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

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
PO Box 1080 Blindern, NO-0316 Oslo, Norway
email: {einarj,olaf,ingridcy}@ifi.uio.no

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 external communication is independent from the implementation of remote objects. 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 report presents the Creol syntax, operational semantics, and type system and shows that runtime type errors do not occur for well-typed programs.

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; e.g., system breakdown may have disastrous consequences. Furthermore, these distributed applications are nonterminating, and are 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 [40]. However, standard object-oriented models do not address the specific challenges of distributed computation structures. 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. We do not believe that distribution should 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. Separating execution threads from distributed objects breaks the modularity and encapsulation of object orientation, leading to a very low-level style of programming. Models of distributed systems based on asynchronously communicating concurrent objects seem much more natural. Intuitive high-level programming constructs are needed to unite object orientation and distribution in a natural way.

This paper presents the high-level object-oriented modeling language Creol [43, 45–47], which addresses distributed systems. The language is based on concurrent objects typed by behavioral interfaces, communication by asynchronous method calls, and a notion of processor release points. Processor release points allow a notion of non-blocking method calls, and allow objects to dynamically change between active and reactive behavior (client and server). The model integrates asynchronous communication and multiple inheritance, allowing 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. The language has an operational semantics defined in rewriting logic [55], which is executable on the Maude tool [20] and provides an interpreter and analysis platform for system models.

This paper extends previous work on Creol by a type system for Creol programs. It is shown that the execution of objects typed by behavioral interfaces and communicating by means of asynchronous method calls, is type safe. Behavioral interfaces make all external method calls virtually bound. The typing of mutually dependent interfaces is controlled by a notion of *contracts*. Furthermore, it is shown that extending the language with high-level constructs for local control, allowing objects to better adapt to the external nondeterminism of the distributed environment at runtime, still supports strong typing. 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.

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 Interfaces

In object-oriented viewpoint modeling an object may assume different roles, depending on the context of interaction. These roles may be captured by specifications of aspects of the externally observable behavior of objects. The specification of an aspect 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 identifier provides 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 provide the behavior described by the interface. A class may implement several interfaces. Objects of different classes may provide 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 binding also applies to external method calls, as the runtime class of the object will not be statically known.

For active objects it may be desirable 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 passively complete invocations of its own methods. This way, callback is supported in the run of a protocol between distributed objects. Use of the **with** clause restricts the communication environment of an object, as considered through the interface, to external objects providing a given *cointerface* [42, 44]. For some objects no such knowledge is required, which is captured by the keyword *Any* in the **with** clause, letting *Any* be the superinterface of all interfaces. *Mutual dependency* is specified if two interfaces have each other as cointerface.

2.1 Syntax

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

$$\mathit{method}\,(\mathit{Name}, \mathit{Co}, \mathit{Inpar}, \mathit{Outpar}, \mathit{Body}).$$

If Mtd is a set of methods, denote the subset of Mtd with methods of a given name by

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

where $\mathit{Name} \in \mathsf{Mtd}$ 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 $\langle \mathit{Var}, \mathit{Code}\rangle$ consisting of lists of variable declarations Vdecl and program statements S.

The elements of a method tuple may be accessed by dot notation. The empty sequence (or list) is denoted ε. To conveniently organize object viewpoints, interfaces may be structured in an inheritance hierarchy. Furthermore, interfaces may have parameters which may be passed on to inherited interfaces.

Definition 2 An interface is represented by a term

$$\mathit{interface}\,(\mathit{Param}, \mathit{Inh}, \mathit{Mtd}, \mathit{Req})$$

of type $\mathcal{I}$, where Param is a list of typed program variables, Inh is a list of instantiated interfaces, defining inheritance, Mtd is a set of methods such that $m.\mathit{Body} = \langle\varepsilon, \varepsilon\rangle$ for all $m \in \mathit{Mtd}$, and Req is a semantic requirement.

Let $\tau_{\mathcal{I}}$ denote the set of interface names, with typical elements I and J. For convenience, dot notation will be used to refer to the different elements of an interface; e.g., $\mathit{interface}\,(P, \mathit{Is}, M, R).\mathit{Mtd} = M$. The name $\mathit{Any} \in \tau_{\mathcal{I}}$ is reserved for $\mathit{interface}(\varepsilon, \varepsilon, \emptyset, \mathsf{true})$, and the name $\mathit{Void} \in \tau_{\mathcal{I}}$ is reserved for type checking purposes. A graphical representation of an interface may be given following the BNF syntax of Figure 1. In the graphical notation, all methods of an interface have the same cointerface, encouraging an aspect oriented specification style [48]. If I inherits J, the methods of both I and J must be available in any class that implements I. Two interfaces with the same set of methods may differ in their semantic constraints, so structural subtyping is too liberal in this setting. The observable behavior of an object as seen through an interface may be defined by restricting the observable behavior of the object to the methods available through the interface.

$$
\begin{array}{lcl}
IL & ::= & [\textbf{interface}\ I\ [(\mathit{Param})]^{?}\ [\textbf{inherits}\ \mathit{InhList}]^{?} \\
 & & \textbf{begin}\ [\textbf{with}\ I\ \mathit{Msig}^{*}]^{?}\ [\textbf{req}\ R]^{?}\ \textbf{end}]^{*} \\
\mathit{InhList} & ::= & [I[(\textsc{e})]^{?}]^{+}_{;} \\
\mathit{Param} & ::= & [v : T]^{+}_{;} \\
\mathit{Msig} & ::= & \textbf{op}\ m\ ([\textbf{in}\ \mathit{Param}]^{?}\ [\textbf{out}\ \mathit{Param}]^{?})
\end{array}
$$

Figure 1: A BNF grammar for 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. $\textsc{e}$ denotes a list of expressions.

A subtype relationship between interfaces may be defined in terms of the observable behavior of interfaces.

The semantic requirements for interfaces are not treated in detail in this paper, but motivate a nominal subtype relation [64]. An interface is a subtype of all its inherited interfaces but the subtype relation may also be extended by subtype declarations, normally accompanied by proof of observable behavior compliance. A simple example of semantic requirements is suggested in Section 5.3.1. An approach to behavioral interfaces with semantic requirements and refinement of behavioral interfaces is studied in [42, 44].

2.2 Example

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 mechanism for downloading packs; i.e., parts of a target file. The *Server* interface is only available to other servers in the network. Due to the **with**-construct and strong typing, any caller of a server request will understand the *enquire* and *getPack* 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      interface Server      interface Peer
begin                 begin                  inherits Client, Server
 with Any              with Server          begin
  op availFiles          op enquire         end
  op reqFile             op getLength
end                      op getPack
                      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, a brief review of basic interaction models for concurrent processes with respect to distributed interaction is given.

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 [61] and Java [35], 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-serialized 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 serialized, which is the approach taken in, e.g., Hybrid. However, when the language is restricted to serialized 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 instances, 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 waits 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* [74] (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 the 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, 41, 73, 74]. This approach seems well-suited to model communication in distributed environments, reflecting the fact that communication in a network is not instantaneous. Generative communication in 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 the Actor model and Linda. Actors do not distinguish replies from invocations, so capturing method calls with Actors quickly becomes unwieldy [3]. Asynchronous message passing gives better control and efficiency, but 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. We believe that a satisfactory notion of method calls for the distributed setting should be asynchronous, combining the advantages of asynchronous message passing with the structuring mechanism provided by the method concept.

3.1 Syntax

A simple language for concurrent objects is now presented, which combines *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 (client and server).

At the imperative level attributes and method declarations are organized in classes, which may have value and object parameters. Objects are dynamically created instances of classes, their persistent state consists of 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 of 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 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 with type constructors.

Definition 3 Let τ_B be a set of basic data types and $\tau_{\mathcal{I}}$ a set of interfaces, such that $\tau_B \cap \tau_{\mathcal{I}} = \emptyset$. Let τ denote the set of all types including the basic and interface types; $\tau_B \subseteq \tau$ and $\tau_{\mathcal{I}} \subseteq \tau$. Type schemes such as parameterized data types may be applied to types in τ to form new types in τ.

It will be 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. Furthermore, it is assumed that $\mathsf{Set}[T]$ and $\mathsf{List}[T]$ are included among the type schemes. Let T_B and T be typical elements of τ_B and τ. Expressions without side effects are given by a functional language $\mathcal{F}$ defined as follows:

Definition 4 Let $\mathcal{F}$ be a strongly typed functional language which consists of expressions $e \in \mathsf{Expr}$ constructed in a type correct way from

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

In particular, we denote by $\mathsf{ObjExpr}$ and $\mathsf{BoolExpr}$ two subtypes of Expr; expressions of these types reduce to object variables (typed by interface) and booleans, respectively. Note that there are no constructors or field access functions for terms of interface types in the functional language $\mathcal{F}$, but object identifiers may be compared by an identity function.

Given a typing environment Γ and an expression $e \in \mathsf{Expr}$, we let $\Gamma \vdash_{\mathcal{F}} e : T$ denote that e has type T in Γ. Γ includes all relevant type information for variables, constants, and functions. It is assumed that $\mathcal{F}$ is *type sound* [64]: well-typed expressions remain well-typed during evaluation and the evaluation of expressions terminates: If $\Gamma \vdash_{\mathcal{F}} e : T$ and $e \longrightarrow e'$ then $\Gamma \vdash_{\mathcal{F}} e' : T'$ such that $T' \prec T$.

Classes. An object-oriented language will now by constructed which extends the functional language $\mathcal{F}$. We denote by Var a set of program variables and by Label a set of method call identifiers. Classes are defined in a traditional way, including declarations of persistent state variables and method definitions.

$$\begin{array}{lll}
CL & ::= & [\mathbf{class}\ C\ [(\mathit{Param})]^{?}\ [\mathbf{contracts}\ \mathit{InhList}]^{?}\ [\mathbf{implements}\ \mathit{InhList}]^{?} \\
 & & [\mathbf{inherits}\ \mathit{ClassInhList}]^{?}\ \mathbf{begin}\ [\mathbf{var}\ \mathit{Vdecl}]^{?}\ [[\mathbf{with}\ I]^{?}\ \mathit{Mdecls}]^{*}\ \mathbf{end}]^{*} \\
\mathit{Vdecl} & ::= & [v : T[= e]^{?}]^{+}_{;} \\
\mathit{Mdecls} & ::= & [\mathit{Msig} == [\mathbf{var}\ \mathit{Vdecl};]^{?}\ \textsc{s}]^{+} \\
\mathit{ClassInhList} & ::= & [C[(\textsc{e})]^{?}]^{+}_{,}
\end{array}$$

Figure 2: An outline of the language syntax for classes, excluding expressions e, expression lists $\textsc{e}$, and statement lists $\textsc{s}$.

Definition 5 (Classes.) 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 interfaces, *Inh* is a list of instantiated classes, defining class inheritance, *Var* is a list of typed program variables (possibly with initial expressions), and *Mtd* is a set of methods.

Each method is equipped with a special field specifying the cointerface associated with the method. For purely internal methods, the cointerface field will simply contain a special name *Void*. Notice that *Impl* represents interfaces supported by this class, whereas *Contract* represents interfaces supported by this 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 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 will be used to check any 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. For convenience, dot notation will be used to denote the different elements of a class; e.g., *Cl.Var* denotes the variable list of a class *Cl*. A graphical representation of a class may be given following the BNF 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 parameter. Overloading of methods is allowed. The pseudo-variable *self* is used for self reference in the language and its value cannot be modified. Issues related to inheritance are considered in Section 5, until then, we consider classes without explicit inheritance.

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

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

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.

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 method instances in another order. A method instance is, roughly speaking, a list S of program statements evaluated in the context of local variables. Due to the possible interleavings of different method executions, the values of an object's program variables are not entirely controlled by a method instance 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. Assignment to local and object variables is expressed as $\mathrm{V} := \mathrm{E}$ for (the same number of) program variables V and expressions E. The syntax for program statements is given in Figure 3.

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 must be called internally if it is internal (with *Void* as interface), otherwise externally. 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 is actually needed by the process.

To fetch the return values from the 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 $x.m(\text{E}; \text{V})$, which is defined by $t!x.m(\text{E})$; $t?(\text{V})$ 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, where the callee will be equal to *self*. Therefore the reply statement $t?(\text{V})$ will enable execution of the call identified by t when this call is local. The language does not support monitor reentrance; mutual or cyclic synchronous calls between objects may therefore lead to deadlock.

Potential suspension is 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 local variable scope, corresponding to a series of potential suspension points. Let S_1 and S_2 denote statement lists; in S_1; **await** g; 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 $t?$ succeeds if the reply to the method invocation with label t has arrived. Guards may be composed: $g_1 \wedge g_2$ and $g_1 \vee g_2$ form compound guards from guards g_1 and g_2. 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 $t?(\text{V})$ = **await** $t?$; $t?(\text{V})$
await $r(\text{E}; \text{V}) = t!r(\text{E})$; **await** $t?(\text{V})$, where t is a fresh label variable.

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 calls may be evaluated while waiting for a reply. If the called object does not reply at all, 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 the reply has arrived, 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 will be virtually bound. Let the nominal subtype relation $\prec$ be a reflexive partial ordering on types, including interfaces. We let $\perp$ represent an undefined (and illegal) type; thus $\perp \prec T$, $T \prec \perp$, and $\perp \prec \perp$ are undefined for any type T. 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 *Void* which is only related to itself. The semantic constraints associated with interfaces motivated a nominal type system such that the semantic constraints restrict a structural subtype relation which ensures substitutability; If $T \prec T'$ then any value of T may masquerade as a value of T' [12]. For product types R and R', $R \prec R'$ is the point-wise extension of the subtype relation; i.e., R and R' have the same length l and $T_i \prec T'_i$ 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, $\prec$ is extended to function spaces $A \rightarrow B$, where A and B are (possibly empty) product types:

$$A \rightarrow B \prec A' \rightarrow B' = A \prec A' \wedge B' \prec B$$

Let the function *Sig* give the signature of a method, defined by $\mathit{Sig}(m) = \mathit{type}(m.\mathit{Inpar}) \rightarrow \mathit{type}(m.\mathit{Outpar})$ where *type* gives the product of the types in a parameter declaration. The static analysis of a synchronous internal call $m(\text{E}; \text{V})$ will assign 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 $T_{\text{E}} \rightarrow T_{\text{V}} \prec T_1 \rightarrow T_2$. 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 will verify that it implements the methods declared in its interfaces. Assuming that any object variable typed by the interface I is an instance of a class implementing I, method binding will succeed regardless of the class of the object. At runtime, the class of the object will be 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 type of the actual parameter values and the actual cointerface are needed in order 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 uniquely 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 will automatically resume if the connection is reestablished. A client may run several downloads concurrently, at different speeds. We assume that every node in the network has an associated database with shared files. Downloaded files are stored in this database, which is not modeled here but implements the interface *DB*:

```
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*. *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. The *Node* class is given in Figure 4. 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 *getPack* a particular pack in the transmission of a file. The list constructor is represented by semicolon. For $x : T$ and $\text{s} : \mathsf{List}[T]$, we let $hd(x; \text{s}) = x$ and $tl(x; \text{s}) = \text{s}$, and $\text{s}[i]$ denotes the i'th element of s, provided $i \leq length(\text{s})$.

```
class Node (db:DB)
  implements Peer
begin
with Server
 op enquire(out files:List[Str]) == await db.listFiles(;files)
 op getLength(in fId:Str out lth:Nat) == await db.getLength(fId;lth)
 op getPack(in fId:Str; pNbr:Nat out pack:List[Data]) == var f:List[Data];
      await db.getFile(fId;f); pack:=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; fId:Str) ==
   var file:List[Data]; pack:List[Data]; lth:Nat;
    await sId.getLength(fId;lth); while (lth > 0) do
     await sId.getPack(fId, lth; pack); file:=(pack ; file); lth:=lth - 1 od;
   !db.storeFile(fId, file)
end
```

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

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, such that the typing environment resulting from 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 will be given as a *family of mappings*: $\Gamma_{\mathcal{I}}$ describes the binding of interface names to interfaces, $\Gamma_{\mathcal{C}}$ the binding of class names to classes, Γ_{V} the binding of program variables to types, and the mappings Γ_{AI} and Γ_{AE} will be related to the binding of asynchronous internal and external method calls. 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 6 Let n be a name, d a declaration, $i \in I$ a mapping index, and $[n{\mapsto_i}d]$ 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 can be extracted from Γ as follows

$$\begin{array}{lcl} \emptyset_i & = & \varepsilon \\ (\Gamma + [n{\mapsto_{i'}}d])_i & = & \textbf{if } i = i' \textbf{ then } \Gamma_i + [n \mapsto d] \textbf{ else } \Gamma_i. \end{array}$$

For an indexed mapping Γ_i, mapping application is defined by

$$\begin{array}{lcl} \varepsilon(n) & = & \bot \\ (\Gamma_i + [n{\mapsto_i}d])(n') & = & \textbf{if } n = n' \textbf{ then } d \textbf{ else } \Gamma_i(n'). \end{array}$$

3.4.1 Typing of Programs

A class or interface declaration binds a name to a class or interface, respectively. Class and interface names need not be distinct. A program consists of a list of interface and class declarations, represented by the mappings $\Gamma_{\mathcal{I}}$ and $\Gamma_{\mathcal{C}}$.

$$\begin{array}{ll} \Gamma_{\mathcal{I}} & = \tau_{\mathcal{I}} \rightarrow \mathcal{I} \\ \Gamma_{\mathcal{C}} & = \tau_{\mathcal{C}} \rightarrow \mathcal{C} \end{array}$$

In a nominal type system, each interface and class is type checked relying on these mappings.

$$\text{(PROG)} \quad \frac{\begin{array}{c} \forall I \in \tau_{\mathcal{I}} \ \cdot \ \Gamma_{\mathcal{I}} + [\mathit{self}{\mapsto_{\mathrm{V}}}I] \vdash \Gamma_{\mathcal{I}}(I) \\ \forall C \in \tau_{\mathcal{C}} \ \cdot \ \Gamma_{\mathcal{I}} + [\mathit{self}{\mapsto_{\mathrm{V}}}C] \vdash \Gamma_{\mathcal{C}}(C) \end{array}}{\emptyset \vdash \Gamma_{\mathcal{I}}, \Gamma_{\mathcal{C}}}$$

When type checking a class or interface, *self* is bound to the class or interface name. Notice that each interface or class declaration is type checked without any knowledge of $\Gamma_{\mathcal{C}}$, which reflects the fact that object variables are typed by interface and allows a modular type checking of classes. We now define a function which verifies that formal and actual parameters of interfaces and classes match.

Definition 7 Let Γ be a typing environment and *inhlist* an inheritance list. Define *matchparam* by

$$\begin{array}{lcl} \mathit{matchparam}_i(\Gamma, \varepsilon) & = & \mathsf{true} \\ \mathit{matchparam}_i(\Gamma, (\mathit{name}(\mathrm{E}), \mathit{inhlist})) & = & \Gamma \vdash_{\mathcal{F}} \mathrm{E} : T \ \wedge \ T \prec \mathit{type}(\Gamma_i(\mathit{name}).\mathit{Param}) \\ & & \wedge\, \mathit{matchparam}_i(\Gamma, \mathit{inhlist}) \end{array}$$

Here *name* is understood as an interface name when the index i is $\mathcal{I}$, and a class name when i is $\mathcal{C}$. The rule for type checking interface declarations may now be given as follows:

$$\text{(INTERFACE)} \quad \frac{\begin{array}{c} \Gamma \vdash_{\mathrm{V}} \ \mathit{Param} \ \langle\, \Delta \,\rangle \qquad\qquad \mathit{matchparam}_{\mathcal{I}}(\Gamma + \Delta, \mathit{Inh}) \\ \forall m : \mathit{Mtd} \ \cdot \ \Gamma + \Delta \vdash_{\mathrm{V}} \ m.\mathit{Inpar}; m.\mathit{Outpar} \ \langle\, \Delta' \,\rangle \end{array}}{\Gamma \vdash \mathit{interface}\,(\mathit{Param}, \mathit{Inh}, \mathit{Mtd}, \mathit{Req})}$$

Although not included in the above rule, it is here assumed that method names are not reused in the same interface. We now consider type checking of classes in detail.

$$\text{(CLASS)}\quad \frac{\begin{array}{c}\Gamma \vdash_{\mathrm{V}} \ \mathit{Param}; \mathit{Var} \ \langle\ \Delta\ \rangle \qquad\qquad \forall m \in \mathit{Mtd} \ \cdot \ \Gamma + \Delta \vdash m \\ \forall I \in (\mathit{Impl}; \mathit{Contract}) \cdot \forall m' \in \Gamma_{\mathcal{I}}(I).\mathit{Mtd} \cdot \exists m \in \mathit{Mtd} \cdot \\ \mathit{Sig}(m') \prec \mathit{Sig}(m) \wedge m'.\mathit{Co} \prec m.\mathit{Co}\end{array}}{\Gamma \vdash \mathit{class}\,(\mathit{Param}, \mathit{Impl}, \mathit{Contract}, \varepsilon, \mathit{Var}, \mathit{Mtd})}$$

A class may implement a number of interfaces. For each interface, we need to check that the class provides type correct method bodies for the signatures of the interface. In this case, the declared signatures of the methods in the class are correct with respect to the interfaces of the class, and we need to check that the bodies of the methods are type correct in the environment of the class parameters and attributes. Thus, before type checking the method bodies of a class, the typing environment is extended with class parameters and class variable declarations. Moreover, class variable declarations will be type checked after the typing environment has been extended with class parameters as variable declarations may include initial expressions that use these parameters. All the methods of the class are then type checked in the typing environment resulting from the typing of attributes and the parameters, including the *caller*. At this point Γ_{AI} and Γ_{AE} are empty, since no asynchronous method invocation has been encountered.

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

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

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

Similar rules may be added for up- and down-casting of object expressions. A qualification may be ignored in the operational semantics except in the case of down-casting, where one must ensure that the current class implements or contracts the down-cast interface.

3.4.2 Typing of Parameter and Variable Declarations

A method may have local variable declarations preceding the program statements of the method. Both class and local variable declarations may extend the typing environment Γ with new variables, provided that these are not previously declared in the typing environment. The type rules for variable and parameter declarations are given in Figure 5.

$$\text{(VAR-AX)}\quad \Gamma \vdash_{\mathrm{V}} \ \varepsilon \qquad\qquad \text{(VAR)}\quad \frac{\Gamma_{\mathrm{V}}(v) = \bot \qquad\qquad T \prec \mathsf{Data}}{\Gamma \vdash_{\mathrm{V}} \ v : T \ \langle\ [v \mapsto_{\mathrm{V}} T]\ \rangle}$$

$$\text{(VAR-EXPR)}\quad \frac{\Gamma \vdash_{\mathrm{V}} \ v : T \ \langle\ [v \mapsto_{\mathrm{V}} T]\ \rangle \quad \Gamma + [v \mapsto_{\mathrm{V}} T] \vdash_{\mathrm{S}} \ v := e}{\Gamma \vdash_{\mathrm{V}} \ v : T = e \ \langle\ [v \mapsto_{\mathrm{V}} T]\ \rangle}$$

Figure 5: Typing of variable and parameter declarations.

3.4.3 Typing of Basic Statements

The typing of basic statements not involving class related issues is given in Figure 6, as well as a statement for object creation. Notice that the typing system for $\mathcal{F}$ is used to type check expressions and is orthogonal to the object-oriented type system. The last premise of the NEW rule ensures that the new object implements an interface which is a subtype of the declared interface of the variable v. Recall that $\Gamma \vdash_{\mathrm{S}} \ \mathrm{S}$ abbreviates $\Gamma \vdash_{\mathrm{S}} \ \mathrm{S} \ \langle\ \emptyset\ \rangle$.

$$\text{(SKIP)}\quad \Gamma \vdash_{\mathrm{S}} \ \mathbf{skip} \qquad\qquad \text{(ASSIGN)}\quad \frac{\Gamma \vdash_{\mathcal{F}} \ \mathrm{E} : T' \quad T' \prec \Gamma_{\mathrm{V}}(\mathrm{V})}{\Gamma \vdash_{\mathrm{S}} \ \mathrm{V} := \mathrm{E}}$$

$$\text{(EMPTY)}\quad \Gamma \vdash_{\mathrm{S}} \ \varepsilon \qquad\qquad \text{(AWAIT-}\vee\text{)}\quad \frac{\Gamma \vdash_{\mathrm{S}} \ \mathbf{await}\ g_1 \quad \Gamma \vdash_{\mathrm{S}} \ \mathbf{await}\ g_2}{\Gamma \vdash_{\mathrm{S}} \ \mathbf{await}\ g_1 \vee g_2}$$

$$\text{(AWAIT-}\mathsf{Bool}\text{)}\quad \frac{\Gamma \vdash_{\mathcal{F}} \ g : \mathsf{Bool}}{\Gamma \vdash_{\mathrm{S}} \ \mathbf{await}\ g} \qquad\qquad \text{(AWAIT-}\wedge\text{)}\quad \frac{\Gamma \vdash_{\mathrm{S}} \ \mathbf{await}\ g_1 \quad \Gamma \vdash_{\mathrm{S}} \ \mathbf{await}\ g_2}{\Gamma \vdash_{\mathrm{S}} \ \mathbf{await}\ g_1 \wedge g_2}$$

$$\text{(NEW)}\quad \frac{\begin{array}{c}\Gamma \vdash_{\mathcal{F}} \ \mathrm{E} : T \quad T \prec \mathit{type}(\Gamma_{\mathcal{C}}(C).\mathit{Param}) \\ \exists I \in (\Gamma_{\mathcal{C}}(C).\mathit{Impl}; \Gamma_{\mathcal{C}}(C).\mathit{Contract}) \ \cdot \ I \prec \Gamma_{\mathrm{V}}(v)\end{array}}{\Gamma \vdash_{\mathrm{S}} \ v := \mathbf{new}\ C(\mathrm{E})}$$

Figure 6: Typing of basic statements.

3.4.4 Typing of Asynchronous Calls

The types of actual and formal parameters must be compared when type checking method calls. The *match* predicate verifies that the actual parameters and cointerface of a given method invocation correspond to the formal parameters and cointerface of a method declaration in a class or interface.

Definition 8 Let T and U be types (or type products), I an interface, and M a set of methods.

$$\begin{array}{l}\mathrm{match}(I, T, U, \emptyset) = \mathsf{false} \\ \mathrm{match}(I, T, U, \{m\} \cup \mathrm{M}) = (I \prec m.\mathit{Co} \wedge T \rightarrow U \prec \mathit{Sig}(m))\end{array}$$

Remark that the set constructor is associative and commutative, following rewriting logic conventions.

For synchronous calls, parameter type checking is straightforward as the types of the actual in- and out-parameters can be derived directly from the calls. In contrast, type checking asynchronous invocations is more involved since the correspondence between in- and out-parameters is controlled by label values. The increased freedom in the language gained from using labels, complicates the type analysis. In order to address asynchronous invocations, we introduce the mappings Γ_{AI} and Γ_{AE} from labels to *sets* of tuples containing the information needed to later type check method calls associated with the labels.

$$\begin{aligned}\Gamma_{AI} &= \mathsf{Label} \rightarrow \mathsf{SetOf}\{\mathsf{List}[\tau_{\mathcal{C}}] \times \mathsf{Mtd} \times \mathsf{Data}\}\\ \Gamma_{AE} &= \mathsf{Label} \rightarrow \mathsf{SetOf}\{\mathsf{List}[\tau_{\mathcal{I}}] \times \mathsf{Mtd} \times \mathsf{Data}\}\end{aligned}$$

The elements in Γ_{AI} represent internal asynchronous pending calls and the elements in Γ_{AE} external ones, where the types of the out-parameters have yet to be resolved. Both mappings Γ_{AI} and Γ_{AE} are included in the typing environment Γ when 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 can be type checked against the supertype Data. For an external asynchronous invocation, the type of the return values is yet unknown, thus, type checking with the exact type of the out variables must be postponed until the corresponding reply sentence arrives at a later time. Therefore, the invocation is added to the set of pending external calls Γ_{AE} with a mapping from its associated label to the interface, method name, and input type of the call. However, already at this stage some erroneous invocations may be eliminated by type checking against the weakest possible type for actual out variables.

For internal calls it is checked that the method is internal, with *Void* as cointerface. For external calls $x.m$ it is checked that the interface of x offers a method m with a cointerface *contracted* by the current class (and therefore supported by the actual calling object). This implies that *remote calls to* self are allowed when the class contracts an interface used as the cointerface of a method m in the current class.

A reply is requested through a reply sentence or a guard, if there are pending invocations with the same label. For a reply sentence, the matching

$$\text{(EXT-SYNC)}\quad \frac{\begin{array}{c}\Gamma \vdash_{\mathcal{F}} e : I \qquad \Gamma \vdash_{\mathcal{F}} \textsc{e} : T\\ \text{match}(\Gamma_{\mathcal{C}}(\Gamma_{\mathrm{V}}(\mathit{self})).\mathit{Contract}, T, \Gamma_{\mathrm{V}}(\textsc{v}), \Gamma_{\mathcal{I}}(I).\mathit{Mtd}(m))\end{array}}{\Gamma \vdash_{\mathrm{S}} e.m(\textsc{e};\textsc{v})}$$

$$\text{(EXT-ASYNC)}\quad \frac{\begin{array}{c}\Gamma \vdash_{\mathcal{F}} e : I \qquad \Gamma \vdash_{\mathcal{F}} \textsc{e} : T\\ \text{match}(\Gamma_{\mathcal{C}}(\Gamma_{\mathrm{V}}(\mathit{self})).\mathit{Contract}, T, \mathsf{Data}, \Gamma_{\mathcal{I}}(I).\mathit{Mtd}(m))\end{array}}{\Gamma \vdash_{\mathrm{S}} !e.m(\textsc{e})}$$

$$\text{(EXT-ASYNC-L)}\quad \frac{\begin{array}{c}\Gamma \vdash_{\mathcal{F}} e : I \qquad \Gamma \vdash_{\mathcal{F}} \textsc{e} : T \qquad \Gamma_{\mathrm{V}}(t) = \mathsf{Label}\\ \text{match}(\Gamma_{\mathcal{C}}(\Gamma_{\mathrm{V}}(\mathit{self})).\mathit{Contract}, T, \mathsf{Data}, \Gamma_{\mathcal{I}}(I).\mathit{Mtd}(m)))\end{array}}{\Gamma \vdash_{\mathrm{S}} t!e.m(\textsc{e}) \ \langle\ [t \mapsto_{AE} \{\langle I, m, T\rangle\}] + [t \mapsto_{AI} \emptyset]\ \rangle}$$

$$\text{(INT-SYNC)}\quad \frac{\text{match}(\mathit{Void}, T, \Gamma_{\mathrm{V}}(\textsc{v}), \Gamma_{\mathcal{C}}(\Gamma_{\mathrm{V}}(\mathit{self})).\mathit{Mtd}(m)) \quad \Gamma \vdash_{\mathcal{F}} \textsc{e} : T}{\Gamma \vdash_{\mathrm{S}} m(\textsc{e};\textsc{v})}$$

$$\text{(INT-ASYNC)}\quad \frac{\text{match}(\mathit{Void}, T, \mathsf{Data}, \Gamma_{\mathcal{C}}(\Gamma_{\mathrm{V}}(\mathit{self})).\mathit{Mtd}(m)) \quad \Gamma \vdash_{\mathcal{F}} \textsc{e} : T}{\Gamma \vdash_{\mathrm{S}} !m(\textsc{e})}$$

$$\text{(INT-ASYNC-L)}\quad \frac{\begin{array}{c}\Gamma_{\mathrm{V}}(t) = \mathsf{Label} \qquad \Gamma \vdash_{\mathcal{F}} \textsc{e} : T\\ \text{match}(\mathit{Void}, T, \mathsf{Data}, \Gamma_{\mathcal{C}}(\Gamma_{\mathrm{V}}(\mathit{self})).\mathit{Mtd}(m))\end{array}}{\Gamma \vdash_{\mathrm{S}} t!m(\textsc{e}) \ \langle\ [t \mapsto_{AI} \{\langle \Gamma_{\mathrm{V}}(\mathit{self}), m, T\rangle\}] + [t \mapsto_{AE} \emptyset]\ \rangle}$$

$$\text{(AWAIT-REPLY)}\quad \frac{\Gamma_{\mathrm{V}}(t) = \mathsf{Label} \qquad \Gamma_{AE}(t) \cup \Gamma_{AI}(t) \neq \emptyset}{\Gamma \vdash_{\mathrm{S}} \textbf{await}\ t?}$$

$$\text{(REPLY)}\quad \frac{\begin{array}{c}\Gamma_{\mathrm{V}}(t) = \mathsf{Label} \qquad \Gamma_{AE}(t) \cup \Gamma_{AI}(t) \neq \emptyset\\ \forall \langle C, m, T\rangle \in \Gamma_{AI}(t) \ \cdot\ \text{match}(\mathit{Void}, T, \Gamma_{\mathrm{V}}(\textsc{v}), \Gamma_{\mathcal{C}}(C).\mathit{Mtd}(m))\\ \forall \langle I, m, T\rangle \in \Gamma_{AE}(t) \ \cdot\ \text{match}(\Gamma_{\mathcal{C}}(\Gamma_{\mathrm{V}}(\mathit{self})).\mathit{Contract}, T, \Gamma_{\mathrm{V}}(\textsc{v}), \Gamma_{\mathcal{I}}(I).\mathit{Mtd}(m))\end{array}}{\Gamma \vdash_{\mathrm{S}} t?(\textsc{v}) \ \langle\ [t \mapsto_{AI} \emptyset] + [t \mapsto_{AE} \emptyset]\ \rangle}$$

Figure 7: Typing of external and internal invocation and reply sentences.

invocations must be type checked again as the types of the out-parameters are now known. If the reply rule succeeds, all pending calls using a label have been type checked against the type of the variables in which the reply values are stored. This type checking depends on the interface of the callee for external calls and the class of *self* for internal calls. These calls can therefore be removed from the sets of pending calls. Note that the type correctness of pending calls without a corresponding reply sentence is given by the typing rules for method invocations. Some initial properties of the type system are now considered.

Lemma 1 *No type checked methods use undeclared program variables.*

Proof. We consider a method body $\langle \mathit{Var}, \mathit{Code}\rangle$. Initial variable declarations *Var* type-correctly extend the typing environment. The typing rule VAR-EXPR ensures that only declared variables are read if initial values are assigned to the new variables. Denote by Γ the extended environment. The proof continues by induction over the construction of program sentences. By Definition 4 the functional language $\mathcal{F}$ is strongly typed; We may assume that $\Gamma \vdash_{\mathcal{F}} \ \mathrm{E} : T$ such that $T \neq \bot$ for all expressions E in *Code*, so the lemma holds for E. For $\mathrm{V} := \mathrm{E}$, we assume that $\Gamma \vdash_{\mathcal{F}} \ \mathrm{E} : T$ and that $T \prec \Gamma_{\mathrm{V}}(\mathrm{V})$. If V has not been declared, $\Gamma_{\mathrm{V}}(\mathrm{V}) = \bot$, which is a contradiction. For object creation $v :=$ **new** $C(\mathrm{E})$, there exists an interface $I \in \Gamma_{\mathcal{C}}(C).\mathit{Impl}; \Gamma_{\mathcal{C}}(C).\mathit{Contract}$ such that $I \prec \Gamma_{\mathrm{V}}(x)$. Consequently $\Gamma_{\mathrm{V}}(x) \neq \bot$. For $e.m(\mathrm{E};\mathrm{V})$, we may assume $\Gamma \vdash_{\mathcal{F}} \ e : I$, $\Gamma \vdash_{\mathcal{F}} \ \mathrm{E} : T$ such that $\mathit{match}(\Gamma_{\mathcal{C}}(\Gamma_{\mathrm{V}}(\mathit{self})).\mathit{Contract}, T, \Gamma_{\mathrm{V}}(\mathrm{V}), \Gamma_{\mathcal{I}}(I).\mathit{Mtd}(m))$, and $\Gamma_{\mathrm{V}}(\mathrm{V}) \neq \bot$. For $e.m(\mathrm{E})$ we can use the same argument, discarding return values. For $t!e.m(\mathrm{E})$ we can use a similar argument, the additional condition $\Gamma_{\mathrm{V}}(t) = \mathsf{Label}$ guarantees that t has been declared. Internal calls follow similar arguments, making use of $\Gamma_{\mathcal{C}}(\Gamma_{\mathrm{V}}(\mathit{self})).\mathit{Mtd}(m)$ for the match function. For $t?(\mathrm{V})$ we may assume that $\Gamma_{\mathrm{V}}(t) = \mathsf{Label}$, so t is declared, and there are pending calls to t, which guarantees that $\Gamma_{\mathrm{V}}(\mathrm{V}) \neq \bot$. All variables in a boolean guard g are declared because $\Gamma \vdash_{\mathcal{F}} \ g : \mathsf{Bool}$. For the reply guard sentence **await** $t?$, t has been declared because $\Gamma_{\mathrm{V}}(t) = \mathsf{Label}$. The sequential composition $\mathrm{S}_1;\mathrm{S}_2$ follows by the induction hypotheses for S_1 and S_2. ■

Lemma 2 *If there exist variables* V *with* $\Gamma_{\mathrm{V}}(\mathrm{V}) = T$ *such that a method call* $e.m(\mathrm{E};\mathrm{V})$ *or* $m(\mathrm{E};\mathrm{V})$ *is type correct, then the asynchronous calls* $!e.m(\mathrm{E})$ *and* $!m(\mathrm{E})$ *are also type correct.*

Proof. If the external call $e.m(\mathrm{E};\mathrm{V})$ is type correct there exists an interface I and types T and T' such that $\Gamma \vdash_{\mathcal{F}} \ e : I$, $\Gamma \vdash_{\mathcal{F}} \ \mathrm{E} : T$, $\Gamma_{\mathrm{V}}(\mathrm{V}) = T'$, and

$$\mathrm{match}(\Gamma_{\mathcal{C}}(\Gamma_{\mathrm{V}}(\mathit{self})).\mathit{Contract}, T, T', \Gamma_{\mathcal{I}}(I).\mathit{Mtd}(m)).$$

As $T' \prec \mathsf{Data}$ for any type T', we have

$$\mathrm{match}(\Gamma_{\mathcal{C}}(\Gamma_{\mathrm{V}}(\mathit{self})).\mathit{Contract}, T, \mathsf{Data}, \Gamma_{\mathcal{I}}(I).\mathit{Mtd}(m)).$$

Consequently the call $!e.m(\mathrm{E})$ is also type correct. The case for the internal calls $m(\mathrm{E};\mathrm{V})$ and $!m(\mathrm{E})$ is similar. ■

It follows from Lemma 2 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 3 *Let* Γ *be a mapping family such that* $\Gamma_{AI} = \Gamma_{AE} = \emptyset$. *For any sentence list* S *with a type judgment* $\Gamma \vdash_{\mathrm{S}} \ \mathrm{S} \langle \Delta \rangle$, *the set* Δ *contains exactly the labeled method invocations that have not been verified as type correct by the analysis of* S *with respect to possible corresponding reply sentences.*

Proof. The proof is by induction over the length of S. For an empty list of program statements ε, we have $\Gamma \vdash \mathrm{S}$. For the induction step, we assume $\Gamma \vdash \mathrm{S} \langle \Delta \rangle$ where Δ contains the method invocations that may need further analysis due to possible corresponding reply sentences. We prove that for $\Gamma \vdash \mathrm{S}; s \langle \Delta + \Delta' \rangle$, the mappings $(\Delta + \Delta')_{AI}$ and $(\Delta + \Delta')_{AE}$ contain the method invocations that may now need further analysis. Clearly the skip, assignment, new, reply, and guard sentences do not result in new method calls, so $\Delta' = \emptyset$. We now consider $e.m(\mathrm{E}; \mathrm{V})$, $m(\mathrm{E}; \mathrm{V})$, $!e.m(\mathrm{E})$, and $!m(\mathrm{E})$. The type system allows the type correctness of these statements to be verified directly. Consequently $\Delta' = \emptyset$. For the asynchronous invocation sentences $t!e.m(\mathrm{E})$ and $t!m(\mathrm{E})$, the typing rules verify correctness with the weakest possible type for actual out-variables. However, a later reply sentence may impose a restriction on the type of the out values. We may here assume that $\Gamma \vdash_{\mathcal{F}} e : I$, $\Gamma_{\mathrm{V}}(\mathit{self}) = C$, and $\Gamma \vdash_{\mathcal{F}} \mathrm{E} : T$. Consequently the invocation is recorded, yielding $\Delta' = [t \mapsto_{AE} \Delta_{AE}(t) \cup \{\langle I, m, T \rangle\}]$ or $\Delta' = [t \mapsto_{AI} \Delta_{AI}(t) \cup \{\langle C, m, T \rangle\}]$, respectively. The reply sentence $t?(v)$ does not result in new invocations. By assumption there are pending calls to t in Δ. The rule for $t?(v)$ will type check the calls in $\Delta_{AI}(t) \cup \Delta_{AE}(t)$. Consequently these calls can be removed from Δ_{AI} and Δ_{AE}; the type rule gives us $\Delta' = [t \mapsto_{AI} \emptyset] + [t \mapsto_{AE} \emptyset]$ and $\Delta + \Delta'$ contains the calls that may need further analysis. Sequential composition $\mathrm{S}; s$ follows directly from the induction hypothesis. ■

It can now be shown that all asynchronous invocations can be precisely type checked.

Lemma 4 *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 the program, with the type judgment $\Gamma \vdash_S \mathrm{S} \langle \Delta \rangle$. As $\Gamma_{AI} = \Gamma_{AE} = \emptyset$, Lemma 3 gives us that Δ contains exactly the method invocations that may need further type checking. However, as the entire method body has been type checked, the invocations in Δ 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 and for any label t in this program, $\#(\Gamma_{AI}(t) \cup \Gamma_{AE}(t)) \leq 1$ and there can be at most one reply sentence in the program corresponding to any such label t. Consequently, the following observation can be made for the language and type system considered in this section:

Lemma 5 *In a well-typed method body, a type correct signature for every method invocation may be deterministically derived from the actual types of input expressions and output variables.*

It is hereafter assumed that this type correct signature is included as an argument to the runtime method invocation. However, there may be several possible valid cointerfaces for a given method invocation. We shall assume that one such is chosen, either by the parser or by some explicit syntax extending the given language, and similarly included in the runtime method invocation.

3.5 Operational Semantics

The operational semantics of the language is defined using rewriting logic [55]. 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, 55], including Petri nets, CCS, Actors, and Unity, as well as the ODP computational model [60]. 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 [33] 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 compositional operational semantics. In fact, structural operational semantics can be uniformly mapped into RL specifications [56].

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. We here assume that the types of the actual in- and out-parameters of the call as well as the cointerface have been added to the method invocation as additional arguments *Sig* and *Co* at compile-time. 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 $m(\mathit{Sig}, \mathit{Co}, \mathrm{E})$ results in the return values E', the call is reflected by two messages $\mathit{invoc}(m, \mathit{Sig}, \mathit{Co}, (n\ o_1\ \mathrm{E}))$ **to** o_2 and $\mathit{comp}(n, \mathrm{E}')$ **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 will be organized around a *message queue* which contains incoming messages and a *process queue* which contains suspended processes; i.e., remaining parts of method instances.

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, as shown in [43, 45].) 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 sentences 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

$$\langle \mathit{Ob} \mid \mathit{Cl}, \mathit{Att}, \mathit{Pr}, \mathit{PrQ}, \mathit{EvQ}, \mathit{Lab}\rangle,$$

where *Ob* is the object identifier, *Cl* the class name, *Att* the object state, *Pr* the active process, *PrQ* a multiset of suspended processes with unspecified queue ordering, and *EvQ* a multiset of unprocessed messages, respectively. Let $T_<$ be a type partially ordered by $<$, with least element 1, and let $\mathit{next} : T_< \to T_<$ be such that $\forall x \,.\, x < \mathit{next}(x)$. *Lab* is the method call identifier corresponding to labels in the language, of type $T_<$. Thus, the object identifier *Ob* and the generated local label value provide a globally unique identifier for each method call.

The classes of Creol are represented by RL objects

$$\langle \mathit{Cl} \mid \mathit{Param}, \mathit{Att}, \mathit{Mtds},\ \mathit{Tok}\rangle,$$

where *Cl* is the class name, *Param* a list of parameters, *Att* a list of attributes, *Mtds* a multiset of methods, and *Tok* an arbitrary term of sort $T_<$. A method has a name, signature, cointerface, 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 by 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*.

3.5.2 Concurrent Transitions

Concurrent change is achieved in the operational semantics by applying concurrent rewrite steps to state configurations. 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 for the program $\text{V} := \text{E}$ binds the values of the expression list E to the list V of local and object variables.
- *Rules for suspension of the active process:* When an active process guard evaluates to false, the process and its local variables are suspended, leaving *Pr* empty.
- *Rules that activate suspended processes:* When *Pr* is empty, suspended processes may be activated. When this happens, the local state is replaced.

In addition, a *transport rule* moves messages into the message queues, representing network flow. When auxiliary functions are needed in the semantics, these are defined in equational logic, and are evaluated in between the state transitions [55]. In the presentation irrelevant attributes are ignored in the style of Full Maude [20]. The operational semantics is given in Figures 8 and 9. The rules are now briefly presented. A detailed discussion may be found in [43].

Whitespace is used as the constructor of multisets, such as *PrQ* and *EvQ*, as well as variable and expression lists, whereas semicolon is used as the constructor of lists of statements in order to improve readability. As before, + is the constructor for mappings. In the assignment rule, a list of expressions is evaluated and bound to a list of program variables. The auxiliary function *eval* evaluates an expression with a given *state*; i.e., a mapping from variable names to values.

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

In the object creation rule, a local state is constructed from the attribute list and parameters of the class, an object identifier for the new object is

$$
\begin{array}{l}
\langle o : Ob \,|\, Att : \textsc{a}, Pr : \langle (v\ \textsc{v} := e\ \textsc{e}); \textsc{s}, \textsc{l}\rangle\ \rangle \\
\quad \longrightarrow\ \textbf{if}\ v\ \mathit{in}\ \textsc{l}\ \textbf{then}\ \langle o : Ob \,|\, Att : \textsc{a}, Pr : \langle (\textsc{v} := \textsc{e}); \textsc{s}, (\textsc{l} + [v \mapsto \mathit{eval}(e, (\textsc{a}; \textsc{l}))])\rangle\ \rangle \\
\qquad\qquad\qquad\quad \textbf{else}\ \ \langle o : Ob \,|\, Att : (\textsc{a} + [v \mapsto \mathit{eval}(e, (\textsc{a}; \textsc{l}))]), Pr : \langle (\textsc{v} := \textsc{e}); \textsc{s}, \textsc{l}\rangle\ \rangle \\
\qquad\ \ \textbf{fi}
\end{array}
$$

$$
\begin{array}{l}
\langle o : Ob \,|\, Att : \textsc{a}, Pr : \langle (v := \mathit{new}\ C(\textsc{e}); \textsc{s}), \textsc{l}\rangle\ \rangle \\
\langle C : Cl \,|\, Param : \textsc{v}, Att : \textsc{a}',\ Tok : n\rangle \\
\quad \longrightarrow\ \langle o : Ob \,|\, Att : \textsc{a}, Pr : \langle (v := (C; n); \textsc{s}), \textsc{l}\rangle\ \rangle \\
\langle C : Cl \,|\, Param : \textsc{v}, Att : \textsc{a}'\ , Tok : \mathit{next}(n)\rangle \\
\langle (C; n) : Ob \,|\, Cl\!: C, Att : (\textsc{v}; \textsc{a}'), Pr : \langle (\textsc{v} := \mathit{eval}(\textsc{e}, (\textsc{a}; \textsc{l})); \mathit{run}), \varepsilon\rangle, \\
\qquad\qquad\quad PrQ\!: \varepsilon, EvQ : \varepsilon, Lab : 1\rangle
\end{array}
$$

$$
\begin{array}{l}
\langle o : Ob \,|\, Att : \textsc{a}, Pr : \langle \mathit{await}\ g; \textsc{s}, \textsc{l}\rangle, EvQ : \textsc{q}\rangle \\
\quad \longrightarrow\ \langle o : Ob \,|\, Att : \textsc{a}, Pr : \langle \textsc{s}, \textsc{l}\rangle, EvQ : \textsc{q}\rangle\ \textbf{if}\ \mathit{enabled}(g, (\textsc{a}, \textsc{l}), \textsc{q})
\end{array}
$$

$$
\begin{array}{l}
\langle o : Ob \,|\, Att : \textsc{a}, Pr : \langle \textsc{s}, \textsc{l}\rangle, PrQ : \textsc{w}, EvQ : \textsc{q}\rangle \\
\quad \longrightarrow\ \langle o : Ob \,|\, Att : \textsc{a}, Pr : \langle \varepsilon, \varepsilon\rangle, PrQ : (\textsc{w}\ \langle \mathit{clear}(\textsc{s}), \textsc{l}\rangle), EvQ : \textsc{q}\rangle \\
\textbf{if}\ \mathit{not}\ \mathit{enabled}(\textsc{s}, (\textsc{a}; \textsc{l}), \textsc{q})
\end{array}
$$

$$
\begin{array}{l}
\langle o : Ob \,|\, Att : \textsc{a}, Pr : \langle \varepsilon, \textsc{l}'\rangle, PrQ : \langle \textsc{s}, \textsc{l}\rangle\ \textsc{w}, EvQ : \textsc{q}\rangle \\
\quad \longrightarrow\ \langle o : Ob \,|\, Att : \textsc{a}, Pr : \langle \textsc{s}, \textsc{l}\rangle, PrQ : \textsc{w}, EvQ : \textsc{q}\rangle\ \textbf{if}\ \mathit{enabled}(\textsc{s}, (\textsc{a}, \textsc{l}), \textsc{q})
\end{array}
$$

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

constructed and the *run* method is invoked. The rules 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. The *enabledness* function is defined by induction over the construction of guards:

$$
\begin{array}{l}
\mathit{enabled}(t?, \textsc{d}, \textsc{q}) = \mathit{eval}(t, \textsc{d})\ \mathit{in}\ \textsc{q} \\
\mathit{enabled}(b, \textsc{d}, \textsc{q}) = \mathit{eval}(b, \textsc{d}) \\
\mathit{enabled}(\mathit{wait}, \textsc{d}, \textsc{q}) = \mathit{false} \\
\mathit{enabled}(g \vee g', \textsc{d}, \textsc{q}) = \mathit{enabled}(g, \textsc{d}, \textsc{q}) \vee \mathit{enabled}(g', \textsc{d}, \textsc{q}) \\
\mathit{enabled}(g \wedge g', \textsc{d}, \textsc{q}) = \mathit{enabled}(g, \textsc{d}, \textsc{q}) \wedge \mathit{enabled}(g', \textsc{d}, \textsc{q})
\end{array}
$$

When a non-enabled guard is encountered, 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:

$$
\begin{array}{l}
\mathit{enabled}(s; \textsc{s}, \textsc{d}, \textsc{q}) = \mathit{enabled}(s, \textsc{d}, \textsc{q}) \\
\mathit{enabled}(\mathit{await}\ g, \textsc{d}, \textsc{q}) = \mathit{enabled}(g, \textsc{d}, \textsc{q}) \\
\mathit{enabled}(s, \textsc{d}, \textsc{q}) = \mathit{true} \quad [\textbf{otherwise}]
\end{array}
$$

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

A synchronous call $r(Sig, Co, \textsc{e}; \textsc{v})$, where V is a list of variables and r is of one of the forms m or $x.m$, is translated into an *asynchronous call*,

$$\begin{array}{l}\langle o\!:\!Ob \mid Pr : \langle (r(Sig, Co, In; \textsc{v}); \textsc{s}), \textsc{l}\rangle, Lab : n\rangle \\ \quad\longrightarrow\ \langle o\!:\!Ob \mid Pr : \langle (!r(Sig, Co, In); n?(\textsc{v}); \textsc{s}), \textsc{l}\rangle, Lab : n\rangle\end{array}$$

$$\begin{array}{l}\langle o : Ob \mid Att : \textsc{a}, Pr : \langle (t!r(Sig, Co, \textsc{e}); \textsc{s}), \textsc{l}\rangle, Lab : n\rangle \\ \quad\longrightarrow\ \langle o : Ob \mid Att : \textsc{a}, Pr : \langle (t := n; !r(Sig, Co, \textsc{e}); \textsc{s}), \textsc{l}\rangle, Lab : n\rangle\end{array}$$

$$\begin{array}{l}\langle o : Ob \mid Att : \textsc{a}, Pr : \langle (!x.m(Sig, Co, \textsc{e}); \textsc{s}), \textsc{l}\rangle, Lab : n\rangle \\ \quad\longrightarrow\ \langle o : Ob \mid Att : \textsc{a}, Pr : \langle \textsc{s}, \textsc{l}\rangle, Lab : next(n)\rangle \\ invoc(m, Sig, Co, (o\ n\ eval(\textsc{e}, (\textsc{a}; \textsc{l})))) \textbf{ to } eval(x, (\textsc{a}; \textsc{l}))\end{array}$$

$$(msg \textbf{ to } o)\ \langle o : Ob \mid EvQ : \textsc{q}\rangle \ \longrightarrow\ \langle o : Ob \mid EvQ : \textsc{q}\ msg\rangle$$

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

$$\begin{array}{l}\langle o : Ob \mid Pr : \langle (t\,?\,(\textsc{v}); \textsc{s}), \textsc{l}\rangle, EvQ : \textsc{q}\ comp(o, n, \textsc{e})\rangle \\ \quad\longrightarrow\ \langle o : Ob \mid Pr : \langle (\textsc{v} := \textsc{e}; \textsc{s}), \textsc{l}\rangle, EvQ : \textsc{q}\rangle \textbf{ if } n = eval(t, \textsc{l})\end{array}$$

$$\begin{array}{l}\langle o : Ob \mid Pr : \langle (t\,?\,(\textsc{v}); \textsc{s}), \textsc{l}\rangle, PrQ : (\textsc{s}', \textsc{l}')\ \textsc{w}\rangle \\ \quad\longrightarrow\ \langle o : Ob \mid Pr : \langle \textsc{s}'; cont(eval(t, \textsc{l})), \textsc{l}'\rangle, PrQ : \langle await\ t\,?\,(\textsc{v}); \textsc{s}, \textsc{l}\rangle\ \textsc{w}\rangle \\ \textbf{if } eval(caller, \textsc{l}') = o \wedge eval(label, \textsc{l}') = eval(t, \textsc{l})\end{array}$$

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

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

$!r(Sig, Co, \textsc{e})$, followed by a blocking *reply statement*, $n?(\textsc{v})$, where n is the label value uniquely identifying the call. Likewise, a labeled asynchronous call is translated into a label assignment and an unlabeled asynchronous call. Internal calls are treated as external calls to *self*, and guarded calls are expanded by means of asynchronous calls and guarded replies, as defined in Section 3.1.

When an object calls a method, a message is placed in the configuration and delivered to the callee by a transport rule. Message overtaking is captured by the nondeterminism inherent in RL: invocation and completion messages *msg* sent by an object to another object in one order may arrive in any order. The call is bound in the following rule by sending a message to the class of the object. Note that for external calls, this rule identifies the class of the callee; consequently external method calls are virtually bound. The *bind* message is handled by the rule below, which identifies the method m in the method multiset $\textsc{m}$ of the class, returns the code associated with the method name from the object's class, and instantiates the method's in-parameters with the call's actual parameters $\textsc{e}$.

$$\begin{array}{l}(bind(m, Sig, Co, \textsc{e}, o) \textbf{ to } C)\ \langle C\!:\!Cl \mid Mtds : \textsc{m}\rangle \\ \quad\longrightarrow\ (bound(get(m, \textsc{m}, \textsc{e})) \textbf{ to } o)\ \langle C\!:\!Cl \mid Mtds : \textsc{m}\rangle \textbf{ if } match(m, Sig, Co, \textsc{m})\end{array}$$

The auxiliary predicate $\mathit{match}(m, \mathit{Sig}, \mathit{Co}, \textsc{m})$ evaluates to true if m is declared in $\textsc{m}$ with a signature Sig' and cointerface Co' such that $\mathit{Sig} \prec \mathit{Sig}'$, $\mathit{Co} \prec \mathit{Co}'$, and the function *get* returns a process with the method's code and local state from the method multiset $\textsc{m}$ of the class, and ensures that a completion message will be emitted upon method termination. Values of the actual in-parameters, the caller, and the label value n are stored locally. The process w resulting from the binding is loaded into the internal process queue:

$$(\mathit{bound}(w)\ \textbf{to}\ o)\ \langle o : \mathit{Ob} \mid \mathit{PrQ} : \textsc{w}\rangle \longrightarrow \langle o : \mathit{Ob} \mid \mathit{PrQ} : \textsc{w}\ w\rangle$$

The rules for the reply sentences fetch the return values corresponding to $\textsc{v}$ from the object's queue. In the model, EvQ is a multiset; thus the rule will match any occurrence of $\mathit{comp}(n, \textsc{e})$ in the queue. The use of rewrite rules rather than equations mimics distributed and concurrent processing of method lookup. Note the special construct $\mathit{cont}(n)$ in the two last rules, used to control local calls in order to avoid deadlock in the case of self reentrance [43].

3.6 Subject Reduction

A subject reduction property for the language is given in this section. We begin with some properties of program execution needed for subject reduction.

Lemma 6 *Given an arbitrary Creol program P. If $\Gamma \vdash P$ then the execution of every sentence $x := \textbf{new}\ C(\textsc{e})$ in P will create a new object of class C with type correct attribute instantiations without causing runtime type errors.*

Proof. From Lemma 3, we know that $\Gamma_{\mathrm{V}}(x)$ is of some type J. The typing rule for object creation ensures that there is an interface

$$I \in \Gamma_{\mathcal{C}}(C).\mathit{Impl}; \Gamma_{\mathcal{C}}(C).\mathit{Contract}$$

such that $I \prec J$, so J must be an interface. Furthermore, $\Gamma_{\mathrm{V}} \vdash_{\mathcal{F}} \textsc{e} : T$ such that $T \prec \mathit{type}(\Gamma_{\mathcal{C}}(C).\mathit{Param})$. We must show that the object creation rule succeeds in creating a new runtime object of class C. Let $\textsc{e}'$ denote the values for $\textsc{e}$ as evaluated in the object executing the creation sentence, consequently $\Gamma \vdash_{\mathcal{F}} \textsc{e}' : T'$ such that $T' \prec T$. With a runtime class representation $\langle C : \mathit{Cl} \mid \mathit{Param} : \textsc{v}, \mathit{Att} : \textsc{a}', \mathit{Init} : (\textsc{s}', \textsc{l}'), \mathit{Tok} : n\rangle$, the new object becomes

$$\langle (C; n) : \mathit{Ob} | \mathit{Cl} : C, \mathit{Att} : (\textsc{v}; \textsc{a}'), \mathit{Pr} : \langle (\textsc{v} := \textsc{e}'; \mathit{run}), \varepsilon\rangle, \mathit{PrQ} : \varepsilon, \mathit{EvQ} : \varepsilon, \mathit{Lab} : 1\rangle.$$

As $T' \prec T \prec \mathit{type}(\Gamma_{\mathcal{C}}(C).\mathit{Param})$, the assignment $\textsc{v} := \textsc{e}'$ is type correct. ■

It follows from Lemma 6 that any program variable typed by an interface will, if not null, point to an object of a class which implements this interface.

Lemma 7 *Let P be an arbitrary Creol program. If $\Gamma \vdash P$, then every method invocation $!x.m(\textsc{e})$ or $!m(\textsc{e})$ with actual parameter type $T \to T'$ in P can be type-correctly bound at runtime to a method such that the return values are of type T'' and $T'' \prec T'$, provided that x is not a null pointer.*

Proof. By Lemma 5, the signature *Sig* and cointerface *Co* of every invocation can be deterministically given by the type system. The proof is then carried out by looking at the evaluation rules for $!x.m(\mathit{Sig}, \mathit{Co}, \textsc{e})$ and $!m(\mathit{Sig}, \mathit{Co}, \textsc{e})$. Ignoring the evaluation of local variables, the evaluation rule for an external method call $!x.m(\mathit{Sig}, \mathit{Co}, \textsc{e})$ creates an invocation message as follows:

$\langle o : \mathit{Ob} \mid \mathit{Pr} : \langle(!x.m(\mathit{Sig}, \mathit{Co}, \textsc{e}); \textsc{s})\rangle, \mathit{Lab} : n\rangle$

$\longrightarrow\ \langle o : \mathit{Ob} \mid \mathit{Pr} : \langle \textsc{s}\rangle, \mathit{Lab} : \mathit{next}(n)\rangle\ \mathit{invoc}(m, \mathit{Sig}, \mathit{Co}, (o\ n\ \textsc{e}))\ \mathbf{to}\ x$

Recall that *Sig* here is the signature identified by the type analysis. The message will eventually arrive at the callee x, where it causes the generation of the message $\mathit{bind}(m, \mathit{Sig}, \mathit{Co}, \textsc{e}, x)$ **to** C. Here, C is identified as the class of x at runtime. It follows from Lemma 6 that if $\Gamma_{\mathrm{V}}(x) = I$ for some interface I, then the class 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 and cointerface Co_C for m in C such that

$$\mathit{Sig} \prec \mathit{Sig}_I \prec \mathit{Sig}_C \text{ and } \mathit{Co} \prec \mathit{Co}_I \prec \mathit{Co}_C$$

The rule for the bind message gives us:

$(\mathit{bind}(m, \mathit{Sig}, \mathit{Co}, \textsc{e}, x)\ \mathbf{to}\ C)\ \langle C : \mathit{Cl} \mid \mathit{Mtds} : \textsc{m}\rangle$

$\longrightarrow\ (\mathit{bound}(\mathit{get}(m, \textsc{m}, \textsc{e}))\ \mathbf{to}\ x)\ \langle C : \mathit{Cl} \mid \mathit{Mtds} : \textsc{m}\rangle\ \mathbf{if}\ \mathit{match}(m, \mathit{Sig}, \mathit{Co}, \textsc{m})$

It then follows from the transitivity of the subtype relation that the runtime *match* function will succeed. For an internal invocation $!m(\mathit{Sig}, \mathit{Void}, \textsc{e})$, the type analysis tells us directly that $\mathit{Sig} \prec \mathit{Sig}_C$ and $\mathit{Void} \prec \mathit{Co}_C$ and the runtime *match* function will succeed. ■

The above lemma also applies to synchronous invocations, because these are expanded to asynchronous invocations at runtime.

Lemma 8 *Given an arbitrary Creol program P. If $\Gamma \vdash P$ then no reply sentence $t?(\textsc{v})$ in P will cause runtime type errors.*

Proof. From the type rule for $t?(\textsc{v})$, we may assume that there is at least one pending method invocation with label t. We need to show that the runtime method lookup will select a method body such that the return values are covered by the type $\Gamma_{\mathrm{V}}(\textsc{v}) = T'$. By assumption, the signature found by the type analysis is included in the runtime method invocation. For this signature, type analysis guarantees that the actual return values are covered by the type of $\textsc{v}$, as stated by Lemma 7. ■

Theorem 9 (Subject reduction) *Runtime type errors do not occur in well-typed programs.*

Proof. We consider a well-typed program P and an initial type correct message **new** $C[\textsc{e}]$. By Lemma 6, this message will succeed in creating an initial runtime object and correctly instantiate object attributes. Because class names are unique in P and **new** creates a unique identity with respect to a given class, all object identifiers will be unique in the runtime configuration, so every message has a unique destination. As there is no remote variable access through dot notation in the language, it is therefore sufficient to consider execution in an arbitrary object in the runtime configuration to verify the subject reduction property. We consider the reduction of an object $\langle o : Ob \mid Att : \textsc{a}, Pr : \langle s; \textsc{s}, \textsc{l}\rangle\rangle$ to $\langle o : Ob \mid Att : \textsc{a}, Pr : \langle s'; \textsc{s}, \textsc{l}\rangle\rangle$.

There are two possible runtime type errors. Type errors in the assignments to attributes or local variables and method invocations which cannot be correctly bound. Lemma 6 guarantees that the evaluation of object creation sentences does not cause runtime type errors. Lemma 7 guarantees that the evaluation of method invocations does not cause runtime type errors. Lemma 8 guarantees that the evaluation of reply sentences does not cause runtime type errors. We now consider the remaining program statements. There are three cases: assignment, guard sentences, and sequential composition. First, consider $(v\ \textsc{v} := e\ \textsc{e})$. Execution of $(v\ \textsc{v} := e\ \textsc{e})$ reduces the sentence to $\textsc{v} := \textsc{e}$. Since the program is well-typed, we can assume that $\Gamma_{\mathrm{V}}(v) = T_v$, $\Gamma \vdash_{\mathcal{F}} e : T$, $T \prec T_v$, and the functional expression e reduces to e' with type $\Gamma \vdash_{\mathcal{F}} e' : T'$ such that $T' \prec T$. By transitivity, $T' \prec T_v$. Second, consider **await** g. Execution reduces the sentence to ε. Since the program is well-typed, we can assume that $\Gamma \vdash_{\mathcal{F}} g : \mathsf{Bool}$ as required by the *enabled* predicate. Third, the execution of sequential composition $s; \textsc{s}$ results in $s'; \textsc{s}$, and by assumption the reduction of s to s' does not cause runtime type errors. It remains to consider $\varepsilon; \textsc{s}$, which trivially reduces to $\textsc{s}$. It is immediate that the message transport rule does not introduce runtime type errors. Finally, process suspension, process activation, and the continuation rule move processes between *PrQ* and *Pr* without affecting the types of the program variables. ■

4 Flexible High-Level Control Structures for Local Computation

Asynchronous method calls, as introduced in Section 3, add flexibility 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 occur. For this purpose, additional composition operators between program sentences are introduced: conditionals, while-loops,

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

Figure 10: The language extended with constructs for local high-level control.

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 may adapt itself to the distributed environment *without* yielding control to competing processes. For example, nondeterministic choice may be used to encode interrupts for process suspension such as timeout and race conditions between competing asynchronous calls.

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, except reply statements $t?(\mathrm{V})$ which may block. Let S_1 and S_2 denote statement lists. 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. Nondeterministic choice between statements, written $\mathrm{S}_1 \Box \mathrm{S}_2$, may compute S_1 once S_1 is enabled or S_2 once S_2 is enabled, 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 $\mathrm{S}_1 \Vert\!\vert \mathrm{S}_2$, evaluates the statements S_1 and S_2 in some interleaved and enabled order. Control flow without potential processor release uses **if** and **while** constructs. Figure 10 gives the extended language syntax.

4.2 Example

Open distributed systems may be highly unstable in the sense that remote objects may become unavailable and communication links may break down. In these situations, an object's services may be severely delayed or even disrupted. The presence of the blocking reply statement may lead to object deadlock in an unstable environment, whereas a reply guard may lead to deadlock in one process in the object. The use of reply guards improves the latter situation as the object remains active, but the waiting process may remain suspended if the guard never becomes enabled. Nondeterministic choice is useful for selection between alternative replies (race conditions), and may be used to mimic exception handling and timeouts. Merge may increase efficiency when communicating with several objects when many replies are requested, by reducing the time needed for passive waiting. The use of nondeterministic choice is here illustrated by an encoding of a timeout mechanism. Consider a local timing mechanism expressing a delay, such as the following operation:

op delay(**in** n:Nat) == **if** n > 0 **then await** *wait*; delay(n–1) **else skip fi**

The timing mechanism essentially corresponds to the time needed to evaluate a series of n unconditional processor suspension points. Remark that this does not provide very exact timing, as the ordering of suspended processes is not specified in the abstract semantics (Section 3.5). In an implementation a more precise (and low level) timing construct could be defined using an underlying clock, or by a deterministic ordering of suspended processes.

A timeout effect for an invocation of a (remote) method m may be obtained by nondeterministically combining the asynchronous call to m with a call to the local timing mechanism. In an object with an internal *delay* method this can be illustrated by two program examples, considering blocking and nonblocking reply statements for some time delay $n \geq 0$:

$t!o.m(\mathrm{E}); t'!\mathrm{delay}(n); \mathrm{S}; (t?(\mathrm{V}); \mathrm{S}_1 \quad \Box\, \textbf{await}\ t'?; \mathrm{S}_2)$ (1)
$t!o.m(\mathrm{E}); t'!\mathrm{delay}(n); \mathrm{S}; (\textbf{await}\ t?; \mathrm{S}_1 \Box\, \textbf{await}\ t'?; \mathrm{S}_2)$ (2)

In case the replies corresponding to the respective reply statements for label t do not arrive within the time taken to evaluate the delay, both programming examples (1) and (2) are able to continue with S_2. This is because the semantics of nondeterministic choice will only commit to an enabled branch which does not immediately block evaluation. Consequently, a timeout branch may take control over a branch starting with a reply statement or guard, provided the timeout occurs before the reply has been received. If neither branch of the nondeterministic choice is enabled, (2) may be suspended. If this happens both replies may arrive before the sentence is reevaluated, in which case either branch may be selected. The semantics does not give priority to a particular branch. However, priority may be given to a branch by extending a guard: If normal execution is preferred to the timeout behavior, this can be achieved by replacing the second branch by **await** $t'? \wedge \neg t?; \mathrm{S}_2$. This construction creates a *race condition* between method calls; methods in several different objects may be invoked and the process proceeds with the first available reply.

$$\text{(EXT-ASYNC-L)}\quad \frac{\begin{array}{c}\Gamma \vdash_{\mathcal{F}} e : I \qquad \Gamma \vdash_{\mathcal{F}} \mathrm{E} : T \qquad \Gamma_{\mathrm{V}}(t) = \mathsf{Label} \\ \mathrm{match}(\Gamma_{\mathcal{C}}(\Gamma_{\mathrm{V}}(\mathit{self})).\mathit{Contract}, T, \mathsf{Data}, \Gamma_{\mathcal{I}}(I).\mathit{Mtd}(m)))\end{array}}{\Gamma \vdash_{\mathrm{S}} t!e.m(\mathrm{E}) \left\langle \begin{array}{l}[t \mapsto_{AE} \{\langle I, m, T\rangle\}] + [t \mapsto_{AI} \emptyset] + \\ [\mathrm{inuse} \mapsto_{\mathcal{L}} \Gamma_{\mathcal{L}}(\mathrm{inuse}) \cup \{t\}]\end{array} \right\rangle}$$

$$\text{(INT-ASYNC-L)}\quad \frac{\begin{array}{c}\Gamma_{\mathrm{V}}(t) = \mathsf{Label} \qquad \Gamma \vdash_{\mathcal{F}} \mathrm{E} : T \qquad \Gamma_{\mathrm{V}}(\mathit{self}) = C \\ \mathrm{match}(\mathit{Void}, T, \mathsf{Data}, \Gamma_{\mathcal{C}}(\Gamma_{\mathrm{V}}(\mathit{self})).\mathit{Mtd}(m))\end{array}}{\Gamma \vdash_{\mathrm{S}} t!m(\mathrm{E}) \left\langle \begin{array}{l}[t \mapsto_{AI} \{\langle C, m, T\rangle\}] + [t \mapsto_{AE} \emptyset] + \\ [\mathrm{inuse} \mapsto_{\mathcal{L}} \Gamma_{\mathcal{L}}(\mathrm{inuse}) \cup \{t\}]\end{array} \right\rangle}$$

$$\text{(MERGE)}\quad \frac{\begin{array}{cc}\mathcal{R}_{AI}^{(\Gamma,\Delta)} \cap \mathcal{R}_{AI}^{(\Gamma,\Delta')} = \emptyset & \Gamma + \emptyset_{\mathcal{L}} \vdash_{\mathrm{S}} \mathrm{s} \langle \Delta \rangle \\ \mathcal{R}_{AE}^{(\Gamma,\Delta)} \cap \mathcal{R}_{AE}^{(\Gamma,\Delta')} = \emptyset & \Gamma + \emptyset_{\mathcal{L}} \vdash_{\mathrm{S}} \mathrm{s}' \langle \Delta' \rangle \\ \multicolumn{2}{c}{\Gamma_{\mathcal{L}}(\mathrm{inuse}) \cap \Delta_{\mathcal{L}}(\mathrm{inuse}) \cap \Delta'_{\mathcal{L}}(\mathrm{inuse}) = \emptyset}\end{array}}{\Gamma \vdash_{\mathrm{S}} \mathrm{s} \| \mathrm{s}' \langle \Delta + \Delta' \rangle}$$

$$\text{(NON-DET)}\quad \frac{\begin{array}{ll}\forall\, t \in \mathcal{R}_{AI}^{(\Gamma,\Delta)} \cap \mathcal{R}_{AI}^{(\Gamma,\Delta')} \cdot \mathit{subType}(t, \Gamma, \mathrm{s}, \mathrm{s}') & \\ \forall\, t \in \mathcal{R}_{AE}^{(\Gamma,\Delta)} \cap \mathcal{R}_{AE}^{(\Gamma,\Delta')} \cdot \mathit{subType}(t, \Gamma, \mathrm{s}, \mathrm{s}') & \\ \forall\, t \cdot\ (\Gamma + \Delta)_{AI}(t) \notin \{\emptyset, \bot\} \vee (\Gamma + \Delta)_{AE}(t) \notin \{\emptyset, \bot\} & \Gamma \vdash_{\mathrm{S}} \mathrm{s} \langle \Delta \rangle \\ \quad \Leftrightarrow (\Gamma + \Delta')_{AI}(t) \notin \{\emptyset, \bot\} \vee (\Gamma + \Delta')_{AE}(t) \notin \{\emptyset, \bot\} & \Gamma \vdash_{\mathrm{S}} \mathrm{s}' \langle \Delta' \rangle\end{array}}{\Gamma \vdash_{\mathrm{S}} \mathrm{s} \,\Box\, \mathrm{s}' \langle (\Gamma + \Delta) \cup (\Gamma + \Delta') \rangle}$$

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

$$\text{(WHILE)}\quad \frac{\Gamma \vdash_{\mathcal{F}} \phi : \mathsf{Bool} \quad \Gamma + \emptyset_{AI} + \emptyset_{AE} \vdash_{\mathrm{S}} \mathrm{s}}{\Gamma \vdash_{\mathrm{S}} \textbf{while } \phi \textbf{ do } \mathrm{s} \textbf{ od}}$$

Figure 11: Typing of local high-level control structures.

4.3 Typing

The type system introduced in Section 3 is now extended to account for merge, conditional, choice, and while statements. It is nondeterministic for choice sentences which branch will be executed at runtime and for merge statements, the order in which all branches are executed. Consequently, the branches must be type checked in the same typing environment. The typing environment resulting from a nondeterministic statement depends on the calls introduced and removed in each branch. In order to control how asynchronous method calls in different branches may interfere, the labels used so far in the type checking of a statement list are recorded in a mapping $\mathcal{L}$ in Γ such that $\Gamma_{\mathcal{L}}(\mathrm{inuse})$ is the set of labels currently in use. Each asynchronous invocation will update this set. The rules for asynchronous invocations are modified accordingly. The new rules are given in Figure 11. In addition, a function is introduced to keep track of pending invocations that have been removed in a branch.

Definition 9 Given a judgment $\Gamma \vdash_{\mathrm{S}} \mathrm{s} \langle \Delta \rangle$. Let $\mathcal{R}^{(\Gamma,\Delta)} \equiv \Gamma \setminus (\Gamma + \Delta)$ denote the pending calls removed from Γ by the analysis of s.

Consequently for a judgment $\Gamma \vdash_{\mathrm{S}} \mathrm{s} \langle \Delta \rangle$, $\mathcal{R}_{AI}^{(\Gamma,\Delta)}$ and $\mathcal{R}_{AE}^{(\Gamma,\Delta)}$ denote the removed pending internal calls and external calls, respectively.

In a merge statement $\mathrm{s}_1 ||| \mathrm{s}_2$, both statement lists will be evaluated. The typing rule requires that reply sentences in the two branches do not correspond to the same pending call. Hence, $\mathcal{R}_{AI}^{(\Gamma,\Delta)} \cap \mathcal{R}_{AI}^{(\Gamma,\Delta')} = \emptyset$ and $\mathcal{R}_{AE}^{(\Gamma,\Delta)} \cap \mathcal{R}_{AE}^{(\Gamma,\Delta')} = \emptyset$. Additionally, all asynchronous invocations introduced in a merge sentence must have unique labels, to avoid interference between calls in the two branches.

For a nondeterministic choice, at most one of the two branches will be evaluated. If there is a reply sentence in only one of the branches that corresponds to a pending call in the typing environment Γ then a reply sentence corresponding to the same call cannot be found in statements succeeding the nondeterministic choice statement, even though the branch with the reply sentence need not be evaluated at runtime. Moreover, if there is a reply sentence in each branch that corresponds to the same pending call in Γ, say $t?(\mathrm{v})$ and $t?(\mathrm{v}')$, the type system requires the types of the out-parameters to have the property $\Gamma_{\mathrm{V}}(\mathrm{v}) \prec \Gamma_{\mathrm{V}}(\mathrm{v}')$ or $\Gamma_{\mathrm{V}}(\mathrm{v}') \prec \Gamma_{\mathrm{V}}(\mathrm{v})$ in order to ensure a deterministic signature of every invocation. This property is ensured by the function *subType*, which compares the types of actual out-parameters of reply sentences when they correspond to the same pending call in Γ. Note that if a label is reused in a new asynchronous invocation the reply sentence in this branch will refer to the new invocation, so the function will consider Data as the type for the out-parameters in this branch. (The function is by induction over the two branches and is not given here.) Furthermore, given the two branches in a nondeterministic choice between s and s', the type system requires that if a call with label t is introduced in s then a call with label t must also be introduced in s'. This ensures that each reply sentence corresponds to a call independent of the branch being evaluated.

The while-rule requires 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 5 holds for the extended language.

Lemma 10 *Let Γ be a mapping family such that $\Gamma_{AI} = \Gamma_{AE} = \emptyset$. For any sentence 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 contains exactly the labeled method invocations that have not been given a type correct signatureby the type analysis of s with respect to possible corresponding reply sentences.*

Proof. The proof is by induction over the length of s, similar to the proof of Lemma 3. For the old cases, the uniqueness of signatures in a single branch of a sentence list follows from Lemma 5. The new cases are now considered for the induction step. We assume $\Gamma \vdash_{\mathrm{S}} \mathrm{s} \langle \Delta \rangle$, where Δ contains the labeled method invocations that may need further analysis due to possible corresponding reply sentences. We prove that for judgment $\Gamma \vdash_{\mathrm{S}} \mathrm{s}; s \langle \Delta + \Delta' \rangle$, where s is a nondeterministic choice, merge, conditional, or while statement, $\Delta + \Delta'$ contains the labeled method invocations that may now need further analysis. For the while-sentence, the type rule does not allow any updates on the typing environment, so $\Delta + \Delta' = \Delta$. We now consider nondeterministic choice $\mathrm{s}_1 \Box \mathrm{s}_2$ and assume that $\Gamma + \Delta \vdash_{\mathrm{S}} \mathrm{s}_1 \langle \Delta_1 \rangle$, and $\Gamma + \Delta \vdash_{\mathrm{S}} \mathrm{s}_2 \langle \Delta_2 \rangle$. We first consider the case where s_1 contains a reply sentence $t?(\mathrm{v}_1)$ with a label t corresponding to a call contained in Δ. There are two possibilities. First, the reply sentence with label t does not occur in s_2. The update of the typing environment removes the pending calls; e.g., $(\Delta_1 + \Delta_2)_{AI}(t) = (\Delta_1 + \Delta_2)_{AE}(t) = \emptyset$. Signature uniqueness is here immediate. Second, the reply sentence $t?(\mathrm{v}_2)$ occurs in s_2. We need to check that the reply sentences will yield a unique signature. By the induction hypothesis, we may assume that there is a unique signature candidate in each branch. The *subType* function evaluates to true if a compatible minimal type can be given for the two reply sentences, which gives us a unique signature. Note that if the reply sentence in one branch refers to a new method invocation with the same label t in that branch, the *subType* function will use Data as the out-parameter type from the branch. We next consider the case where s_1 introduces a new invocation with label t. The condition ensures that there will also be a pending call to t if branch s_2 is chosen, this pending call 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 $(\Delta + \Delta_1) \cup (\Delta + \Delta_2)$ and may therefore be given a more precise signature if corresponding reply sentences are encountered. The conditional sentence follows from the nondeterministic choice sentence.

Finally, we consider nondeterministic merge $\mathrm{s}_1 ||| \mathrm{s}_2$ and assume that $\Gamma + \Delta + \emptyset_{\mathcal{L}} \vdash_{\mathrm{S}} \mathrm{s}_1 \langle \Delta_1 \rangle$, and $\Gamma + \Delta + \emptyset_{\mathcal{L}} \vdash_{\mathrm{S}} \mathrm{s}_2 \langle \Delta_2 \rangle$ where Δ_1 and Δ_2 contain exactly the labeled method invocations that have not been given a precise signature by the type analysis of s_1 and s_2 with respect to possible corresponding reply sentences in each branch. The condition $(\Gamma + \Delta)_{\mathcal{L}}(\text{inuse}) \cap \Delta_{1\mathcal{L}}(\text{inuse}) \cap \Delta_{2\mathcal{L}}(\text{inuse}) = \emptyset$ ensures that there is no interference between the labels of method invocations in s and the two branches s_1 and s_2 of the merge sentence. Similarly, the conditions $\mathcal{R}_{AI}^{(\Gamma+\Delta,\Delta_1)} \cap \mathcal{R}_{AI}^{(\Gamma+\Delta,\Delta_2)} = \emptyset$ and $\mathcal{R}_{AE}^{(\Gamma+\Delta,\Delta_1)} \cap \mathcal{R}_{AE}^{(\Gamma+\Delta,\Delta_2)} = \emptyset$ ensure that there is no interference between reply sentences in the two branches. Consequently, Δ, 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

$$\begin{array}{l}\langle o : Ob \mid Att : \textsc{a}, Pr : \langle (\textsc{s}_1 \Box \textsc{s}_2); \textsc{s}_3, \textsc{l}\rangle, EvQ : \textsc{q}\rangle \\ \quad \longrightarrow \langle o : Ob \mid Att : \textsc{a}, Pr : \langle \textsc{s}_1; \textsc{s}_3, \textsc{l}\rangle, EvQ : \textsc{q}\rangle \ \textbf{if}\ \mathit{ready}(\textsc{s}_1, (\textsc{a}; \textsc{l}), \textsc{q})\end{array}$$

$$\begin{array}{l}\langle o : Ob \mid Att : \textsc{a}, Pr : \langle ((s; \textsc{s}_1) /\!/\!/ \textsc{s}_2); \textsc{s}_3, \textsc{l}\rangle, EvQ : \textsc{q}\rangle \\ \quad \longrightarrow \ \textbf{if}\ \mathit{enabled}(s, (\textsc{a}; \textsc{l}), \textsc{q})\ \textbf{then}\ \langle o : Ob \mid Att : \textsc{a}, Pr : \langle s; (\textsc{s}_1 /\!/\!/ \textsc{s}_2); \textsc{s}_3, \textsc{l}\rangle, EvQ : \textsc{q}\rangle \\ \qquad\quad \textbf{else}\ \langle o : Ob \mid Att : \textsc{a}, Pr : \langle ((s; \textsc{s}_1) \mathbin{|||} \textsc{s}_2); \textsc{s}_3, \textsc{l}\rangle, EvQ : \textsc{q}\rangle \ \textbf{fi}\end{array}$$

$$\begin{array}{l}\langle o : Ob \mid Att : \textsc{a}, Pr : \langle (\textsc{s}_1 \mathbin{|||} \textsc{s}_2); \textsc{s}_3, \textsc{l}\rangle, EvQ : \textsc{q}\rangle \\ \quad \longrightarrow \langle o : Ob \mid Att : \textsc{a}, Pr : \langle (\textsc{s}_1 /\!/\!/ \textsc{s}_2); \textsc{s}_3, \textsc{l}\rangle, EvQ : \textsc{q}\rangle \ \textbf{if}\ \mathit{ready}(\textsc{s}_1, (\textsc{a}; \textsc{l}), \textsc{q})\end{array}$$

$$\begin{array}{l}\langle o : Ob \mid Att : \textsc{a}, Pr : \langle (\mathit{if}\ b\ \mathit{then}\ \textsc{s}_1\ \mathit{else}\ \textsc{s}_2\ \mathit{fi}; \textsc{s}_3), \textsc{l}\rangle\rangle \\ \quad \longrightarrow \ \textbf{if}\ \mathit{eval}(b, (\textsc{l}; \textsc{a}))\ \textbf{then}\ \langle o : Ob \mid Att : \textsc{a}, Pr : \langle (\textsc{s}_1; \textsc{s}_3), \textsc{l}\rangle\ \rangle \\ \qquad\quad \textbf{else}\ \langle o : Ob \mid Att : \textsc{a}, Pr : \langle (\textsc{s}_2; \textsc{s}_3), \textsc{l}\rangle\ \rangle\ \textbf{fi}\end{array}$$

$$\begin{array}{l}\langle o : Ob \mid Att : \textsc{a}, Pr : \langle (\mathit{while}\ b\ \mathit{do}\ \textsc{s}_1\ \mathit{od}; \textsc{s}_2), \textsc{l}\rangle\rangle \\ \quad \longrightarrow \langle o : Ob \mid Att : \textsc{a}, Pr : \langle (\mathit{if}\ b\ \mathit{then}\ (\textsc{s}_1; \mathit{while}\ b\ \mathit{do}\ \textsc{s}_1\ \mathit{od})\ \mathit{else}\ \mathit{skip}\ \mathit{fi}); \textsc{s}_2, \textsc{l}\rangle\ \rangle\end{array}$$

Figure 12: An operational semantics for local high-level control structures.

$\Delta + \Delta_1 + \Delta_2$ contains exactly the labeled method invocations that have not been given a precise signature by the type analysis of $\textsc{s}; \textsc{s}_1 \mathbin{|||} \textsc{s}_2$. ■

4.4 Operational Semantics

The operational semantics of Section 3.5 is now extended with local control structures. Selection of a branch $\textsc{s}_1$ in a nondeterministic choice statement $\textsc{s}_1 \Box \textsc{s}_2$ is modeled by the first rule in Figure 12. Combined with the associativity and commutativity of the $\Box$ operator, this rule covers the selection of any branch in a compound nondeterministic choice sentence. 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

$$\begin{array}{l}\mathit{ready}(s; \textsc{s}, \textsc{d}, \textsc{q}) = \mathit{ready}(s, \textsc{d}, \textsc{q}) \\ \mathit{ready}(t?(\textsc{v}), \textsc{d}, \textsc{q}) = \mathit{eval}(t, \textsc{d})\ \mathit{in}\ \textsc{q} \\ \mathit{ready}(s, \textsc{d}, \textsc{q}) = \mathit{enabled}(s, \textsc{d}, \textsc{q}) \qquad [\,\textbf{otherwise}\,]\end{array}$$

As long as neither $\textsc{s}_1$ nor $\textsc{s}_2$ 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 $\mathbin{|||}$ interleaves the execution of two statement lists $\textsc{s}_1$ and $\textsc{s}_2$. A naive approach is to define merge in terms of the nondeterministic choice $\textsc{s}_1; \textsc{s}_2 \Box \textsc{s}_2; \textsc{s}_1$. To improve efficiency, a more fine-grained interleaving is preferred. However, in order to comply with the suspension technique of the language, interleaving will only be allowed at processor release points in the branches. An associative but not commutative auxiliary operator $/\!/\!/$ is introduced, with the second rule of Figure 12. This rule has the following property: Whenever evaluation of the selected (left) branch leads to nonenabledness, execution has arrived at a suspension point and it is safe to pass control back to the $\|\!|$ operator. The rule for merge decides whether to block, select the other branch, or suspend. The $\|\!|$ operator is associative, commutative, and has identity element ε (i.e., $\varepsilon \|\!| \mathrm{S} = \mathrm{S}$). The operational semantics for conditionals and while-loops are as expected. Finally, the enabledness predicate is extended to nondeterministic choice and merge as follows:

$$\begin{aligned} &\mathit{enabled}(\mathrm{S} \,\Box\, \mathrm{S}', \mathrm{D}, \mathrm{Q}) = \mathit{enabled}(\mathrm{S}, \mathrm{D}, \mathrm{Q}) \vee \mathit{enabled}(\mathrm{S}', \mathrm{D}, \mathrm{Q}) \\ &\mathit{enabled}(\mathrm{S} \|\!| \mathrm{S}', \mathrm{D}, \mathrm{Q}) = \mathit{enabled}(\mathrm{S}, \mathrm{D}, \mathrm{Q}) \vee \mathit{enabled}(\mathrm{S}', \mathrm{D}, \mathrm{Q}) \end{aligned}$$

4.5 Subject Reduction

A subject reduction property for the extended language is given in this section. It follows from Lemma 10 that Lemmas 7 and 8 hold for the extended language:

Lemma 11 *Let P be an arbitrary Creol program. If $\Gamma \vdash P$, then every method invocation $!x.m(\mathrm{E})$ or $!m(\mathrm{E})$ with actual parameter type $T \rightarrow T'$ in P can be type-correctly bound at runtime to a method such that the return values are of type T'' and $T'' \prec T'$, provided that x is not a null pointer.*

Lemma 12 *Let P be an arbitrary Creol program. If $\Gamma \vdash P$ then no reply sentence $t?(\mathrm{V})$ in P will cause runtime type errors.*

The subject reduction property may now be proved for the extended language.

Theorem 13 (Subject reduction) *Runtime type errors do not occur in well-typed programs.*

Proof. The proof extends the proof of Theorem 9. We consider the reduction of an 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. There are two possible runtime type errors. Type errors in the assignments to attributes or local variables, and method invocations which cannot be correctly bound to their corresponding declarations. It follows from Lemmas 11 and 12 that method invocations will be correctly bound and the reply values will be assigned to attributes or local variables without causing runtime type errors. Consequently, we assume as an induction hypothesis that the subject reduction property holds for subsentences S_1 and S_2 of s.

We first consider nondeterministic choice $\mathrm{S}_1 \Box \mathrm{S}_2$ which may reduce to either S_1 or S_2. Both S_1 and S_2 have been type checked independently and by the induction hypothesis the subject reduction property holds for both S_1 and S_2. We next consider the conditional **if** ϕ **then** S_1 **else** S_2 **fi**. By assumption $\Gamma \vdash_{\mathcal{F}} \phi : \mathsf{Bool}$, and S_1 and S_2 are well-typed. Also by assumption ϕ may be successfully reduced to a boolean value, and the conditional may reduce to either S_1 or S_2.

We then consider the while-sentence **while** E **do** S_1 **od**. The typing rule ensures that a traversal of S_1 does not modify the typing environment. Consequently, S_1; **while** E **do** S_1 **od** is well-typed. The typing rule for while ensures that there is no interference between the different traversals of S_1, so the conditions of the typing rule NON-DET are satisfied and the derived conditional sentence **if** E **then** S_1; **while** E **do** S_1 **od else** ε **fi** will also be well-typed. By assumption, E will be reduced to a boolean value and the reduction of the derived conditional sentence will succeed.

We finally consider the merge sentence $\mathrm{S}_1 \mathbin{|||} \mathrm{S}_2$. If $\mathrm{S}_1 = \varepsilon$, then $\mathrm{S}_1 \mathbin{|||} \mathrm{S}_2 = \mathrm{S}_2$, for which subject reduction holds by assumption. Now assume that S_1 is nonempty. The operational semantics reduces $\mathrm{S}_1 \mathbin{|||} \mathrm{S}_2$ to $\mathrm{S}_1 \mathbin{///} \mathrm{S}_2$, which in turn reduces S_1 to S_1', where S_1' is a tail sequence of S_1, and returns $\mathrm{S}_1' \mathbin{|||} \mathrm{S}_2$. By assumption the evaluation of the head sequence of S_1 will not cause runtime type errors and the subject reduction property holds for S_1'. Since the type system guarantees that S_1 does not interfere with S_2, $\mathrm{S}_1' \mathbin{|||} \mathrm{S}_2$ will be well-typed. Finally, process suspension and process activation move processes depending on the enabledness of choice and merge sentences without affecting the types of program variables. ■

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, 69]. 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 [52, 69], 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 which share the same internal structure; i.e., attributes and method definitions. Code inheritance provides a powerful mechanism for defining, specializing, and understanding the imperative structure of classes through code reuse and modification. Class extension and method redefinition are convenient both for development and understanding of code. Calling superclass methods in a subclass method enables *reuse in redefined methods*, making the relationship between the method versions explicit. A denotational semantics for code sharing and reuse based on single inheritance is given in [21].

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 [54, 58]. Reasoning considerations therefore suggest that all attributes and methods of a superclass are inherited, but method redefinition may violate the semantic requirements of an interface of the superclass.

5.1 Syntax

A mechanism for multiple inheritance 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 an *inheritance list*; i.e., a list of class names $C(\mathrm{E})$ where E provides the actual class parameters.

Let a class hierarchy be a directed acyclic graph of parameterized classes. Each class consists of lists of class parameters and inherited classes, a set of attributes, and method definitions. 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, methods defined in a superclass may be accessed from the subclass by qualified references. A name conflict is *vertical* if a name occurs in a class and in one of its ancestors. A name conflict is *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 matching definition with respect to the types of the actual parameters is chosen while ascending a branch of the inheritance tree. Horizontal name conflicts will be 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

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 from 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 [68]. In order to allow the reuse of attribute names, these will always be 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 now 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 $m@C(\text{E}; \text{V})$ 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 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 13.

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

Figure 13: A language extension for static and virtual internal calls.

5.2 Virtual Binding

When multiple inheritance is included in the language, it is necessary to reconsider the mechanism for virtual binding. Let a class C be *below* a class C' if C is C', or C is a direct or indirect subclass of C'. A method declaration in a class C is *constrained* by C' if C is required to be below to C'. The virtual binding of method calls is now explained. At runtime, a call to a method of an object o will always be 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 will identify 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 will be bound to the first matching method definition above C, in a left-first depth-first order. (An arbitrary order may be obtained by replacing the inheritance list by a multiset.) The three forms of method binding are illustrated in Figure 14.

5.3 Example: Combining Authorization Policies

In a database containing sensitive information and different authorization policies, the information returned for a request will depend on the clearance

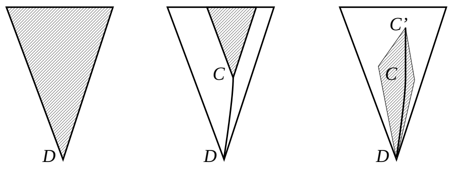

Figure 14: Binding calls to m, $m@C$, and $m < C'$ from class D.

level of the agent making the request. Let *Agent* denote the interface of arbitrary agents, and let *Auth* denote an authorization interface with methods *grant*(x), *revoke*(x), *auth*(x), and *delay* for agents x. The two classes *SAuth* and *MAuth*, both implementing *Auth*, implement 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 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 delay == await (gr = null)
  op grant == grant(caller)
  op revoke == revoke(caller)
  op auth == auth(caller)
end
```

```
class MAuth implements Auth
begin
  var gr: Set[Agent] = ∅
  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 delay == await (gr = ∅)
  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
```

As agents need not be entitled to high authorization, the *openH* method returns with a boolean reply.

Let a class *DB* provide the actual operations on the database. We assume given the internal operations *access*(**in** k:Key, high:Bool **out** y:Data), where *high* defines the access level, 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 will authorize low clearance, and *SAuth* will authorize 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
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) ==
     auth(caller);
      await access@DB(k,true; y)
  op closeH == revoke(caller)
end
```

```
class LAuth implements Low
  inherits MAuth, DB
begin
with Agent
  op openL == grant(caller)
  op access(in k:Key out y:Data) ==
      auth(caller);
      await access@DB(k,false; 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 *LAuth* and *HAuth* classes may not be used through the *Auth* interface. This would have been harmful for the authorization provided in the example. For instance, a call to *grant* to a *HAuth* object could then 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(k; y) else access@LAuth(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* will trigger calls to the *HAuth* methods, and from these methods the virtual internal calls to *grant*, *revoke* and *auth* will now bind to those of the *MAuth* class, if selected in a left-most depth-first traversal of the inheritance tree of the actual class, *HLAuth*. Note that if the inheritance ordering in *HLAuth* were reversed, similar problems occur with the binding of *Low* interaction.

This *pruned* virtual binding strategy ensures that the virtual internal calls constrained by classes *HAuth* and *LAuth* will be 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*.

5.4 Typing

In order to extend the type system to classes with inheritance, it suffices to 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 15 and 16. The reflexive and transitive closure of the subclass relation is denoted by $\prec_{\mathcal{C}}$. Thus, $C \prec_{\mathcal{C}} C'$ expresses that C is a direct or indirect subclass of C', or is the same class as C'.

Definition 10 Let m be a method, $\Gamma_{\mathcal{C}}$ a class mapping, C and C' be class names and $\textsc{c}$ and $\textsc{c}'$ class inheritance lists. Define

$$\begin{aligned}
&\mathit{matchcall}(m, \Gamma_{\mathcal{C}}, \varepsilon) = \varepsilon\\
&\mathit{matchcall}(m, \Gamma_{\mathcal{C}}, C(\textsc{e}); \textsc{c}) = \textbf{if } \exists m' \in \Gamma_{\mathcal{C}}(C).\mathit{Mtd} \cdot (m.\mathit{Name} = m'.\mathit{Name}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge \mathit{Sig}(m) \prec \mathit{Sig}(m') \wedge m.\mathit{Co} \prec m'.\mathit{Co})\\
&\qquad\qquad\qquad\qquad \textbf{then } C \textbf{ else } \varepsilon \textbf{ fi } ; \mathit{matchcall}(m, \Gamma_{\mathcal{C}}, C.\mathit{Inh}; \textsc{c})\\
&\mathit{bounded}(\varepsilon, C') = \varepsilon\\
&\mathit{bounded}((C; \textsc{c}), C') = \textbf{if } C \prec_{\mathcal{C}} C' \textbf{ then } C \textbf{ else } \varepsilon \textbf{ fi } ; \mathit{bounded}(\textsc{c}, C')
\end{aligned}$$

Initial expressions in variable declarations and program statements in method bodies of a given class C may refer to variables declared in its superclasses. Consequently, the typing environment Γ must be extended with inherited attributes before type checking class C. This extension is obtained by traversing the inheritance list *Inh* of C depth first and using the mapping $\Gamma_{\mathcal{C}}$ to gather *Var* and *Param* from each superclass. The function *InhAttr*

$$\text{(CLASS-INH)}\quad \frac{\begin{array}{c}\Gamma \vdash_{\mathrm{V}} \ \mathit{InhAttr}(\mathit{Inh}, \Gamma_{\mathcal{C}}), \mathit{Param} \ \langle\ \Delta\ \rangle \qquad \Gamma + \Delta \vdash_{\mathrm{V}} \ \mathit{Var} \ \langle\ \Delta'\ \rangle \\ \mathit{matchparam}_{\mathcal{C}}\ (\Gamma + \Delta, \mathit{Inh}) \qquad \forall m \in \mathit{Mtd}\ \cdot\ \Gamma + \Delta + \Delta' \vdash m \\ \forall I \in (\mathit{Impl}; \mathit{Contract}) \cdot \forall m' \in \Gamma_{\mathcal{I}}(I).\mathit{Mtd} \cdot \mathit{matchcall}(m', \Gamma_{\mathcal{C}}, \Gamma(\mathit{self})) \neq \varepsilon\end{array}}{\Gamma \vdash \mathit{class}\,(\mathit{Param}, \mathit{Impl}, \mathit{Contract}, \mathit{Inh}, \mathit{Var}, \mathit{Mtd})}$$

$$\text{(INT-SYNC)}\quad \frac{\begin{array}{c}\Gamma_{\mathrm{V}}(\mathit{self}) \prec_{\mathcal{C}}\ C \qquad \Gamma \vdash_{\mathcal{F}}\ \textsc{e} : T \\ \mathit{matchcall}(\mathit{method}(m, \mathit{Void}, T, \Gamma_{\mathrm{V}}(\textsc{v}), \varepsilon), \Gamma_{\mathcal{C}}, C) \neq \varepsilon\end{array}}{\Gamma \vdash_{\mathrm{S}}\ m@C(\textsc{e}; \textsc{v})}$$

$$\text{(INT-ASYNC)}\quad \frac{\begin{array}{c}\Gamma \vdash_{\mathcal{F}}\ \textsc{e} : T \qquad \Gamma_{\mathrm{V}}(\mathit{self}) \prec_{\mathcal{C}}\ C \\ \mathit{matchcall}(\mathit{method}(m, \mathit{Void}, T, \mathsf{Data}, \varepsilon), \Gamma_{\mathcal{C}}, C) \neq \varepsilon\end{array}}{\Gamma \vdash_{\mathrm{S}}\ !m@C(\textsc{e})}$$

$$\text{(INT-ASYNC-L)}\quad \frac{\begin{array}{c}\Gamma \vdash_{\mathcal{F}}\ \textsc{e} : T \qquad \Gamma_{\mathrm{V}}(t) = \mathsf{Label} \qquad \Gamma_{\mathrm{V}}(\mathit{self}) \prec_{\mathcal{C}}\ C \\ \mathit{matchcall}(\mathit{method}(m, \mathit{Void}, T, \mathsf{Data}, \varepsilon), \Gamma_{\mathcal{C}}, C) \neq \varepsilon\end{array}}{\Gamma \vdash_{\mathrm{S}}\ t!m@C(\textsc{e})\ \langle\ [t \mapsto (\Gamma_{AI}(t); \langle C, m, T\rangle)]\ \rangle}$$

Figure 15: Typing of multiple inheritance.

in the typing rule for classes returns a list of typed variables inherited from the classes above C. Furthermore, for each interface that C implements, C must provide, either by inheritance or by local declaration, at least one type correct method body for each method in the interface. The typing rules for internal invocations are similar to those of Section 4.3, but the method lookup function *matchcall* covers the inheritance tree above *self*.

Internal calls. The typing of external calls is controlled by interfaces, and is not affected by class inheritance. The typing of internal calls, however, must inspect the inheritance graph, using the *matchcall* function, rather than the simpler *match* function used in the case without inheritance. Similarly, the typing of replies uses *matchcall* in cases of internal calls. For static calls $m@C$, the match starts from the given class C, rather than from the actual class. For bounded calls $m < C$, the *bounded* function is used to check that there is a match below C in the inheritance tree above the actual class.

5.5 Operational Semantics

The operational semantics is now adapted to incorporate multiple inheritance. The classes of Creol are extended to include the names of inherited classes and are now represented by the RL objects

$$\langle \mathit{Cl} \mid \mathit{Param}, \mathit{Inh}, \mathit{Att}, \mathit{Mtds},\ \mathit{Tok}\rangle,$$

where *Cl* is the class name, *Param* a list of parameters, *Inh* is the inheritance list, *Att* a list of attributes, *Mtds* a multiset of methods, and *Tok* is an arbitrary term of sort $T_<$. When an object needs a method, it is bound to a

$$
(\text{BND-SYNC}) \quad \dfrac{\begin{array}{c}\Gamma_{\mathrm{V}}(\mathit{self}) \prec_{\mathcal{C}} C \qquad \Gamma \vdash_{\mathcal{F}} \mathrm{E} : T \\ \mathit{bounded}(\mathit{matchcall}(\mathit{method}(m, \mathit{Void}, T, \Gamma_{\mathrm{V}}(\mathrm{V}), \varepsilon), \Gamma_{\mathcal{C}}, \Gamma_{\mathrm{V}}(\mathit{self})), C) \neq \varepsilon\end{array}}{\Gamma \vdash_{\mathrm{S}} m < C(\mathrm{E}; \mathrm{V})}
$$

$$
(\text{BND-ASYNC}) \quad \dfrac{\begin{array}{c}\Gamma_{\mathrm{V}}(\mathit{self}) \prec_{\mathcal{C}} C \qquad \Gamma \vdash_{\mathcal{F}} \mathrm{E} : T \\ \mathit{bounded}(\mathit{matchcall}(\mathit{method}(m, \mathit{Void}, T, \mathsf{Data}, \varepsilon), \Gamma_{\mathcal{C}}, \Gamma_{\mathrm{V}}(\mathit{self})), C) \neq \varepsilon\end{array}}{\Gamma \vdash_{\mathrm{S}} !m < C(\mathrm{E})}
$$

$$
(\text{BND-ASYNC-L}) \quad \dfrac{\begin{array}{c}\Gamma \vdash_{\mathcal{F}} \mathrm{E} : T \qquad \mathit{C\text{-}list} \neq \varepsilon \\ \Gamma_{\mathrm{V}}(t) = \mathsf{Label} \qquad \Gamma_{\mathrm{V}}(\mathit{self}) \prec_{\mathcal{C}} C \\ \mathit{C\text{-}list} = \mathit{bounded}(\mathit{matchcall}(\mathit{method}(m, \mathit{Void}, T, \mathsf{Data}, \varepsilon), \Gamma_{\mathcal{C}}, \Gamma_{\mathrm{V}}(\mathit{self})), C)\end{array}}{\Gamma \vdash_{\mathrm{S}} t!m < C(\mathrm{E}) \ \langle\ [t \mapsto \Gamma_{AI}(t) \cup \{\langle \mathit{C\text{-}list}, m, T\rangle\}]\ \rangle}
$$

$$
(\text{REPLY}) \quad \dfrac{\begin{array}{l}\Gamma_{\mathrm{V}}(t) = \mathsf{Label} \qquad \Gamma_{AE}(t) \cup \Gamma_{AI}(t) \neq \emptyset \\ \forall \langle (\mathrm{C}; C; \mathrm{C}'), m, T\rangle \in \Gamma_{AI}(t) \cdot \\ \qquad \mathit{matchcall}(\mathit{method}(m, \mathit{Void}, T, \Gamma_{\mathrm{V}}(\mathrm{V}), \varepsilon), \Gamma_{\mathcal{C}}, \Gamma_{\mathrm{V}}(\mathit{self})) \neq \varepsilon \\ \forall \langle I, m, T\rangle \in \Gamma_{AE}(t) \cdot \\ \qquad \mathrm{match}(\Gamma_{\mathcal{C}}(\Gamma_{\mathrm{V}}(\mathit{self})).\mathit{Contract}, T, \Gamma_{\mathrm{V}}(\mathrm{V}), \Gamma_{\mathcal{I}}(I).\mathit{Mtd}(m))\end{array}}{\Gamma \vdash_{\mathrm{S}} t?(\mathrm{V}) \ \langle\ [t \mapsto_{AI} \emptyset] + [t \mapsto_{AE} \emptyset]\ \rangle}
$$

Figure 16: Typing of bounded method calls.

definition in the *Mtds* multiset of its class or of a superclass. Previous class definitions without inheritance can be extended with an empty inheritance list to be valid in the extended semantics. The additional rules are given in Figure 17.

5.5.1 Virtual and Static Binding of Method Calls

It is syntactically prohibited to make qualified external method invocations; external invocations are not affected by the inclusion of inheritance in the language. As before an internal call gives rise to an invocation message, but r may be of the form $m@C$ or $m < C$. The constraint C will be used in the virtual binding as described below.

In order to allow concurrent and dynamic execution, the inheritance graph will not be statically given. Rather, the binding mechanism dynamically inspects the current class hierarchy as present in the configuration. Our approach to virtual binding is to use a *bind* message which is sent from a class to its superclasses, resulting in a *bound* message returned to the object generating the *bind* message. This way, the inheritance graph is explored dynamically and as far as necessary when needed. When the external invocation of a method m is found in the message queue of an object o, a message $\mathit{bind}(m, \mathit{Sig}, \mathit{Co}, \mathrm{E}, o)$ **to** C can be sent after dynamically retrieving the class C of the object, as shown in Figure 9. For internal static calls $m@C$, the

$$
\begin{array}{l}
\langle o\!:\!Ob \mid Att : \textsc{a}, Pr : \langle\, !r(Sig, Co, \textsc{e}); \textsc{s}, \textsc{l}\rangle, Lab : n\rangle \\
\quad \longrightarrow \ \langle o\!:\!Ob \mid Att : \textsc{a}, Pr : \langle \textsc{s}, \textsc{l}\rangle, Lab : next(n)\rangle \\
\qquad invoc(p, Sig, Co, (n \ o \ eval(\textsc{e}, (\textsc{a}; \textsc{l})))) \ \textbf{to} \ o
\end{array}
$$

$$
\begin{array}{l}
\langle o\!:\!Ob \mid EvQ : invoc(m@C, Sig, Co, \textsc{e}) \ \textsc{q}\rangle \\
\quad \longrightarrow \langle o\!:\!Ob \mid EvQ : \textsc{q}\rangle \ (bind(m, Sig, Co, \textsc{e}, o) \ \textbf{to} \ C)
\end{array}
$$

$$
\begin{array}{l}
(bind(m, Sig, Co, \textsc{e}, o) \ \textbf{to} \ C \ \textsc{i}) \ \langle C\!:\!Cl \mid Inh : \textsc{i}', Mtds : \textsc{m}\rangle \\
\quad \longrightarrow \ \textbf{if} \ match(m, Sig, Co, \textsc{m}) \ \textbf{then} \ bound(get(m, \textsc{m}, \textsc{e})) \ \textbf{to} \ o \\
\qquad\qquad\qquad\qquad \textbf{else} \ bind(m, Sig, Co, \textsc{e}, o) \ \textbf{to} \ (\textsc{i}' \ \textsc{i}) \ \textbf{fi} \ \langle C\!:\!Cl \mid Inh : \textsc{i}', Mtds : \textsc{m}\rangle
\end{array}
$$

$$
\begin{array}{l}
(bind(m < C', Sig, Co, \textsc{e}, o) \ \textbf{to} \ C \ \textsc{i}) \ \langle C\!:\!Cl \mid Inh : \textsc{i}', Mtds : \textsc{m}\rangle \\
\quad \longrightarrow \ \textbf{if} \ match(m, Sig, Co, \textsc{m}) \\
\qquad\qquad\qquad \textbf{then} \ (find(C', C) \ \textbf{to} \ C) \ (stopbind(m < C', Sig, Co, \textsc{e}, o) \ \textbf{to} \ C \ \textsc{i}) \\
\qquad\qquad\qquad \textbf{else} \ bind(m < C', Sig, Co, \textsc{e}, o) \ \textbf{to} \ (\textsc{i}' \ \textsc{i}) \ \textbf{fi} \ \langle C\!:\!Cl \mid Inh : \textsc{i}', Mtds : \textsc{m}\rangle
\end{array}
$$

$$
\begin{array}{l}
(found(b, C') \ \textbf{to} \ C) \ (stopbind(m, Sig, Co, \textsc{e}, o) \ \textbf{to} \ C \ \textsc{i}) \ \langle C\!:\!Cl \mid Inh : \textsc{i}', Mtds : \textsc{m}\rangle \\
\quad \longrightarrow \ \textbf{if} \ b \ \textbf{then} \ bound(get(m, \textsc{m}, \textsc{e})) \ \textbf{to} \ o \ \textbf{else} \ bind(m, Sig, Co, \textsc{e}, o) \ \textbf{to} \ \textsc{i} \ \textbf{fi} \\
\qquad\quad \langle C\!:\!Cl \mid Inh : \textsc{i}', Mtds : \textsc{m}\rangle
\end{array}
$$

$$
\begin{array}{l}
find(C, C'') \ \textbf{to} \ \varepsilon \ \longrightarrow \ found(false, C) \ \textbf{to} \ C'' \\
find(C, C'') \ \textbf{to} \ \textsc{i} \ C \ \textsc{i}' \ \longrightarrow \ found(true, C) \ \textbf{to} \ C'' \\
(find(C, C'') \ \textbf{to} \ C' \ \textsc{i}) \ \langle C'\!:\!Cl \mid Inh : \textsc{i}'\rangle \\
\quad \longrightarrow \ (find(C, C'') \ \textbf{to} \ \textsc{i} \ \textsc{i}') \ \langle C'\!:\!Cl \mid Inh : \textsc{i}'\rangle \ \textbf{if} \ (C \neq C')
\end{array}
$$

$$
\begin{array}{l}
\langle o\!:\!Ob \mid Att : \textsc{a}, Pr : \langle (v := new \ C(\textsc{e}); \textsc{s}), \textsc{l}\rangle\rangle \ \langle C\!:\!Cl \mid Tok : n\rangle \\
\quad \longrightarrow \ \langle o\!:\!Ob \mid Att : \textsc{a}, Pr : \langle (v := newid; \textsc{s}), \textsc{l}\rangle\rangle \ \langle C\!:\!Cl \mid Tok : next(n)\rangle \\
\langle newid\!:\!Ob \mid Cl : C, Att : \varepsilon, Pr : \langle run, \varepsilon\rangle, PrQ : \varepsilon, EvQ : \varepsilon, Lab : 1\rangle \\
inherit(newid, \varepsilon) \ \textbf{to} \ C(eval(\textsc{e}, (\textsc{a}, \textsc{l})))
\end{array}
$$

Figure 17: An operational semantics with multiple inheritance.

bind message can be sent 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. In order to easily traverse the inheritance graph, an inheritance list 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 Internal Virtual Binding

The binding of an internal virtual call $m < C'$ constrained by C' is slightly more complex. When a match in a class C is found, the inheritance graph of C is inspected to ensure that $C \prec_{\mathcal{C}} C'$, otherwise the binding must resume. Note the additional *stopbind* message used to suspend binding while checking that $C \prec_{\mathcal{C}} C'$. This is done by two auxiliary messages: The message *find*(C', C) **to** I represents that C is asking I if C' is found in I or further

$$\begin{aligned}
&\mathit{inherit}(o, \textsc{ia})\ \textbf{to}\ \mathit{nil} = \mathit{inherited}(\mathit{evalS}((\mathit{self} \mapsto o)\ \textsc{ia}), \varepsilon)\ \textbf{to}\ o\\
&\mathit{inherit}(o, \textsc{ia})\ \textbf{to}\ (\textsc{i}\ C(\mathit{In}))\ \ \langle C : \mathit{Cl} \mid \mathit{Param} : \textsc{v}, \mathit{Inh} : \textsc{i}', \mathit{Att} : \textsc{ia}' \rangle\\
&\quad = \mathit{inherit}(o, (\mathit{pass}(\textsc{v}, \mathit{evalS}(\mathit{In}, \textsc{ia}))\ \textsc{ia}'\ \textsc{ia}))\ \textbf{to}\ (\textsc{i}\ \textsc{i}')\\
&\qquad \langle C : \mathit{Cl} \mid \mathit{Param} : \textsc{v}, \mathit{Inh} : \textsc{i}', \mathit{Att} : \textsc{ia}' \rangle
\end{aligned}$$

$$\begin{aligned}
&\mathit{pass}(\textsc{v}, \varepsilon) = \textsc{v}\\
&\mathit{pass}((v\ \textsc{v}), e'\ \textsc{e}') = (v \mapsto e')\ \mathit{pass}(\textsc{v}, \textsc{e}')
\end{aligned}$$

$$\begin{aligned}
&\mathit{evalS}(\varepsilon, \textsc{a}) = \varepsilon\\
&\mathit{evalS}((v \mapsto e)\ \textsc{ia}, \textsc{a}) = (v \mapsto \mathit{eval}(e, \textsc{a}))\ \mathit{evalS}(\textsc{ia}, (v \mapsto \mathit{eval}(e, \textsc{a}))\ \textsc{a})
\end{aligned}$$

$$(\mathit{inherited}(\textsc{a})\ \textbf{to}\ o)\ \ \langle o : \mathit{Ob} \mid \mathit{Att} : \varepsilon \rangle = \langle o : \mathit{Ob} \mid \mathit{Att} : \textsc{a} \rangle$$

$$\textsc{a}\ (v \mapsto e)\ \textsc{a}'\ (v \mapsto e') = \textsc{a}\ (v \mapsto e)\ \textsc{a}'$$

Figure 18: State instantiation equations for multiple inheritance.

up in the hierarchy, whereas $\mathit{found}(b, C')$ **to** C gives the answer to C, where the boolean b is true if the request was successful. The rewrite rules formalizing this are given in Figure 17, ignoring class parameter lists. This search corresponds to breadth-first, left-first traversal of the inheritance graph.

5.5.3 Object Creation and Attribute Instantiation

Object creation results in a new object with a unique identifier, calling its *run* method (if present in the class) synchronously. New object identifiers are created 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. The rule is given in Figure 17. In the figure, *newid* denotes the new identifier. Before the new object can be activated, its initial state must be created. This is done by equations in rewriting logic, given in Figure 18. Notice again that the use of equations enables a new object to be created and initialized in a single rewriting step. The equations collect attribute lists, which consist of program variables bound to initial expressions, from the classes inherited by C. The initial expressions must be reduced to values and bound to the program variables in the state. Class parameters and inherited attributes provide a mechanism to pass values to the initial expressions of the inheritance list in a class. The variables bound by the class parameters are stored first in the attribute list of a class in the textual order.

The auxiliary function *pass* associates actual parameters to formal class parameters and *inherit* builds an attribute list from class parameters and attributes during traversal of the inheritance tree in a right-first depth-first order. Inherited attributes and their initializing expressions are dynamically accumulated. The traversal results in a list of attributes with initializing expressions, which are evaluated by *evalS* from left to right and delivered to the new object. The attribute list is ordered such that the attributes of a superclass precede those of a subclass, for all classes above the class of the object. Consequently, the type system can guarantee that all variables occurring in an initial expression of a program variable v have been instantiated before v is instantiated. The resulting state is consumed by the new object. In the presence of multiple inheritance, a class C may inherit a superclass several times. The final equation on attribute lists ensures that an attribute is only stored once. Thus multi-inheritance of the same class is the same as inheriting the class once, keeping the leftmost instantiation.

5.6 Subject Reduction

A subject reduction property for the extended language is given in this section. We first prove some preliminary properties.

Lemma 14 *Let P be an arbitrary Creol program. If $\Gamma \vdash P$ then the execution of every **new** $C(\textsc{e})$ statement in P will create a new object of class C with type correct attribute instantiations without causing runtime type errors.*

Proof. The proof is by induction over the depth of the inheritance tree above C. At each step of the induction, we assume that attributes are correctly collected from the inheritance tree above the current considered class and that all variables occurring in the initial expressions of the attributes have been instantiated.

Let C' be a leaf above C with formal class parameters $\textsc{v}$ of type T' and let In be the actual parameter values passed to C'. For well-typed programs, we know that $\Gamma \vdash_{\mathcal{F}} \mathit{In} : T \prec T'$ (this is checked by the *matchparam* function). Consequently, In can be type-correctly assigned to $\textsc{v}$. Let $\textsc{ia}$ an accumulated attribute list (if we are instantiating a leaf class directly, $\textsc{ia}$ will be empty). The operational semantics gives us

$$\begin{aligned}
&\mathit{inherit}(o, \textsc{ia})\ \textbf{to}\ C'(\mathit{In})\ \langle C' : \mathit{Cl} \mid \mathit{Param} : \textsc{v}, \mathit{Inh} : \mathit{nil}, \mathit{Att} : \textsc{ia}' \rangle \\
&\quad = \mathit{inherit}(o, (\mathit{pass}(\textsc{v}, \mathit{evalS}(\mathit{In}, \textsc{ia}))\ \textsc{ia}'\ \textsc{ia}))\ \ \textbf{to}\ \mathit{nil} \\
&\qquad \langle C' : \mathit{Cl} \mid \mathit{Param} : \textsc{v}, \mathit{Inh} : \mathit{nil}, \mathit{Att} : \textsc{ia}' \rangle
\end{aligned}$$

For C', the function $\mathit{InhAttr}(\mathit{Inh}, \Gamma_{\mathcal{C}}) = \varepsilon$, so initial expressions in $\textsc{ia}'$ have been type checked from left to right in the typing environment extended with class parameters. Initial expressions are evaluated from left to right by *evalS*, which corresponds to how the typing environment is constructed by the typing rule. Consequently, the evaluation of initial expressions in $\textsc{ia}'$ is type correct. Note that the attributes of C' precede the accumulated attribute list $\textsc{ia}$, so initial expressions in $\textsc{ia}$ may safely refer to variables in $\mathit{pass}(\textsc{v}, \mathit{In})\ \textsc{ia}'$.

For the induction step, we consider a class C' and assume that the attribute instantiations of its superclasses are type correct, as the actual parameter values are type correct for the formal parameters of each superclass. The *inherit* function traverses the inheritance tree of C' in a right first depth first order, passing on appropriate class parameter values from the inheritance list and collecting attributes from each superclass. The following equation from the operational semantics applies:

$$\begin{aligned}&\mathit{inherit}(o, \textsc{ia})\ \textbf{to}\ (\textsc{i}\ C'(\mathit{In}))\ \ \langle C' : \mathit{Cl} \mid \mathit{Param} : \textsc{v}, \mathit{Inh} : \textsc{i}', \mathit{Att} : \textsc{ia}'\rangle\\ &\quad = \mathit{inherit}(o, (\mathit{pass}(\textsc{v}, \mathit{evalS}(\mathit{In}, \textsc{ia}))\ \textsc{ia}'\ \textsc{ia}))\ \textbf{to}\ (\textsc{i}\ \textsc{i}')\\ &\qquad \langle C' : \mathit{Cl} \mid \mathit{Param} : \textsc{v}, \mathit{Inh} : \textsc{i}', \mathit{Att} : \textsc{ia}'\rangle\end{aligned}$$

The attribute list $\textsc{ia}''$ collected from all superclasses precede the attribute list of C'. The induction hypothesis tells us that the evaluation of initial expressions in $\textsc{ia}''$ is type correct. Consequently the evaluation of initial expressions in $(\textsc{ia}''\ \textsc{ia}')$ is type correct.

If C' is the actual class of the object, the actual class parameters passed to C' come from the statement **new** $C(\mathit{In})$. If the formal parameters $\textsc{v}$ have type T, the functional language $\mathcal{F}$ ensures that the expression *In* evaluates to a value of type T' such that $T' \prec T$ and *In* can be type-correctly assigned to $\textsc{v}$. In this case there is no accumulated attribute list (i.e., $\textsc{ia} = \varepsilon$) and it follows from the induction hypothesis that all attributes above C' are collected and the evaluation of initial expressions $\textsc{ia}''\ \textsc{ia}'$ will not cause type errors. Thus, as the new unique object o is of class C and the function that accumulates attributes is restricted to classes above C, o will start with the desired initial state. As the collection of the object state is done by equations, the state will be present before the *run* method starts. ■

Lemma 15 *Let P be an arbitrary Creol program. If $\Gamma \vdash P$, then every method invocation $!x.m(\textsc{e})$, $!m@C(\textsc{e})$ and $!m < C(\textsc{e})$ in P can be type-correctly bound at runtime to the method provided that x is not a null pointer.*

Proof. We consider how the invocation message is bound for the evaluation rules for $!x.m(\mathit{Sig}, \mathit{Co}, \textsc{e})$, $!m@C(\mathit{Sig}, \mathit{Co}, \textsc{e})$, and $!m < C(\mathit{Sig}, \mathit{Co}, \textsc{e})$. The proof is similar to the proof of Lemma 7 but accounts for dynamic traversal of the inheritance graph. As the call to m is well-typed, we may assume that o is typed by an interface I and that o 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.

The evaluation rule for an external method call $!o.m(\mathit{Sig}, \mathit{Co}, \textsc{e})$ eventually creates a bind message $\mathit{bind}(m, \mathit{Sig}, \mathit{Co}, \textsc{e}, o)$ **to** C, if C is the dynamically identified class of o. Method lookup proceeds by the rule

$$\begin{aligned}&(\mathit{bind}(m, \mathit{Sig}, \mathit{Co}, \textsc{e}, o)\ \textbf{to}\ C\ \textsc{i}')\langle C : \mathit{Cl} \mid \mathit{Inh} : \textsc{i}, \mathit{Mtds} : \textsc{m}\rangle\\ &\quad \longrightarrow \textbf{if}\ \mathit{match}(m, \mathit{Sig}, \mathit{Co}, \textsc{m})\ \textbf{then}\ \mathit{bound}(\mathit{get}(m, \textsc{m}, \textsc{e}))\ \textbf{to}\ o\\ &\qquad\qquad\qquad\qquad \textbf{else}\ \mathit{bind}(m, \mathit{Sig}, \mathit{Co}, \textsc{e}, o)\ \textbf{to}\ (\textsc{i}\ \textsc{i}')\ \textbf{fi}\end{aligned}$$

where $\textsc{i}'$ 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 $!m@C(Sig, Co, \mathrm{E})$ is similar to the previous case. It results in an invocation message $invoc(m@C, Sig, Co, \mathrm{E})$ **to** o which will cause a bind message $bind(m, Sig, Co, \mathrm{E}, o)$ **to** C to be generated, where C is the specified class. As in the previous case, the method m will be correctly bound if there is a method m declared above C with signature Sig' and cointerface Co' such that $Sig \prec Sig'$ and $Co \prec Co'$. It is guaranteed by the static type analysis that C inherits a method m with a matching signature and cointerface. Consequently, the *bind* message will succeed in binding the call to a method declared above C when traversing the inheritance tree above C.

For an internal bounded call $m < C(Sig, Co, \mathrm{E})$ the resulting invocation message $invoc(m < C, Sig, Co, \mathrm{E})$ **to** o will generate a bind message $bind(m < C, Sig, Co, In, o)$ **to** C I. The bounded binding rule

$$
\begin{array}{l}
(bind(m < C, Sig, Co, In, o) \textbf{ to } C' \ \mathrm{I}) \ \langle C' : Cl \mid Inh : \mathrm{I}', Mtds : \mathrm{M}\rangle \\
\longrightarrow \textbf{ if } match(m, Sig, Co, \mathrm{M}) \\
\qquad\qquad \textbf{then } (find(C, C') \textbf{ to } C') \ (stopbind(m < C, Sig, Co, In, o) \textbf{ to } C' \ \mathrm{I}) \\
\qquad\qquad \textbf{else } bind(m < C, Sig, Co, In, o) \textbf{ to } (\mathrm{I}' \ \mathrm{I}) \textbf{ fi } \langle C' : Cl \mid Inh : \mathrm{I}', Mtds : \mathrm{M}\rangle
\end{array}
$$

traverses the inheritance graph of C', breadth first, searching for a matching method declaration located below C. For every match in the inheritance graph, say in a class C', a find message $find(C, C')$ **to** C' is generated, which means that the class C' is a candidate for binding m. This message returns true if C is above C' and false otherwise. In the latter case, the breadth first search continues. The type system guarantees that there is at least one match in a class below C, so the search will succeed. ■

Theorem 16 (Subject reduction) *Runtime type errors do not occur in well-typed programs.*

Proof. The proof follows directly from the proofs of Theorems 9 and 13. It remains to consider object creation and method binding. Remark that external calls are type checked as internal calls to the actual class of the object. Object creation and bounded calls are covered by Lemmas 14 and 15. ■

6 Related Work

Many object-oriented languages offer constructs for concurrency; a survey is given in [63]. A common approach has been to rely on the tight synchronization of RPC and keep activity (threads) and objects distinct, as done in Hybrid [61] and Java [35], or on the rendezvous concept in concurrent objects languages such as Ada and POOL [5]. For distributed systems, with potential delays and even loss of communication, these approaches seem less desirable. 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 at the level of the operational semantics: no new threads and active loop 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 [36] 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. 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 (discarding local process variables). In contrast to monitors sufficient signaling is ensured at the semantic level, which significantly simplifies reasoning [24]. Internal reasoning control is facilitated by the explicit declaration of release points, at which class invariants are expected to hold [30].

Languages based on the Actor model [3, 4] take asynchronous messages as the communication primitive, focusing 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 [74], Argus [51], ConcurrentSmalltalk [73], Eiffel// [16], CJava [23], and in the Join calculus [32] based languages Polyphonic C$^\sharp$ [8] and Join Java [41]. 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.

Most languages supporting asynchronous methods either disallow inheritance [41, 74] or impose redefinition of asynchronous methods [16]. Multiple inheritance is supported in languages such as C++ [70], CLOS [27], Eiffel [57], 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 [62, 67] also depend upon linearization to be merged correctly into the single inheritance chain. Linearization changes the parent-child relationship between classes in the inheritance hierarchy [68], and understanding method binding quickly becomes difficult.

Maude's inherent object concept [20, 55] represents an object's state as a subconfiguration, as we have done in this paper, but in contrast to our approach object behavior is captured directly by rewrite rules. Both Actor-style asynchronous messages and synchronous transitions (rewrite rules which involve more than one object) 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. 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$^\sharp$ 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 a deterministic binding strategy. 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 [37, 59]. Process algebra is usually based on synchronous communication. In contrast to, e.g., the asynchronous π-calculus [38], 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 [65, 71, 72]. 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 [34] 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, 34].

A concurrent object calculus with single inheritance is presented by Laneve [50]. 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 in the literature are usually based on the *objects-as-records* paradigm. This approach focuses on subtyping issues related to subclassing, but issues related to method binding are not easily captured. Even access to methods of superclasses is not addressed in Cardelli's denotational semantics of multiple inheritance [14]. Rossi, Friedman, and Wand [66] propose a formal definition of multiple inheritance based on *subobjects*, a runtime data structure used for virtual pointer tables [49, 70]. This formalism focuses 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, as targeted by Creol [47]. In particular, we have given an operational semantics based on dynamic and distributed traversal of the available classes, rather than through 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 [39], 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 [62] uses a nominal type system for mixin-based traits, extending a single inheritance relation. Asynchronous method calls, interfaces, and cointerfaces necessitate a more refined type system, including typing effects [53]. A type and effect system provides an elegant way of adding context information to the type analysis. 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 [31]. For Creol, an effect system is used to guarantee type correct signatures for asynchronous method calls.

7 Conclusion

This paper has presented Creol, a 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 presented 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*.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, New York, NY, 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-Verlag, 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, Paris, 1996. Chapman & Hall.

[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, New York, NY, Oct. 1990. ACM Press.

[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$^\sharp$. *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-Verlag, 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, Ottawa, Canada, 1990. ACM Press.

[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-Verlag, Berlin, 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-Verlag, 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, London, 1997. Springer-Verlag.

[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), Revised Lectures*, volume 2852 of *Lecture Notes in Computer Science*, pages 71–98. Springer-Verlag, 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-Verlag, 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-Verlag, 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] 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.

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

[32] 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.

[33] 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.

[34] 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 (Nice, France, September 12, 1998)*, volume 16(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998.

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

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

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

[38] 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-Verlag, 1991.

[39] 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.

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

[41] 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-Verlag, 2003.

[42] 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.

[43] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. In *Proc. 2nd IEEE International Conference on Software Engineering and Formal Methods (SEFM'04)*, pages 188–197. IEEE Computer Society Press, Sept. 2004.

[44] 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-Verlag, 2004.

[45] 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.

[46] 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.

[47] 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-Verlag, June 2005.

[48] 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-Verlag, June 1997.

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

[50] 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.

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

[52] 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.

[53] 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.

[54] 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, Cambridge, Mass., 1993.

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

[56] 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-Verlag, 2004.

[57] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ., 2 edition, 1997.

[58] 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.

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

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

[61] 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.

[62] 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-Verlag, 2003.

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

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

[65] 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.

[66] 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-Verlag, July 1996.

[67] 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-Verlag, 2003.

[68] 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.

[69] 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.

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

[71] 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-Verlag, 1994.

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

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

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