$ and consequently $\mathrm{match}(m, T \to \mathsf{Data}, Co, \Gamma_{\mathcal{I}}(I).\mathit{Mtd})$ holds. Rule EXT-ASYNC then asserts that the call $!e.m(\mathrm{E})$ is type-correct. The case for the internal calls $m(\mathrm{E}; \mathrm{V})$ and $!m(\mathrm{E})$ is similar. ■

It follows from Lemma 3 that a minimal requirement for successful binding is that an invocation can be bound with Data as the type of the actual out-parameter. This is reflected in the typing rules. This minimal requirement is

sufficient to show that the call may be bound correctly unless the return values from the asynchronous call are assigned to program variables.

Lemma 4 *Let* Γ *be a mapping family such that* $\Gamma_P = \epsilon$*. For any statement list* S *with a type judgment* $\Gamma \vdash_{\mathrm{S}} \mathrm{S} \langle\, \Delta \,\rangle$*, the set* Δ_P *contains exactly the labeled method invocations for which the return values may still be assigned to program variables after* S*.*

Proof. The proof is by induction over the length of S. If $\mathrm{S} = \epsilon$, we get $\Delta_P = \epsilon$. Assume as induction hypothesis that for $\Gamma \vdash_{\mathrm{S}} \mathrm{S} \langle\, \Delta \,\rangle$, Δ_P contains the invocations for which the return values may still be assigned to program variables after S. For the induction step, we prove that for $\Gamma \vdash_{\mathrm{S}} \mathrm{S}; s \langle\, \Delta + \Delta' \,\rangle$, $(\Delta + \Delta')_P$ contains the method invocations for which the return values may still be assigned to program variables after $\mathrm{S}; s$, by case analysis of s.

- For **skip**, $\mathrm{V} := \mathrm{E}$, $x :=$ **new** $C(\mathrm{E})$, and **await** g, there are no new method calls, so $\Delta'_P = \epsilon$ and $(\Delta + \Delta')_P$ contains exactly the method invocations that may need further analysis by the induction hypothesis.
- For $e.m(\mathrm{E}; \mathrm{V})$, $m(\mathrm{E}; \mathrm{V})$, $!e.m(\mathrm{E})$, and $!m(\mathrm{E})$, the type system allows the type-correctness of these statements to be verified directly. Consequently $\Delta'_P = \epsilon$ and $(\Delta + \Delta')_P$ contains the method invocations for which the return values may still be assigned to program variables by the induction hypothesis.
- For $t_i!e.m(\mathrm{E})$, an eventual reply statement may later impose a restriction on the type of the out-values. Note that previous calls pending on t are no longer accessible to such a reply statement. The effect of the EXT-ASYNC-L rule records only the new call as pending on t, yielding $\Delta'_P = [t \overset{P}{\mapsto} \{i\}]$. It follows that $(\Delta+\Delta')_P$ contains exactly the method invocations for which the return values may still be assigned to program variables after $\mathrm{S}; t_i!e.m(\mathrm{E})$.
- For $t_i!m(\mathrm{E})$, the case is similar. The effect of the INT-ASYNC-L rule yields $\Delta'_P = [t \overset{P}{\mapsto} \{i\}]$. It follows that $(\Delta + \Delta')_P$ contains exactly the the method invocations for which the return values may still be assigned to program variables after $\mathrm{S}; t_i!m(\mathrm{E})$.
- For $t?(v)$ the REPLY rule states that $\Gamma_P(t) \neq \emptyset$, so there are pending calls to t in Δ_P which are type checked with the new out-parameter type and removed from Δ_P. The effect of REPLY is $\Delta'_P = [t \overset{P}{\mapsto} \emptyset]$ and it follows from the induction hypothesis that $(\Delta+\Delta')_P$ contains exactly the invocations for which the return values may be assigned to program variables after $\mathrm{S}; t?(\mathrm{V})$.
- S_1; S_2 follows from the induction hypothesis. ■

It follows that all asynchronous invocations can be precisely type checked.

Lemma 5 *In a well-typed program, all method invocations have been verified as type-correct by the type analysis.*

Proof. We consider method invocations in the code S of an arbitrary method body in a class of the program, with the type judgment $\Gamma \vdash_{\mathrm{S}} \ \mathrm{S} \langle\, \Delta \,\rangle$. As $\Gamma_P = \epsilon$, Lemma 4 states that Δ_P contains exactly the method invocations that may need further type checking. As the entire method body S has been type checked, the invocations in Δ_P may be safely bound with the weakest possible type for actual out-variables, which has already been checked. ■

Note that for any typing environment Γ used in type checking a well-typed program of Section 3 and for any label t in this program, $\#\Gamma_P(t) \leq 1$ and there can be at most one reply statement in the program corresponding to any label t. Consequently, the following lemma holds for the language and type system considered in this section:

Lemma 6 *In a well-typed method, a signature and cointerface can be derived for every method invocation in the body, such that the invocation is covered.*

Proof. Let S be the code in a well-typed method such that $\Gamma \vdash_{\mathrm{S}} \ \mathrm{S} \langle\, \Delta \,\rangle$. The proof is by induction over S. For $\mathrm{S} = \epsilon$, the lemma holds trivially. Now let $\mathrm{S} = \mathrm{S}_0; s; \mathrm{S}_1$ and assume as induction hypothesis that well-typed signatures and cointerfaces have been derived for every invocation in S_0, we now show that this is also the case for $\mathrm{S}_0; s$. Let $\Gamma \vdash_{\mathrm{S}} \ \mathrm{S}_o; s \langle\, \Delta' \,\rangle$. The proof is by case analysis of s; only statements that invoke methods are discussed. (The remaining cases follow directly from the induction hypothesis.)

- For $o.m(\mathrm{E};\mathrm{V})$, rule EXT-SYNC provides the signature $T \to \Gamma_{\mathrm{V}}(V)$, where $\Gamma \vdash_{\mathrm{F}} \ e : T$, and cointerface Co, such that the invocation is covered.
- For $!o.m(\mathrm{E})$, rule EXT-ASYNC provides the signature $T \to \mathsf{Data}$, where $\Gamma \vdash_{\mathrm{F}} \ e : T$, and cointerface Co, such that the invocation is covered.
- For $t_i!o.m(\mathrm{E})$, rule EXT-ASYNC-L provides the signature $T \to \mathsf{Data}$, where $\Gamma \vdash_{\mathrm{F}} \ e : T$, and cointerface Co, for which the rule gives a match. This signature and cointerface Co are stored in $\Delta'_{Sig}(t_i)$. If there is a reply statement $t?(\mathrm{V})$ in S_1 before an invocation labeled t_j $(i \neq j)$, the return values from the call $t_i!o.m(\mathrm{E})$ will be assigned to V. Hence, the REPLY rule refines the signature to $T \to \Gamma_{\mathrm{V}}(\mathrm{V})$ in $\Delta_{Sig}(t_i)$, such that the invocation is covered by the new signature and Co. Otherwise, the return values from the call are not accessible and $\Delta_{Sig}(t_i) = \Delta'_{Sig}(t_i)$. In both cases, $\Delta_{Sig}(t_i)$ provides a signature and cointerface such that the invocation is covered.
- For $m(\mathrm{E};\mathrm{V})$, the INT-SYNC rule provides the signature $T \to \Gamma_{\mathrm{V}}(V)$, where $\Gamma \vdash_{\mathrm{F}} \ e : T$, and cointerface Co.
- For $!m(\mathrm{E})$, the INT-ASYNC rule provides the signature and cointerface.
- For $t_i!m(\mathrm{E})$, the INT-ASYNC rule provides a signature and cointerface. The signature may be refined by a reply statement as for $t_i!o.m(\mathrm{E})$. The call is covered by the signature and cointerface provided by $(\Gamma + \Delta)'_{Sig}(t_i)$ ■

The operational semantics of the language is defined in rewriting logic [57]. A rewrite theory is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$, where the signature Σ defines the function symbols of the language, E defines equations between terms, L is a set of labels, and R is a set of labeled rewrite rules. From a computational viewpoint, a rewrite rule $t \longrightarrow t'$ may be interpreted as a *local transition rule* allowing an instance of the pattern t to evolve into the corresponding instance of the pattern t'. Each rewrite rule describes how a part of a configuration can evolve in one transition step. If rewrite rules may be applied to non-overlapping subconfigurations, the transitions may be performed in parallel. Consequently, concurrency is implicit in rewriting logic (RL). A number of concurrency models have been successfully represented in RL [20,57], including Petri nets, CCS, Actors, and Unity, as well as the ODP computational model [63]. RL also offers its own model of object orientation [20].

Informally, a state configuration in RL is a multiset of terms of given types. Types are specified in (membership) equational logic (Σ, E), the functional sublanguage of RL which supports algebraic specification in the OBJ [34] style. When modeling computational systems, configurations may include the local system states. Different parts of the system are modeled by terms of the different types defined in the equational logic.

RL extends algebraic specification techniques with transition rules: The dynamic behavior of a system is captured by rewrite rules, supplementing the equations which define the term language. Assuming that all terms can be reduced to normal form, rewrite rules transform terms modulo the equations in E. Conditional rewrite rules are allowed, where the condition is formulated as a conjunction of rewrites and equations which must hold for the main rule to apply:

$$\textit{subconfiguration} \longrightarrow \textit{subconfiguration}\ \textbf{\textit{if}}\ \textit{condition}.$$

Rules in RL may be formulated at a high level of abstraction, closely resembling a structural operational semantics. In fact, structural operational semantics can be uniformly mapped into RL specifications [58].

3.5.1 System Configurations

Synchronous and asynchronous method calls are given a uniform representation in the operational semantics: Objects communicate by sending messages. Messages have the general form *message* **to** *dest* where *dest* is a single object or class, or a list of classes. The actual signature and cointerface of a method call, as derived during type checking (Lemma 6), are now assumed to be *included* as arguments to the method invocations of the runtime system. (After the signatures and cointerfaces of invocations have been included, the label

indices for asynchronous method invocations are erased.) If an object o_1 calls a method m of an object o_2, with actual type *Sig*, cointerface *Co*, and actual parameters E, and the execution of $m(Sig, Co, \text{E})$ results in the return values E′, the call is reflected by two messages $invoc(m, Sig, Co, (n\ o_1\ \text{E}))$ **to** o_2 and $comp(n, \text{E}')$ **to** o_1, which represent the invocation and completion of the call, respectively. In the asynchronous setting invocation messages will include the caller's identity, which ensures that completions can be transmitted to the correct destination. Objects may have several pending calls to another object, so the completion message includes a locally unique label value n, generated by the caller. Object activity is organized around a *message queue* which contains incoming messages and a *process queue* which contains suspended processes; i.e., remaining parts of method activations.

A state configuration is a multiset combining Creol objects, classes, and messages. (In order to increase the parallelism in the model, message queues could be external to object bodies [46, 47].) In RL, objects are commonly represented by terms of the type $\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$ where O is the object's identifier, C is its class, the a_i's are the names of the object's attributes, and the v_i's are the corresponding values [20]. We adopt this form of presentation and define Creol objects and classes as RL objects. Let a process be a pair consisting of a sequence of program statements and a local state, given by a *mapping* which binds program variables to values of their declared types. Omitting RL types, a Creol object is represented by an RL object $\langle Ob \mid Cl, Att, Pr, PrQ, EvQ, Lab \rangle$, where Ob is the object identifier, Cl the class name, Att the object state, Pr the active process, PrQ a multiset of suspended processes, EvQ a multiset of unprocessed messages, and Lab of type Label is the method call identifier, respectively. Thus, the object identifier Ob and the generated local label value provide a globally unique identifier for each method call.

At runtime, classes are represented by RL objects $\langle Cl \mid Par, Att, Mtds, Tok \rangle$, where Cl is the class name, Par and Att are lists of parameter and attribute declarations, $Mtds$ is a multiset of methods, and Tok is an arbitrary term of sort Label. A method has a name, signature, cointerface, in-parameter, and body. When an object needs a method, it is bound to a definition in the $Mtds$ multiset of its class. In RL's object model [20], classes are not represented explicitly in the system configuration. This leads to ad hoc mechanisms to handle object creation, which we avoid with explicit class representations. The Creol construct *new* $C(\text{E})$ creates a new object with a unique object identifier, attributes as listed in the class parameter list and in Att, and places the code from the *run* method in Pr. An *initial (state) configuration* consists of class representations and an initial **new** message.

3.5.2 Executions

An *execution* of a program P is a sequence of state configurations such that there is a rewrite step in the operational semantics between every two consecutive configurations. The operational semantics is given in Figures 8 and 9. There are three main kinds of rewrite rules:

- *Rules that execute code from the active process:* For every program statement there is at least one rule. For example, the assignment rule R1 for the statement $\text{V} := \text{E}$ binds the values of the expression list E to the list V of local and object variables.
- *Rule R5 suspends the active process:* When an active process guard evaluates to false, the process and its local variables are suspended, leaving *Pr* empty.
- *Rule R6 activates a suspended process:* If *Pr* is empty, suspended processes may be activated. The rule selects an arbitrary suspended and enabled process for activation.

In addition, *transport rules* (R11 and R17) move messages into the message queues, representing network flow. The rules are now briefly presented. Auxiliary functions are defined in equational logic and are therefore evaluated in between the state transitions [57]; e.g., the equation $(\epsilon := \epsilon) = \epsilon$ removes empty assignments. Irrelevant attributes are ignored in the style of Full Maude [20]. A detailed discussion may be found in [47].

Whitespace is used as the constructor of multisets, such as *PrQ* and *EvQ*, as well as variable and expression lists, whereas semicolon (with ϵ as left and right identity) is used as the constructor of lists of statements in order to improve readability. The **skip** statement is understood as ϵ. As before, + is the constructor for mappings. In the assignment rule R1, a list of expressions is evaluated and bound to a list of program variables. The auxiliary function *eval* evaluates an expression in a given *state*; the equations for the functional language $\mathcal{F}$ extend state lookup, given by

$$
\begin{aligned}
&\mathit{eval}(\epsilon, \text{L}) = \epsilon \\
&\mathit{eval}(v, \text{L} + [v' \mapsto d]) = \textbf{if } v = v' \textbf{ then } d \textbf{ else } \mathit{eval}(v, \text{L}) \textbf{ fi} \\
&\mathit{eval}(v\ \text{V}, \text{L}) = (\mathit{eval}(v, \text{L})\ \mathit{eval}(\text{V}, \text{L}))
\end{aligned}
$$

In the object creation rules R2 and R3, an object state is constructed from the class parameters and attribute list, an object identifier for the new object is constructed, and the *run* method is synchronously invoked. Let *self* of type *Any* be the self reference in a runtime object. The object identifier $(C; n)$ is, by the typing rule NEW, typed by an interface I such that $I \preceq \mathit{Any}$. The parameter is represented as a typed list of variables to accommodate the assignment rule. In order to apply R1 to the initialization of the object state, a *type erasure* function on attribute lists is introduced, defined recursively by $(\epsilon)\downarrow = \epsilon$ and $(\text{V}{:}\,T = \text{E}; \text{S})\downarrow = \text{V} := \text{E}; (\text{S})\downarrow$. New object identifiers are created

$$\text{(R1)}\quad \langle o : Ob \mid Att : \text{A}, Pr : \langle (v\ \text{V} := e\ \text{E}); \text{S}, \text{L}\rangle\ \rangle \longrightarrow \textbf{if}\ v\ \textit{in}\ \text{L}\ \textbf{then}\ \langle o : Ob \mid Att : \text{A}, Pr : \langle (\text{V} := \text{E}); \text{S}, (\text{L} + [v \mapsto \textit{eval}(e, (\text{A}; \text{L}))])\rangle\ \rangle\ \textbf{else}\ \langle o : Ob \mid Att : (\text{A} + [v \mapsto \textit{eval}(e, (\text{A}; \text{L}))]), Pr : \langle (\text{V} := \text{E}); \text{S}, \text{L}\rangle\ \rangle\ \textbf{fi}$$

$$\text{(R2)}\quad \textit{new}\ C(\text{E})\ \langle C : Cl \mid Par : (\text{V}: T), Att : \text{A}', Tok : n\rangle \longrightarrow \langle C : Cl \mid Par : (\text{V}: T), Att : \text{A}', Tok : \textit{next}(n)\rangle\ \langle (C; n): Ob \mid Cl: C, Att: \epsilon, Pr: \langle ((\textit{self} : Any = (C; n); \text{V}: T := \text{E}; \text{A}')\downarrow; \textit{run}), \epsilon\rangle, PrQ: \epsilon, EvQ : \epsilon, Lab : 1\rangle$$

$$\text{(R3)}\quad \langle o : Ob \mid Att : \text{A}, Pr : \langle (v := \textit{new}\ C(\text{E}); \text{S}), \text{L}\rangle\ \rangle\ \langle C : Cl \mid Par : (\text{V}: T), Att : \text{A}',\ Tok : n\rangle \longrightarrow \langle o: Ob \mid Att: \text{A}, Pr: \langle (v := (C; n); \text{S}), \text{L}\rangle\ \rangle\ \langle (C; n): Ob \mid Cl: C, Att : \epsilon, Pr: \langle ((\textit{self} : Any = (C; n); \text{V}: T = \textit{eval}(\text{E}, (\text{A}; \text{L})); \text{A}')\downarrow; \textit{run}), \epsilon\rangle, PrQ: \epsilon, EvQ: \epsilon, Lab : 1\rangle\ \langle C: Cl \mid Par : (\text{V}: T), Att: \text{A}', Tok : \textit{next}(n)\rangle$$

$$\text{(R4)}\quad \langle o : Ob \mid Att : \text{A}, Pr : \langle \textit{await}\ g; \text{S}, \text{L}\rangle, EvQ : \text{Q}\rangle \longrightarrow \langle o : Ob \mid Att : \text{A}, Pr : \langle \text{S}, \text{L}\rangle, EvQ : \text{Q}\rangle\ \textbf{if}\ \textit{enabled}(g, (\text{A}, \text{L}), \text{Q})$$

$$\text{(R5)}\quad \langle o : Ob \mid Att : \text{A}, Pr : \langle \text{S}, \text{L}\rangle, PrQ : \text{W}, EvQ : \text{Q}\rangle \longrightarrow \langle o : Ob \mid Att : \text{A}, Pr : \langle \epsilon, \epsilon\rangle, PrQ : (\text{W}\ \langle \textit{clear}(\text{S}), \text{L}\rangle), EvQ : \text{Q}\rangle\ \textbf{if}\ \textit{not}\ \textit{enabled}(\text{S}, (\text{A}; \text{L}), \text{Q})$$

$$\text{(R6)}\quad \langle o : Ob \mid Att : \text{A}, Pr : \langle \epsilon, \text{L}'\rangle, PrQ : \langle \text{S}, \text{L}\rangle\ \text{W}, EvQ : \text{Q}\rangle \longrightarrow \langle o : Ob \mid Att : \text{A}, Pr : \langle \text{S}, \text{L}\rangle, PrQ : \text{W}, EvQ : \text{Q}\rangle\ \textbf{if}\ \textit{enabled}(\text{S}, (\text{A}, \text{L}), \text{Q})$$

$$\text{(R7)}\quad \langle o: Ob \mid Pr : \langle (r(Sig, Co, \text{E}; \text{V}); \text{S}), \text{L}\rangle, Lab : n\rangle \longrightarrow \langle o: Ob \mid Pr : \langle (!r(Sig, Co, \text{E}); n?(\text{V}); \text{S}), \text{L}\rangle, Lab : n\rangle$$

$$\text{(R8)}\quad \langle o : Ob \mid Att : \text{A}, Pr : \langle (t!r(Sig, Co, \text{E}); \text{S}), \text{L}\rangle, Lab : n\rangle \longrightarrow \langle o : Ob \mid Att : \text{A}, Pr : \langle (t := n; !r(Sig, Co, \text{E}); \text{S}), \text{L}\rangle, Lab : n\rangle$$

$$\text{(R9)}\quad \langle o : Ob \mid Att : \text{A}, Pr : \langle (!x.m(Sig, Co, \text{E}); \text{S}), \text{L}\rangle, Lab : n\rangle \longrightarrow \langle o : Ob \mid Att : \text{A}, Pr : \langle \text{S}, \text{L}\rangle, Lab : \textit{next}(n)\rangle\ \textit{invoc}(m, Sig, Co, (o\ n\ \textit{eval}(\text{E}, (\text{A}; \text{L}))))\ \textbf{to}\ \textit{eval}(x, (\text{A}; \text{L}))$$

Figure 8. An operational semantics in rewriting logic (1).

by concatenating tokens n from the unbounded set *Tok* to the class name. The identifier is returned to the object which initiated the object creation. Before the new object can be activated, its state must be initialized. This is done by assigning actual values to class parameters and then evaluating the attribute list. The rules R4, R5, and R6 for guards depend on an *enabledness* function. Let D denote a state and let the infix function *in* check whether a completion message corresponding to a given label value is in a message queue Q. The *enabledness* function is defined by induction over the construction of guards:

$$\textit{enabled}(t?, \text{D}, \text{Q}) = \textit{eval}(t, \text{D})\ \textit{in}\ \text{Q}$$
$$\textit{enabled}(b, \text{D}, \text{Q}) = \textit{eval}(b, \text{D})$$
$$\textit{enabled}(\textit{wait}, \text{D}, \text{Q}) = \textit{false}$$
$$\textit{enabled}(g \vee g', \text{D}, \text{Q}) = \textit{enabled}(g, \text{D}, \text{Q}) \vee \textit{enabled}(g', \text{D}, \text{Q})$$
$$\textit{enabled}(g \wedge g', \text{D}, \text{Q}) = \textit{enabled}(g, \text{D}, \text{Q}) \wedge \textit{enabled}(g', \text{D}, \text{Q})$$

$$\text{(R10)}\quad \begin{array}{l}\langle o : Ob \mid Att : \text{A}, Pr : \langle (!m(Sig, Co, \text{E}); \text{S}), \text{L}\rangle, Lab : n\rangle \\ \quad\longrightarrow\ \langle o : Ob \mid Att : \text{A}, Pr : \langle \text{S}, \text{L}\rangle, Lab : next(n)\rangle \\ invoc(m, Sig, Co, (o\ n\ eval(\text{E}, (\text{A}; \text{L})))) \textbf{ to } o\end{array}$$

$$\text{(R11)}\quad (msg \textbf{ to } o)\ \langle o : Ob \mid EvQ : \text{Q}\rangle \longrightarrow \langle o : Ob \mid EvQ : \text{Q}\ msg\rangle$$

$$\text{(R12)}\quad \begin{array}{l}\langle o : Ob \mid Cl : C, EvQ : \text{Q}\ invoc(m, Sig, Co, \text{E})\rangle \\ \quad\longrightarrow\ \langle o : Ob \mid Cl : C, EvQ : \text{Q}\rangle\ bind(m, Sig, Co, \text{E}, o) \textbf{ to } C\end{array}$$

$$\text{(R13)}\quad \begin{array}{l}\langle o : Ob \mid Pr : \langle (t\,?\,(\text{V}); \text{S}), \text{L}\rangle, EvQ : \text{Q}\ comp(n, \text{E})\rangle \\ \quad\longrightarrow\ \langle o : Ob \mid Pr : \langle (\text{V} := \text{E}; \text{S}), \text{L}\rangle, EvQ : \text{Q}\rangle \textbf{ if } n = eval(t, \text{L})\end{array}$$

$$\text{(R14)}\quad \begin{array}{l}\langle o : Ob \mid Pr : \langle (t\,?\,(\text{V}); \text{S}), \text{L}\rangle, PrQ : (\text{S}', \text{L}')\ \text{W}\rangle \\ \quad\longrightarrow\ \langle o : Ob \mid Pr : \langle \text{S}'; cont(eval(t, \text{L})), \text{L}'\rangle, PrQ : \langle await\ t\,?\,(\text{V}); \text{S}, \text{L}\rangle\ \text{W}\rangle \\ \textbf{if } eval(caller, \text{L}') = o \wedge eval(label, \text{L}') = eval(t, \text{L})\end{array}$$

$$\text{(R15)}\quad \begin{array}{l}\langle o : Ob \mid Pr : \langle cont(n), \text{L}\rangle, PrQ : \langle (await\ (t'?); \text{S}), \text{L}'\rangle\ \text{W}\rangle \\ \quad\longrightarrow\ \langle o : Ob \mid Pr : \langle \text{S}, \text{L}'\rangle, PrQ : \text{W}\rangle \textbf{ if } eval(t', \text{L}') = n\end{array}$$

$$\text{(R16)}\quad \begin{array}{l}(bind(m, Sig, Co, \text{E}, o) \textbf{ to } C)\ \langle C : Cl \mid Mtds : \text{M}\rangle \\ \qquad\longrightarrow\ \textbf{if } match(m, Sig, Co, \text{M}) \\ \qquad\qquad \textbf{then } (bound(get(m, \text{M}, \text{E})) \textbf{ to } o) \\ \qquad\qquad \textbf{else } (bind(m, Sig, Co, \text{E}, o) \textbf{ to } \epsilon) \textbf{ fi } \langle C : Cl \mid Mtds : \text{M}\rangle\end{array}$$

$$\text{(R17)}\quad (bound(w) \textbf{ to } o)\ \langle o : Ob \mid PrQ : \text{W}\rangle \longrightarrow \langle o : Ob \mid PrQ : \text{W}\ w\rangle$$

$$\text{(R18)}\quad \begin{array}{l}\langle o : Ob | Pr : (return(\text{V})); \text{P}, Lvar : \text{L}, Att : \text{A}\rangle \\ \quad\longrightarrow \langle o : Ob | Pr : \text{P}, Lvar : \text{L}, Att : \text{A}\rangle \\ comp(eval(label, \text{L}), eval(\text{V}, (\text{L}))) \textbf{ to } eval(caller, \text{L})\end{array}$$

Figure 9. An operational semantics in rewriting logic (2).

When a non-enabled guard is encountered in R5, the active process is suspended on the process queue. In this rule, the auxiliary function *clear* removes occurrences of *wait* from any leading guards. The enabledness predicate is extended to *statements* as follows:

$$\begin{array}{l}enabled(s; \text{S}, \text{D}, \text{Q}) = enabled(s, \text{D}, \text{Q}) \\ enabled(await\ g, \text{D}, \text{Q}) = enabled(g, \text{D}, \text{Q}) \\ enabled(s, \text{D}, \text{Q}) = true \qquad [\textbf{otherwise}]\end{array}$$

The **otherwise** attribute of the last equation states that this equation is taken when no other equation matches.

In R7, a synchronous call $r(Sig, Co, \text{E}; \text{V})$, where V is a list of variables and r is either m or $x.m$, is translated into an *asynchronous call*, $!r(Sig, Co, \text{E})$, followed by a blocking *reply statement*, $n?(\text{V})$, where n is the label value uniquely identifying the call. In R8 a labeled asynchronous call is translated into a label assignment and an unlabeled asynchronous call, which results in an invocation message in R9. Internal calls are treated as external calls to *self* in R10. Guarded calls are expanded to asynchronous calls and guarded replies, as defined in Section 3.1.

When an object calls a method, a message is emitted into the configuration (R9 or R10) and delivered to the callee (R11). Message overtaking is captured by the nondeterminism inherent in RL: invocation and completion messages sent by an object to another object in one order may arrive in any order. The call is bound by sending a *bind* message to the class of the object (R12). Note that for external calls, the class of the callee is first identified in this rule; consequently external method calls are virtually bound.

The *bind* message is handled by R16, which identifies the method m in the method multiset M of the class. The auxiliary predicate $match(m, Sig, Co, \mathrm{M})$ evaluates to true if m is declared in M with a signature Sig' and cointerface Co' such that $Sig' \preceq Sig$, $Co \preceq Co'$. Note that a *method-not-understood* error is represented by a *bind* message sent to an empty list of classes. Furthermore, the auxiliary function *get* returns a process with the method's code and local state (instantiating the method's in-parameters with the call's actual parameters E and binding local variable declarations to initial expressions) from the method multiset M of the class, and ensures that a completion message is emitted upon method termination. The values of the actual in-parameters, the caller, and the label value n are stored locally; the caller and label value are stored in the local variables *caller* and *label*. Finally, if the method is declared with formal out-parameters V, a special construct $return(\mathrm{V})$ is appended to the method code. The process w resulting from the binding is loaded into the internal process queue in R17. The special construct $return(\mathrm{V})$ is used in R18 to return a uniquely labeled completion message to the caller.

The reply statement fetches the return values corresponding to V from the completion message in the object's queue (R13). In the model, *EvQ* is a multiset; thus the rule matches any occurrence of $comp(n, \mathrm{E})$ in the queue. The use of rewrite rules rather than equations mimics distributed and concurrent processing of method lookup. Note the special construct $cont(n)$ in R14 and R15, which is used to control local calls in order to avoid deadlock in the case of self reentrance [47].

Example. We consider an execution sequence inspired by the example of Section 3.4.4. Let C' be a class similar to class C, but without an active *run* method. The runtime representation of class C' is given as

$$\begin{aligned}\langle C' : Cl \,|\, Par : \epsilon, Att : \epsilon, Mtds = \{&\langle run, \epsilon \to \epsilon, \varsigma, \epsilon, \langle return(\epsilon), \epsilon\rangle\rangle \\ &\langle m, \epsilon \to \mathsf{Bool}, B, \epsilon, \langle (x := \mathsf{true}, return(x)), x \mapsto d_{\mathrm{Bool}}\rangle\rangle \\ &\langle m, \epsilon \to \mathsf{Nat}, A, \epsilon, \langle (x := 0, return(x)), x \mapsto d_{\mathrm{Nat}}\rangle\rangle\}, Tok : 1\rangle.\end{aligned}$$

For convenience, we denote the method multiset of C' by M, $next(n)$ by $n+1$, and ignore equational reduction. Figure 10 presents an execution sequence in which an object of class C creates an instance of C' and makes an asynchronous call to the new object. The call $t!o.m()$ of the Creol code is expanded to $t!o.m(\epsilon \to \mathsf{Nat}, AB, \epsilon)$ after the type analysis. This call causes an invocation

$\langle C' : Cl \,|\, Par : \epsilon, Att : \epsilon, Mtds = \text{M}, Tok : 1\rangle$
$\langle (C;1) : Ob \,|\, Cl : C, Att : \epsilon, Pr : \langle (o := \textbf{new}\ C'; t!o.m(\epsilon \to \text{Nat}, AB, \epsilon); \textbf{await}\ t?; t?(x)),$
$(x \mapsto d_{\text{Nat}}, o \mapsto null, t \mapsto d_{\text{Label}})\rangle, PrQ : \epsilon, EvQ : \epsilon, Lab : 2\rangle$
$\longrightarrow$ R2
$\langle C' : Cl \,|\, Par : \epsilon, Att : \epsilon, Mtds = \text{M}, Tok : 2\rangle$
$\langle (C;1) : Ob \,|\, Cl : C, Att : \epsilon, Pr : \langle (o := (C';1); t!o.m(\epsilon \to \text{Nat}, AB, \epsilon); \textbf{await}\ t?; t?(x)),$
$(x \mapsto d_{\text{Nat}}, o \mapsto null, t \mapsto d_{\text{Label}})\rangle, PrQ : \epsilon, EvQ : \epsilon, Lab : 2\rangle$
$\langle (C';1) : Ob \,|\, Cl : C, Att : \epsilon, Pr : \langle run(\epsilon \to \epsilon, \varsigma, \epsilon); \epsilon\rangle, PrQ : \epsilon, EvQ : \epsilon, Lab : 1\rangle$
$\longrightarrow$ R1, $\longrightarrow$ R8, we omit the default invocation of the empty *run* method (e.g., **skip**) in $(C';1)$.
$\langle C' : Cl \,|\, Par : \epsilon, Att : \epsilon, Mtds = \text{M}, Tok : 2\rangle$
$\langle (C;1) : Ob \,|\, Cl : C, Att : \epsilon, Pr : \langle (t := 2; !o.m(\epsilon \to \text{Nat}, AB, \epsilon); \textbf{await}\ t?; t?(x)),$
$(x \mapsto d_{\text{Nat}}, o \mapsto (C';1), t \mapsto d_{\text{Label}})\rangle, PrQ : \epsilon, EvQ : \epsilon, Lab : 2\rangle$
$\langle (C';1) : Ob \,|\, Cl : C, Att : \epsilon, Pr : \langle \epsilon, \epsilon\rangle, PrQ : \epsilon, EvQ : \epsilon, Lab : 2\rangle$
$\longrightarrow$ R1, $\longrightarrow$ R9
$\langle C' : Cl \,|\, Par : \epsilon, Att : \epsilon, Mtds = \text{M}, Tok : 2\rangle$
$\langle (C;1) : Ob \,|\, Cl : C, Att : \epsilon, Pr : \langle \textbf{await}\ t?; t?(x), (x \mapsto d_{\text{Nat}}, o \mapsto (C';1), t \mapsto 2)\rangle, PrQ : \epsilon, EvQ : \epsilon, Lab : 3\rangle$
$\langle (C';1) : Ob \,|\, Cl : C, Att : \epsilon, Pr : \langle 1?(\epsilon), \epsilon\rangle, PrQ : \epsilon, EvQ : \epsilon, Lab : 2\rangle$
$invoc(m, \epsilon \to \text{Nat}, AB, (C;1)\ 2)$ **to** $(C';1)$
$\longrightarrow$ R11, $\longrightarrow$ R12
$\langle C' : Cl \,|\, Par : \epsilon, Att : \epsilon, Mtds = \text{M}, Tok : 2\rangle$
$\langle (C;1) : Ob \,|\, Cl : C, Att : \epsilon, Pr : \langle \textbf{await}\ t?; t?(x), (x \mapsto d_{\text{Nat}}, o \mapsto (C';1), t \mapsto 2)\rangle, PrQ : \epsilon, EvQ : \epsilon, Lab : 3\rangle$
$\langle (C';1) : Ob \,|\, Cl : C, Att : \epsilon, Pr : \langle \epsilon, \epsilon\rangle, PrQ : \epsilon, EvQ :, Lab : 2\rangle$
$bind(m, \epsilon \to \text{Nat}, AB, (C;1)\ 2)$ **to** C'
$\longrightarrow$ R16
$\langle C' : Cl \,|\, Par : \epsilon, Att : \epsilon, Mtds = \text{M}, Tok : 2\rangle$
$\langle (C;1) : Ob \,|\, Cl : C, Att : \epsilon, Pr : \langle \textbf{await}\ t?; t?(x), (x \mapsto d_{\text{Nat}}, o \mapsto (C';1), t \mapsto 2)\rangle, PrQ : \epsilon, EvQ : \epsilon, Lab : 3\rangle$
$\langle (C';1) : Ob \,|\, Cl : C, Att : \epsilon, Pr : \langle \epsilon, \epsilon\rangle, PrQ : \epsilon, EvQ : \epsilon, Lab : 2\rangle$
$bound(\langle (x := 0, return(x)), (caller \mapsto (C;1), label \mapsto 2, x \mapsto d_{\text{Nat}})\rangle)$ **to** $(C';1)$
$\longrightarrow$ R17, $\longrightarrow$ R6
$\langle C' : Cl \,|\, Par : \epsilon, Att : \epsilon, Mtds = \text{M}, Tok : 2\rangle$
$\langle (C;1) : Ob \,|\, Cl : C, Att : \epsilon, Pr : \langle \textbf{await}\ t?; t?(x), (x \mapsto d_{\text{Nat}}, o \mapsto (C';1), t \mapsto 2)\rangle, PrQ : \epsilon, EvQ : \epsilon, Lab : 3\rangle$
$\langle (C';1) : Ob \,|\, Cl : C, Att : \epsilon, Pr : \langle (x := 0, return(x)), (caller \mapsto (C;1), label \mapsto 2, x \mapsto d_{\text{Nat}})\rangle,$
$PrQ : \epsilon, EvQ : \epsilon, Lab : 2\rangle$
$\longrightarrow$ R1, $\longrightarrow$ R18
$\langle C' : Cl \,|\, Par : \epsilon, Att : \epsilon, Mtds = \text{M}, Tok : 2\rangle$
$\langle (C;1) : Ob \,|\, Cl : C, Att : \epsilon, Pr : \langle \textbf{await}\ t?; t?(x), (x \mapsto d_{\text{Nat}}, o \mapsto (C';1), t \mapsto 2)\rangle, PrQ : \epsilon, EvQ : \epsilon, Lab : 3\rangle$
$\langle (C';1) : Ob \,|\, Cl : C, Att : \epsilon, Pr : \langle \epsilon, (caller \mapsto (C;1), label \mapsto 2, x \mapsto 0)\rangle, PrQ : \epsilon, EvQ : \epsilon, Lab : 2\rangle$
$comp(2,0)$ **to** $(C;1)$
$\longrightarrow$ R10, $\longrightarrow$ R4
$\langle C' : Cl \,|\, Par : \epsilon, Att : \epsilon, Mtds = \text{M}, Tok : 2\rangle$
$\langle (C;1) : Ob \,|\, Cl : C, Att : \epsilon, Pr : \langle t?(x), (x \mapsto d_{\text{Nat}}, o \mapsto (C';1), t \mapsto 2)\rangle, PrQ : \epsilon, EvQ : comp(2,0), Lab : 3\rangle$
$\langle (C';1) : Ob \,|\, Cl : C, Att : \epsilon, Pr : \langle \epsilon, (caller \mapsto (C;1), label \mapsto 2, x \mapsto 0)\rangle, PrQ : \epsilon, EvQ : \epsilon, Lab : 2\rangle$
$\longrightarrow$ R13, since $t \mapsto 2$
$\langle C' : Cl \,|\, Par : \epsilon, Att : \epsilon, Mtds = \text{M}, Tok : 2\rangle$
$\langle (C;1) : Ob \,|\, Cl : C, Att : \epsilon, Pr : \langle x := 0, (x \mapsto d_{\text{Nat}}, o \mapsto (C';1), t \mapsto 2)\rangle, PrQ : \epsilon, EvQ : \epsilon, Lab : 3\rangle$
$\langle (C';1) : Ob \,|\, Cl : C, Att : \epsilon, Pr : \langle \epsilon, (caller \mapsto (C;1), label \mapsto 2, x \mapsto 0)\rangle, PrQ : \epsilon, EvQ : \epsilon, Lab : 2\rangle$
$\longrightarrow$ R1
$\langle C' : Cl \,|\, Par : \epsilon, Att : \epsilon, Mtds = \text{M}, Tok : 2\rangle$
$\langle (C;1) : Ob \,|\, Cl : C, Att : \epsilon, Pr : \langle \epsilon, (x \mapsto 0, o \mapsto (C';1), t \mapsto 2)\rangle, PrQ : \epsilon, EvQ : \epsilon, Lab : 3\rangle$
$\langle (C';1) : Ob \,|\, Cl : C, Att : \epsilon, Pr : \langle \epsilon, (caller \mapsto (C;1), label \mapsto 2, x \mapsto 0)\rangle, PrQ : \epsilon, EvQ : \epsilon, Lab : 2\rangle$

Figure 10. An example of an execution sequence. The representation of C as well as some intermediary states are omitted, $\longrightarrow$ RX denotes the application of rule RX.

to the C' object, of the method m with Nat output, which again causes a completion with the value 0. The object of class C assigns 0 to its local variable x and the execution terminates.

The soundness of the type system is established in this section. First, we define well-typed runtime objects, configurations, and executions. Then, we show that when applying rewrite rules to the final state of a well-typed execution, the execution remains well-typed. In particular, method-not-understood errors do not occur. Say that a runtime object is of a program P if it is an instance of a class defined in P.

Definition 10 Let $\langle o : Ob \mid Att : \text{A}, Pr : \langle \text{S}, \text{L} \rangle, PrQ : \langle \text{S}_1, \text{L}_1 \rangle \ldots \langle \text{S}_n, \text{L}_n \rangle \rangle$ represent a runtime object of a program P. If A, L, $\text{L}_1, \ldots, \text{L}_n$ are well-typed states in the typing environment of P, then the runtime object is *well-typed*.

An object has been *initialized* when the object state has been constructed and the object is ready to call *run*. Recall that method-not-understood errors are captured technically by *bind* messages with no destination address. Say that a configuration is of a program P if all objects in the configuration are of P.

Definition 11 In a *well-typed configuration* of a program P, there are no method-not-understood errors and every object in the configuration is a well-typed runtime object of P with a unique identity. A *well-typed initial configuration* of a program P is an initial configuration of a well-typed program P. A *well-typed execution* of a program P is an execution that starts in an initial configuration and in which every configuration is well-typed.

Note that all objects in a configuration of a well-typed execution follow the naming convention of the object creation rules R2 and R3. Furthermore, all messages in a configuration of a well-typed execution are generated directly or indirectly by a method call in a well-typed object; i.e., all invocation messages are generated from the asynchronous call statement (R9 and R10), all *bind* messages are generated from these invocation messages (R12), all *bound* messages result from *bind* messages (R16), and all completion messages result from method termination (R18).

Lemma 7 *Given an arbitrary Creol program P and a well-typed execution ρ of P. The execution of a statement $x :=$ **new** $C(\text{E})$ in the final configuration of ρ results in well-typed configurations of P while the new object is initialized.*

Proof. Let o be a runtime object executing $x :=$ **new** $C(\text{E})$ (by applying rule R3) in the final configuration ρ_i of ρ (and let o' denote o after executing the statement). Since ρ is well-typed, o is well-typed. Since P is well-typed, Lemma 2 asserts that $\Gamma_{\text{V}}(x)$ has a type J and the typing rule for object creation ensures that there is an interface $I \in \Gamma_{\mathcal{C}}(C).Impl; \Gamma_{\mathcal{C}}(C).Contract$ such that $I \preceq J$. Consequently, the new object reference $(C; n)$ may be typed by J and o' is well-typed.

It remains to show that object creation results in a new well-typed initialized runtime object of class C with a unique identifier. Let E' denote E evaluated in o. Given a runtime class representation $\langle C : Cl \mid Par : (\text{V}{:}\, T), Att : \text{A}, Tok : n\rangle$ in ρ_i, the configuration ρ_{i+1} includes a runtime object

$$\langle (C;n){:}\, Ob \mid Cl{:}\, C, Att : \epsilon, Pr : \langle((\text{V} : T = \text{E}'; \text{A})\downarrow; run), \epsilon\rangle, PrQ{:}\, \epsilon, EvQ : \epsilon, Lab : 1\rangle,$$

which is well-typed. All object identifiers in ρ_i have been constructed by applications of R2 and R3, as pairs consisting of a class identifier and an element of type Label. In particular, no identifiers for instances of other classes than C contain the class identifier C, all instances of class C may be ordered by the relation $<$ on Label, and for any instance $(C;n')$ of C in ρ_i we have $n' < n$. Consequently, $(C;n)$ is an unused identifier in ρ_i. As the application of R3 locks the class in the rewrite step from ρ_i to ρ_{i+1}, $(C;n)$ is a unique identifier in ρ_{i+1}. Assuming that other concurrent activity in the configuration preserves well-typedness, ρ_{i+1} is well-typed. We now show that object initialization preserves well-typedness; i.e., for some well-typed state σ the object reduces to

$$\langle (C;n){:}\, Ob \mid Cl{:}\, C, Att : \sigma, Pr : \langle run, \epsilon\rangle, PrQ{:}\, \epsilon, EvQ : \epsilon, Lab : 1\rangle$$

in a new well-typed configuration ρ_{i+j}. Note that the rewrite steps involved in object initialization, i.e., the repeated application of R1, are internal to $(C;n)$. Consequently, concurrent activity does not influence the initialization and for simplicity we assume that such activity preserves configuration well-typedness. For class C, rule CLASS asserts that $\Gamma \vdash_{\text{V}} \ Param; Var \ \langle\ \Delta\ \rangle$. It follows by induction over the length of *Var* that the assignment of initial expressions to the program variables in the list $(self : Any = (C;n); \text{V} : T = \text{E}'; \text{A})\downarrow$ is type-correct and results in a well-typed state σ which is defined for *self* and the variables declared in *Param*; *Var*.

For $Var = \epsilon$, we have $\Gamma_{\mathcal{F}}(self) = Any$ and $\Gamma_{\mathcal{F}}((C;n)) = I$ so $I \preceq Any$ and rule NEW asserts that $\Gamma \vdash_{\text{F}} \ \text{E} : T'$ with $T' \preceq T$. Since E is well-typed, we have $\Gamma \vdash_{\text{F}} \ \text{E}' : T''$ such that $T'' \preceq T'$. Since $T'' \preceq T' \preceq T$, the assignment $(\text{V}{:}\, T = \text{E}')\downarrow$ is type-correct. By induction over the length k of *Par*, we show that a well-typed state σ is built by the multiple assignment $(\text{V}{:}\, T = \text{E}')\downarrow$. For $k = 0$, there are no parameters and $\sigma = \epsilon$ is well-typed (as is ρ_{i+1}). For the induction step, assume that ρ_{i+k-1} is a well-typed configuration in which σ_{k-1} $(1 < k \leq n)$ is the well-typed state of $(C;n)$ constructed by assigning values to the variables $v_1, \ldots, v_{k-1}$. Rule PAR asserts that the variable name v_k of type T_k is new, so an assignment to v_k does not override a previous variable in σ_{k-1}. By rule NEW, the assignment $v_k := e'_k$ is type-correct, so the application of R1 results in a well-typed state $\sigma_{k-1} \cup \{v_i \mapsto e_i\}$. It follows that the object state $\sigma = \sigma_n$ is well-typed and defined for the variables V, and that ρ_{i+k} is a well-typed configuration.

For the induction step, assume $\Gamma \vdash_{\text{V}} \ Param; Var \ \langle\ \Delta\ \rangle$ such that the assignment list $(\text{A})\downarrow$ is type-correct, resulting in a well-typed state σ, defined for *self* and the variables declared in *Param*; *Var*. If $\Gamma + \Delta \vdash_{\text{V}} \ v{:}\, T = e \ \langle\ \Delta'\ \rangle$

for some type T, then $(\text{A}; v{:}T = e)\downarrow$ is also a type-correct assignment list, resulting in a well-typed state σ' defined for *self*, the variables declared in *Param*; *Var*, and for v. Since $\Gamma + \Delta \vdash_{\text{V}} \ v{:}T = e \ \langle\ \Delta'\ \rangle$, the variable name v is new, $\Gamma + \Delta \vdash_{\text{F}} \ e{:}T'$ such that $T' \preceq T$, and e reduces to a value e' such that $\Gamma + \Delta \vdash_{\text{F}} \ e'{:}T''$ and $T'' \preceq T' \preceq T$. It follows that applying R1 results in a well-typed state $\sigma' = \sigma \cup \{v \mapsto e'\}$. There are no other processes with local state in $(C; n)$, so the initialized runtime representation of $(C; n)$ is well-typed. Assuming that other concurrent rewrites in the transition from ρ_i to ρ_{i+j} preserve well-typedness, $\rho_{i+1}, \ldots, \rho_{i+j}$ are well-typed configurations. ■

Using the same argument, we can show that a **new** message in a well-typed initial configuration of a program P creates a well-typed configuration of P (by R2). It follows by Lemma 7 that any program variable typed by an interface I will, if not null, point to an object of a class which implements I.

Lemma 8 *Let P be an arbitrary program. If $\Gamma_{\mathcal{F}} \vdash P$, then every method invocation $!x.m(T_{in} \to T_{out}, Co, \text{E})$ or $!m(T_{in} \to T_{out}, Co, \text{E})$ in a well-typed configuration of P can be type-correctly bound at runtime to a method such that the return values from the method are of type T'_{out} and $T'_{out} \preceq T_{out}$, provided that x is not a null pointer.*

Proof. By Lemma 6, the signature $Sig = T_{in} \to T_{out}$ and cointerface Co of every invocation is derived by the type system. The proof considers the evaluation rule R9 for $!x.m(Sig, Co, \text{E})$ and R10 for $!m(Sig, Co, \text{E})$. Let E, x evaluate to E', x' such that $\Gamma(\text{E}') \preceq \Gamma(\text{E})$.For an external method call $!x.m(Sig, Co, \text{E})$, rule R9 creates an invocation message in the following rewrite step

$$\langle o : Ob \mid Pr : \langle (!x.m(Sig, Co, \text{E}); \text{S})\rangle, Lab : n\rangle$$
$$\longrightarrow\ \langle o : Ob \mid Pr : \langle \text{S}\rangle, Lab : next(n)\rangle\ invoc(m, Sig, Co, (o\ n\ \text{E}'))\ \textbf{to}\ x'.$$

Let C be the runtime class of x. After $invoc(m, Sig, Co, (o\ n\ \text{E}'))$ **to** x' eventually arrives at x' by application of R11, the application of R12 generates a message $bind(m, Sig, Co, (o\ n\ \text{E}'), x')$ **to** C. Lemma 7 asserts that if $\Gamma_{\text{V}}(x') = I$ for some interface I, then C implements I. It follows from the type analysis that there must be a signature Sig_I and cointerface Co_I for m in I and a signature $Sig_C = T'_{in} \to T'_{out}$ and cointerface Co_C for m in C such that

$$Sig_C \preceq Sig_I \preceq Sig \text{ and } Co \preceq Co_I \preceq Co_C,$$

so $T'_{out} \preceq T_{out}$. As the subtype relation is transitive the runtime *match* function succeeds and the application of R16 results in the rewrite step

$$(bind(m, Sig, Co, (o\ n\ \text{E}'), x')\ \textbf{to}\ C)\ \langle C : Cl \mid Mtds : \text{M}\rangle$$
$$\longrightarrow\ (bound(get(m, \text{M}, (o\ n\ \text{E}')))\ \textbf{to}\ x')\ \langle C : Cl \mid Mtds : \text{M}\rangle.$$

For an internal runtime invocation $!m(Sig, \varsigma, \text{E})$, we get $Sig_C \preceq Sig$ and $\varsigma \preceq \varsigma$ directly from rule INT-ASYNC and the runtime *match* function succeeds. ■

All invocations in Creol are expanded into asynchronous invocations at runtime and all *bind* messages in a well-typed execution are generated from these invocations. Consequently, Lemma 8 implies that for every *bind* message $bind(m, T_1 \to T_2, Co, \text{E}, o)$ **to** C in a well-typed execution, $\Gamma_{\text{V}}(\text{E}) \preceq T_1$ and the matching of $T_1 \to T_2$ and Co with the formal declaration of m in C succeeds. Since $\Gamma_{\text{V}}(\text{E}) \preceq T_1$, the instantiation of local variables for the new process results in a well-typed state. Thus, loading a new process into the process queue of a well-typed object by R17 results in a well-typed runtime object.

Lemma 9 *Given an arbitrary Creol program P such that $\Gamma_{\mathcal{F}} \vdash P$ and a well-typed execution ρ of P. The execution of a statement $t?(\text{V})$ in the final configuration of ρ results in a well-typed configuration of P.*

Proof. We consider an object executing the (enabled) statement $t?(\text{V})$ by applying R13 on the final configuration of a well-typed execution ρ of P. Rule REPLY asserts that there is a pending method invocation with label t, say $t_i!o.m(\text{E})$ for some index i, object o, method name m, and data E (where $\Gamma \vdash_{\text{F}} \text{E} : T$). We need to show that the runtime method lookup selects a method body such that the return values are of a subtype of the type $\Gamma_{\text{V}}(\text{V})$, in order to ensure that the object remains well-typed after applying R13.

By Lemma 6, the signature derived by the type analysis for every call is such that the call is covered. This signature $Sig = T \to \Gamma_{\text{V}}(\text{V})$ and cointerface Co are used by the runtime system, yielding $t_i!o.m(Sig, Co, \text{E})$. By Lemma 8 the signature $Sig' = T_{in} \to T_{out}$ of the method selected at runtime is such that $Sig' \preceq Sig$, so $T_{out} \preceq \Gamma_{\text{V}}(\text{V})$. Since ρ is well-typed, the return values E' assigned to the out-parameters of the call by 18 are such that $\Gamma(\text{E}') \preceq T_{out}$. Consequently, $\Gamma(\text{E}') \preceq \Gamma_{\text{V}}(\text{V})$ and the result of applying R13 is a well-typed configuration of P. ■

Theorem 10 (Type soundness) *All executions of programs starting in a well-typed initial configuration, are well-typed.*

Proof. We consider a well-typed execution ρ of a program P. The proof is by the induction over the length of $\rho = \rho_0, \rho_1, \ldots$. By assumption, ρ_0 is a well-typed initial configuration of P; i.e., ρ_0 consists of class representations and a **new** message. Only R2 is applicable to ρ_0 and, by Lemma 7, ρ_1 is well-typed.

For the induction step we show that, for any well-typed configuration ρ_i, the successor configuration ρ_{i+1} is well-typed by case analysis over the rewrite rules of the operational semantics. We first consider the reduction of an object $\langle o : Ob \mid Att : \text{A}, Pr : \langle s; \text{S}, \text{L} \rangle \rangle$ to $\langle o : Ob \mid Att : \text{A}', Pr : \langle s'; \text{S}, \text{L}' \rangle \rangle$ and then the remaining rewrite rules.

- For $s = (v\ \mathrm{V} := e\ \mathrm{E})$ and the application of R1, the execution of $(v\ \mathrm{V} := e\ \mathrm{E})$ reduces the statement to $\mathrm{V} := \mathrm{E}$. Since the program is well-typed, we can assume that $\Gamma_{\mathrm{V}}(v) = T_v$, $\Gamma \vdash_{\mathrm{F}} e : T$, $T \preceq T_v$, and the functional expression e reduces to e' with type $\Gamma \vdash_{\mathrm{F}} e' : T'$ such that $T' \preceq T$. By transitivity $T' \preceq T_v$ and ρ_{i+1} is well-typed.
- Consider $s =$ **new** $C(\mathrm{E})$ and the application of R3. Lemma 7 guarantees that the evaluation of object creation statements does not result in object representations which are not well-typed, so ρ_{i+1} and the successor configurations from the initialization of the new object are well-typed.
- For $s =$ **await** g, either R4 or R5 may be applied, depending on the enabledness of g. If g is enabled and R4 is applied, the state variables do not change and ρ_{i+1} is well-typed. If g is not enabled and R5 is applied, the active process with the well-typed state L is moved to *PrQ* without modifying the state variables. The state of the active process is ϵ and hence also well-typed. Consequently, ρ_{i+1} is well-typed.
- For $s = r(\mathit{Sig}, \mathit{Co}, \mathrm{E}; \mathrm{V})$ and the application of R7, the state variables are not changed and ρ_{i+1} is well-typed.
- For $s = t!r(\mathit{Sig}, \mathit{Co}, \mathrm{E})$ and the application of R8, the statement reduces to $t := n; !r(\mathit{Sig}, \mathit{Co}, \mathrm{E})$. As n is of type Label, $\Gamma \vdash_{\mathrm{S}} t := n$. The state variables are not changed and ρ_{i+1} is well-typed.
- For $s = !r(\mathit{Sig}, \mathit{Co}, \mathrm{E})$ and the application of R9 and R10, the state variables are not changed and ρ_{i+1} is well-typed.
- For $s = t?(\mathrm{V})$ and the application of R13, there is a message $\mathit{comp}(n\ \mathrm{E})$ in *EvQ* such that t is bound to n in o. It follows from Lemma 9 that $\Gamma(\mathrm{E}) \preceq \Gamma_{\mathrm{V}}(\mathrm{V})$. Consequently, ρ_{i+1} is well-typed and the further assignment of values E to variables V will preserve well-typedness.
- For $s = t?(\mathrm{V})$ and the application of R14, the active process with state L is suspended and a suspended process with state L′ is activated. Since ρ_i is well-typed both L and L′ are well-typed, and consequently ρ_{i+1} is well-typed.
- For $s = \mathit{cont}(n)$ and the application of R15, a process with state L′ is activated. Since ρ_i is well-typed, L′ is well-typed, and ρ_{i+1} is well-typed.
- For $s = \mathit{return}(\mathrm{E})$ and the application of R18, the state variables do not change and ρ_{i+1} is well-typed.
- For sequential composition $s = s_1; \mathrm{S}_1$, all cases are covered except $\epsilon; \mathrm{S}_1$, which trivially reduces to S_1 by the left identity of sequential composition without modifying state variables. Consequently ρ_{i+1} is well-typed.

We now consider the remaining rewrite rules.

- The application of R6 activates a suspended process with a state L. Since ρ_i is well-typed, L is well-typed and, consequently, ρ_{i+1} is well-typed.
- Applying R11 or R12 does not modify state variables, and ρ_{i+1} is well-typed.
- For R16, observe that since ρ is well-typed a message $\mathit{bind}(m, \mathit{Sig}, \mathit{Co}, \mathrm{E}, o)$ must have been generated from a message $\mathit{invoc}(m, \mathit{Sig}, \mathit{Co}, \mathrm{E})$ **to** o by application of R12 and that the message $\mathit{invoc}(m, \mathit{Sig}, \mathit{Co}, \mathrm{E})$ **to** o must

Syntactic categories.		*Definitions.*
g in Guard	v in Var	$g ::= \mathit{wait} \mid b \mid t? \mid g_1 \wedge g_2 \mid g_1 \vee g_2$
t in Label	s in Stm	$r ::= x.m \mid m$
m in Mtd	r in MtdCall	$\mathrm{S} ::= s \mid s; \mathrm{S}$
e in Expr	b in BoolExpr	$s ::= \mathbf{skip} \mid (\mathrm{S}) \mid \mathrm{V} := \mathrm{E} \mid v := \mathbf{new}\ C(\mathrm{E})$
x in ObjExpr		$\mid\ !r(\mathrm{E}) \mid t!r(\mathrm{E}) \mid t?(\mathrm{V}) \mid r(\mathrm{E};\mathrm{V}) \mid \mathbf{while}\ b\ \mathbf{do}\ \mathrm{S}\ \mathbf{od}$
		$\mid\ \mathbf{await}\ g \mid \mathbf{await}\ t?(\mathrm{V}) \mid \mathbf{await}\ r(\mathrm{E};\mathrm{V})$
		$\mid\ \mathrm{S}_1 \,\square\, \mathrm{S}_2 \mid \mathrm{S}_1 \Vert \mathrm{S}_2 \mid \mathbf{if}\ b\ \mathbf{then}\ \mathrm{S}_1\ \mathbf{else}\ \mathrm{S}_2\ \mathbf{fi}$

Figure 11. The language extended with constructs for local high-level control.

have been generated by R9 or R10 from a statement $!x.m(\mathit{Sig}, \mathit{Co}, \mathrm{E})$ or $!m(\mathit{Sig}, \mathit{Co}, \mathrm{E})$. Lemma 8 asserts that a call $!x.m(\mathit{Sig}, \mathit{Co}, \mathrm{E})$ or $!m(\mathit{Sig}, \mathit{Co}, \mathrm{E})$ in P can be type-correctly bound at runtime. Consequently, the conditional test $\mathit{match}(m, \mathit{Sig}, \mathit{Co}, \mathrm{M})$ will succeed in R16, resulting in a *bound* message. It follows that ρ_{i+1} is well-typed.

- For R17, since ρ is well-typed the message $\mathit{bound}(\langle \mathrm{S}, \mathrm{L} \rangle)$ **to** o comes from applying R16. Therefore the function *get* has instantiated the formal parameters V of some method m with actual values E such that the call is covered. Consequently $\Gamma(\mathrm{E}) \preceq \Gamma_{\mathrm{V}}(\mathrm{V})$ and L is well-typed. It follows that since o was well-typed in ρ_i, o is also well-typed in ρ_{i+1} and ρ_{i+1} is well-typed. ■

In a well-typed execution, objects only communicate by method calls. When a **new** $C(\mathrm{E})$ statement is evaluated in a well-typed configuration, no other objects of class C can be created in the same rewrite step, Consequently, the above theorem also applies to concurrent processes; i.e., the parallel reduction of a well-typed configuration ρ_i results in a well-typed configuration ρ_{i+1}.

4 Flexible High-Level Control Structures for Local Computation

Asynchronous method calls, as introduced in Section 3, add flexibility to method calls in the distributed setting because waiting activities may yield processor control to suspended and enabled processes. Further, this allows active and reactive behavior in a concurrent object to be naturally combined. However a more fine-grained control may be desirable, in order to allow different tasks within the same process to be selected depending on the order in which communication with other objects actually occur. For this purpose, additional composition operators between program statements are introduced: conditionals, while-loops, nondeterministic *choice*, and nondeterministic *merge*. The latter operators introduce high-level branching structures which allow the local computation in a concurrent object to take advantage of nondeterministic delays in the environment in a flexible way. By means of these operators, the local computation adapts itself to the distributed environment *without* yielding control to competing processes. For example, nondetermin-

$$\text{(MERGE)}\ \frac{\Gamma \vdash_{\mathrm{S}} \mathrm{s} \langle\, \Delta \,\rangle \quad \Gamma \vdash_{\mathrm{S}} \mathrm{s}' \langle\, \Delta' \,\rangle \quad \mathrm{Dom}(\Delta_P) \cap \mathrm{Dom}(\Delta'_P) = \emptyset}{\Gamma \vdash_{\mathrm{S}} \mathrm{s} \| \mathrm{s}' \langle\, \Delta + \Delta' \,\rangle}$$

$$\text{(NON-DET)}\ \frac{\Delta_{\mathrm{Sig}} \cup \Delta'_{\mathrm{Sig}} \neq \bot \quad \Gamma \vdash_{\mathrm{S}} \mathrm{s} \langle\, \Delta \,\rangle \quad \Gamma \vdash_{\mathrm{S}} \mathrm{s}' \langle\, \Delta' \,\rangle}{\Gamma \vdash_{\mathrm{S}} \mathrm{s} \,\Box\, \mathrm{s}' \langle\, (\Gamma + \Delta) \cup (\Gamma + \Delta') \,\rangle}$$

$$\text{(COND)}\ \frac{\Gamma \vdash_{\mathrm{F}} \phi : \mathsf{Bool} \quad \Gamma \vdash_{\mathrm{S}} \mathrm{s} \,\Box\, \mathrm{s}' \langle\, \Delta \,\rangle}{\Gamma \vdash_{\mathrm{S}} \textbf{if}\ \phi\ \textbf{then}\ \mathrm{s}\ \textbf{else}\ \mathrm{s}'\ \textbf{fi} \langle\, \Delta \,\rangle}$$

$$\text{(WHILE)}\ \frac{\Gamma \vdash_{\mathrm{F}} \phi : \mathsf{Bool} \quad \Gamma + \emptyset_P \vdash_{\mathrm{S}} \mathrm{s} \langle\, \Delta \,\rangle}{\Gamma \vdash_{\mathrm{S}} \textbf{while}\ \phi\ \textbf{do}\ \mathrm{s}\ \textbf{od} \langle\, \Delta_{\mathrm{Sig}} \,\rangle}$$

Figure 12. Typing of local high-level control structures.

istic choice may be used to encode interrupts for process suspension such as timeout and race conditions between competing asynchronous calls [47].

4.1 Syntax

Statements can be composed in different ways, reflecting the requirements to the internal control flow in the objects. Recall that unguarded statements are always enabled, and that reply statements $t?(\mathrm{v})$ may block. Let s_1 and s_2 be statements. Nondeterministic choice between statements s_1 and s_2, written $\mathrm{s}_1 \,\Box\, \mathrm{s}_2$, may compute s_1 once s_1 is ready or s_2 once s_2 is ready, and suspends if neither branch is enabled. (Remark that to avoid deadlock the semantics additionally will not commit to a branch which starts with a blocking reply statement.) Nondeterministic merge, written $\mathrm{s}_1 \| \mathrm{s}_2$, evaluates the statements s_1 and s_2 in some interleaved and enabled order, and suspends if neither branch is enabled. Control flow without potential processor release uses **if** and **while** constructs. The conditional **if** g **then** s_1 **else** s_2 **fi** selects s_1 if g evaluates to true and otherwise s_2, and the loop **while** g **do** s_1 **od** repeats s_1 until g is not true. Figure 11 gives the extended language syntax.

4.2 Typing

The type system introduced in Section 3 is now extended to account for choice, merge, conditional, and while statements. It is nondeterministic for choice statements which branch will be executed at runtime, and for merge statements the order in which the statements of the different branches are executed. Consequently, the branches must be type checked in the same typ-

ing environment. The typing environment resulting from a nondeterministic statement depends on the calls introduced and removed in each branch. The additional rules are given in Figure 12. We now define a union operator for typing environments (recall that $\Gamma_{\mathcal{I}}$, $\Gamma_{\mathcal{C}}$, and $\Gamma_{\mathcal{F}}$ are static tables and that Γ_{V} is not modified by the type checking of program statements).

Definition 12 Let Γ and Γ' be typing environments and let N and M be sets. The *union of typing environments* $\Gamma \cup \Gamma'$ is defined as $\Gamma_{\mathcal{F}} + \Gamma_{\mathcal{I}} + \Gamma_{\mathcal{C}} + \Gamma_{\mathrm{V}} + (\Gamma_{Sig} \cup \Gamma'_{Sig}) + (\Gamma_P \cup \Gamma'_P)$. The *union of signature mappings*, $\Gamma_{Sig} \cup \Gamma'_{Sig}$, is defined as follows:

$$
\begin{aligned}
&([t_i \overset{Sig}{\mapsto} \langle m, Co, T_1 \to T_2\rangle] + \Gamma_{Sig}) \cup ([t_i \overset{Sig}{\mapsto} \langle m, Co, T_1 \to T'_2\rangle] + \Gamma'_{Sig}) \\
&\qquad = [t_i \overset{Sig}{\mapsto} \langle m, Co, T_1 \to T_2\rangle] \oplus [t_i \overset{Sig}{\mapsto} \langle m, Co, T_1 \to T'_2\rangle] + (\Gamma_{Sig} \cup \Gamma'_{Sig}) \\
&([t_i \overset{Sig}{\mapsto} \langle m, Co, T_1 \to T_2\rangle] + \Gamma_{Sig}) \cup \Gamma'_{Sig} \\
&\qquad = [t_i \overset{Sig}{\mapsto} \langle m, Co, T_1 \to T_2\rangle] + (\Gamma_{Sig} \cup \Gamma'_{Sig}) \quad \text{if} \quad \Gamma'_{Sig}(t_i) = \bot \\
&\emptyset_{Sig} \cup \Gamma'_{Sig} = \Gamma'_{Sig}
\end{aligned}
$$

Let $\bot + \Gamma_{Sig} = \bot$. The operator $\oplus$ on signature mappings is defined as

$$
[t_i \overset{Sig}{\mapsto} \langle m, Co, T_1 \to T_2\rangle] \oplus [t_i \overset{Sig}{\mapsto} \langle m, Co, T_1 \to T'_2\rangle] = \begin{cases} [t_i \overset{Sig}{\mapsto} \langle m, Co, T_1 \to (T_2 \cap T'_2)\rangle] & \text{if} \quad T_2 \cap T'_2 \neq \bot \\ \bot \qquad [\,\textbf{otherwise}\,] \end{cases}
$$

The *union of pending mappings*, $\Gamma_P \cup \Gamma'_P$, is defined as follows:

$$
([t \overset{P}{\mapsto} N] + \Gamma_P) \cup ([t \overset{P}{\mapsto} M] + \Gamma'_P) = [t \overset{P}{\mapsto} N] \otimes [t \overset{P}{\mapsto} M] + (\Gamma_P \cup \Gamma'_P)
$$

The operator $\otimes$ on pending mappings is defined as

$$
[t \overset{P}{\mapsto} N] \otimes [t \overset{P}{\mapsto} M] = \begin{cases} [t \overset{P}{\mapsto} N \cup M] & \text{if} \quad N \neq \emptyset \wedge M \neq \emptyset \\ [t \overset{P}{\mapsto} \emptyset] \qquad [\,\textbf{otherwise}\,] \end{cases}
$$

In a merge statement $\mathrm{S}_1 \| \mathrm{S}_2$, both statement lists will be evaluated. The MERGE rule ensures that reply statements in the two branches do not correspond to the same pending call. Furthermore all asynchronous invocations introduced in a merge statement must have unique labels to avoid interference between calls in the two branches, caused by the interleaved execution of S_1 and S_2. We then have that for each labeled invocation, the invocation cannot be matched by a reply statement in another branch of the merge. For this purpose, the MERGE rule ensures that $\mathrm{Dom}(\Delta_P) \cap \mathrm{Dom}(\Delta'_P) = \emptyset$.

For nondeterministic choice, at most one of the two branches is evaluated. If only one branch has a reply statement which corresponds to a pending call in Γ then a reply statement corresponding to the same call is not allowed in

statements succeeding the nondeterministic choice, although the branch with the reply statement need not be chosen at runtime. Moreover, if there is a reply statement in each branch that corresponds to the same pending call in Γ, say $t?(\mathrm{V})$ and $t?(\mathrm{V}')$, the NON-DET rule ensures that the types of the out-parameters have a common subtype (i.e., $\Gamma_{\mathrm{V}}(\mathrm{V}) \cap \Gamma_{\mathrm{V}}(\mathrm{V}') \neq \bot$) in order to ensure a deterministic signature for the invocation. This property is ensured by $\Delta_{\mathrm{Sig}} \cup \Delta'_{\mathrm{Sig}} \neq \bot$, which compares the types of the actual out-parameters of the reply statements in the branches when they correspond to the same pending call in Γ. The subtype $\Gamma_{\mathrm{V}}(\mathrm{V}) \cap \Gamma_{\mathrm{V}}(\mathrm{V}')$ is then selected as the type of the out-parameter of the call. Note that if a label is reused in a new asynchronous invocation, the reply statement in this branch will refer to the new invocation and Data is used as the type for the out-parameters of the previous call in this branch. Furthermore, given two branches S and S′ in a nondeterministic choice, the type system ensures that a reply statement with label t can occur after the nondeterministic choice statement only if a call with label t is pending after the evaluation of either S or S′. This ensures that each reply statement corresponds to a call independent of the branch selection.

The WHILE rule ensures that reply statements in the body S of the while-loop must correspond to invocations in the same traversal of S. Similarly, calls initiated in S must have their corresponding reply statements within the same traversal of S. This is guaranteed by the fact that the WHILE rule does not update the typing environment.

Lemma 11 *Let Γ be a mapping family such that $\Gamma_P = \epsilon$. For any statement list* S *in the code of an arbitrary method body with a type judgment $\Gamma \vdash_{\mathrm{S}} \mathrm{S} \langle\, \Delta \,\rangle$, the set Δ_P contains exactly the labeled method invocations for which the return value may still be assigned to program variables after* S.

Proof. The proof is by induction over the length of S, similar to the proof of Lemma 4. For the old cases, the signatures and cointerfaces in a single branch of a statement list follow from Lemma 6. The new cases are now considered for the induction step. We assume given $\Gamma \vdash_{\mathrm{S}} \mathrm{S} \langle\, \Delta \,\rangle$ where Δ_P contains the labeled method invocations for which the return values may still be assigned to program variables after S. We prove that for a judgment $\Gamma \vdash_{\mathrm{S}} \mathrm{S}; s \langle\, \Delta + \Delta' \,\rangle$, where s is a nondeterministic choice, merge, conditional, or while statement, $(\Delta + \Delta')_P$ contains the labeled method invocations for which the return values may still be assigned to program variables after S; s.

- For the while statement **while** ϕ **do** S' **od**, the type system does not allow any updates on the typing environment, so $(\Delta + \Delta')_P = \Delta_P$.
- For nondeterministic choice $\mathrm{S}_1 \,\square\, \mathrm{S}_2$, the induction hypothesis gives us $\Gamma + \Delta \vdash_{\mathrm{S}} \mathrm{S}_1 \langle\, \Delta_1 \,\rangle$ and $\Gamma + \Delta \vdash_{\mathrm{S}} \mathrm{S}_2 \langle\, \Delta_2 \,\rangle$ such that the lemma holds.

  Case 1: S_1 contains a reply statement $t?(\mathrm{V}_1)$ with a label t corresponding to a call on t_i such that $i \in \Delta_P(t)$. There are two possibilities. First, a

reply statement with label t does not occur in S_2. The pending call is type checked with the new out-parameter type in S_1, and the update of the typing environment removes the pending calls; i.e., $(\Delta_1 \cup \Delta_2)_P(t) = \emptyset$. Signature and cointerface uniqueness is here immediate. Second, a reply statement $t?(\mathrm{V}_2)$ occurs in S_2. We must check that the reply statements yield a unique signature and cointerface. By the induction hypothesis, there is a unique signature candidate in each branch. $(\Delta_1 \cup \Delta_2)_{\mathrm{Sig}} \neq \bot$ evaluates to true if a compatible minimal type can be given for the two reply statements, which gives us a unique signature and cointerface. Note that if the reply statement in one branch refers to a new method invocation with the same label t in that branch, Data is used as the out-parameter type for the first call in that branch. The update of the typing environment removes the pending calls.

Case 2: S_1 introduces a new invocation with label t. The effect of the NON-DET rule records the pending call if there is also a pending call to t if S_2 is chosen. (This pending call after S_2 may either correspond to an invocation in S_2 or a pending call in Δ which has been overwritten in S_1.) The invocations from both branches are captured in the effect of the NON-DET rule $(\Delta + \Delta_1) \cup (\Delta + \Delta_2)$. It follows from the induction hypothesis that $\Delta + (\Delta + \Delta_1)_P \cup (\Delta + \Delta_2)_P$ contains the labeled invocations for which the return values may be assigned to program variables after $\mathrm{S}; \mathrm{S}_1 \,\square\, \mathrm{S}_2$.

- The conditional statement follows from the case for nondeterministic choice.
- For nondeterministic merge $\mathrm{S}_1 \| \mathrm{S}_2$ the induction hypothesis is that for $\Gamma + \Delta \vdash_{\mathrm{S}} \mathrm{S}_1 \langle\, \Delta_1 \,\rangle$ and $\Gamma + \Delta \vdash_{\mathrm{S}} \mathrm{S}_2 \langle\, \Delta_2 \,\rangle$, $(\Delta_1)_P$ and $(\Delta_2)_P$ contain exactly the labeled method invocations for which the return value may still be assigned to program variables after S_1 and S_2. The condition $\mathrm{Dom}((\Delta_1)_P) \cap \mathrm{Dom}((\Delta_2)_P) = \emptyset$, ensures that labels used in branches S_1 and S_2 of the merge statement are non-overlapping, so a call with label t cannot be matched by a reply statement in another branch. Consequently, Δ_1 and Δ_2 are disjoint, so $\Delta_1 + \Delta_2 = \Delta_2 + \Delta_1$ and the order in which the updates are applied to Δ is insignificant. It then follows directly from the induction hypothesis that $(\Delta + \Delta_1 + \Delta_2)_P$ contains exactly the labeled method invocations for which the return value may still be assigned to program variables after $\mathrm{S}; \mathrm{S}_1 \| \mathrm{S}_2$. ■

We show that every call is covered for the derived signature and cointerface.

Lemma 12 *In a well-typed method, a signature and cointerface can be derived for every method invocation in the body, such that the invocation is covered.*

Proof. The proof extends the proof of Lemma 6. Let S be the code of an arbitrary well-typed method, such that $\Gamma \vdash_{\mathrm{S}} \mathrm{S} \langle\, \Delta \,\rangle$. The proof is by induction over the length of S. Let $\mathrm{S} = \mathrm{S}_0; s$ and assume as induction hypothesis that well-typed signatures and cointerfaces have been derived for every invocation in $\Gamma \vdash_{\mathrm{S}} \mathrm{S}_0 \langle\, \Delta_{\mathrm{S}_0} \,\rangle$. We show that well-typed signatures and cointerfaces have been derived for every invocation in $\Gamma \vdash_{\mathrm{S}} \mathrm{S}_0; s \langle\, \Delta' \,\rangle$. Here, only the cases for choice, merge, conditional, and while statements are considered.

(R19) $\langle o : Ob \mid Att : \text{A}, Pr : \langle(\text{S}_1 \,\square\, \text{S}_2); \text{S}_3, \text{L}\rangle, EvQ : \text{Q}\rangle$
$\longrightarrow \langle o : Ob \mid Att : \text{A}, Pr : \langle \text{S}_1; \text{S}_3, \text{L}\rangle, EvQ : \text{Q}\rangle$ **if** $ready(\text{S}_1, (\text{A}; \text{L}), \text{Q})$

(R20) $\langle o : Ob \mid Att : \text{A}, Pr : \langle(\text{S}_1 \| \text{S}_2); \text{S}_3, \text{L}\rangle, EvQ : \text{Q}\rangle$
$\longrightarrow \langle o : Ob \mid Att : \text{A}, Pr : \langle(\text{S}_1 /\!/\!/ \text{S}_2); \text{S}_3, \text{L}\rangle, EvQ : \text{Q}\rangle$ **if** $ready(\text{S}_1, (\text{A}; \text{L}), \text{Q})$

(R21) $\langle o : Ob \mid Att : \text{A}, Pr : \langle((s; \text{S}_1) /\!/\!/ \text{S}_2); \text{S}_3, \text{L}\rangle, EvQ : \text{Q}\rangle$
$\longrightarrow$ **if** $enabled(s, (\text{A}; \text{L}), \text{Q})$
then $\langle o : Ob \mid Att : \text{A}, Pr : \langle s; (\text{S}_1 /\!/\!/ \text{S}_2); \text{S}_3, \text{L}\rangle, EvQ : \text{Q}\rangle$
else $\langle o : Ob \mid Att : \text{A}, Pr : \langle((s; \text{S}_1) \| \text{S}_2); \text{S}_3, \text{L}\rangle, EvQ : \text{Q}\rangle$ **fi**

(R22) $\langle o : Ob \mid Att : \text{A}, Pr : \langle(\textit{if } b \textit{ then } \text{S}_1 \textit{ else } \text{S}_2 \textit{ fi}; \text{S}_3), \text{L}\rangle\rangle$
$\longrightarrow$ **if** $eval(b, (\text{L}; \text{A}))$ **then** $\langle o : Ob \mid Att : \text{A}, Pr : \langle(\text{S}_1; \text{S}_3), \text{L}\rangle \,\rangle$
else $\langle o : Ob \mid Att : \text{A}, Pr : \langle(\text{S}_2; \text{S}_3), \text{L}\rangle \,\rangle$ **fi**

(R23) $\langle o : Ob \mid Att : \text{A}, Pr : \langle(\textit{while } b \textit{ do } \text{S}_1 \textit{ od}; \text{S}_2), \text{L}\rangle\rangle$
$\longrightarrow \langle o : Ob \mid Att : \text{A}, Pr : \langle(\textit{if } b \textit{ then } (\text{S}_1; \textit{while } b \textit{ do } \text{S}_1 \textit{ od}) \textit{ else } \epsilon \textit{ fi}); \text{S}_2, \text{L}\rangle \,\rangle$ **fi**

Figure 13. An operational semantics for local high-level control structures.

- For $s =$ **while** ϕ **do** S' **od**, by the induction hypotheses every invocation introduced in S_0 and S' has a well-typed signature and cointerface. Traversing S' does not modify the typing environment, signatures in $(\Delta_{\text{S}_0})_{Sig}$ are not refined after $\text{S}_0; s$. Well-typed signatures and cointerfaces for invocations in $\text{S}_0; s$ follow from the induction hypotheses and the invocations are covered.
- For $s = \text{S}_1 \,\square\, \text{S}_2$, by the induction hypothesis, the signature and cointerface of every invocation in S_1 and S_2 are derived. By rule NON-DET, if one of the two branches contains a reply statement to an invocation in S_0, the signature of the invocation in $(\Delta_{\text{S}_0})_{Sig}$ will be refined in Δ'_{Sig}. If both branches contain such reply statements, then the signature is refined in Δ'_{Sig} with a common subtype, such that independent of the branch being evaluated, the invocation is covered with the new signature and the cointerface.
- The conditional statement follows from the case for nondeterministic choice.
- For $s = \text{S}_1 \| \text{S}_2$, the signature and cointerface for every invocation in S_1 and S_2 is derived by the induction hypothesis. The signatures in $(\Delta_{\text{S}_0})_{Sig}$ may be refined if the corresponding reply statement is contained in the merge statement. By rule MERGE, reply statements in the two branches do not correspond to the same pending call, so the sets of labels used in the two branches are disjoint. Consequently, the signature can only be refined once in Δ'_{Sig}, and the invocation is covered by the new signature. Regardless of the order of the evaluation of branches, every call in $\text{S}_0; s$ is covered. ■

4.3 Operational Semantics

The operational semantics of Section 3.5 is extended with rules for the local control structures in Figure 13. The selection of a branch S_1 in a nondeterministic choice statement $\text{S}_1 \,\square\, \text{S}_2$ is modeled by R19. Combined with the associativity and commutativity of the $\square$ operator, this rule covers the selection

of any branch in a compound nondeterministic choice statement. Here, the *ready* predicate tests if a process is ready to execute; i.e., the process does not immediately need to wait for a guard to become true or for a completion message. The *ready* predicate is defined as follows:

$$\begin{array}{l} \mathit{ready}(s; \mathrm{S}, \mathrm{D}, \mathrm{Q}) = \mathit{ready}(s, \mathrm{D}, \mathrm{Q}) \\ \mathit{ready}(t?(\mathrm{V}), \mathrm{D}, \mathrm{Q}) = \mathit{eval}(t, \mathrm{D})\ \mathit{in}\ \mathrm{Q} \\ \mathit{ready}(s, \mathrm{D}, \mathrm{Q}) = \mathit{enabled}(s, \mathrm{D}, \mathrm{Q}) \qquad [\textbf{otherwise}] \end{array}$$

As long as neither S_1 nor S_2 is ready, the active process is blocked if enabled and suspended if not enabled. Consequently, selecting a branch which immediately blocks or suspends execution is avoided if possible.

The merge operator $\|$ interleaves the execution of two statement lists S_1 and S_2. A naive approach is to define merge in terms of the nondeterministic choice $\mathrm{S}_1; \mathrm{S}_2 \,\square\, \mathrm{S}_2; \mathrm{S}_1$. To improve efficiency, a more fine-grained interleaving is preferred. However, in order to comply with the suspension technique of the language, interleaving is only allowed at processor release points in the branches. An associative but not commutative auxiliary operator $/\!/\!/$ is introduced in R20 and R21. The latter rule has the following property: Whenever evaluation of the selected (left) branch leads to non-enabledness, execution has arrived at a suspension point and it is safe to pass control back to the $\|$ operator. Rule R20 for merge decides whether to block or select a branch. (Suspension is handled by R5.) The $\|$ operator is associative, commutative, and has identity element ϵ (i.e., $\epsilon \| \mathrm{S} = \mathrm{S}$). The operational semantics for conditionals (R22) and while-loops (R23) are as expected. Finally, the enabledness predicate is extended to nondeterministic choice and merge as follows:

$$\begin{array}{l} \mathit{enabled}(\mathrm{S} \,\square\, \mathrm{S}', \mathrm{D}, \mathrm{Q}) = \mathit{enabled}(\mathrm{S}, \mathrm{D}, \mathrm{Q}) \vee \mathit{enabled}(\mathrm{S}', \mathrm{D}, \mathrm{Q}) \\ \mathit{enabled}(\mathrm{S} \| \mathrm{S}', \mathrm{D}, \mathrm{Q}) = \mathit{enabled}(\mathrm{S}, \mathrm{D}, \mathrm{Q}) \vee \mathit{enabled}(\mathrm{S}', \mathrm{D}, \mathrm{Q}) \end{array}$$

4.4 Type Soundness

The soundness of the type system for the extended language is established in this section. By Lemma 12, the signature and cointerface of every method invocation is deterministically given by the extended type system. Consequently, Lemmas 8 and 9 hold for the extended language. We first show a property of the execution sequences of nondeterministic merge.

Lemma 13 *Let* $\mathrm{S}_1 \| \mathrm{S}_2$ *be well-typed in a typing environment* Γ *and let* S *be an interleaving of* S_1 *and* S_2*. Then* S *is well-typed in* Γ*.*

Proof. Let $\Gamma \vdash_{\mathrm{S}} \mathrm{S}_1 \langle \Delta_1 \rangle$ and $\Gamma \vdash_{\mathrm{S}} \mathrm{S}_2 \langle \Delta_2 \rangle$ such that $\Gamma \vdash_{\mathrm{S}} \mathrm{S}_1 \| \mathrm{S}_2 \langle \Delta_1 + \Delta_2 \rangle$. The proof is by induction over the length of S. For $\mathrm{S} = \epsilon$, S is trivially well-

typed. For the induction step, let $\mathrm{S}; s_i; \ldots; s_1$ be an interleaving of S_1 and S_2 such that $\Gamma \vdash_{\mathrm{S}} \mathrm{S}; s_i; \ldots; s_1 \langle\, \Delta \,\rangle$. We prove that $\mathrm{S}; s; s_i; \ldots; s_1$ is well-typed if $(\mathrm{S}_1; s) \| \mathrm{S}_2$ is well-typed. Let $\Gamma \vdash_{\mathrm{S}} \mathrm{S}_1; s \langle\, \Delta_1' \,\rangle$ such that $\Gamma \vdash_{\mathrm{S}} (\mathrm{S}_1; s) \| \mathrm{S}_2 \langle\, \Delta_1' + \Delta_2 \,\rangle$. (The case for $\mathrm{S}_2; s$ is similar.) The proof proceeds by induction over i and by case analysis over s. Observe that $s_i; \ldots; s_1$ $(i \geq 0)$ is a tail sequence from S_2 and that for well-typedness, only the statements $t!p(\mathrm{E})$, $t?(\mathrm{V})$, and **await** t actually depend on the dynamically decided typing environment. We consider $s = t?(\mathrm{V})$.

- For $i = 0$, we have the interleaving $\mathrm{S}; s$. Since $\Gamma \vdash_{\mathrm{S}} \mathrm{S}_1; s \langle\, \Delta_1' \,\rangle$, it follows that $(\Gamma + \Delta_1)_{\mathrm{V}}(\mathrm{V}) \neq \bot$ and $(\Gamma + \Delta_1)_P(t) \neq \emptyset$, and hence $(\Gamma + \Delta_1)_{\mathrm{V}}(t) = \mathsf{Label}$. Since $(\Gamma + \Delta_1)_{\mathrm{V}} = (\Gamma + \Delta)_{\mathrm{V}}$, it follows that t and V are declared variables in $\Gamma + \Delta$. Since $\Gamma \vdash_{\mathrm{S}} \mathrm{S}_1 \| \mathrm{S}_2 \langle\, \Delta_1 + \Delta_2 \,\rangle$, $\mathrm{Dom}((\Delta_1)_P) \cap \mathrm{Dom}((\Delta_2)_P) = \emptyset$. Consequently, $(\Gamma + \Delta_1)_P(t) = (\Gamma + \Delta_1 + \Delta_2)_P(t) = (\Gamma + \Delta)_P(t)$ and $\mathrm{S}; s$ is well-typed in Γ. The case for **await** t is similar, the other cases are straightforward.
- For $i+1$, the induction hypothesis is that $\mathrm{S}; s_{i+1}; s; s_i; \ldots; s_1$ is a well-typed interleaving and s_{i+1} is a statement from S_2. Since s_{i+1} is a statement from S_2, $\mathrm{Dom}((\Delta_1)_P)(t) \neq \bot$, and $\mathrm{Dom}((\Delta_1)_P) \cap \mathrm{Dom}((\Delta_2)_P) = \emptyset$, exchanging s and s_{i+1} does not affect well-typedness and $\mathrm{S}; s; s_{i+1}; s_i; \ldots; s_1$ is well-typed.

The cases for $t!p(\mathrm{E})$ and **await** t are similar, the others straightforward. ■

It follows from Lemma 13 that if $(\mathrm{S}_1; \mathrm{S}_2) \| \mathrm{S}_3$ is well-typed in a typing environment Γ, then so is $\mathrm{S}_1; (\mathrm{S}_2 \| \mathrm{S}_3)$.

Theorem 14 (Type soundness) *All executions of programs starting in a well-typed initial configuration, are well-typed.*

Proof. We consider a well-typed execution ρ of a program P. The proof is by the induction over the length of $\rho = \rho_0, \rho_1, \ldots$ and extends the proof of Theorem 10. We show that for any well-typed configuration ρ_i, the successor configuration ρ_{i+1} is also well-typed, by case analysis over the rewrite rules. Here, only the new rewrite rules are considered, i.e., the reduction of a well-typed runtime object $\langle o : Ob \mid Att : \mathrm{A}, Pr : \langle s; \mathrm{S}, \mathrm{L} \rangle \rangle$ to $\langle o : Ob \mid Att : \mathrm{A}', Pr : \langle s'; \mathrm{S}, \mathrm{L} \rangle \rangle$ where s is a nondeterministic choice, merge, conditional, or while statement. (The old cases are covered by the proof of Theorem 10.) For the induction hypothesis, we assume that the type soundness property holds for well-typed branches S_1 and S_2 of s.

- For $s = \mathrm{S}_1 \,\square\, \mathrm{S}_2$, the application of R19 reduces s to either S_1 or S_2. As the program is well-typed, $\Gamma \vdash_{\mathrm{S}} \mathrm{S}_1$ and $\Gamma \vdash_{\mathrm{S}} \mathrm{S}_2$. The state variables do not change and ρ_{i+1} is well-typed.
- Consider $s = \mathrm{S}_1 \parallel \mathrm{S}_2$. If $\mathrm{S}_1 = \epsilon$ then $\mathrm{S}_1 \parallel \mathrm{S}_2 = \mathrm{S}_2$, for which type soundness holds by assumption. Now assume that S_1 is nonempty. By R20, $\mathrm{S}_1 \parallel \mathrm{S}_2$ reduces to $\mathrm{S}_1 /\!/\!/ \mathrm{S}_2$, so ρ_{i+1} is well-typed since state variables do not change.
- For $\mathrm{S}_1 /\!/\!/ \mathrm{S}_2$, we proceed by induction over the number n of rewrite steps

using R21 immediately preceding ρ_i. If $n = 0$, ρ_i must have been obtained by application of R20. By the induction hypothesis $\mathrm{S}_1 \parallel \mathrm{S}_2$ is well-typed and by Lemma 13 any $\mathrm{S}_1'; (\mathrm{S}_1'' \| \mathrm{S}_2)$ is well-typed. For the induction step, observe that the repeated application of R21 must eventually result in a statement $\mathrm{S}_1'' \| \mathrm{S}_2$ in a state $\rho_{i+n'}$ $(n' \geq n)$ after executing S_1'. It follows from the induction hypothesis and Lemma 13 that all states $\rho_i, \ldots, \rho_{i+n'}$ are well-typed.

- For $s =$ **if** ϕ **then** S_1 **else** S_2 **fi**, the application of R22 reduces s to either S_1 or S_2. Since the program is well-typed, we know that $\Gamma \vdash_{\mathrm{F}} \phi : \mathsf{Bool}$, $\Gamma \vdash_{\mathrm{S}} \mathrm{S}_1$, and $\Gamma \vdash_{\mathrm{S}} \mathrm{S}_2$. The functional expression ϕ will be successfully reduced to a Boolean value, as expected by the conditional test. The state variables do not change and ρ_{i+1} is well-typed.
- For $s =$ **while** E **do** S_1 **od**, the application of R23 reduces s to $s' =$ **if** E **then** $(\mathrm{S}_1;$ **while** E **do** S_1 **od**$)$ **else** ϵ **fi**. The typing rule WHILE ensures that a traversal of S_1 does not modify the typing environment (Γ_{Sig} and Γ_P). Consequently, $\mathrm{S}_1;$ **while** E **do** S_1 **od** is well-typed. Moreover, since there is no interference between the different traversals of S_1, the conditions of rule NON-DET are satisfied and s' is also well-typed. The state variables do not change and ρ_{i+1} is well-typed. ■

5 An Extension with Multiple Inheritance

Many languages identify the subclass and subtype relations, in particular for parameter passing, although several authors argue that inheritance relations for code and for behavior should be distinct [5, 13, 22, 72]. From a pragmatic point of view, combining these relations leads to severe restrictions on code reuse which seem unattractive to programmers. From a reasoning perspective, the separation of these relations allows greater expressiveness while providing type-safety. In order to solve the conflict between unrestricted code reuse in subclasses, and behavioral subtyping and incremental reasoning control [54, 72], we use interfaces to type object variables and external calls. Multiple inheritance is allowed for both interfaces and classes. Whereas subinterfacing is restricted to a form of behavioral subtyping, subclassing is unrestricted in the sense that implementation claims (and class invariants) are not in general inherited. However, the mutual dependencies introduced by cointerfaces makes the inheritance of contracts necessary in subclasses.

A class describes a collection of objects with similar internal structure; i.e., attributes and method definitions. Class inheritance is a powerful mechanism for defining, specializing, and understanding the imperative class structures through code reuse and modification. Class extension and method redefinition are convenient both for the development and understanding of code. Calling superclass methods in a subclass method enables *reuse in redefined methods*, while method redefinition allows *specialization* in subclasses.

With distinct inheritance and subtyping hierarchies, class inheritance could allow a subset of the attributes and methods of a class to be inherited. However, this would require considerable work establishing invariants for parts of the superclass that appear desirable for inheritance, either anticipating future needs or while designing subclasses. The *encapsulation principle* for class inheritance states that it should suffice to work at the subclass level to ensure that the subclass is well-behaved when inheriting from a superclass: Code design as well as new proof obligations should occur in the subclass only. Situations that break this principle are called inheritance anomalies [56,61] (see also the fragile base class problem [60]). Reasoning considerations therefore suggest that all attributes and methods of a superclass are inherited, but method redefinition may violate the requirements of the interfaces of the superclass.

5.1 Syntax

A mechanism for multiple inheritance at the class level is now considered, where all attributes and methods of a superclass are inherited by the subclass, and where superclass methods may be redefined. In the syntax the keyword **inherits** is introduced followed by a list of instantiated class names $C(\text{E})$, where E provides the actual class parameters.

Let a class hierarchy be a directed acyclic graph of classes. Each class consists of lists of class parameters and instantiated class names (for superclasses), a set of attributes, and method definitions. Let a class C be *below* a class C' if C is C', or if C is a direct or indirect subclass of C' and *above* C' if C is C', or if C is a direct or indirect superclass of C'. The encapsulation provided by interfaces suggests that external calls to an object of class C are virtually bound to the closest method definition above C. However, the object may internally invoke methods of its superclasses. In the setting of multiple inheritance and overloading, ambiguities may occur when attributes or methods are accessed. A name conflict is *vertical* if a name occurs in a class and in one of its ancestors, and *horizontal* if the name occurs in distinct branches of the graph. Vertical name conflicts for method names are resolved in a standard way: the first definition matching the types of the actual parameters is chosen while ascending a branch of the inheritance tree. Horizontal name conflicts are resolved dynamically depending on the class of the object and the context of the call.

5.1.1 Qualified Names

Qualified names may be used to internally refer to an attribute or method in a class in a unique way. For this purpose, we adapt the **qua** construct of

Syntactic categories.		*Definitions.*
g in Guard	v in Var	$g ::= wait \mid b \mid t? \mid g_1 \wedge g_2 \mid g_1 \vee g_2$
t in Label	s in Stm	$r ::= x.m \mid m \mid m@C \mid m < C$
m in Mtd	r in MtdCall	$\mathrm{S} ::= s \mid s; \mathrm{S}$
e in Expr	b in BoolExpr	$s ::= \mathbf{skip} \mid (\mathrm{S}) \mid \mathrm{V} := \mathrm{E} \mid v := \mathbf{new}\ C(\mathrm{E})$
x in ObjExpr		$\mid\ !r(\mathrm{E}) \mid t!r(\mathrm{E}) \mid t?(\mathrm{V}) \mid r(\mathrm{E};\mathrm{V}) \mid \mathbf{while}\ b\ \mathbf{do}\ \mathrm{S}\ \mathbf{od}$
		$\mid \mathbf{await}\ g \mid \mathbf{await}\ t?(\mathrm{V}) \mid \mathbf{await}\ r(\mathrm{E};\mathrm{V})$
		$\mid \mathrm{S}_1 \,\square\, \mathrm{S}_2 \mid \mathrm{S}_1 \Vert \mathrm{S}_2 \mid \mathbf{if}\ b\ \mathbf{then}\ \mathrm{S}_1\ \mathbf{else}\ \mathrm{S}_2\ \mathbf{fi}$

Figure 14. A language extension for static and virtual internal calls.

Simula [25] to the setting of multiple inheritance with virtual binding. For an attribute v or a method m declared in a class C, we denote by $v@C$ and $m@C$ the qualified names which provide static references to v and m. By extension, if v or m is *not* declared in C, but inherited from the superclasses of C, the qualified reference $m@C$ binds as an unqualified reference m above C.

Attribute names are not visible through an object's external interfaces. Consequently, attribute names should not be merged if inheritance leads to name conflicts and attributes of the same name should be allowed in different classes of the inheritance hierarchy [71]. In order to allow the reuse of attribute names, these are always expanded into qualified names. This is desirable in order to avoid runtime errors that may occur if methods of superclasses assign to overloaded attributes. This convention has the following consequence: unlike C++, there is no duplication of attributes when branches in the inheritance graph have a common superclass. Consequently, if multiple copies of the superclass' attributes are needed, one has to rely on delegation techniques.

5.1.2 Instantiation of Attributes

At object creation time, attributes are collected from the object's class and superclasses. Recall that an attribute in a class C is declared by $x : T = e$, where x is the name of the attribute, T its type, and e its initial value. The expression e may refer to the values of inherited attributes by means of qualified references, in addition to the values of the class parameter variables V. The initial state values of an object of class C then depend on the actual parameter values bound to V. These may be passed as actual class parameter values to inherited classes in order to derive values for the inherited attributes, which in turn may be used to instantiate the locally declared attributes.

5.1.3 Accessing Inherited Attributes and Methods

If C is a superclass of C' we introduce the syntax $m@C(\mathrm{E};\mathrm{V})$ for synchronous internal invocation of a method above C in the inheritance graph, and similarly for external and asynchronous invocations. These calls may be bound without

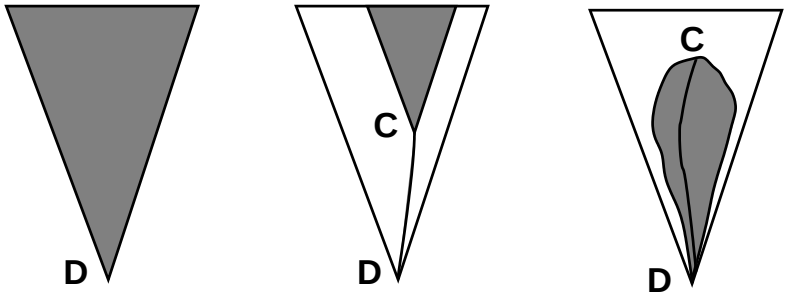

Figure 15. Binding calls to m, $m@C$, and $m < C$ from class D.

knowing the exact class of *self*, so they are called *static*, in contrast to calls without @, called *virtual*. We assume that attributes have unique names in the inheritance graph; this may be enforced at compile-time by extending each attribute name x with the name of the class C in which it is declared, which implies that attributes are bound statically. Consequently, a method declared in a class C may only access attributes declared above C. In a subclass, an attribute x of a superclass C is accessed by the qualified reference $x@C$. The extended language syntax is given in Figure 14.

5.2 Virtual Binding

When multiple inheritance is included in the language, it is necessary to reconsider the mechanism for virtual binding. A method declaration in a class C is *constrained* by C' if C is required to be below C'. The virtual binding of method calls is now explained. At runtime, a call to a method of an object o is always bound above the class of o. Let m be a method declared in an interface I and let o be an instance of a class C implementing I. There are two cases:

(1) m is called *externally*, in which case C is not statically known. In this case, C is dynamically identified as the class of o.
(2) m is called *internally* from C', a class above the actual class C of o. In this case static analysis identifies the call with a declaration of m above C', say in C''. Consequently we let the call be constrained by C'', and compilation replaces the reference to m with a reference to $m < C''$.

The dynamically decided context of a call may eliminate parts of the inheritance graph above the actual class of the callee with respect to the binding of a specific call. If a method name is ambiguous within the dynamic constraint, we assume that any solution is acceptable. For a natural and simple model of priority, the call is bound to the first matching method definition above C, in a left-first depth-first order as given by the textual declarations of the instantiated class names. (An arbitrary order may be obtained by replacing the list of instantiated class names by a multiset.) The three forms of method binding are illustrated in Figure 15.

5.3 Example: Combining Authorization Policies

In a database containing sensitive information and different authorization policies, the information returned for a request depends on the clearance level of the agent making the request. Let *Agent* denote the interface of arbitrary agents and *Auth* an authorization interface with methods *grant*(x), *revoke*(x), and *auth*(x) for agents x. The two classes *SAuth* and *MAuth*, which both implement *Auth*, provide single and multiple authorization policies, respectively. *SAuth* authorizes one agent at a time and *MAuth* authorizes multiple agents. The method *grant*(x) returns when x becomes authorized, and authorization is removed by *revoke*(x). The method *auth*(x) suspends until x is authorized, and *delay* returns once no agent is authorized.

```
class SAuth implements Auth
begin
 var gr: Agent = null
 op delay == await (gr = null)
 op grant(in x:Agent)== delay(); gr:=x
 op auth(in x:Agent)== await (gr=x)
 op revoke(in x:Agent) ==
  if gr = x then gr := null else skip fi
with Agent
 op grant == grant(caller)
 op revoke == revoke(caller)
 op auth == auth(caller)
end
```

```
class MAuth implements Auth
begin
 var gr: Set[Agent] = ∅
 op delay == await (gr = ∅)
 op grant(in x:Agent) == gr := gr ∪ {x}
 op auth(in x:Agent) == await (x ∈ gr)
 op revoke(in x:Agent) == gr := gr\{x}

with Agent
 op grant == grant(caller)
 op revoke == revoke(caller)
 op auth == auth(caller)
end
```

5.3.1 Authorization Levels

We now consider concurrent access to the database. *Low clearance* agents may share access to unclassified data while *high clearance* agents have unique access to (classified) data. Proper usage is defined by two interfaces, defining open and close operations at both access levels:

```
interface High
begin
with Agent
 op openH(out ok:Bool)
 op access(in k:Key out y:Data)
 op closeH
end
```

```
interface Low
begin
with Agent
 op openL
 op access(in k:Key out y:Data)
 op closeL
end
```

Not all agents are entitled to high authorization, so *openH* returns a Boolean.

Let a class *DB* provide the actual operations on the database. We assume given the internal operations *access*(**in** k:Key, level:Bool **out** y:Data), where

level defines the access level (high or low), and *clear*(**in** x:*Agent* **out** b:Bool) to give clearance to sensitive data for agent x. Any agent may get low access rights, while only agents cleared by the database may be granted exclusive high access. Consequently, the *MAuth* class authorizes low clearance and *SAuth* authorizes high clearance. Since the attribute *gr* in *SAuth* is implemented as an object identifier, only one agent is authorized full access at a time.

```
class HAuth implements High
 inherits SAuth, DB
begin
 op access(in x:Agent;k:Key out y:Data)
 == auth(x); await access@DB(k,high;y)
with Agent
 op openH(out ok:Bool) ==
     await clear(caller;ok);
     if ok then grant(caller) else skip fi
 op access(in k:Key out y:Data) ==
     access(caller,k; y)
 op closeH == revoke(caller)
end
```

```
class LAuth implements Low
 inherits MAuth, DB
begin
 op access(in x:Agent;k:Key out y:Data)
 == auth(x); await access@DB(k,low;y)
with Agent
 op openL == grant(caller)
 op access(in k:Key out y:Data) ==
     access(caller,k; y)
 op closeL == revoke(caller)
end
```

The code given here uses asynchronous calls whenever an internal deadlock would be possible. Thus, objects of the four classes above may respond to new requests even when used improperly, for instance when agent access is not initiated by open.

The database itself has no interface containing *access*, therefore all database access is through the *High* and *Low* interfaces. Notice also that objects of the *HAuth* and *LAuth* classes may not be used through the *Auth* interface. This would have been harmful for the authorization provided in the example. For instance, an external call to the *grant* method of a *HAuth* object could result in high *access* without clearance of the calling agent! This supports the approach not to inherit implementation clauses.

5.3.2 Combining Authorization Levels

High and low authorization policies may be combined in a subclass *HLAuth* which implements both interfaces, inheriting *LAuth* and *HAuth*.

```
class HLAuth implements High, Low
 inherits LAuth, HAuth
begin
with Agent
 op access(in k:Key out y:Data) ==  if caller=gr@SAuth
  then access@HAuth(caller,k; y) else access@LAuth(caller,k; y) fi
end
```

Although the *DB* class is inherited twice, for both *High* and *Low* interaction, *HLAuth* gets only one copy (see Section 5.1.1).

The example demonstrates natural usage of classes and multiple inheritance. Nevertheless, it reveals problems with the combination of inheritance and *statically ordered* virtual binding: Objects of the classes *LAuth* and *HAuth* work well, in the sense that agents opening access through the *Low* and *High* interfaces get the appropriate access, but the addition of the common subclass *HLAuth* is detrimental: When used through the *High* interface, this class allows multiple high access to data! Calls to the *High* operations of *HLAuth* trigger calls to the *HAuth* methods. From these methods the virtual internal calls to *grant*, *revoke*, and *auth* now binds to those of the *MAuth* class, if selected in a left-first depth-first traversal of the inheritance tree of the actual class *HLAuth*. If the inheritance ordering in *HLAuth* were reversed, similar problems occur with the binding of *Low* interaction.

The *pruned* virtual binding strategy ensures that the virtual internal calls constrained by classes *HAuth* and *LAuth* are bound in classes *SAuth* and *MAuth*, respectively, regardless of the actual class of the caller (*HAuth*, *LAuth*, or *HLAuth*), and of the inheritance ordering in *HLAuth*. In an object of class *HLAuth*, the local calls to *grant*, *revoke*, and *auth* in code from class *HAuth* is understood as *grant*<*Sauth*, *revoke*<*Sauth*, and *auth*<*Sauth*. These may not be bound in the *Mauth* class since *Mauth* is not a subclass of *Sauth*.

5.4 Typing

In order to extend the type system to classes with inheritance we revise the rules for classes, internal calls, and replies, and add rules for the new method notations $m@C$ and $m < C$. These are given in Figures 16 and 17. Let $\sqsubseteq$ be the reflexive and transitive closure of the subclass relation; $C \sqsubseteq C'$ expresses that C is a direct or indirect subclass of C', or is the same class as C'.

Definition 13 Let Γ be a typing environment, C be a class name, E a list of expressions, and C a list of instantiated class names. Define

$$\begin{aligned} \mathit{matchparam}(\Gamma, \epsilon) &= \mathsf{true} \\ \mathit{matchparam}(\Gamma, C(\mathrm{E}); \mathrm{C}) &= \Gamma \vdash_{\mathrm{F}} \mathrm{E} : T \ \wedge \ T \preceq \mathit{type}(\Gamma_{\mathcal{C}}(C).\mathit{Param}) \\ &\quad \wedge \mathit{matchparam}(\Gamma, \mathrm{C}) \end{aligned}$$

For a class C, the formal parameters of C may be instantiated with values passed to C from its subclasses. Thus, to ensure type-correct instantiations of superclasses, the type of the actual parameters of the instantiated superclass names must be type checked with respect to the type of the formal parameters of the superclasses. This is done by the auxiliary function *matchparam* in

$$\text{(CLASS-INH)}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathrm{V}} \ \mathit{Param}\ \langle\ \Delta\ \rangle \qquad \Gamma + \Delta \vdash_{\mathrm{V}} \ \mathit{InhAttr}(\mathit{Inh}, \Gamma_{\mathcal{C}}); \mathit{Var}\ \langle\ \Delta'\ \rangle \\ \mathit{matchparam}\,(\Gamma + \Delta, \mathit{Inh}) \qquad \forall m \in \mathit{Mtd}\ \cdot\ \Gamma + \Delta + \Delta' \vdash m\ \langle\ \Delta^m\ \rangle \\ \forall I \in (\mathit{Impl}; \mathit{Contract}) \cdot \forall m' \in \Gamma_{\mathcal{I}}(I).\mathit{Mtd} \cdot \exists\ C' \in \Gamma_{\mathcal{C}}\ \cdot \\ \quad \mathit{match}(m'.\mathit{Name}, \mathit{sig}(m'), m'.\mathit{Co}, \Gamma_{\mathcal{C}}(C').\mathit{Mtd}) \wedge \Gamma_{\mathrm{V}}(\mathit{self}) \sqsubseteq C'\end{array}}{\Gamma \vdash \mathit{class}\,(\mathit{Param}, \mathit{Impl}, \mathit{Contract}, \mathit{Inh}, \mathit{Var}, \mathit{Mtd})\ \langle\ \bigcup_{m \in \mathit{Mtd}} \Delta^m_{\mathit{Sig}}\ \rangle}$$

$$\text{(INT-SYNC)}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathrm{F}}\ \mathrm{E} : T \qquad \Gamma_{\mathrm{V}}(\mathit{self}) \sqsubseteq C \\ \exists\ C' \in \Gamma_{\mathcal{C}}\ \cdot\ \mathit{match}(m, T \to \Gamma_{\mathrm{V}}(\mathrm{V}), \varsigma, \Gamma_{\mathcal{C}}(C').\mathit{Mtd}) \wedge C \sqsubseteq C'\end{array}}{\Gamma \vdash_{\mathrm{S}}\ m@C(\mathrm{E}; \mathrm{V})}$$

$$\text{(INT-ASYNC)}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathrm{F}}\ \mathrm{E} : T \qquad \Gamma_{\mathrm{V}}(\mathit{self}) \sqsubseteq C \\ \exists\ C' \in \Gamma_{\mathcal{C}}\ \cdot\ \mathit{match}(m, T \to \mathsf{Data}, \varsigma, \Gamma_{\mathcal{C}}(C').\mathit{Mtd}) \wedge C \sqsubseteq C'\end{array}}{\Gamma \vdash_{\mathrm{S}}\ !m@C(\mathrm{E})}$$

$$\text{(INT-ASYNC-L)}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathrm{F}}\ \mathrm{E} : T \qquad \Gamma_{\mathrm{V}}(t) = \mathsf{Label} \qquad \Gamma_{\mathrm{V}}(\mathit{self}) \sqsubseteq C \\ \exists\ C' \in \Gamma_{\mathcal{C}}\ \cdot\ \mathit{match}(m, T \to \mathsf{Data}, \varsigma, \Gamma_{\mathcal{C}}(C').\mathit{Mtd})\ \wedge\ C \sqsubseteq C'\end{array}}{\Gamma \vdash_{\mathrm{S}}\ t_i!m@C(\mathrm{E})\ \langle\ [t \overset{P}{\mapsto} \{i\}] + [t_i \overset{\mathit{Sig}}{\mapsto} \langle \varsigma, m_{C'}, \varsigma, T \to \mathsf{Data}\rangle]\ \rangle}$$

Figure 16. Typing of multiple inheritance. Here *sig*(m) returns the signature of m.

the typing rule CLASS-INH, which takes the typing environment and a list of instantiated class names, and compares the actual and formal parameters.

The initial expressions in variable declarations and the program statements in method bodies of a class C may refer to variables declared in its superclasses. Consequently, the typing environment Γ must be extended with inherited attributes before type checking variables and methods of class C. This extension is obtained by traversing the instantiated class names *Inh* of C depth first and using the mapping $\Gamma_{\mathcal{C}}$ to gather *Param* and *Var* from each superclass. The function *InhAttr* in the typing rule for classes returns the list of all typed variables inherited from the classes above C. (It follows that Lemma 2 holds for the extended language.) Furthermore, for each interface that C implements C must provide at least one type-correct method body for each method in the interface, either by inheritance or by local declaration.

Method calls. The typing of external calls is controlled by interfaces and is not affected by class inheritance. The rules for internal invocations ressemble those in Section 3.4.4, but the analysis may now depend on the inheritance tree above *self*. The typing of internal calls inspects the inheritance graph, choosing a class such that the invocation is covered. As before, the signature of a call may be refined by a reply statement, but the signature is fixed to a class which need not be the class of *self*. For static calls $m@C$ the match starts from C, and not from the class of *self*. For bounded calls $m < C$, the match must be found below C in the inheritance tree above the class of *self*.

$$\text{(BND-SYNC)}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathrm{F}}\ \mathrm{E} : T \\ \exists C' \in \Gamma_{\mathcal{C}}\ \cdot (\Gamma_{\mathrm{V}}(\mathit{self}) \sqsubseteq C' \sqsubseteq C) \\ \qquad \wedge\ \mathit{match}(m, T \to \Gamma_{\mathrm{V}}(\mathrm{V}), \varsigma, \Gamma_{\mathcal{C}}(C').\mathit{Mtd})\end{array}}{\Gamma \vdash_{\mathrm{S}}\ m < C(\mathrm{E};\mathrm{V})}$$

$$\text{(BND-ASYNC)}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathrm{F}}\ \mathrm{E} : T \\ \exists C' \in \Gamma_{\mathcal{C}}\ \cdot (\Gamma_{\mathrm{V}}(\mathit{self}) \sqsubseteq C' \sqsubseteq C) \\ \qquad \wedge\ \mathit{match}(m, T \to \mathsf{Data}, \varsigma, \Gamma_{\mathcal{C}}(C').\mathit{Mtd})\end{array}}{\Gamma \vdash_{\mathrm{S}}\ !m < C(\mathrm{E})}$$

$$\text{(BND-ASYNC-L)}\ \frac{\begin{array}{c}\Gamma \vdash_{\mathrm{F}}\ \mathrm{E} : T \qquad \Gamma_{\mathrm{V}}(t) = \mathsf{Label} \\ \exists C' \in \Gamma_{\mathcal{C}}\ \cdot (\Gamma_{\mathrm{V}}(\mathit{self}) \sqsubseteq C' \sqsubseteq C) \\ \qquad \wedge\ \mathit{match}(m, T \to \mathsf{Data}, \varsigma, \Gamma_{\mathcal{C}}(C').\mathit{Mtd})\end{array}}{\Gamma \vdash_{\mathrm{S}}\ t_i!m < C(\mathrm{E})\ \langle\ [t \overset{P}{\mapsto} \{i\}] + [t_i \overset{\mathit{Sig}}{\mapsto} \langle \varsigma, m_{C'}, \varsigma, T \to \mathsf{Data}\rangle]\ \rangle}$$

$$\text{(REPLY)}\ \frac{\begin{array}{c}\Gamma_P(t) \neq \emptyset \\ \forall i \in \Gamma_P(t) \cdot \Gamma_{\mathit{Sig}}(t_i) = \langle I, m_s, \mathit{Co}, T_1 \to T_2\rangle \wedge (T_2 \cap \Gamma_{\mathrm{V}}(\mathrm{V})) \neq \bot \\ \quad \wedge\ (I = \varsigma \wedge s \neq \_) \Rightarrow \mathit{match}(m, T_1 \to (T_2 \cap \Gamma_{\mathrm{V}}(\mathrm{V})), \mathit{Co}, \Gamma_{\mathcal{C}}(s).\mathit{Mtd})) \\ \quad \wedge\ (I \neq \varsigma \wedge s = \_) \Rightarrow \mathit{match}(m, T_1 \to (T_2 \cap \Gamma_{\mathrm{V}}(\mathrm{V})), \mathit{Co}, \Gamma_{\mathcal{I}}(I).\mathit{Mtd}))\end{array}}{\Gamma \vdash_{\mathrm{S}}\ t?(\mathrm{V})\ \langle\ [t \overset{P}{\mapsto} \emptyset] + \bigcup_{i \in \Gamma_P(t)} [t_i \overset{\mathit{Sig}}{\mapsto} \langle I, m_s, \mathit{Co}, T_1 \to (T_2 \cap \Gamma_{\mathrm{V}}(\mathrm{V}))]\ \rangle}$$

Figure 17. Typing of bounded method calls. The subscript "_" in the REPLY rule denotes an empty class subscript, representing an external invocation.

5.5 Operational Semantics

The operational semantics is adapted to incorporate multiple inheritance in Figure 18. Creol classes are extended to include the instantiated class names of inherited classes and are given as RL objects $\langle \mathit{Cl} \,|\, \mathit{Par}, \mathit{Inh}, \mathit{Att}, \mathit{Mtds}, \mathit{Tok}\rangle$, where Cl is the class name, Par a list of parameters, Inh is a list of instantiated class names, Att a list of attributes, Mtds a multiset of methods, and Tok is an arbitrary term of sort Label. When an object needs a method, it is bound to a definition in the Mtds multiset of its class or of a superclass. Previous class definitions (without inheritance) can be extended with an empty list of instantiated class names to be valid in the extended semantics.

5.5.1 Virtual and Static Binding of Method Calls

Qualified external method invocations are syntactically excluded; external invocations cannot access the internal structure of the callee. Internal calls give rise to invocation messages, but in R10′ the qualified method name mq may be of the form $m@C$ or $m < C$, where the constraint C is used in the binding.

$$\text{(R2}') \quad \begin{array}{l} \mathit{new}\ C(\mathrm{E})\ \langle C : \mathit{Cl} \mid \mathit{Tok} : n\rangle \\ \quad\longrightarrow\ \langle (C;n) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \epsilon, \mathit{Pr} : \langle \epsilon, \epsilon\rangle, \mathit{PrQ} : \epsilon, \mathit{EvQ} : \epsilon, \mathit{Lab} : 1\rangle \\ \qquad \langle C : \mathit{Cl} \mid \mathit{Tok} : \mathit{next}(n)\rangle \\ \qquad \mathit{inherit}((C;n), \mathit{self} : \mathit{Any} = (C;n), \epsilon)\ \mathbf{to}\ C(\mathit{eval}\,(\mathrm{E}, (\mathrm{A}, \mathrm{L}))) \end{array}$$

$$\text{(R3}') \quad \begin{array}{l} \langle o : \mathit{Ob} \mid \mathit{Att} : \mathrm{A}, \mathit{Pr} : \langle (v := \mathit{new}\ C(\mathrm{E}); \mathrm{S}), \mathrm{L}\rangle\rangle\ \langle C : \mathit{Cl} \mid \mathit{Tok} : n\rangle \\ \quad\longrightarrow\ \langle o : \mathit{Ob} \mid \mathit{Att} : \mathrm{A}, \mathit{Pr} : \langle (v := (C;n); \mathrm{S}), \mathrm{L}\rangle\rangle\ \langle C : \mathit{Cl} \mid \mathit{Tok} : \mathit{next}(n)\rangle \\ \qquad \langle (C;n) : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{Att} : \epsilon, \mathit{Pr} : \langle \epsilon, \epsilon\rangle, \mathit{PrQ} : \epsilon, \mathit{EvQ} : \epsilon, \mathit{Lab} : 1\rangle \\ \qquad \mathit{inherit}((C;n), \mathit{self} : \mathit{Any} = (C;n), \epsilon)\ \mathbf{to}\ C(\mathit{eval}\,(\mathrm{E}, (\mathrm{A}, \mathrm{L}))) \end{array}$$

$$\text{(R10}') \quad \begin{array}{l} \langle o : \mathit{Ob} \mid \mathit{Att} : \mathrm{A}, \mathit{Pr} : \langle\, !\mathit{mq}(\mathit{Sig}, \mathit{Co}, \mathrm{E}); \mathrm{S}, \mathrm{L}\rangle, \mathit{Lab} : n\rangle \\ \quad\longrightarrow\ \langle o : \mathit{Ob} \mid \mathit{Att} : \mathrm{A}, \mathit{Pr} : \langle \mathrm{S}, \mathrm{L}\rangle, \mathit{Lab} : \mathit{next}(n)\rangle \\ \qquad \mathit{invoc}(\mathit{mq}, \mathit{Sig}, \mathit{Co}, (n\ o\ \mathit{eval}(\mathrm{E}, (\mathrm{A}; \mathrm{L}))))\ \mathbf{to}\ o \end{array}$$

$$\text{(R24)} \quad \begin{array}{l} \langle o : \mathit{Ob} \mid \mathit{EvQ} : \mathit{invoc}(m@C, \mathit{Sig}, \mathit{Co}, \mathrm{E})\ \mathrm{Q}\rangle \\ \quad\longrightarrow \langle o : \mathit{Ob} \mid \mathit{EvQ} : \mathrm{Q}\rangle\ (\mathit{bind}(m, \mathit{Sig}, \mathit{Co}, \mathrm{E}, o)\ \mathbf{to}\ C) \end{array}$$

$$\text{(R25)} \quad \begin{array}{l} \langle o : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{EvQ} : \mathit{invoc}(m < C', \mathit{Sig}, \mathit{Co}, \mathrm{E})\ \mathrm{Q}\rangle \\ \quad\longrightarrow \langle o : \mathit{Ob} \mid \mathit{Cl} : C, \mathit{EvQ} : \mathrm{Q}\rangle\ (\mathit{bind}(m < C', \mathit{Sig}, \mathit{Co}, \mathrm{E}, o)\ \mathbf{to}\ C) \end{array}$$

$$\text{(R16}') \quad \begin{array}{l} (\mathit{bind}(m, \mathit{Sig}, \mathit{Co}, \mathrm{E}, o)\ \mathbf{to}\ C\ \mathrm{I})\ \langle C : \mathit{Cl} \mid \mathit{Inh} : \mathrm{I}', \mathit{Mtds} : \mathrm{M}\rangle \\ \quad\longrightarrow\ \mathbf{if}\ \mathit{match}(m, \mathit{Sig}, \mathit{Co}, \mathrm{M}) \\ \qquad \mathbf{then}\ \mathit{bound}(\mathit{get}(m, \mathrm{M}, \mathrm{E}))\ \mathbf{to}\ o \\ \qquad \mathbf{else}\ \mathit{bind}(m, \mathit{Sig}, \mathit{Co}, \mathrm{E}, o)\ \mathbf{to}\ (\mathrm{I}'\ \mathrm{I})\ \mathbf{fi}\ \langle C : \mathit{Cl} \mid \mathit{Inh} : \mathrm{I}', \mathit{Mtds} : \mathrm{M}\rangle \end{array}$$

$$\text{(R26)} \quad \begin{array}{l} (\mathit{bind}(m < C', \mathit{Sig}, \mathit{Co}, \mathrm{E}, o)\ \mathbf{to}\ C\ \mathrm{I})\ \langle C : \mathit{Cl} \mid \mathit{Inh} : \mathrm{I}', \mathit{Mtds} : \mathrm{M}, \mathit{Tok} : n\rangle \\ \quad\longrightarrow\ \langle C : \mathit{Cl} \mid \mathit{Inh} : \mathrm{I}', \mathit{Mtds} : \mathrm{M}, \mathit{Tok} : \mathit{next}(n)\rangle\ (\mathbf{if}\ \mathit{match}(m, \mathit{Sig}, \mathit{Co}, \mathrm{M}) \\ \qquad \mathbf{then}\ (\mathit{find}(n, C', C)\ \mathbf{to}\ C)\ (\mathit{stopbind}(n, m < C', \mathit{Sig}, \mathit{Co}, \mathrm{E}, o)\ \mathbf{to}\ C\ \mathrm{I}) \\ \qquad \mathbf{else}\ \mathit{bind}(m < C', \mathit{Sig}, \mathit{Co}, \mathrm{E}, o)\ \mathbf{to}\ (\mathrm{I}'\ \mathrm{I})\ \mathbf{fi}) \end{array}$$

$$\text{(R27)} \quad \begin{array}{l} (\mathit{found}(n, b, C')\ \mathbf{to}\ C)\ (\mathit{stopbind}(n, m, \mathit{Sig}, \mathit{Co}, \mathrm{E}, o)\ \mathbf{to}\ C\ \mathrm{I}) \\ \langle C : \mathit{Cl} \mid \mathit{Inh} : \mathrm{I}', \mathit{Mtds} : \mathrm{M}\rangle \\ \quad\longrightarrow\ \mathbf{if}\ b\ \mathbf{then}\ \mathit{bound}(\mathit{get}(m, \mathrm{M}, \mathrm{E}))\ \mathbf{to}\ o\ \mathbf{else}\ \mathit{bind}(m, \mathit{Sig}, \mathit{Co}, \mathrm{E}, o)\ \mathbf{to}\ \mathrm{I}\ \mathbf{fi} \\ \qquad \langle C : \mathit{Cl} \mid \mathit{Inh} : \mathrm{I}', \mathit{Mtds} : \mathrm{M}\rangle \end{array}$$

$$\text{(R28)} \quad \mathit{find}(n, C, C'')\ \mathbf{to}\ \epsilon\ \longrightarrow\ \mathit{found}(n, \mathit{false}, C)\ \mathbf{to}\ C''$$

$$\text{(R29)} \quad \mathit{find}(n, C, C'')\ \mathbf{to}\ \mathrm{I}\ C\ \mathrm{I}'\ \longrightarrow\ \mathit{found}(n, \mathit{true}, C)\ \mathbf{to}\ C''$$

$$\text{(R30)} \quad \begin{array}{l} (\mathit{find}(n, C, C'')\ \mathbf{to}\ C'\ \mathrm{I})\ \langle C' : \mathit{Cl} \mid \mathit{Inh} : \mathrm{I}'\rangle \\ \quad\longrightarrow\ (\mathit{find}(n, C, C'')\ \mathbf{to}\ \mathrm{I}\ \mathrm{I}')\ \langle C' : \mathit{Cl} \mid \mathit{Inh} : \mathrm{I}'\rangle\ \mathbf{if}\ (C \neq C') \end{array}$$

Figure 18. An operational semantics with multiple inheritance. Note that the rules R2′, R3′, R10′, and R16′ redefine the previous rules R2, R3, R10, and R16. In R10′, mq denotes either $m@C$ or $m < C$.

In order to allow concurrent and dynamic execution, the inheritance graph is not statically given. Rather, the binding mechanism dynamically inspects the class hierarchy in the configuration. Our approach to virtual binding uses a *bind* message, which is sent from a class to its superclasses, resulting in a *bound* message returned to the object requesting the method binding. This way, the inheritance graph is explored dynamically and only as far as necessary. When the external invocation of a method m is found in the message queue of an object o, a message $\mathit{bind}(m, \mathit{Sig}, \mathit{Co}, \mathrm{E}, o)\ \mathbf{to}\ C$ is sent in R12, after retrieving

the class C of the object. For internal static calls $m@C$, the *bind* message is sent by R24 without inspecting the *actual* class of the callee, thus surpassing local definitions. If a suitable m is defined locally in C, a process with the method code and local state is returned in a *bound* message. Otherwise, the *bind* message is *retransmitted* to the superclasses of C in a left-first depth-first order by application of R16′. In order to facilitate the traversal of the inheritance graph, a list of instantiated class names is used as the destination of the *bind* message. The process resulting from the binding is loaded into the internal process queue of the callee as before.

5.5.2 Pruned Virtual Binding

The binding of an internal virtual call $m < C'$ is more involved. When a match in a class C is found in the application of R26, the inheritance graph of C is inspected to ensure that $C \sqsubseteq C'$, otherwise the binding must resume. Note the additional *stopbind* message with token n, which suspends binding while checking that $C \sqsubseteq C'$. This is done by two auxiliary messages, captured in R27 – R30: The message $\mathit{find}(n, C', C)$ **to** I represents that C is asking a list I of instantiated class names (ignoring the actual parameter values for readability) if C' may be found in I or further up in the hierarchy. The corresponding message $\mathit{found}(n, b, C')$ **to** C returns an answer to C. In this message the Boolean b is true if the request was successful and n matches the token of the *stopbind* message, identifying the call being bound. This search corresponds to left-first breadth-first traversal of the inheritance graph.

5.5.3 Object Creation and Attribute Instantiation

The object creation rules R2 and R3 are redefined to address the dynamic inheritance graph (see R2′ and R3′). In order to initialize the state of the new object the inheritance graph is traversed and inherited state variables collected, using the equations given in Figure 19. Recall that using equations enables object creation and attribute collection in one rewrite step. The equations convert class parameters and actual parameter values to attribute declarations which textually precede the attribute list of each class. Inherited parameters and attribute lists textually precede the attribute list of a subclass. The collected attribute declarations are evaluated in the new object in order to initialize the object state. Class parameters and inherited attributes provide a mechanism to pass values to the initial expressions of the inheritance list in a class. The order in which parameters are collected (Figure 19) ensures that multi-inheritance of the same class is the same as inheriting the class once, keeping the leftmost instantiation of the state variables.

$inherit(o, \text{S}, \text{A})$ **to** $nil = inherited(\text{S}; \text{A})$ **to** o

$inherit(o, \text{S}, \text{A})$ **to** $(\text{I}\ C(In))\ \ \langle C : Cl \mid Param : \text{V}, Inh : \text{I}', Att : \text{A}' \rangle$
$= inherit(o, (\text{S}; \text{V} = In), \text{A}'; \text{A}))$ **to** $(\text{I}\ \text{I}')\ \langle C : Cl \mid Param : \text{V}, Inh : \text{I}', Att : \text{A}' \rangle$

$(inherited(\text{A})$ **to** $o)\ \langle o : Ob \mid Pr : \langle \epsilon, \epsilon \rangle \rangle = \langle o : Ob \mid Pr : \langle (\text{A}) \downarrow; run, \epsilon \rangle \rangle$

Figure 19. State instantiation equations for multiple inheritance.

5.6 Type soundness

The soundness of the full language is established in this section. We first show that the class hierarchy is correctly unfolded when initializing the object state.

Lemma 15 *If* $inherit(o, \text{S}, \text{A})$ ***to*** I *is a message in a configuration of a well-typed execution of a program* P*, then* $\Gamma_{\mathcal{F}} \vdash_{\text{V}} \text{S} \langle\, \Delta \,\rangle$.

Proof. Let $\text{S} = \text{V}_0 : T_0 = \text{E}_0; \ldots; \text{V}_n : T_n = \text{E}_n$ be an attribute list and let Γ^i_{V} be a mapping such that $\Gamma^i_{\text{V}}(\text{V}_j) = T_j$ for $1 \leq j < i$. We show that $\Gamma_{\mathcal{F}} + \Gamma^n_{\text{V}} \vdash_{\text{F}} \text{E}_n \langle\, \Delta \,\rangle$. The proof is by induction over n. For $n = 0$, $\text{S} = self : Any = (C; n)$, $\Gamma_{\mathcal{F}}(self) = Any$, and $\Gamma_{\mathcal{F}}((C; n)) = I$ so $I \preceq Any$ and $\Gamma_{\mathcal{F}} \vdash_{\text{V}} \text{S} \langle\, \Delta \,\rangle$. For $n = 1$, the message must have been caused by application of R2′ or R3′. By Lemma 2, the evaluation of an expression in a well-typed method successfully dereferences all program variables in the expression. Since E_1 is an expression which has been evaluated either in R2′ or R3′, E_1 does not refer to program variables and consequently $\Gamma_{\mathcal{F}} \vdash_{\text{F}} \text{E}_1$. For the induction step we assume that the lemma holds for $n + 1$ and show that it also holds for $n + 2$, in which case V_{n+2} is a class parameter to some class C' which is inherited by a class C such that V_i $(i \leq n + 1)$ is the class parameter of C. Since C is well-typed, CLASS-INH asserts that $\Gamma_{\mathcal{F}} + \Gamma^i_{\text{V}} \vdash_{\text{F}} \text{E}_{n+2}$ through the *matchparam* predicate. It follows that $\Gamma_{\mathcal{F}} \vdash_{\text{V}} \text{S} \langle\, \Delta \,\rangle$. ■

Lemma 16 *Given an arbitrary Creol program* P *and a well-typed execution* ρ *of* P*. The execution of a statement* $x :=$ ***new*** $C(\text{E})$ *in the final configuration of* ρ *results in well-typed configurations of* P *while the new object is initialized.*

Proof. Let o' be a runtime object executing $x :=$ **new** $C(\text{E})$ (by the application of R3′) in the final configuration ρ_i of ρ. Let o be the new object reference. The well-typedness of o' before and after the execution of $x :=$ **new** $C(\text{E})$ and the uniqueness of reference o are covered in the proof of Lemma 7. The configuration ρ_{i+1} includes a runtime object

$\langle o : Ob \mid Cl : C, Att : \epsilon, Pr : \langle ((\text{A}_0) \downarrow; run), \epsilon \rangle, PrQ : \epsilon, EvQ : \epsilon, Lab : 1 \rangle$

which is well-typed. Let E' be the result of evaluating E in o'. We need to show that the object remains well-typed while the assignment list $(\text{A}_0) \downarrow$ is executed and that the assignment list instantiates all program variables in the object state. This is done by showing that the attribute list A_0 is well-

typed, so that object instantiation (by the repeated application of R1) results in well-typed configurations. The proof is by induction over the depth of the inheritance tree above C. At each step of the induction we assume that the parameter and attribute lists are type-correctly collected from the inheritance tree above the currently considered class, and show that when type-checking the parameter and attribute list from left to right, all variables occurring in the initial expressions have been instantiated.

For the basis step, let C' be a leaf above C with formal class parameters V of type T' and let In be the actual parameter values passed to C'. For well-typed programs, we know that $\Gamma \vdash_{\mathrm{F}} In : T$ such that $T \prec T'$ (this is checked by the *matchparam* function). Consequently, In can be type-correctly assigned to V. Let S and A be accumulated class parameter and class attribute assignments, and I be a list of superclasses of C. (If we are instantiating a leaf class directly, S, A, and I are empty.) The operational semantics gives us

$$(\mathit{inherit}(o, \mathrm{S}, \mathrm{A})\ \mathbf{to}\ \mathrm{I}\ C'(\mathit{In}))\ \langle C' : \mathit{Cl} \mid \mathit{Param} : \mathrm{V} : T', \mathit{Inh} : \mathit{nil}, \mathit{Att} : \mathrm{A}' \rangle$$
$$= (\mathit{inherit}(o, (\mathrm{S}; \mathrm{V} : T' = \mathit{In}), (\mathrm{A}'; \mathrm{A}))\ \ \mathbf{to}\ \mathrm{I})\ \langle C' : \mathit{Cl} \mid \mathit{Param} : \mathrm{V} : T', \mathit{Inh} : \mathit{nil}, \mathit{Att} : \mathrm{A}' \rangle.$$

By Lemma 15, $\Gamma \vdash_{\mathrm{V}}\ \mathrm{S}; \mathrm{V} : T' = \mathit{In}\ \langle\ \Delta\ \rangle$. We need to show that $\Gamma \vdash_{\mathrm{V}}\ \mathrm{S}; \mathrm{V} : T' = \mathit{In}; \mathrm{A}'$. Since C' is a leaf the function $\mathit{InhAttr}(\mathit{Inh}, \Gamma_{\mathcal{C}}) = \epsilon$ and, by the typing rule CLASS-INH, $\Gamma + \Delta \vdash_{\mathrm{V}}\ \mathrm{A}'$. It follows that $\Gamma \vdash_{\mathrm{V}}\ \mathrm{S}; \mathrm{V} : T' = \mathit{In}; \mathrm{A}'$. It is immediate that $\mathrm{S}; \mathrm{V} : T' = \mathit{In}; \mathrm{A}'$ contains all state variables declared in C'. Note that the attribute assignments of C' precede the accumulated attribute assignment list A, so initial expressions in A may safely refer to variables in $\mathrm{S}; \mathrm{V} : T' = \mathit{In}; \mathrm{A}'$.

For the induction step, we consider a class C' and assume that the assignment list from each of its superclasses is well-typed (with respect to the accumulated parameter assignment list) and contains all state variables declared in the superclasses of C'. Due to the qualified name convention, the concatenation of these assignments lists is also well-typed with respect to the accumulated parameter assignment list. The following equation applies:

$$\mathit{inherit}(o, \mathrm{S}, \mathrm{A})\ \mathbf{to}\ (\mathrm{I}\ C'(\mathit{In}))\ \ \langle C' : \mathit{Cl} \mid \mathit{Param} : \mathrm{V} : T, \mathit{Inh} : \mathrm{I}', \mathit{Att} : \mathrm{A}' \rangle$$
$$= \mathit{inherit}(o, (\mathrm{S}; \mathrm{V} : T = \mathit{In}), \mathrm{A}'; \mathrm{A}))\ \mathbf{to}\ (\mathrm{I}\ \mathrm{I}')\ \langle C' : \mathit{Cl} \mid \mathit{Param} : \mathrm{V} : T, \mathit{Inh} : \mathrm{I}', \mathit{Att} : \mathrm{A}' \rangle.$$

The message $\mathit{inherit}(o, (\mathrm{S}; \mathrm{V}{:}\, T = \mathit{In}), \mathrm{A}'; \mathrm{A}))\ \mathbf{to}\ (\mathrm{I}\ \mathrm{I}')$ is further reduced to $\mathit{inherit}(o, (\mathrm{S}; \mathrm{V}{:}\, T = \mathit{In}; \mathrm{S}'), \mathrm{A}''; \mathrm{A}'; \mathrm{A}))\ \mathbf{to}\ \mathrm{I}$, where the superclasses of C' have been expanded. Here, S' and A'' are the accumulated parameter and attribute assignment lists from the classes in I'. By Lemma 15 and the induction hypothesis, $\Gamma \vdash_{\mathrm{V}}\ \mathrm{S}; \mathrm{V}{:}\, T = \mathit{In}; \mathrm{S}'; \mathrm{A}''\ \langle\ \Delta\ \rangle$. We need to show that $\Gamma \vdash_{\mathrm{V}}\ \mathrm{S}; \mathrm{V}{:}\, T = \mathit{In}; \mathrm{S}'; \mathrm{A}''; \mathrm{A}'$. By typing rule CLASS-INH, $\Gamma + \Delta \vdash_{\mathrm{V}}\ \mathrm{A}'$, since Δ contains all class parameters and inherited attributes. It follows that $\Gamma \vdash_{\mathrm{V}}\ \mathrm{S}; \mathrm{V}{:}\, T = \mathit{In}; \mathrm{S}'; \mathrm{A}''; \mathrm{A}'$ and $\mathrm{S}; \mathrm{V}{:}\, T = \mathit{In}; \mathrm{S}'; \mathrm{A}''; \mathrm{A}'$ contains all state variables declared above C'.

Finally, if C' is the actual class C of the object, the actual class parameters passed to C' come from the statement **new** $C(\text{E})$. Lemma 15 asserts that E can be type-correctly assigned to the formal parameters of C', $\text{S} = \mathit{self} : \mathit{Any} = (C; n)$, and $\text{A} = \epsilon$. It follows that the attribute list $\mathit{self}\colon \mathit{Any} = (C; n); \text{V}\colon T = \text{E}; \text{S}'; \text{A}''; \text{A}'$ is well-typed and declares all state variables of C'. Since the attribute list is well-typed, every application of R1 to $(\text{A}_0)\downarrow$ will result in a well-typed object. Consequently the evaluation of $(\text{A}_0)\downarrow$ constructs a well-typed state σ containing all object variables, and the object reduces to

$\langle o\colon \mathit{Ob} \mid \mathit{Cl}\colon C, \mathit{Att} : \sigma, \mathit{Pr} : \langle \mathit{run}, \epsilon\rangle, \mathit{PrQ}\colon \epsilon, \mathit{EvQ} : \epsilon, \mathit{Lab} : 1\rangle$.

Object initialization preserves well-typedness. ■

Lemma 17 *Let P be an arbitrary Creol program. If $\Gamma_{\mathcal{F}} \vdash P$, then every method invocation $!x.m(T_{in} \to T_{out}, \mathit{Co}, \text{E})$, $!m@C(T_{in} \to T_{out}, \mathit{Co}, \text{E})$, or $!m < C(T_{in} \to T_{out}, \mathit{Co}, \text{E})$ in a well-typed configuration of P can be type-correctly bound at runtime to a method such that the return values from the method are of type T'_{out} and $T'_{out} \preceq T_{out}$, provided that x is not a null pointer.*

Proof. We consider how the invocation message is bound in the evaluation rules for $!x.m(\mathit{Sig}, \mathit{Co}, \text{E})$, $!m@C(\mathit{Sig}, \mathit{Co}, \text{E})$, and $!m < C(\mathit{Sig}, \mathit{Co}, \text{E})$. The proof is similar to the proof of Lemma 8, but accounts for the dynamic traversal of the inheritance graph. As the call to m is well-typed, we may assume that x is typed by an interface I and that x is an instance of a class C which implements I. Consequently, there is at least a class C' above C in which m is declared with an appropriate signature and cointerface. Let E evaluate to E' such that $\Gamma(\text{E}') \preceq \Gamma(\text{E})$. Applying R9 to an external method call $!x.m(\mathit{Sig}, \mathit{Co}, \text{E})$ creates a message $\mathit{bind}(m, \mathit{Sig}, \mathit{Co}, \text{E}', x)$ **to** C, if C is the dynamically identified class of x. Method lookup proceeds by R16′:

$$(\mathit{bind}(m, \mathit{Sig}, \mathit{Co}, \text{E}', x)\ \textbf{to}\ C\ \text{I}')\langle C : \mathit{Cl} \mid \mathit{Inh} : \text{I}, \mathit{Mtds} : \text{M}\rangle$$
$$\longrightarrow \textbf{if}\ \mathit{match}(m, \mathit{Sig}, \mathit{Co}, \text{M})\ \textbf{then}\ \mathit{bound}(\mathit{get}(m, \text{M}, \text{E}'))\ \textbf{to}\ x$$
$$\textbf{else}\ \mathit{bind}(m, \mathit{Sig}, \mathit{Co}, \text{E}', x)\ \textbf{to}\ (\text{I}\ \text{I}')\ \textbf{fi}$$

where I' is initially empty. The method lookup function may potentially traverse the entire inheritance tree above C, including C'. Method binding is guaranteed to succeed at C'.

An internal method call $!m@C(\mathit{Sig}, \mathit{Co}, \text{E})$ in an object o ressembles the previous case. By R10′ the call results in a message $\mathit{invoc}(m@C, \mathit{Sig}, \mathit{Co}, \text{E}')$ **to** o which, by applying R24, generates a message $\mathit{bind}(m, \mathit{Sig}, \mathit{Co}, \text{E}', o)$ **to** C where C is the specified class. The call is correctly bound if there is a method m declared above C with signature $\mathit{Sig}' = T'_{in} \to T'_{out}$ and cointerface Co' such that $\mathit{Sig}' \preceq \mathit{Sig}$ and $\mathit{Co} \preceq \mathit{Co}'$, so $T'_{out} \preceq T_{out}$. The type analysis guarantees that C inherits a method m with a matching signature and cointerface. Consequently, the *bind* message succeeds in binding the call to a method declared above C when traversing the inheritance tree above C, by repeated applications of R16′.

Let o be an object of class C'. An internal bounded call $m < C(Sig, Co, \text{E})$ in o results in the message $invoc(m < C, Sig, Co, \text{E}')$ **to** o, which by R25 generates a message $bind(m < C, Sig, Co, \text{E}', o)$ **to** C'. The (repeated) application of R26 inspects the inheritance graph above C', searching for a matching method declaration located below C. For every match in the inheritance graph, say in a class C'', a message $find(n, C, C'')$ **to** C'' is generated, which means that the class C'' is a candidate for binding m. By application of rules R28–R30 this message returns true with token n if C is above C'' and false otherwise, and the token identifies the corresponding *stopbind* message. If the result is false, rule R27 continues the search. Well-typedness guarantees that there is at least one match in a class below C with signature $T'_{in} \to T'_{out}$ and Co' such that $T'_{out} \preceq T_{out}$, so the search eventually succeeds. ■

Theorem 18 (Type soundness) *All executions of Creol programs starting in a well-typed initial configuration, are well-typed.*

Proof. We consider a well-typed execution ρ of a program P. The proof is by the induction over the length of $\rho = \rho_0, \rho_1, \ldots$ and extends the proofs of Theorems 10 and 14. By assumption, ρ_0 is a well-typed initial configuration of P. As only R2′ is applicable to ρ_0, it follows from Lemma 16 that object creation results in a well-typed successor configuration. For the induction step we show that for any well-typed configuration ρ_i, the successor configuration ρ_{i+1} is also well-typed, by case analysis of the rewrite rules. Only the new rules of Figure 18 are discussed, the other rules are covered by the proofs of Theorems 10 and 14. We first consider object reductions; i.e., rules that reduce an object $\langle o : Ob \,|\, Att : \text{A}, Pr : \langle s; \text{S}, \text{L}\rangle\rangle$ to $\langle o : Ob \,|\, Att : \text{A'}, Pr : \langle s'; \text{S}, \text{L}\rangle\rangle$, and then the remaining rewrite rules.

- Consider $s = v := \textbf{new}\ C(\text{E})$ and the application of R3′. By Lemma 16, the evaluation of an object creation statement gives a well-typed successor configuration ρ_{i+1}. Moreover, the successor configurations from the initialization of the new object are also well-typed.
- For $s = mq(Sig, Co, \text{E})$ and the application of R10′, the state variables are not changed and ρ_{i+1} is well-typed.
- Applying R24 and R25 does not modify the state, so ρ_{i+1} is well-typed.

We now consider the remaining new rewrite rules.

- Rules R16′ and R26. Let I be a list of instantiated class names. As ρ_i is a configuration of the well-typed execution ρ, a message $bind(m, Sig, Co, \text{E}, o)$ **to** I must have been generated from a message $invoc(m@C, Sig, Co, \text{E})$ by application of R24 or from a message $invoc(m, Sig, Co, \text{E})$ by application of R12 for external calls. Similarly, a message $bind(m < C', Sig, Co, \text{E}, o)$ **to** I must have been generated from a message $invoc(m < C', Sig, Co, \text{E})$ by application of R25. These *invoc* messages must have been generated by applying R10′ to a

statement $!mq(Sig, Co, \text{E})$ or R9 to a statement $!x.m(Sig, Co, \text{E})$ for external calls. By Lemma 17, any such call $m@C(Sig, Co, \text{E})$, $m < C'(Sig, Co, \text{E})$, or $!x.m(Sig, Co, \text{E})$ in P can be type-correctly bound at runtime. Consequently, the *match* function in R16′ and R26 will eventually succeed before $\text{I} = \epsilon$, resulting in a *bound* or a *find* message, and method-not-understood errors cannot occur in ρ_{i+1}. It follows that ρ_{i+1} is well-typed.

- Rule R27. As ρ is well-typed, the *found* and *stopbind* messages are originated from the application of R26 to a message $bind(m < C', Sig, Co, \text{E}, o)$ **to** I, in a well-typed configuration. By Lemma 17, the invocation can be bound by traversing the inheritance tree above C and below C'. Consequently, the binding will succeed before $\text{I} = \epsilon$ generating a *bound* message. It follows that method-not-understood errors do not occur in ρ_{i+1}, so ρ_{i+1} is well-typed.
- Rules R28, R29, and R30 do not modify the state, so ρ_{i+1} is well-typed. ■

6 Related Work

Many object-oriented languages offer constructs for concurrency; a survey is given in [66]. A common approach is to rely on the tight synchronization of RPC, separating activity (threads) and objects, as done in Hybrid [64] and Java [36], or on the rendezvous concept in concurrent objects languages such as Ada and POOL [5]. These approaches seem less desirable for distributed systems, with potential delays and communication loss. Hybrid offers *delegation* to (temporarily) branch an activity thread. Asynchronous method calls may be seen as a form of delegation and can be implemented in, e.g., Java by explicitly creating new threads to handle calls [23]. In Creol, polling for replies to asynchronous calls is handled by the operational semantics: no new threads and active loops are needed to poll for replies to delegated activity. UML offers asynchronous event communication and synchronous method invocation but does not integrate these, resulting in significantly more complex formalizations [26] than ours. To facilitate the programmer's task and reduce the risk of errors, implicit control structures based on asynchronous method calls seem more attractive, allowing a higher level of abstraction in the language.

The internal concurrency model of concurrent objects in Creol may be compared to monitors [38] or to thread pools executing on a single processor, with a shared state space given by the object attributes. In contrast to monitors, explicit signaling is avoided. Sufficient signaling is ensured by the semantics, which significantly simplifies reasoning [24]. However, general monitors may be encoded in the language [47]. In contrast to thread pools, processor release is explicit. In Creol, the activation of suspended processes is nondeterministically handled by an unspecified scheduler. Consequently, intra-object concurrency is similar to the interleaving semantics of concurrent process languages [6, 29], where each Creol process resembles a series of guarded atomic actions (dis-

carding local process variables). Internal reasoning control is facilitated by the explicit declaration of release points, at which class invariants should hold [31].

Languages based on the Actor model [3, 4] take asynchronous messages as the communication primitive, focussing on loosely coupled processes with less synchronization. This makes Actor languages conceptually attractive for distributed programming. The interpretation of method calls as asynchronous messages has lead to the notion of future variables which may be found in languages such as ABCL [80], Argus [53], ConcurrentSmalltalk [79], Eiffel// [16], CJava [23], and in the Join calculus [33] based languages Polyphonic C♯ [8] and Join Java [43]. Our communication model is also based on asynchronous messages and the proposed asynchronous method calls resemble programming with future variables, but Creol's processor release points further extend this approach to asynchrony with additional flexibility.

Languages supporting asynchronous methods generally either disallow inheritance [43, 80] or impose redefinition of asynchronous methods [16]. Multiple inheritance is supported in languages such as C++ [73], CLOS [27], Eiffel [59], POOL [5], and Self [19]. Horizontal name conflicts in C++, POOL, and Eiffel are removed by explicit resolution, after which the inheritance graph may be linearized. A natural semantics for virtual binding in Eiffel is proposed in [7]. This work is similar in spirit to ours and models the binding mechanism at the abstraction level of the program, capturing Eiffel's renaming mechanism. Mixin-based inheritance [10] and traits [65, 70] depend upon linearization to be merged correctly into the single inheritance chain. Linearization changes the parent-child relationship between classes in the inheritance hierarchy [71], and understanding method binding quickly becomes difficult.

Maude's inherent object concept [20, 57] represents an object's state as a subconfiguration, as we have done here, but in contrast to our approach object behavior is captured directly by rewrite rules. Both Actor-style asynchronous messages and synchronous transitions (rewrite rules involving several objects) are allowed, which makes Maude's object model very flexible. However, asynchronous method calls and processor release points as proposed in this paper are hard to represent within this model. Both Maude and the Join-calculus model multiple inheritance by disjoint union of methods. Name ambiguity lets method definitions compete for selection. The definition selected when an ambiguously named method is called, is nondeterministically chosen. In Polyphonic C♯ this nondeterminism is supplemented by a substitution mechanism for inherited code. CJava, restricted to outer guards and single inheritance, allows separate redefinition of synchronization code and bodies in subclasses. Programmer control may be improved if inherited classes are ordered [19, 27], resulting in deterministic binding. However, the ordering of superclasses may result in surprising but "correct" behavior. The example of Section 5.3 displays such surprising behavior regardless of how the inherited classes are ordered.

The statements for high-level control of local computation in Creol are inspired by notions from process algebra [39, 62]. Process algebra is usually based on synchronous communication. In contrast to, e.g., the asynchronous π-calculus [40], which encodes asynchronous communication in a synchronous framework by dummy processes, our communication model is truly asynchronous and without channels: message overtaking may occur. Further, Creol differs from process algebra in its integration of processes in an object-oriented setting using methods, including active and passive object behavior, and self reference rather than channels. In formalisms based on process algebra the operation of returning a result is not directly supported, but typically encoded as sending a message on a return channel [68,76,77]. Finally, Creol's high-level integration of asynchronous and synchronous communication and the organization of pending processes and interleaving at release points within class objects seem hard to capture naturally in process algebra.

Formal models clarify the intricacies of object orientation and may thus contribute to better programming languages in the future, making programs easier to understand, maintain, and analyze. Object calculi such as the ς-calculus [1] and its concurrent extension [35] aim at a direct expression of object-oriented features such as self-reference, encapsulation, and method calls, but asynchronous invocation of methods is not addressed. This also applies to Obliq [15], a programming language based on similar primitives which targets distributed concurrent objects. The concurrent object calculus of Di Blasio and Fisher [28] provides both synchronous and asynchronous invocation of methods. In contrast to Creol, return values are discarded when methods are invoked asynchronously and the two ways of invoking a method have different semantics. Class inheritance is not addressed in [1, 28, 35].

In the concurrent object calculus with single inheritance studied by Laneve [52], methods of superclasses are accessible and virtual binding is addressed by a careful renaming discipline. A denotational semantics for single inheritance with similar features is studied by Cook and Palsberg [21]. Multiple inheritance is not addressed in these works. Formalizations of multiple inheritance are usually based on the *objects-as-records* paradigm and focusses on subtyping issues related to subclassing. Issues related to method binding are not easily captured in this approach: Even access to superclass' methods is not addressed in Cardelli's denotational semantics of multiple inheritance [14]. Rossi, Friedman, and Wand [69] propose a formalization of multiple inheritance based on *subobjects*, a runtime data structure used for virtual pointer tables [51,73]. Their work focusses on compile time issues and does not clarify multiple inheritance at the abstraction level of the programming language.

The dynamically typed prototype-based language Self [19] proposes an elegant *prioritized binding strategy* to solve horizontal name conflicts, although a formal semantics is not given. The strategy is based on combining ordered and

unordered multiple inheritance. Each superclass is annotated with a priority, and many superclasses may have the same priority. A name is only ambiguous if it occurs in two superclasses with the same priority, in which case a class related to the actual class is preferred. However, explicit class priorities may have surprising effects in large class hierarchies: names may become ambiguous through inheritance. If neither class is related to the caller the binding does not succeed, resulting in a method-not-understood error.

The *pruned binding strategy* proposed in this paper solves these issues without the need for manually declaring (equal) class priorities and without the possibility of method-not-understood errors: Calls are only bound to intended method redefinitions. This binding strategy seems particularly useful during system maintenance to avoid introducing unintentional errors in evolving class hierarchies, supported in Creol [49]. In particular, Creol's operational semantics is based on the dynamic and distributed traversal of the class hierarchy, rather than on virtual pointer tables. Our approach may therefore be combined with dynamic constructs for changing the class inheritance structure, such as adding a class C and enriching an existing class with C as a new superclass.

The type system presented in this paper resembles that of Featherweight Java [41], a core calculus for Java, because of its nominal approach. Featherweight Java is class based and uses a class table to represent class information in its type system. Subtyping is the reflexive and transitive closure of the subclass relation. In contrast the type system of Creol cleanly distinguishes classes and types, which results in both a class and an interface table. Furthermore, Featherweight Java does not address issues related to assignment, overloading, and interfaces. A subtype discipline is required for method overriding, which allows significantly simpler definitions of method lookup (virtual binding is trivial in this setting). Multiple inheritance, interfaces and cointerfaces, nondeterministic merge and choice, and asynchronous method calls are not found in (Featherweight) Java. PolyToil [13] separates subtyping and (single) inheritance. Object types resemble Creol's interfaces, but there is only one type per class and no notion of cointerface. Scala [65] uses a nominal type system for mixin-based traits, extending a single inheritance relation. Asynchronous method calls, interfaces, and cointerfaces in Creol necessitate a more refined type system, including an effect system [55]. A type and effect system provides an elegant way of adding context information to the type analysis [74]. Type and effect systems have been used to ensure that, e.g., guards controlling method availability do not have side effects [28] and to estimate the effects of a reclassification primitive [32]. For Creol, the effect system derives type-correct signatures for asynchronous method calls. The system may be extended to ensure the absence of null pointers, using initialization restrictions [30] and checks on remote calls guaranteeing that called objects are not null.

7 Conclusion

This paper has presented the Creol model of distributed concurrent objects communicating by means of asynchronous method calls. The approach emphasizes flexibility with respect to the possible delays and instabilities of distributed computing but also with respect to code reuse through a liberal notion of multiple inheritance. The model makes a clear distinction between inheritance and subtyping, in particular subtyping is not required for method redefinition. Object variables are typed by interface, abstracting from the actual class of external objects. An object may be typed by many interfaces, expressing different roles of the object. Interfaces may require cointerfaces, expressing dependencies which facilitate protocol sessions in the distributed environment. The concept of *contracts* is used to statically control the typing of mutually dependent classes in presence of inheritance. Creol is formalized with an operational semantics defined in rewriting logic, providing a detailed account of, e.g., asynchronous method calls, object creation, and late binding. A type system for Creol has been introduced in this paper, distinguishing data types, interfaces, and classes. Type checking asynchronous method calls is based on a type and effect system. It is shown that runtime type errors do not occur for well-typed programs, including asynchronous method calls, nondeterministic choice and merge, multiple inheritance, object creation, and late binding of internal methods using the *pruned binding strategy*.

Acknowledgements

We are grateful to Dave Clarke for interesting discussions and to the anonymous referees for excellent feedback, significantly improving the paper.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.

[2] E. Ábrahám-Mumm, F. S. de Boer, W.-P. de Roever, and M. Steffen. Verification for Java's reentrant multithreading concept. In *International Conference on Foundations of Software Science and Computation Structures (FOSSACS'02)*, volume 2303 of *Lecture Notes in Computer Science*, pages 5–20. Springer, Apr. 2002.

[3] G. A. Agha. Abstracting interaction patterns: A programming paradigm for open distributed systems. In E. Najm and J.-B. Stefani, editors, *Proc. 1st IFIP*

International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'96), pages 135–153. Chapman & Hall, 1996.

[4] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, Jan. 1997.

[5] P. America and F. van der Linden. A parallel object-oriented language with inheritance and subtyping. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 25(10), pages 161–168. ACM Press, Oct. 1990.

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

[7] I. Attali, D. Caromel, and S. O. Ehmety. A natural semantics for Eiffel dynamic binding. *ACM Transactions on Programming Languages and Systems*, 18(6):711–729, 1996.

[8] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C♯. *ACM Transactions on Programming Languages and Systems*, 26(5):769–804, Sept. 2004.

[9] L. Bettini, V. Bono, R. D. Nicola, G. L. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The Klaim project: Theory and practice. In C. Priami, editor, *Global Computing. Programming Environments, Languages, Security, and Analysis of Systems*, volume 2874 of *Lecture Notes in Computer Science*, pages 88–150. Springer, 2003.

[10] G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / European Conference on Object-Oriented Programming*, pages 303–311. ACM Press, 1990.

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

[12] K. B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics.* The MIT Press, Cambridge, Mass., 2002.

[13] K. B. Bruce, A. Schuett, R. van Gent, and A. Fiech. PolyTOIL: A type-safe polymorphic object-oriented language. *ACM Transactions on Programming Languages and Systems*, 25(2):225–290, 2003.

[14] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2-3):138–164, 1988.

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

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

[17] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[18] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 157–200. Springer, 1999.

[19] C. Chambers, D. Ungar, B.-W. Chang, and U. Hölzle. Parents are shared parts of objects: Inheritance and encapsulation in SELF. *Lisp and Symbolic Computation*, 4(3):207–222, 1991.

[20] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, Aug. 2002.

[21] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. *Information and Computation*, 114(2):329–350, Nov. 1994.

[22] W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In *17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 125–135. ACM Press, Jan. 1990.

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

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

[25] O.-J. Dahl, B. Myrhaug, and K. Nygaard. (Simula 67) Common Base Language. Technical Report S-2, Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway, May 1968.

[26] W. Damm, B. Josko, A. Pnueli, and A. Votintseva. Understanding UML: A formal semantics of concurrency and communication in Real-Time UML. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *First International Symposium on Formal Methods for Components and Objects (FMCO 2002)*, volume 2852 of *Lecture Notes in Computer Science*, pages 71–98. Springer, 2003.

[27] L. G. DeMichiel and R. P. Gabriel. The Common Lisp Object System: An overview. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *European Conference on Object-Oriented Programming (ECOOP'87)*, volume 276 of *Lecture Notes in Computer Science*, pages 151–170. Springer, 1987.

[28] P. Di Blasio and K. Fisher. A calculus for concurrent objects. In U. Montanari and V. Sassone, editors, *7th International Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 655–670. Springer, Aug. 1996.

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

[30] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[31] J. Dovland, E. B. Johnsen, and O. Owe. Verification of concurrent objects with asynchronous method calls. In *Proceedings of the IEEE International Conference on Software - Science, Technology & Engineering (SwSTE'05)*, pages 141–150. IEEE Computer Society Press, Feb. 2005.

[32] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object re-classification: $\text{Fickle}_{\text{II}}$. *ACM Transactions on Programming Languages and Systems*, 24(2):153–191, 2002.

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

[34] J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. A. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*, Advances in Formal Methods, chapter 1, pages 3–167. Kluwer Academic Publishers, 2000.

[35] A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. In U. Nestmann and B. C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages*, volume 16(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998.

[36] J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java language specification*. Java series. Addison-Wesley, 2nd edition, 2000.

[37] B. Hailpern and H. Ossher. Extending objects to support multiple interfaces and access control. *IEEE Transactions on Software Engineering*, 16(11):1247–1257, 1990.

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

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

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

[41] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[42] International Telecommunication Union. Open Distributed Processing - Reference Model parts 1–4. Technical report, ISO/IEC, Geneva, July 1995.

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

[44] E. B. Johnsen and O. Owe. A compositional formalism for object viewpoints. In B. Jacobs and A. Rensink, editors, *Proc. 5th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'02)*, pages 45–60. Klüwer Academic Publishers, Mar. 2002.

[45] E. B. Johnsen and O. Owe. Object-oriented specification and open distributed systems. In O. Owe, S. Krogdahl, and T. Lyche, editors, *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, volume 2635 of *Lecture Notes in Computer Science*, pages 137–164. Springer, 2004.

[46] E. B. Johnsen and O. Owe. Inheritance in the presence of asynchronous method calls. In *Proc. 38th Hawaii International Conference on System Sciences (HICSS'05)*. IEEE Computer Society Press, Jan. 2005.

[47] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 2006. To appear. A short version appeared in the proceedings of *SEFM 2004*.

[48] E. B. Johnsen, O. Owe, and E. W. Axelsen. A run-time environment for concurrent objects with asynchronous method calls. In N. Martí-Oliet, editor, *Proc. 5th International Workshop on Rewriting Logic and its Applications (WRLA'04), Mar. 2004*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 375–392. Elsevier Science Publishers, Jan. 2005.

[49] E. B. Johnsen, O. Owe, and I. Simplot-Ryl. A dynamic class construct for asynchronous concurrent objects. In M. Steffen and G. Zavattaro, editors, *Proc. 7th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)*, volume 3535 of *Lecture Notes in Computer Science*, pages 15–30. Springer, June 2005.

[50] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proc. 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, June 1997.

[51] S. Krogdahl. Multiple inheritance in Simula-like languages. *BIT*, 25(2):318–326, 1985.

[52] C. Laneve. Inheritance in concurrent objects. In H. Bowman and J. Derrick, editors, *Formal methods for distributed processing – a survey of object-oriented approaches*, pages 326–353. Cambridge University Press, 2001.

[53] B. H. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In D. S. Wise, editor, *Proc. SIGPLAN Conference on Programming Lanugage Design and Implementation (PLDI'88)*, pages 260–267. ACM Press, June 1988.

[54] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994.

[55] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th Symposium on Principles of Programming Languages (POPL'88)*, pages 47–57. ACM Press, 1988.

[56] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. The MIT Press, 1993.

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

[58] J. Meseguer and G. Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In D. A. Basin and M. Rusinowitch, editors, *Proceedings of the Second International Joint Conference on Automated Reasoning (IJCAR 2004)*, volume 3097 of *Lecture Notes in Computer Science*, pages 1–44. Springer, 2004.

[59] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2 edition, 1997.

[60] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In E. Jul, editor, *12th European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–382. Springer, 1998.

[61] G. Milicia and V. Sassone. The inheritance anomaly: ten years after. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 1267–1274. ACM Press, 2004.

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

[63] E. Najm and J.-B. Stefani. A formal semantics for the ODP computational model. *Computer Networks and ISDN Systems*, 27:1305–1329, 1995.

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

[65] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In L. Cardelli, editor, *Proc. 17th European Conference on Object-Oriented Programming (ECOOP 2003)*, volume 2743 of *Lecture Notes in Computer Science*, pages 201–224. Springer, 2003.

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

[67] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

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

[69] J. G. Rossie Jr., D. P. Friedman, and M. Wand. Modeling subobject-based inheritance. In P. Cointe, editor, *10th European Conference on Object-Oriented Programming (ECOOP'96)*, volume 1098 of *Lecture Notes in Computer Science*, pages 248–274. Springer, July 1996.

[70] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In L. Cardelli, editor, *Proc. 17th European Conference on Object-Oriented Programming (ECOOP 2003)*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274. Springer, 2003.

[71] A. Snyder. Inheritance and the development of encapsulated software systems. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 165–188. The MIT Press, 1987.

[72] N. Soundarajan and S. Fridella. Inheritance: From code reuse to reasoning reuse. In P. Devanbu and J. Poulin, editors, *Proc. Fifth International Conference on Software Reuse (ICSR5)*, pages 206–215. IEEE Computer Society Press, 1998.

[73] B. Stroustrup. Multiple inheritance for C++. *Computing Systems*, 2(4):367–395, Dec. 1989.

[74] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

[75] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'96)*, pages 359–369. ACM Press, 1996.

[76] V. T. Vasconcelos. Typed concurrent objects. In M. Tokoro and R. Pareschi, editors, *Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP'94)*, volume 821 of *Lecture Notes in Computer Science*, pages 100–117. Springer, 1994.

[77] D. Walker. Objects in the π-calculus. *Information and Computation*, 116(2):253–271, Feb. 1995.

[78] R. J. Wirfs-Brock and R. E. Johnson. Surveying current research in object-oriented design. *Communications of the ACM*, 33(9):104–124, 1990.

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

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