Softw Syst Model
DOI 10.1007/s10270-006-0011-2

SPECIAL SECTION PAPER

# An asynchronous communication model for distributed concurrent objects

**Einar Broch Johnsen · Olaf Owe**

Received: 15 February 2005 / Revised: 7 February 2006 / Accepted: 7 February 2006
© Springer-Verlag 2006

**Abstract** Distributed systems are often modeled by objects that run concurrently, each with its own processor, and communicate by synchronous remote method calls. This may be satisfactory for tightly coupled systems, but in the distributed setting synchronous external calls lead to much waiting; at best resulting in inefficient use of processor capacity, at worst resulting in deadlock. Furthermore, it is difficult to combine active and passive behavior in concurrent objects. This paper proposes an object-oriented solution to these problems by means of asynchronous method calls and conditional processor release points. Although at the cost of additional internal nondeterminism in the objects, this approach seems attractive in asynchronous or unreliable environments. The concepts are integrated in a small object-oriented language with an operational semantics defined in rewriting logic, and illustrated by examples.

**Keywords** Asynchronous method calls · Concurrent objects · Distributed systems · Rewriting logic

This paper extends and combines results which have appeared in the proceedings of the 2004 International Conference on Software Engineering and Formal Methods (SEFM) [37] and the 2004 International Workshop on Rewriting Logic and its Applications (WRLA) [41].

E. B. Johnsen (✉) · O. Owe
Department of Informatics,
University of Oslo, Blindern,
P.O. Box 1080, 0316 Oslo, Norway
e-mail: einarj@ifi.uio.no

O. Owe
e-mail: olaf@ifi.uio.no

## 1 Introduction

Inter-process communication is becoming increasingly important with the development of distributed computing, both over the Internet and over local networks. Object orientation appears as the leading paradigm for concurrent and distributed systems, and has been recommended by the RM-ODP [34], but standard models of object interaction seem less appropriate for distributed concurrent objects. Object interaction based on method calls is usually synchronous. The mechanism of remote method calls has been derived from the setting of sequential systems, and is well suited for tightly coupled systems. It is clearly less suitable in a distributed setting where the components are loosely coupled. In this setting, synchronous communication gives rise to undesired and uncontrolled waiting, and possibly deadlock. 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 still unsettled, especially with respect to inheritance and redefinition.

In order to unite object orientation and distribution in a natural way, we need intuitive high-level programming constructs. In this paper programming constructs for concurrent objects are discussed, based on *processor release points* and a notion of *asynchronous method call*. Processor release points influence the implicit internal control flow in concurrent objects. This reduces time spent waiting for replies to method calls in a distributed environment and allows objects to dynamically change between active and reactive behavior (client and server). The suitability of these concepts for distributed object systems is motivated through integration in an object-oriented language with a simple operational semantics, while maintaining the efficiency control of asynchronous message passing. The language is object-oriented in the sense that all object interaction happens through method calls, the language is class-based and supports inheritance. The proposed language constructs are explored in the small object-oriented language Creol. The operational semantics of the language is defined in rewriting logic [49] and is executable as a language interpreter in the tool Maude [16]. Our experiments suggest that rewriting logic and Maude provide a well-suited platform for experimentation with language constructs and concurrent environments.

The three basic interaction models for concurrent processes are shared variables, remote method calls, and message passing [6], which we review with respect to interaction between distributed concurrent objects. As shared memory models do not generalize well to distributed environments, shared variables are discarded as inappropriate to capture object interaction in the distributed setting. With *remote method invocations* (RMI), 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 call have been received. A similar approach is taken with the execution threads of, e.g., Hybrid [53] and Java [30], where concurrency is achieved through multi-threading. The interference problem for shared variables reemerges when threads operate concurrently in the same object, which happens with nonserialized methods in Java. Reasoning about programs in this setting is highly complex [2,14]: Safety is by convention rather than by language design [10]. Verification considerations therefore suggest that all methods should be serialized, which is the approach taken in 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 proceeding with its activity, and 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 does not transfer control between concurrent objects. A method call can here be modeled by an invocation and a reply message. Message passing may be synchronized, as in Ada's Rendezvous mechanism, in which case both the sender and receiver process must be ready before communication can occur. Hence, the objects synchronize on message transmission. Remote method invocations may be captured in this model if the calling object must wait 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 [63] (or eager invocation [21]). For distributed systems, even such synchronization must necessarily result in much waiting.

Message passing may also be asynchronous. In this 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 which support future variables are usually based on asynchronous message passing; 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 [8,12,17,35,62,63]. This approach seems well-suited for distributed environments, reflecting the fact that communication in a network takes time. Generative communication in Linda [13] is an approach in between shared variables and asynchronous message passing, where messages without an explicit destination address are shared in a tuple space. 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]. We believe that a satisfactory notion of method calls for the distributed setting should be asynchronous, combining asynchronous message passing with the structuring mechanism provided by the method concept.

In the Creol language, method calls are taken as the communication primitive for concurrent objects. In the operational semantics of the language, method calls are represented by pairs of asynchronous messages, allowing message overtaking. We do not believe that distribution should be kept transparent to the programmer as in the RMI communication model, rather communication in the distributed setting should be explicitly asynchronous. Also, separating execution threads from objects breaks the modularity and encapsulation of object orientation, leading to a very low-level style of programming. To model real world systems in an object-oriented manner, asynchronously communicating concurrent objects appear as a much more natural approach.

This paper considers the *communication model* of Creol, extending [37,41] as follows: a mechanism for time-out and race conditions is added, a more abstract and complete semantics is given in which the intra-object process interleaving mechanism is significantly improved, and several new examples are provided. Regarding *class inheritance* in Creol, dynamic binding is discussed in [39] and the inheritance anomaly in [40]. The *type system* for Creol is given in [43] and Hoare-style *proof rules* in [23].

The paper is organized as follows. Section 2 outlines the overall setting of the approach. Section 3 introduces Creol, focusing on asynchronous method calls and processor release points. Section 4 models a peer-to-peer network and Sect. 5 the dining philosophers in Creol. Section 6 illustrates how explicit synchronization may be obtained. Section 7 models a coordinator class for joint computation, and Sect. 8 a class for implicit method invocation in Creol. Section 9 considers a mechanism capturing method time out and method race conditions. Section 10 defines the operational semantics of Creol in rewriting logic and Sect. 11 discusses executing the operational semantics. Sections 12 and 13 consider related and future work, and Sect. 14 concludes the paper.

## 2 Background

According to the RM-ODP, we can represent components by (collections of) objects that run in parallel and communicate asynchronously by means of remote method calls with input and output parameters. Often, objects are supplied by third-party manufacturers unwilling to reveal their implementation details. Therefore, reasoning should be done relying on abstract specifications of the system's components. In this setting specifications of observable behavior seem particularly attractive. A *behavioral interface* consists of a set of method names with signatures and semantic constraints on the use of these methods. We assume that components come equipped with behavioral interfaces that instruct us on how to use them. A component may have multiple interfaces, which correspond to the specifications of different viewpoints, thus providing a separation of concerns between the different services offered by the component.

### 2.1 Strong typing

We consider typing where two kinds of variables are declared; an object variable is typed by an interface and a data variable is typed by a data type. For each method invocation $o.m(inputs; outputs)$, where interface $I$ is the declared type of $o$, strong typing ensures that the actual object $o$ (if not nil) will support $I$, the method $m$ will be declared in $I$, and the call will be understood in the sense that the method is implemented in the class of $o$, and formal and actual parameters match. Explicit hiding of class attributes and methods is not needed, because typing of object variables is based on interfaces and only methods mentioned in the interface (or its super-interfaces) are visible. Multiple inheritance is allowed at the interface level, restricted to a form of behavioral subtyping [46]. A class may implement several interfaces, provided that it satisfies the syntactic and semantic requirements of these interfaces. An object of class $C$ supports an interface $I$ if $C$ implements $I$. 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 subtype of I* in a context where $I$ is expected, although the classes of the two objects may differ.

An interface may specify observable behavior in the form of an assumption-guarantee specification [44] on the local communication history. The assumption is a requirement on the behavior of objects in the environment. As customary in the assumption-guarantee paradigm, the guaranteed invariant need only hold when the assumption is respected by the environment. In our setting, the paradigm is adjusted to deal with input and output aspects of distributed communicating systems. The semantic requirements of an interface rely on the present communication history of an object offering the interface. A compositional formalism for reasoning about behavioral interfaces, and an associated interface refinement relation, is given in previous work [36,38]. At the imperative level, reasoning about class invariants in terms of class attributes and the local history can be done locally in each class [23], extending standard assertional reasoning. In this paper, these semantic requirements on interfaces are not considered in detail.

## 3 The Creol language

The Creol language proposes programming constructs for distributed concurrent objects based on asynchronous method calls and processor release points. Concurrent objects are potentially active: an object encapsulates an execution thread. Consequently, elements of basic data types (referred to as data) are not considered as objects in the language. Rather, Creol's concurrent objects resemble top-level objects in, e.g., Hybrid. Objects have identity: an object's name is unique, communication takes place between named objects, and object names may be exchanged between objects. As motivated above, Creol objects are typed by interfaces, resembling CORBA's IDL [61], but extended with mechanisms for static type control in dynamically reconfigurable systems. Types, interfaces, and classes may be parameterized by formal interfaces and types, allowing generalized definitions. All use of parameterized constructs requires actual parameters. The language supports strong typing, based on a nominal type-system which ensures that invoked methods are supported by the callee (when not nil) and that formal and actual parameters match [43].

## 3.1 Interfaces

Interfaces describe viewpoints to objects and provide a typing mechanism for object variables. In order to support generic descriptions, interfaces may be parameterized by types, but also by data and objects, typed by data types and interfaces, respectively. Interface parameters describe the minimal environment that any object offering the interface needs at the point of creation, and may be passed on to inherited interfaces.

In order to support protocol sessions between concurrent objects, an object may require access to an interface of the calling object. This way, the callee may safely invoke methods of the caller and not only passively process calls to its own methods. Using a **with** clause statically restricts the communication environment to external objects offering a so-called *cointerface*, while admitting remote calls to the methods of the caller declared in the cointerface [36,38]. (An example is given in Sect. 8.) At the imperative level, access to the caller is given by an implicit parameter, *caller*. A **with** clause in an interface applies to the methods declared in that interface; for inherited methods the associated **with** clause applies. This gives strong typing in the asynchronous setting. When call-back access to the caller is not required, the keyword *Any* is used in the **with** clause. Mutual dependency is captured if two interfaces have each other as cointerface. The syntax for interface declarations (*IF*) and type expressions (*Type*) is given in Fig. 1.

Standard type systems lack expressiveness regarding component connectors [58]. Using interfaces to type variables provides abstractions for connecting to objects; in particular, object parameters capture static links to environment resources, cointerfaces capture requirements to a counterpart in a protocol between concurrent objects (while specification predicates capture the intended sequence of interactions in the protocol).

## 3.2 Example

We consider the interfaces of a node in a peer-to-peer file sharing network. A *toClient* 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 *toServer* interface offers a method for

$$
\begin{array}{lll}
IF & ::= & \textbf{interface}\, N\, \{[\{N\}^{+}_{,}]\}^{?}\, \{(Par)\}^{?}\, \{\textbf{inherits}\, Inh\}^{?} \\
 & & \textbf{begin}\ \{\textbf{with}\ Type\ Msig^{+}\}^{?}\ \textbf{end} \\
Type & ::= & N\{[\{Type\}^{+}_{,}]\}^{?} \\
Inh & ::= & \{Type\{(\mathrm{E})\}^{?}\}^{+}_{,} \\
Par & ::= & \{\{v\}^{+}_{,} : Type\}^{+}_{,} \\
Msig & ::= & \textbf{op}\ N\ \{(\{\textbf{in}\ Par\}^{?}\ \{\textbf{out}\ Par\}^{?})\}^{?} \\
CL & ::= & \textbf{class}\, N\, \{[\{N\}^{+}_{,}]\}^{?}\, \{(Par)\}^{?}\, \{\textbf{implements}\, Inh\}^{?} \\
 & & \textbf{begin}\ Vdecl^{?}\ \{\{\textbf{with}\ Type\}^{?}\ Mtds\}^{*}\ \textbf{end} \\
Vdecl & ::= & \textbf{var}\ \{\{v\}^{+}_{,} : Type\{= e\}^{?}\}^{+}_{,} \\
Mtds & ::= & \{Msig == \{Vdecl;\}^{?}\ \mathrm{S}\}^{+}
\end{array}
$$

**Fig. 1** BNF grammar for interface and class declarations. *Curly brackets* are used as meta parenthesis, superscript ? for optional parts, superscript * for repetition zero or more times, whereas $\{\ldots\}^{+}_{,}$, denotes repetition one or more times with "," as delimiter. Identifiers *N* denote interface, class, type, or method names. The syntax for variables *v*, expressions *e*, expression lists E, and statement lists S, is given in Fig. 2

obtaining a list of files available from the node, and a mechanism for downloading packs, i.e., parts of a target file. The *toServer* interface is available to servers in the network.

```
interface toClient     interface toServer
begin                  begin
 with Any               with toServer
  op availFiles           op enquire
  op reqFile              op getLength
end                       op getPack
                       end
```

Due to the **with**-construct and strong typing, the *enquire*, *getLength*, and *getPack* methods may only be called by other objects supporting the *toServer* interface. To save space, discussion of method parameters is postponed to Sect. 4. The two interfaces may be inherited by a third interface *Peer* which describes nodes able to act according to both the *toClient* and the *toServer* role. The *Peer* interface becomes:

```
interface Peer
 inherits toClient, toServer
begin  end
```

## 3.3 Classes and objects

At the imperative level, attributes (program variables) and method declarations are organized in classes. Classes may include class parameters, which become assignable attributes of object instances as in Simula [19]. The syntax for classes (*CL*) is given in Fig. 1. Objects are dynamically created instances of classes, supplied with actual type, data, and object parameters. The attributes of an object are encapsulated and can only be accessed via the object's methods. Among the declared methods, we distinguish two methods *init* and *run*, which are given special treatment operationally. The method *init* is invoked at object creation to instantiate attributes. After initialization the method *run* is started, if it is provided. Apart from *init* and *run*, declared methods may be invoked internally and by other objects of appropriate interfaces. When called from other objects, these methods reflect passive or reactive behavior in the object, whereas *run* initiates active behavior. Methods need not terminate and all method instances may be temporarily *suspended*.

In order to focus the paper on asynchronous method calls and processor release points in method bodies, other language aspects are not discussed in detail, including inheritance and typing. To simplify the exposition, we assume a common type Data of basic data values, such as the natural numbers Nat, strings Str, and the object identifiers Obj, which may be passed as arguments to methods. Expressions Expr evaluate to Data, and do not have side effects. In particular we denote by ObjExpr and BoolExpr two subtypes of Expr; expressions of these types reduce to object identifiers and Booleans, respectively. Let Var denote the set of program variables, Mtd the set of method names, and Label a set of method call identifiers. There is read-only access to method in-parameters, to *self*, used for self-reference, to *caller*, used for return calls. Label variables may not be used in expressions.

## 3.4 Asynchronous methods

An object offers methods to its environment, specified through a number of interfaces and cointerfaces. All interaction with an object happens through method calls. In the asynchronous setting method calls can always be emitted, because the receiving object cannot block communication. *Method overtaking* is allowed: if methods offered by an object are invoked in one order, the object may start the method instances in another order. A method instance is, roughly speaking, program code with nested processor release points, 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 variables. In particular, the values of formal parameters are stored locally, but other local variables may also be created. Semantically, a method activation is represented by a *process* $\langle \mathrm{S},\mathrm{L}\rangle$ where S is a sequence of statements and L:Var $\rightarrow$ Data the local variable bindings. Consider an object $o$ which offers the method

$$\textbf{op}\ m(\textbf{in}\ x : \mathsf{Nat}\ \textbf{out}\ y : \mathsf{Data}) == \textbf{var}\ z : \mathsf{Nat} = 0;\ \mathrm{s}$$

to the environment. Accepting a call to $m$ with argument 2 from another object $o'$ creates a process $\langle \mathrm{s}, \{caller \mapsto o', label \mapsto l, x \mapsto 2, y \mapsto nil, z \mapsto 0\}\rangle$ in the object $o$. An object can have several (suspended) instances of a method, possibly with different values for local variables. The values of the implicit parameters *label* and *caller* are used to correctly identify the call at run-time, in order to bind the return values in the calling process. Figure 2 gives the language syntax for program sentences except for variable declarations, given in Fig. 2. In particular, the syntax **var** $x, y : T = e$ is syntactic sugar for **var** $x : T = e, y : T = e$.

An asynchronous method call is made with the statement $t!o.m(\mathrm{E})$, where $t \in$ Label provides a symbolic reference to the call, $o$ 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 *local* when $o$ is omitted or evaluates to the same value as *self*, otherwise the call is *remote*. As no synchronization is involved, process execution can proceed after an asynchronous call, until the return values are actually needed by the process. The label may be omitted if a reply is not explicitly requested. Label variables may be reused when the reference to the original call is no longer needed.

To fetch the return values from the call, say in a variable list V, we ask for the reply to our call: $t?(\mathrm{V})$, where the label name $t$ statically binds the reply to an invocation by the type analysis in order to type check the out-parameters. The *reply statement* $t?(\mathrm{V})$ treats V as a list of future variables [63]. If the reply to the call has arrived, return values are assigned to V and the execution continues without delay. If the reply has not arrived, process execution is blocked. In order to avoid blocking in the asynchronous case, processor release points

| *Syntactic categories.* | *Definitions.* |
|---|---|
| $t$ in Label | $g ::= wait \mid b \mid t? \mid \neg t? \mid g \wedge g$ |
| $g$ in Guard | $p ::= x.m \mid m$ |
| $p$ in MtdCall | $\mathrm{S} ::= \varepsilon \mid s; \mathrm{S}$ |
| $s$ in Stm | $s ::= (\mathrm{S}) \mid (\mathrm{S}\Box\mathrm{S}) \mid (\mathrm{S}\Vert\mathrm{S})$ |
| $v$ in Var | $\mid \mathrm{V} := \mathrm{E} \mid v := \textbf{new}\ Type(\mathrm{E})$ |
| $e$ in Expr | $\mid \textbf{if}\ b\ \textbf{then}\ \mathrm{S}\ \textbf{else}\ \mathrm{S}\ \textbf{fi}$ |
| $m$ in Mtd | $\mid \textbf{while}\ b\ \textbf{do}\ \mathrm{S}\ \textbf{od}$ |
| $x$ in ObjExpr | $\mid\ !p(\mathrm{E}) \mid t!p(\mathrm{E}) \mid t?(\mathrm{V}) \mid p(\mathrm{E}; \mathrm{V})$ |
| $b$ in BoolExpr | $\mid \textbf{await}\ g \mid \textbf{await}\ t?(\mathrm{V}) \mid \textbf{await}\ p(\mathrm{E}; \mathrm{V})$ |

**Fig. 2** The language syntax for method definitions, with typical terms for each category. *Capitalized terms* such as E, V, and S, denote lists of the syntactic categories of the corresponding lowercase terms

are introduced for reply guards (Sect. 3.5). In this case, process execution is *suspended* rather than blocked.

The use of label variables is subject to static language restrictions to ensure natural and deterministic static binding [43]: a reply statement $t?(\mathrm{V})$ may only occur in a method or loop body if a "pending" call to $t$ is statically guaranteed in the body. A call to $t$ is pending after an invocation to $t$, and before a reply statement to $t$ or another invocation to $t$. Any number of reply guards to $t$ may occur when a call to $t$ is pending.

All methods may be invoked synchronously as well as asynchronously. Synchronous (RMI) method calls immediately block the processor while waiting for the reply and have the syntax $o.m(\mathrm{E};\mathrm{V})$, defined as $t!o.m(\mathrm{E})$; $t?(\mathrm{V})$ for some fresh label $t$. 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 invocation of its methods. It is clear that in order to reply to local calls, the calling method must eventually suspend its own execution. Therefore the reply statement $t?(\mathrm{V})$ 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. Local calls need not be prefixed by an object identifier, in which case they may be identified syntactically, otherwise equality between caller and callee is determined at runtime.

### 3.5 Processor release points

Nondeterministic choice may be captured as a basic programming construct using guarded statements [22]. 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 guard statement **await** $g$. Guard statements can be nested within the same local variable scope, corresponding to a series of processor release points. A guard statement is *enabled* if its guard evaluates to true. Other basic statements are always enabled. When a statement which is not enabled is encountered during process execution, the process is suspended and the processor released. In contrast enabled statements which are blocked, do not release the processor. After processor release, the object's suspended and enabled processes compete for the free processor; any suspended process may be selected for execution.

The type Guard is constructed inductively:

- $wait \in \mathsf{Guard}$ (explicit release)
- $t?, \neg t? \in \mathsf{Guard}$, where $t \in \mathsf{Label}$
- $b \in \mathsf{Guard}$, where $b$ is a Boolean expression over local and object variables
- $g_1 \wedge g_2 \in \mathsf{Guard}$, where $g_1, g_2 \in \mathsf{Guard}$

The *wait* guard explicitly releases the processor. The *reply guard* $t?$ evaluates to true if the reply to the invocation with label $t$ has arrived. Evaluation of guard statements is atomic. Let **await** $t?(\mathrm{V})$ be an abbreviation for **await** $t?; t?(\mathrm{V})$, and **await** $p(\mathrm{E};\mathrm{V})$ for $t!p(\mathrm{E})$; **await** $t?(\mathrm{V})$, a typical form of nonblocking asynchronous method call.

Statements can be *composed* in different ways, reflecting the requirements to the internal control flow in the objects. An unguarded statement list is always enabled, but reply statements $t?(\mathrm{V})$ may block. Let $\mathrm{S}_1$ and $\mathrm{S}_2$ denote statement lists. Sequential composition may introduce guards; in a program statement $\mathrm{S}_1$; **await** $g$; $\mathrm{S}_2$ the guard $g$ corresponds to a potential inner processor release point. Nondeterministic choice between statements, written $\mathrm{S}_1 \Box \mathrm{S}_2$, may compute $\mathrm{S}_1$ once $\mathrm{S}_1$ is enabled, or $\mathrm{S}_2$ once $\mathrm{S}_2$ is enabled, and suspends if neither branch is enabled. (Observe that to avoid deadlock, the semantics additionally will not commit to a branch which starts with a blocking reply statement.) Nondeterministic merge, written $\mathrm{S}_1 \interleave \mathrm{S}_2$, evaluates the statements $\mathrm{S}_1$ and $\mathrm{S}_2$ in some interleaved and enabled order. Control flow without potential processor release uses **if** and **while** constructs, and multiple assignment to local and object variables is expressed as $\mathrm{V} := \mathrm{E}$ for (the same number of) program variables $\mathrm{V}$ and expressions $\mathrm{E}$.

With nested processor release points, 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. If the called object does not reply at all, deadlock is avoided in the sense that other activity in the object is possible. However, when the reply has arrived, the *continuation* of the original process must compete with other enabled suspended processes.

## 4 Example: a peer-to-peer network

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 toServer
  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 implement the *Peer* interface and are 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 Fig. 3. In the class, the method *availFiles* returns a list of pairs each consisting of 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 "::". For $x : T$ and $s : \mathsf{List}[T]$, let $hd(x :: s) = x$, $tl(x :: s) = s$, let s[*i*] denote the *i*'th element of s, for $i \leq length(s)$, and let $rem(x, s)$ be s with all occurrences of *x* removed (cf. Sect. 8).

The example demonstrates high level synchronization and efficiency control, without explicit signaling. In addition, the abbreviation **await** $p(\text{E}; \text{V})$ for label free invocation may be generalized by allowing conjunctions of guards and method calls in **await** statements. For example, the body of the *availFiles* method could then be rewritten as

```
  var fList:List[Str];
  if (sList = empty) then files:=empty else
  await hd(sList).enquire() ∧ self.availFiles(tl(sList));
  files:=((hd(sList),fList)::files) fi
```

This gives a *label free programming style*, applicable to a large class of programs.

```
class Node (db:DB)
  implements Peer
begin
with toServer
  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[List[Data]];
         await db.getFile(fId;f); pack:=f[pNbr]

with Any
  op availFiles (in sList:List[toServer]
                  out files:List[toServer ×List[Str]])
    == var l1:Label, l2:Label, fList:List[Str];
    if (sList = empty) then files:=empty
    else l1!hd(sList).enquire();
      l2!self.availFiles(tl(sList));
      await l1? ∧ l2?; l1?(fList); l2?(files);
      files:=((hd(sList),fList)::files) fi
  op reqFile(in sId:toServer, fId:Str)
    == var file, 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
```

**Fig. 3** A class capturing nodes in a peer-to-peer network

## 5 Example: the dining philosophers

This section considers a model of the dining philosophers in Creol, demonstrating how to combine active and reactive object behavior. Philosophers are active as they think, eat, and digest in an interleaved manner. Each philosopher is equipped with one chopstick, but needs two chopsticks in order to eat. To acquire chopsticks, the philosophers interact with each other. In this model, philosophers can be added dynamically to the system by a *seat* method, which seats the new object between the callee and its neighbor. A philosopher may borrow and return its neighbor's chopstick. Interaction between the philosophers is restricted by the interface. This results in a clear distinction between internal methods and methods that are externally available to other objects. Strong typing and cointerfaces guarantee that only philosophers can call the methods of the *Phil* interface.

```
interface Phil
begin
  with Phil
    op borrowStick
    op returnStick
    op seat
end
```

In this approach, each philosopher controls one chopstick, and must both retain its own and borrow its neighbor's chopstick in order to eat. Thus philosophers have their internal activity, in addition they respond to calls from the environment.

## 5.1 Implementing the philosophers

Philosophers are active, which implies that the *Philosopher* class will include a *run* method. The *run* method is defined in terms of several nonterminating internal methods representing different activities within a philosopher: *think*, *eat*, and *digest*. In *run*, the internal methods are invoked *asynchronously*. Consequently all three methods are activated before any of them can begin execution, and the calling order is arbitrary here. (A merge of the three calls would not improve efficiency.) Thus, the internal methods will be interleaved in a nonterminating manner, illustrating the processor release point construct. All three methods depend on the value of the class attribute *hungry*, which is a shared variable for the object methods. The *think* method is a loop which suspends its own evaluation before each iteration, whereas *eat* attempts to grab the philosopher's and its neighbor's chopsticks in order to satisfy its hunger. The philosopher has to wait until both chopsticks are available before it can eat. In order to avoid blocking the object processor, the *eat* method is therefore suspended after asking for the neighbor's chopstick; further processing of the method can happen once the guard is satisfied. The *digest* method represents the action of becoming hungry. The *Philosopher* class is defined in Fig. 4.

The standard configuration of five philosophers can be obtained by the sequence

```
var p:Phil; p:=new Philosopher(null);
p:=new Philosopher(p); p:=new Philosopher(p);
p:=new Philosopher(p); p:=new Philosopher(p);
```

where **new** *Philosopher(null)* initiates a cyclic structure, and **new** *Philosopher(p)*, when *p* not null, preserves cyclicity. New philosophers can be added dynamically by all objects with access to an existing philosopher.

The proposed model favors implicit control of the object's active behavior. Caromel and Rodier [12] argue

```
class Philosopher(ngb:Phil) implements Phil
begin
  var hungry:Bool= false, chopstick:Bool= true
  op init == await ngb≠null; ngb.seat(;ngb)
  op run == !think(); !eat(); !digest()
  op think == await not hungry; ⟨thinking⟩;
      await wait; !think()
  op eat == var l :Label;
      await hungry; l!ngb.borrowStick();
      await (chopstick ∧ l?); ⟨eating⟩; hungry:=false;
      !ngb.returnStick(); await wait; !eat()
  op digest == await not hungry; hungry:=true;
      await wait; !digest()

 with Phil
  op borrowStick == await chopstick;
      chopstick:=false
  op returnStick == chopstick:=true
  op seat(out n:Phil) == if ngb=null then n:=self
      else n:=ngb fi; ngb:=caller
end
```

**Fig. 4** A class capturing dining philosophers

that facilities for both implicit and explicit control are needed in languages which address concurrent programming. Explicit activity control can be programmed in Creol by using a **while** loop in the *run* method. In asynchronous distributed systems, we believe that communication introduces so much nondeterminism that explicit control structures quickly lead to program over-specification and possibly to unnecessary active waiting.

## 6 Example: explicit synchronization

Creol supports high-level implicit synchronization of interleaved method executions in concurrent objects. Consequently, objects may be regarded as abstract monitors [31], without the need for explicit signaling. Explicit signaling adds an additional level of complexity to programming and reasoning with monitors [18]. In Creol, signaling is guaranteed by the semantics and is therefore not the responsibility of the programmer. When explicit signaling is desirable, it can be achieved by encoding monitors with different signaling disciplines in Creol. We here present the encoding of a class implementing general monitors with the signal and continue discipline [6], for simplicity restricted to one condition variable. The condition variable is encoded as a triple $\langle s, d, q \rangle$ of natural numbers; $s$ represents signals to the condition variable, $d$ the number of the delayed process in the queue of the condition variable, and $q$ the number of delayed processes that have been reactivated. Queues on condition variables are FIFO ordered.

```
class Monitor
begin
  var s, d, q:Nat = 0
with Any
  op wait == var myturn:Nat=d+1; d:=d+1
    await (s>0 ∧ q=myturn); s:=s−1; q:=q+1
  op signal == if (d−q>0) then s:=s+1 else ε fi
  op signalAll == s:=(d−q)
end
```

The counters representing the queue of the condition variable may be reset when no processes are suspended on the queue, by adding an additional line at the end of the *wait* method: **if** (d=q) **then** d:=0; q:=0 **fi**.

## 7 Example: coordinators

This section introduces a class which coordinates the activities of surrounding objects and performs a joint computation. We consider *k* objects which should contribute to the inputs and share the resulting output values of a coordinated computation *body*. For each participating object *i*, we define a method $m_i$ which has as input parameters the data contributed by the object and as output parameters the result values desired by the object. In order to perform the coordinated computation, all *k* objects must have contributed their input data. The *Coordinator* class is given in Fig. 5. For simplicity, we assume that there is a single, locally defined *body* method. In the general case, the active method *run* may invoke different bodies depending on the input values, and these may be external.

We now identify two special cases of the *Coordinator* class, which correspond to synchronization constructs in the literature. First, we consider the case where the methods $m_1,\dots,m_k$ do not contain input and output parameters and let

body == $\varepsilon$.

This corresponds to *barrier synchronization* [6]. We note that if a method $m_i$ is invoked asynchronously, only the invoking process in the caller is synchronized and other processes in the caller object may proceed while waiting for the synchronization barrier. Second we consider a *join pattern* or *chord* in the Join calculus [26], Join Java [35], and Polyphonic C# [8]. A chord consists of a header and a body. The header is a set of method declarations and the body is only executed once all the methods in the header have been called. Method calls are queued up until there is a matching chord. In any given chord, at most one method may be synchronous, i.e., result in output. It is easy to see that the chord construct is a special case of the *Coordinator* class: let methods $m_1,\dots,m_{k-1}$ contain input parameters but no output parameters and $m_k$ contain output parameters but no input parameters. The body of the chord corresponds exactly to the coordinated computation of the *body* method. If there are several chords coordinating on the synchronization methods, the chords would correspond to branches in the *run* method composed by the nondeterministic choice operator.

```
class Coordinator
begin
  var In₁,...,Inₖ:Data, Out₁,...,Outₖ:Data,
    s₁,...,sₖ:Bool=false, sync:Bool=true
  op body (in x1,...,xk:Data out y1,...,yk:Data) ==
    ⟨ do coordinated computation ⟩
  op run == await (s₁ ∧ ... ∧ sₖ);
    body(In₁,...,Inₖ; Out₁,...,Outₖ)
    sync:=false; await (¬s₁ ∧ ... ∧ ¬sₖ);
    sync:=true; !run()
with Any
  op m₁ (in In:Data; out Out:Data) ==
    await (sync∧¬s₁); s₁:=true; In₁:=In;
      await ¬sync; s₁:=false; Out:=Out₁
  op m₂ ...
      ⋮
  op mₖ (in In:Data; out Out:Data) ==
    await (sync∧¬sₖ); sₖ:=true; Inₖ:=In;
      await ¬sync; sₖ:=false; Out:=Outₖ
end
```

**Fig. 5** A *Coordinator* class in Creol

## 8 Example: implicit invocation

Publish/subscribe interaction addresses scalable and distributed systems by decoupling the interacting parties with respect to space, time, and synchronization [24]. This means that the interacting parties should not know about each other, they need not participate in the interaction at the same time, and the interaction should not directly interfere with the other on-going activities of the parties. Note that there may be zero or more publishers and subscribers to an event at a particular time, and the number of publishers as well as of subscribers may vary over time. This section introduces a class which organizes implicit or anonymous method calls, allowing interested objects to subscribe and unsubscribe to particular events. An *EventManager* class is defined, which is parameterized by the signature of the event parameters, allowing new instances of the class to be defined and subscribed to as part of the normal execution flow. This approach allows static type checking of the events [54], while the asynchronous invocation mechanism of Creol naturally models concurrent notification.

An object which subscribes to particular events must understand event notifications, as captured through an *EventListener* interface:

```
interface EventListener[T]
begin
  with EventSender[T]
    op notify(in par : T)
end
```

The corresponding interface of the event manager offers methods to subscribe and unsubscribe to event notification. The manager also has an interface to publisher objects. These interfaces are defined as follows:

```
interface EventSender[T]       interface EventReceiver[T]
begin                          begin
  with EventListener[T]          with Any
    op subscribe                   op publish(in p : T)
    op unsubscribe             end
end
```

The *EventManager* class given in Fig. 6 implements *EventSender*, which will only allow event listeners to subscribe and unsubscribe to the event; i.e., an event listener must understand the *notify* method with parameter of type $T$. Notice that the explicit use of *caller* as an *EventListener* is due to the **with** clause, and allows the subscribers (*subs*) to be a list of event listeners, making the remote call to *notify* type correct. A class implementing *EventListener*[$T$] may define a variable $x$ of interface *EventSender*[$T$] and a method

**op** *notify*(**in** *par* : $T$) == **if** *caller* == $x$ **then** ... **fi**

In this case, the implicit *caller* variable may be used to distinguish similar events. In order to publish an event, an object declares an instance of the appropriate *EventManager* class and (asynchronously) invokes its *publish* method. The instances of the *EventManager* may be locally created, or obtained from some shared resource.

```
class EventManager[T]
  implements EventSender[T], EventReceiver[T]
begin
  var subs:List[EventListener[T]]= ε
with EventListener[T]
  op subscribe== subs:=(caller; subs)
  op unsubscribe==subs:=rem(caller,subs)
with Any
  op publish(in p : T)==var i: Nat = 1;
     while i ≤ length(subs)
       do !subs[i].notify(p); i:=i+1 od
end
```

**Fig. 6** An *EventManager* class in Creol

## 9 Time-outs and race conditions

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. It is therefore desirable to extend the model with a time-out mechanism, which will be defined in terms of the previously defined language.

Consider a local timing mechanism expressing a delay, such as the following operation:

**op** *delay*($n$:Nat) ==
  **if** $n > 0$ **then await** *wait*; *delay*($n-1$) **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 (Sect. 10). In an implementation of the operational semantics, a more precise (and low level) timing construct could be defined in terms of a counter on the number of rewrite steps, by some kind of underlying clock, or by defining a deterministic ordering of suspended processes.

A time-out effect for an invocation of a (remote) method $m$ may be obtained by nondeterministically combining the asynchronous call to $m$ with a call to a 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 $n \geq 0$:

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

$t!o.m(\textsc{e}); t'!delay(n); \textsc{s}; (\textbf{await}\ t?(\textsc{v}); \textsc{s}_1 \Box\ \textbf{await}\ t'?; \textsc{s}_2)$ (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 $\textsc{s}_2$. This is because the semantics of nondeterministic choice will only commit to an enabled branch which does not immediately block evaluation. Consequently, a time-out branch may take control over a branch starting with a reply statement or guard, provided the time-out 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 time-out behavior, this can be achieved by replacing the second branch by **await** $t'? \wedge \neg t?; \textsc{s}_2$.

This construction can be used to create race conditions between method calls. It is often desirable to invoke methods in several different objects and proceed with the first available reply. It may therefore be convenient to introduce a more direct syntax for time-outs and method race conditions, clearly separating the different branches. Denote by $s_{t?}$ a reply statement for a label identifier $t$, i.e., either **await** $g \wedge t?$ or $t?(\textsc{v})$. Let $t$ and $t'$ be label identifiers and assume that the possibly empty statement lists $\textsc{s}_1$ and $\textsc{s}'_1$ do not contain reply statements for $t$ and $t'$. The construct

$(t!p(\textsc{e}); \textsc{s}_1; s_{t?}; \textsc{s}_2 \mid t'!p'(\textsc{e}'); \textsc{s}'_1; s'_{t'?}; \textsc{s}'_2)$

is defined as

$t!p(\textsc{e}); t'!p'(\textsc{e}'); (\textsc{s}_1 \| \textsc{s}'_1); (s_{t?}; \textsc{s}_2 \Box s'_{t'?}; \textsc{s}'_2)$

Due to the expansion of synchronous calls, it follows that

$(o.m(\textsc{e}; \textsc{v}); \textsc{s} \mid delay(n); \textsc{s}')$

will evaluate s if the call to $m$ completes within the time-out, and otherwise s′. This construct will not suspend while waiting for the completion of $m$ or $delay$ since it is enabled (but blocked). In contrast, the statement

$(\textbf{await}\ o.m(\textsc{e}; \textsc{v}); \textsc{s} \mid \textbf{await}\ delay(n); \textsc{s}')$

will be able to suspend since it is not necessarily enabled. It will select s if evaluated before the time-out and the call to $m$ is completed, otherwise s′ may also be chosen. Remark that for time-out mechanisms priority to normal execution is determined by the size of the time-out.

The suggested construct may also be used for programming method race conditions. For instance

$(p(\textsc{e}; \textsc{v}); \textsc{s} \mid p'(\textsc{e}'; \textsc{v}'); \textsc{s}')$

will evaluate s if the $p$ call completes before the $p'$ call and otherwise evaluates s′. In this case equal priority to the branches seems most desirable, which is provided by the semantics.

### 9.1 Example

Reconsider the model of the dining philosophers from Sect. 5. In the model, a deadlock situation may occur in which every philosopher is hungry and has made a request for its neighbor's chopstick. While waiting for the reply to the request, the *eat* method suspends, allowing *borrowStick* to lend its own chopstick to a neighbor. The deadlock situation occurs when every philosopher is in possession of its neighbor's chopstick but not of its own. (Remark that although deadlocked, the philosopher may still *digest*.) It is here assumed that the *Philosopher* class contains the *delay* method and an attribute $n$ : Nat bound to a suitably large value. To avoid this deadlock situation, the *eat* method is now redefined.

```
op eat == var l:label; await hungry;
 (l!ngb.borrowStick(); await (chopstick ∧ l?);
    ⟨eating⟩; hungry:=false
  | await delay(n); await l? );
 !ngb.returnStick(); await wait; !eat()
```

After the delay, the reply guard **await** $l?$ ensures that only borrowed sticks are returned.

## 10 An operational semantics for Creol

The operational semantics of Creol is defined using rewriting logic [49], emphasizing simplicity and abstraction while modeling the essential aspects of concurrency, distribution, and communication. In the operational semantics, it is assumed that programs have been type checked. In particular, this applies to cointerface restrictions: at run-time, objects invoke methods without dynamic (cointerface) restrictions. At run-time, implicit parameters are treated as ordinary program variables; the run-time system assigns values to these parameters, while read-only access from program code is enforced by static checking. Also the initialization of variable declarations is replaced by assignments in *init*.

A rewrite theory is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$, where the signature $\Sigma$ 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 $t'$. If several rules can be applied to distinct subconfigurations, they can be executed in a *concurrent rewrite step*. As a result, concurrency is implicit in rewriting logic (RL) semantics. Many concurrency models have been successfully represented in RL [49,16]; these include Petri nets, CCS, Actors, and Unity, as well as the ODP computational model [52]. RL also offers its own model of object orientation [16].

Informally, a state configuration in RL is a multiset of terms of given types. These types are specified in (membership) equational logic $(\Sigma, E)$, the functional sublanguage of RL which supports algebraic specification in the OBJ [28] style. When modeling computational systems, configurations may include the local system states, where 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 defining equations of $E$. Conditional rewrite rules are allowed, where the condition can be formulated as a conjunction of rewrites and equations which must hold for the main rule to apply:

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

Rules in RL may be formulated at a high level of abstraction, closely resembling a compositional operational semantics. In fact, RL provides a semantic framework unifying equational and operational semantics [50].

## 10.1 System configurations

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 which consists of Creol objects, classes, and messages. (To increase parallelism in the model, message queues may be made external to object bodies as shown in [37].)

In RL, objects are commonly represented by terms $\langle o : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$ 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 [16]. We adopt this form of presentation and define both Creol objects and classes as RL objects, omitting RL types. A *Creol object* is represented by an RL object

$$\langle Ob \mid Cl, Pr, PrQ, Lvar, Att, Lab, EvQ \rangle,$$

where $Ob$ is the object identifier, $Cl$ the class name, $Pr$ the active process code, $PrQ$ a multiset of suspended processes with unspecified queue ordering, $EvQ$ a multiset of unprocessed messages, and $Lvar$ and $Att$ the local and object state variables, respectively. Let $\tau$ be a sort partially ordered by $<$, with least element 1, and let $next : \tau \to \tau$ be such that $\forall x \,.\, x < next(x)$. $Lab$ is a method call identifier corresponding to labels in the language, of sort $\tau$. The object identifier and the generated label value will provide a globally unique identifier for each method call. In the semantics, whitespace is used as the associative and commutative constructor of multisets with identity element empty, such as $PrQ$ and $EvQ$, as well as the associative constructor of variable and expression lists, also with identity element empty, whereas semicolon is used as the similar constructor of statement lists and variable bindings to improve readability. Capitalized variables are reserved for lists and multisets. Note that matching is modulo associativity, commutativity, and identity (ACI) for the multiset constructor, and modulo associativity and identity for the list constructor. In particular, matching modulo the identity $s; \varepsilon = s$ ensures that left hand sides with patterns $s; s$ (in $Pr$) match $s$.

A *Creol class* is represented by an RL object

$$\langle Cl \mid Par, Att, init, Mtds, Tok \rangle,$$

where $Cl$ is the class name, $Par$ the list of class parameters, $Att$ a list of attributes, $init$ the initialization method, $Tok$ is an unbounded set of tokens of sort $\tau$, and $Mtds$ a multiset of methods (including $run$). When an object needs a method, it is loaded from the $Mtds$ multiset of its class. In RL's object model [16], 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 representation. Let $o$ and $C$ be variables of the types $\mathsf{Obj}$ of object identifiers and $\mathsf{Class}$ of class identifiers, respectively.

## 10.2 Concurrent transitions

Concurrent change is achieved by applying concurrent rewrite steps to state configurations in the operational semantics. There are four different kinds of rewrite rules:

- *Rules that execute code from the active process:* for every program statement there is at least one rule.
- *Rules for suspension of the active process:* when an active process guard evaluates to $\mathsf{false}$, the process is suspended, leaving $Pr$ empty.

- *Rules that activate suspended processes:* when *Pr* is empty, suspended processes may be activated. When this happens, the local variable bindings are replaced.
- *Transport rules* move messages into message queues, representing distributed communication.

When auxiliary functions are needed in the semantics, these are defined in equational logic. Equations are evaluated in between the state transitions [49]. Some equations defining normal forms are introduced, including

$$\begin{array}{ll} !m(\textsc{e}) & = !\mathit{self}.m(\textsc{e}) \\ \mathit{await}\ t?(\textsc{v}) & = \mathit{await}\ t?; t?(\textsc{v}) \\ (\varepsilon \parallel \textsc{s}) & = \textsc{s} \\ (\varepsilon \,\Box\, \textsc{s}) & = \textsc{s} \\ (\varepsilon := \varepsilon) & = \varepsilon \end{array}$$

Two new primitives are introduced in the operational semantics to control process termination: *cont(inue)* and *return*. These are not available for the programmer.

*Assignment.* A standard program statement is illustrated by the multiple assignment $\textsc{v} := \textsc{e}$, which binds the value of the expression list $\textsc{e}$ to $\textsc{v}$ within the lists of local and object variables. The rewrite rule for this transition is written as follows, ignoring irrelevant attributes in the style of Full Maude [16]:

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

The auxiliary function *eval* evaluates an expression with a given *state*, i.e., a list of variable to value bindings.

*Object creation.* Next, the creation of new objects is considered. Class parameters are stored among object attributes. A new object with a unique identifier is created. New object identifiers are created by concatenating tokens from the unbounded set *Tok* to the class name. The new object identifier, denoted *C#n*, is returned to the object initiating the object creation.

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

In the new object, self is bound to the new identifier, and class parameter values are stored in the attribute list of the class and instantiated by assignment. After this assignment, *init* is executed and finally, a synchronous call is made to *run* (if present in the class).

*Guard statements.* There are three types of basic guards representing potential processor release points: a Boolean expression guard, a wait guard, and a reply guard. When a guard is encountered, execution continues if the guard is enabled:

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

Enabledness is defined by induction over the construction of guards by the predicate

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

where $\textsc{d}$ denotes a list of state variables, and the function *in* checks whether a completion message corresponding to a given label value is present in the message queue $\textsc{q}$.

When a nonenabled statement is encountered, the active process is suspended on the process queue:

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

where the enabledness predicate is extended to *statements* as follows:

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

The **otherwise** attribute of the last equation states that this equation is taken for all other statements.

The *ready* predicate expresses that a process is ready to execute; i.e., the process is neither waiting for a guard to become true nor for acompletion message. The predicate is defined just as the *enabled* predicate, except that blocking reply statements are *not* ready:

$$\begin{aligned}
&ready(s;\textsc{s},\textsc{d},\textsc{q}) &&= ready(s,\textsc{d},\textsc{q})\\
&ready(t?(\textsc{v}),\textsc{d},\textsc{q}) &&= enabled(await\ t?,\textsc{d},\textsc{q})\\
&ready(\textsc{s}\Box\textsc{s}',\textsc{d},\textsc{q}) &&= ready(\textsc{s},\textsc{d},\textsc{q})\vee ready(\textsc{s}',\textsc{d},\textsc{q})\\
&ready(\textsc{s}|||\textsc{s}',\textsc{d},\textsc{q}) &&= ready(\textsc{s},\textsc{d},\textsc{q})\vee ready(\textsc{s}',\textsc{d},\textsc{q})\\
&ready(s,\textsc{d},\textsc{q}) &&= enabled(s,\textsc{d},\textsc{q}) \quad [\textbf{otherwise}]
\end{aligned}$$

If there is no active process, a suspended process can be reactivated if it is ready:

$$\begin{aligned}
&\langle o : Ob \,|\, Pr : \varepsilon, PrQ : \langle \textsc{s},\textsc{l}\rangle\ \textsc{w}, Lvar : \textsc{l}', Att : \textsc{a}, EvQ : \textsc{q}\rangle\\
&\quad\longrightarrow\\
&\langle o : Ob \,|\, Pr : \textsc{s}, PrQ : \textsc{w}, Lvar : \textsc{l}, Att : \textsc{a}, EvQ : \textsc{q}\rangle\\
&\textbf{if}\ ready(\textsc{s},(\textsc{a};\textsc{l}),\textsc{q})
\end{aligned}$$

This rule allows any ready process to continue because *PrQ* is a multiset. Explicit waiting is resolved by the rule

$$\begin{aligned}
&\langle o : Ob \,|\, Pr : \varepsilon, PrQ : \langle await\ wait \wedge g;\textsc{s},\textsc{l}\rangle\ \textsc{w}\rangle\\
&\quad\longrightarrow\\
&\langle o : Ob \,|\, Pr : \varepsilon, PrQ : \langle await\ g;\textsc{s},\textsc{l}\rangle\ \textsc{w}\rangle
\end{aligned}$$

using ACI matching for conjunctions. Similar rules treat branches of $|||$ and $\Box$ which start with explicit waiting, e.g., a process $\langle(await\ wait \wedge g;\textsc{s}_1)\Box\textsc{s}_2,\textsc{l}\rangle$ in *PrQ* is reduced to $\langle(await\ g;\textsc{s}_1)\Box\textsc{s}_2,\textsc{l}\rangle$.

*Control flow.* Some rules for control flow are now considered. Selection of a branch $\textsc{s}_1$ in a nondeterministic choice statement $\textsc{s}_1\Box\textsc{s}_2$ is modeled by the following rule:

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

Combined with associativity and commutativity of the $\Box$ operator, this rule covers the selection of any branch in a compound nondeterministic choice. When neither $\textsc{s}_1$ nor $\textsc{s}_2$ is ready the active process is blocked if enabled, but suspended if not enabled. Consequently, selecting a branch which immediately blocks or suspends execution is avoided if possible.

The merge operator $|||$ interleaves the execution of two statement lists $\textsc{s}_1$ and $\textsc{s}_2$. A naive approach is to define merge in terms of the nondeterministic choice operator: $\textsc{s}_1;\textsc{s}_2\Box\textsc{s}_2;\textsc{s}_1$. 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. Define an associative, but not commutative, auxiliary operator $/\!/\!/$:

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

Whenever evaluation of the selected (left) branch leads to non-enabledness, execution has arrived at a suspension point and it is safe to pass control back to the $|||$ operator to decide whether to block, select the other branch, or suspend. This is in contrast to the left merge operator of Bergstra and Klop [9], which always returns control to merge after execution of one statement. The $|||$ operator is associative and commutative, and is defined by the following rule:

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

An enabled merge blocks unless this rule is applicable.

*Synchronous and asynchronous method calls.* In the operational semantics, two messages are used to encode a method call. If an object $o_1$ calls a method $m$ of an object $o_2$, with arguments $\textsc{in}$, and the execution results in the return values $\textsc{out}$, the call is reflected by the messages $invoc(o_2,m,o_1\ l\ \textsc{in})$ and $comp(o_1\ l\ \textsc{out})$, representing the invocation and completion of the call, respectively. Objects communicate by asynchronously exchanging these kinds of messages. In the asynchronous setting, the invocation message must include the reply address of the caller, so the completion can be transmitted to the correct destination. As an object may have several pending calls to another object, the completion message includes a locally unique label value $l$, automatically generated by the caller and included in the invocation message.

The Creol semantics handles all invocation mechanisms in a uniform manner using the primitives for asynchronous communication; i.e., asynchronous calls, reply statements, and reply guards. Synchronous and nonblocking calls are reduced by the following equations:

$$\begin{aligned}
&\langle o : Ob \,|\, Pr : p(\textsc{e};\textsc{v});\textsc{s}, Lab : n\rangle\\
&= \langle o : Ob \,|\, Pr :!p(\textsc{e}); n?(\textsc{v});\textsc{s}, Lab : n\rangle
\end{aligned}$$

$$\begin{aligned}
&\langle o : Ob \,|\, Pr : await\ p(\textsc{e};\textsc{v});\textsc{s}, Lab : n\rangle\\
&= \langle o : Ob \,|\, Pr :!p(\textsc{e}); await\ n?(\textsc{v});\textsc{s}, Lab : n\rangle
\end{aligned}$$

When an object calls a method, a message is placed in the configuration. The rewrite rule for this transition is as follows, assigning a value to the label name:

$$\begin{array}{l}\langle o : Ob \mid Pr : t!x.m(\mathrm{E}); \mathrm{S}, Lvar : \mathrm{L}, Att : \mathrm{A}, Lab : n\rangle \\ \quad\longrightarrow \\ \langle o : Ob \mid Pr : t := n; \mathrm{S}, Lvar : \mathrm{L}, Att : \mathrm{A}, Lab : next(n)\rangle \\ \ invoc(eval(x, (\mathrm{A}; \mathrm{L})), m, (o\ n\ eval(\mathrm{E}, (\mathrm{A}; \mathrm{L}))))\end{array}$$

Note that the caller identity and the label value are included as actual parameters. Transport rules take charge of the message, which eventually arrives at the callee's message queue:

$$\begin{array}{l} invoc(\mathrm{o},\ \mathrm{m},\ \mathrm{E})\ \langle o : Ob \mid EvQ : \mathrm{Q}\rangle \\ \quad\longrightarrow \\ \langle o : Ob \mid EvQ : \mathrm{Q}\ invoc(o, m, \mathrm{E})\rangle\end{array}$$

This way, communication between objects requires several rewrite steps, mimicking a distributed environment. A similar transport rule handles completion messages emitted into the configuration upon method execution. Message overtaking is captured by ACI matching messages sent by an object to another object in one order may arrive in any order. (Lossy communication may be explicitly allowed by adding a rule of the form *Configuration Message* ⟶ *Configuration*.)

When an invocation is found in the message queue of an object $o$, the class of $o$, which was not statically known by the caller, is identified. The call is dynamically bound in this class and loaded into the object's internal process queue:

$$\begin{array}{l}\langle o : Ob \mid Cl : C, PrQ : \mathrm{W}, EvQ : \mathrm{Q}\ invoc(o, m, \mathrm{E})\rangle \\ \langle C : Cl \mid Mtds : \mathrm{MT}\rangle \\ \quad\longrightarrow \\ \langle o : Ob \mid Cl : C, PrQ : (\mathrm{W}\ \ bind(m, \mathrm{MT}, \mathrm{E})), EvQ : \mathrm{Q}\rangle \\ \langle C : Cl \mid Mtds : \mathrm{MT}\rangle\end{array}$$

The auxiliary function *bind* fetches method *m* in the method multiset MT 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 E. Note that the code is loaded into $o$ and evaluated in the context of the object's state variables, in which *self* is bound to $o$. The *bind* function ensures that a completion message will be emitted upon method termination by suffixing the code with a statement *return*(V), where V are the formal out-parameters:

$$\begin{array}{l}\langle o : Ob \mid Pr : return(\mathrm{V}); \mathrm{S}, Lvar : \mathrm{L}, Att : \mathrm{A}\rangle \\ \quad\longrightarrow \\ \langle o : Ob \mid Pr : \mathrm{S}, Lvar : \mathrm{L}, Att : \mathrm{A}\rangle \\ comp(eval((caller\ label\ \mathrm{V}), (\mathrm{A}; \mathrm{L})))\end{array}$$

Here *caller* and *label* are the reserved formal parameter names referring to the caller and label values of a call.

The reply statement blocks object activity until the appropriate reply message arrives in the message queue.

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

In the case of a local call, the reply statement allows the call to be loaded in *Pr* by introducing a language primitive *cont*(*n*) as follows:

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

This use of the primitive *cont*(*n*) enforces a LIFO discipline on *PrQ* for local synchronous calls. Similar rules handle branches of $\|\!|$ and $\Box$ starting with a reply of a local call. When evaluation of the new call is completed, the return values are placed in the message queue as usual and the continuation primitive is evaluated:

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

The label value is here sufficient to identify the caller, as reply statements and guards refer to local calls.

## 11 Executable semantics

Specifications in RL are executable on the Maude modeling and analysis tool [16]. This makes RL well-suited for experimenting with programming constructs and language prototypes, combined with Maude's various rewrite strategies and search and model-checking abilities. Thus, development of the language constructs and testing them is done incrementally. In fact, Creol's operational semantics has been used as a language interpreter to test the behavior of Creol programs [41]. The interpreter consists of 700 lines of code, including auxiliary functions and equational specifications, and it has 24 rewrite rules.

Although the proposed operational semantics is highly nondeterministic, Maude rewriting is deterministic in its choice of which rule to apply to a given configuration. For the evaluation of specifications of nondeterministic systems in Maude, as targeted by Creol, this limitation restricts the applicability of the tool as every run of the specification will be identical. However, RL is reflective [15], which allows execution strategies for Maude programs to be written in RL. A strategy based on a pseudo-random number generator is proposed in [41]. Using this strategy, it is easy to test a specification in a series of different runs by providing different seeds to the random number generator.

By executing the operational semantics, Maude may be used as a program analysis tool. Maude's search and model checking facilities can be employed to look for specific configurations or configurations satisfying a given condition. In particular, breadth first search provides a semi-decision procedure for finding failures of safety properties [50].

## 12 Related work

The Creol process statements are inspired by notions from process algebra [32,51]. Process algebra is usually based on synchronous communication. In contrast to the asynchronous $\pi$-calculus [33], 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 fresh return channel [56,57,60]. This provides a unique reference to a call, similar to the values bound to Creol labels at runtime. As shown, label free abstractions may be used in Creol for high level asynchronous method calls.

Integrated formal methods that combine state-based object-oriented structuring languages such as Object-Z and B with process algebras such as CSP and CCS exploit process algebra to express channel communication and synchronization [25,59]. In this vein of work, TCOZ [47] addresses asynchronous communication explicitly through actuators and sensors which represent the local channel ends of asynchronous channels, making global information unnecessary. However, channel-based communication in integrated approaches based on process algebra fixes the communication medium and disallows message overtaking. Finally, Creol's high-level integration of asynchronous and synchronous communication, in which a method may be invoked in both ways, and the organization of pending processes and interleaving at release points within objects seem hard to capture naturally in process algebra and integrated approaches which fix the communication structure.

Object calculi such as the $\varsigma$-calculus [1] and its concurrent extension [29] aim at a direct expression of object-oriented features, supporting, e.g., the return of result values, but asynchronous invocation of methods is not addressed. This also applies to Obliq [11], a programming language based on similar primitives which targets distributed concurrent objects. The concurrent object calculus of [21] 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.

The internal concurrency model of concurrent objects in Creol may be compared to monitors [31] 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. The activation of suspended processes is nondeterministically handled by an unspecified scheduler. Consequently, intra-object concurrency in Creol is similar to the interleaving semantics of concurrent process languages [22,6], 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 [18]. Internal reasoning control is facilitated by the explicit declaration of release points, at which class invariants are expected to hold [23].

Many object-oriented languages offer constructs for concurrency; a survey is given in [55]. A common approach has been to keep activity (threads) and objects distinct, as done in Hybrid [53] and Java [30]. These languages rely on the tightly synchronized RMI model of method calls, forcing the calling method instance to block while waiting for the reply to a call. Verification considerations suggest that methods should be serialized [10], which would block all activity in the calling object. Closely related are method calls based on the rendezvous concept in languages where objects encapsulate activity threads, such as Ada [6] and POOL-T [5].

For distributed systems, with potential delays and even loss of communication, activity threads as well as the tight synchronization of the RMI model seem less desirable. Hybrid offers *delegation* as an explicit construct to (temporarily) branch an activity thread. Clearly, asynchronous method calls may be seen as a form of delegation. Asynchronous method calls can be implemented in, e.g., Java by explicitly creating new threads to handle calls [17]. In Creol, polling for replies to asynchronous calls is handled at the level of the operational semantics: no active loop is 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 [20] than ours. To facilitate the programmer's task and reduce the risk of errors, implicit control structures combined with asynchronous method calls as proposed in Creol seem more attractive, allowing a higher level of abstraction in the language.

Publish/subscribe systems support anonymous and indirect method invocation [24], which complements explicit invocation with a mechanism by which objects subscribing to an event are notified without targeted communication. The approach taken in Sect. 8 follows an approach taken by Notkin et al. [54] for C++. However, asynchronous invocation in Creol directly captures concurrent notification without synchronization.

Languages based on the Actor model [4,3] 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 [63], Argus [45], Concurrent-Smalltalk [62], Eiffel// [12], CJava [17], and in the Join calculus [26] based languages Polyphonic C$^\sharp$ [8] and Join Java [35]. 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.

Maude's inherent object concept [49,16] 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.

## 13 Future work

Creol models as presented in this paper, abstract from any local scheduling policies as well as from any particular network properties. No assumptions are made about the network between objects: it may be channel-based, allow or disallow message loss, etc. In future work, we plan to combine Creol objects with a modeling language expressing properties of network interaction and coordination [7].

The long term goal of our research is to study openness in distributed object systems. This paper has focused on communication aspects in the asynchronous setting and related papers [40,39] on class inheritance, including an approach to dynamic binding and the inheritance anomaly. We believe the language presented here offers interesting possibilities for reasoning about system behavior in the presence of dynamic change. An obvious way to provide some openness is to allow the addition of new (sub)classes and new (sub)interfaces. However, old objects may not use new interfaces that require new methods. A natural way to overcome this limitation is through a dynamic class construct, allowing a class to be *replaced* by a subclass [42]. Thus a class *C* may be modified by adding attributes (with initialization) and methods, redefining methods, as well as extending the inheritance and implements relationships. Unlike standard subclassing, all existing objects of class *C* or a subclass of *C* become renewed in this case and support the new interfaces. The run-time implementation of dynamic class constructs is nontrivial [48], even typing and virtual binding need special considerations. Reasoning control is maintained when the dynamic class construct is restricted to a form of behavioral subtyping [46]. As a special case of class modification, one may posteriorly add super-classes to an established class hierarchy. This answers a major criticism against object-oriented design [27], namely that the class hierarchy severely limits restructuring the system design.

Currently, type systems, and reasoning about inheritance and dynamic classes, are being investigated. More elaborate case studies to test the mechanisms of the language are on the way. The framework provided by rewriting logic and Maude is promising for experimentation with dynamic classes, as the semantic model supports formal reasoning as well as execution and testing.

## 14 Conclusion

Object orientation is the leading framework for distributed systems, but the common approaches to combining concurrency with object-oriented method invocations seem less satisfactory. Communication is either based on synchronous method calls, best suited for tightly coupled processes, or on asynchronous messages, with no direct support for the abstraction and structuring mechanism provided by methods in object-oriented design. Consequently, method calls in the distributed setting become either very inefficient or difficult to program and reason about, requiring explicit low-level synchronization of activity and communication.

In order to facilitate the design of distributed concurrent objects, high-level implicit control structures are needed to organize method invocations and internal object activity. In this paper, we have integrated remote and local, asynchronous and synchronous, method calls with nested processor release points in method bodies for this purpose. The language semantics has been fully formalized in rewriting logic. The approach improves on the efficiency of future variables and allows implicit control of interleaved intra-object concurrency between invoked methods. Active and reactive behavior in an object are thereby easily combined. The proposed interleaving of method executions is more flexible than serialized methods, allowing method overtaking, while maintaining the ease of code verification lost for non-serialized methods. In fact, it suffices that class invariants hold at processor release points.

**Acknowledgements** Marte Arnestad has contributed to the development and implementation of the ideas presented in this paper. The comments of Willem-Paul de Roever, Ivar Alm, and the anonymous referees have resulted in an additional example and several clarifications in the presentation.

## References

1. Abadi, M., Cardelli, L.: A Theory of Objects. Monographs in Computer Science. Springer, Berlin Heidelberg New York 1996
2. Ábrahám-Mumm, E., de Boer F.S., de Roever, W.-P., Steffen, M.: Verification for Java's reentrant multithreading concept. In: International Conference on Foundations of Software Science and Computation Structures (FOSSACS'02). Lecture Notes in Computer Science, vol. 2303, pp. 5–20. Springer, Berlin Heidelberg New York (2002)
3. Agha, G.A.: Abstracting interaction patterns: a programming paradigm for open distributed systems. In: Najm, E., Stefani, J.-B.: (eds.) Proceedings 1st IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'96). Paris pp. 135–153 Chapman & Hall, London 1996
4. Agha, G.A., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. J. Funct. Program. **7**(1), 1–72 (1997)
5. America, P.: POOL-T: A parallel object-oriented language. In: Yonezawa, A., Tokoro, M., (eds.) Object-Oriented Concurrent Programming, pp. 199–220. The MIT Press, Cambridge 1987
6. Andrews, G.R.: Foundations of Multithreaded, Parallel, and Distributed Programming. Addison-Wesley Reading (2000)
7. Arbab, F., Reo: A channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(3), 329–366 (2004)
8. Benton, N., Cardelli, L., Fournet, C.: Modern concurrency abstractions for C$^\sharp$. ACM Trans. Program. Lang. Syst. **26**(5), 769–804 (2004)
9. Bergstra, J., Klop, J.W.: Process algebra for synchronous communication. Inf. Control. **60**(1–3), 109–137 (1984)
10. Brinch Hansen, P.: Java's insecure parallelism. ACM SIGPLAN Notices **34**(4), 38–45 (1999)
11. Cardelli, L.: A language with distributed scope. Comput. Syst. **8**(1), 27–59 (1995)
12. Caromel, D., Roudier, Y.: Reactive programming in Eiffel//. In: Briot, J.-P., Geib, J.M. Yonezawa, A. (eds.) Proceedings of the Conference on Object-Based Parallel and Distributed Computation. Lecture Notes in Computer Science, vol. 1107, pp. 125–147. Springer, Berlin Heidelberg New York 1996
13. Carriero, N., Gelernter, D.: Linda in context. Commun. ACM **32** (4), 444–458 (1989)
14. Cenciarelli, P., Knapp, A., Reus, B., Wirsing, M.: An event-based structural operational semantics of multi-threaded Java. In: Alves-Foss, J. (ed.) Formal Syntax and Semantics of Java. Lecture Notes in Computer Science, vol. 1523, pp. 157–200. Springer, Berlin Heidelberg New York 1999
15. Clavel, M.: Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications. CSLI Publications, Stanford 2000
16. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. Theor. Comput. Sci. **285**, 187–243 2002
17. Cugola, G., Ghezzi, C.: CJava: introducing concurrent objects in Java. In: Orlowska, M.E., Zicari, R. (eds.) 4th International Conference on Object Oriented Information Systems (OOIS'97), London, pp. 504–514. Springer, Berlin Heidelberg New York (1997)
18. Dahl, O.-J.: Monitors revisited. In: Roscoe, A.W. (ed.) A Classical Mind, Essays in Honour of C.A.R. Hoare, pp. 93–103. Prentice-Hall, Englewood Cliffs (1994)
19. Dahl, O.-J., Myrhaug, B., Nygaard, K.: (Simula 67) Common base language. Technical Report S-2. Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway (1968)
20. Damm, W., Josko, B., Pnueli, A., Votintseva, A.: Understanding UML: a formal semantics of concurrency and communication in real-time UML. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) First International Symposium on Formal Methods for Components and Objects (FMCO 2002), Revised Lectures. Lecture Notes in Computer Science, vol. 2852, pp. 71–98. Springer, Berlin Heidelberg New York (2003)
21. Di Blasio, P., Fischer, K.: a calculus for concurrent objects. In: Montanari, U., Sassone, V. (eds.) 7th International Conference on Concurrency Theory (CONCUR'96). Lecture Notes in Computer Science, vol. 1119, pp. 655–670. Springer, Berlin Heidelberg New York (1996)
22. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
23. Dovland, J., Johnsen, E.B., Owe, O.: Verification of concurrent objects with asynchronous method calls. In: Proceedings of the IEEE International Conference on Software Science, Technology & Engineering (SwSTE'05), pp. 141–150. IEEE Computer Society Press, Los Alamitos (2005)
24. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.-M.: The many faces of publish/subscribe. ACM Comput. Surv. **35**(2), 114–131 (2003)
25. Fischer, C.: CSP-OZ: a combination of Object-Z and CSP. In: Bowman, H., Derrick, J. (eds.) Proceedings of the 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS), pp. 423–438. Chapman & Hall, London (1997)

26. Fournet, C., Gonthier, G.: The reflexive chemical abstract machine and the join-calculus. In: ACM Symposium on Principles of Programming Languages (POPL), pp. 372–385 (1996)
27. Ghezzi, C., Jazayeri, M.: Programming Language Concepts, 3rd edn. Wiley, (1998)
28. Goguen, J.A., Winkler, T., Meseguer, J., Futatsugi, K., Jouannaud, J.-P.: Introducing OBJ. In: Goguen, J.A., Malcolm, G. (eds.) Software Engineering with OBJ: Algebraic Specification in Action, chapter 1, pp. 3–167. Kluwer Dordrecht (2000)
29. Gordon A.D., Hankin, P.D.: A concurrent object calculus: reduction and typing. In: Nestmann U., Pierce, B.C. (eds) HLCL '98: High-Level Concurrent Languages, Nice, France, September 12, 1998. Electronic Notes in Theoretical Computer Science, vol. 16(3). Elsevier, Amsterdam (1998)
30. Gosling, J., Joy, B., Steele, G.L., Bracha, G.: The Java language specification, 2nd edn. Java series. Addison-Wesley, Reading (2000)
31. Hoare, C.A.R.: Monitors: an operating systems structuring concept. Commun. ACM, **17**(10), 549–557 (1974)
32. Hoare, C.A.R.: Communicating Sequential Processes. International Series in Computer Science. Prentice-Hall, Englewood Cliffs (1985)
33. Honda, K., Tokoro, M.: An object calculus for asynchronous communication. In: America, P. (ed.) Proceedings of the European Conference on Object-Oriented Programming (ECOOP'91). Lecture Notes in Computer Science, vol. 512, pp. 133–147. Springer, Berlin Heidelberg New York (1991)
34. International Telecommunication Union: Open distributed processing – reference model parts 1–4. Technical report. ISO/IEC, Geneva (1995)
35. Itzstein, G.S., Jasiunas, M.: On implementing high level concurrency in Java. In: Omondi, A., Sedukhin, S. (eds.) Proceeding of the 8th Asia-Pacific Computer Systems Architecture Conference (ACSAC 2003). Lecture Notes in Computer Science, vol. 2823, pp. 151–165. Springer, Berlin Heidelberg New York (2003)
36. Johnsen, E.B., Owe, O.: A compositional formalism for object viewpoints. In: Jacobs, B., Rensink, A. (eds.) Proceedings of the 5th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'02), pp. 45–60. Kluwer Dordrecht, (2002)
37. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. In; Proceedings of the 2nd IEEE International Conference on Software Engineering and Formal Methods (SEFM'04), pp. 188–197. IEEE Computer Society Press, Los Alamitos (2004)
38. Johnsen, E.B., Owe, O.: Object-oriented specification and open distributed systems. In: Owe, O., Krogdahl, S., Lyche, T. (eds.) From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl. Lecture Notes in Computer Science, vol. 2635, pp. 137–164. Springer, Berlin Heidelberg New York (2004)
39. Johnsen, E.B., Owe, O.: A dynamic binding strategy for multiple inheritance and asynchronously communicating objects. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) Proceedings of the 3rd International Symposium on Formal Methods for Components and Objects (FMCO 2004). Lecture Notes in Computer Science, vol. 3657, pp. 274–295. Springer, Berlin Heidelberg New York (2005)
40. Johnsen, E.B., Owe, O.: Inheritance in the presence of asynchronous method calls. In: Proceedings of the 38th Hawaii International Conference on System Sciences (HICSS'05). IEEE Computer Society Press, Los Alamites (2005)
41. Johnsen, E.B., Owe, O., Axelsen, E.W.: A run-time environment for concurrent objects with asynchronous method calls. In: N. Martí-Oliet, (ed.) Proceedings of the 5th International Workshop on Rewriting Logic and its Applications (WRLA'04), March 2004. Electronic Notes in Theoretical Computer Science, vol. 117, pp. 375–392. Elsevier, Amsterdam (2005)
42. Johnsen, E.B., Owe, O., Simplot-Ryl, I.: A dynamic class construct for asynchronous concurrent objects. In: Steffen, M., Zavattaro, G. (eds.) Proceedings of the 7th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05). Lecture Notes in Computer Science, vol. 3535, pp. 15–30. Springer, Berlin Heidelberg New York (2005)
43. Johnsen, E.B., Owe, O., Yu, I.C.: Creol: a type-safe object-oriented model for distributed concurrent systems. Research Report 327. Department of Informatics, University of Oslo, Norway (2005)
44. Jones, C.B.: Development methods for computer programmes including a notion of interference. PhD thesis, Oxford University (1981)
45. Liskov, B.H., Shrira, L.: Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In: Wise, D.S. (ed.) Proceedings of the SIGPLAN Conference on Programming Lanugage Design and Implementation (PLDI'88). Atlanta, GE, USA, June 1988, pp. 260–267. ACM Press, New York (1988)
46. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. on Program. Lang. and Syst. **16**(6), 1811–1841 (1994)
47. Mahony, B.P., Dong, J.S.: Sensors and actuators in TCOZ. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) World Congress on Formal Methods (FM'99), Proceedings, Volume II. Lecture Notes in Computer Science, vol. 1709, pp. 1166–1185. Springer, Berlin Heidelberg New York (1999)
48. Malabarba, S., Pandey, R., Gragg, J., Barr, E., Barnes, J.F.: Runtime support for type-safe dynamic Java classes. In: Bertino, E. (ed.) 14th European Conference on Object-Oriented Programming (ECOOP 2000). Lecture Notes in Computer Science, vol. 1850, pp. 337–361. Springer, Berlin Heidelberg New York (2000)
49. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. **96**, 73–155 (1992)
50. Meseguer, J., Rosu, G.: Rewriting logic semantics: from language specifications to formal analysis tools. In: Basin, D.A., Rusinowitch, M. (eds.) Automated Reasoning – Proceeding of the Second International Joint Conference (IJCAR 2004). Lecture Notes in Computer Science, vol. 3097, pp. 1–44. Springer, Berlin Heidelberg New York (2004)
51. Milner, R.: Communicating and Mobile Systems: The $\pi$-Calculus. Cambridge University Press, Cambridge (1999)
52. Najm, E., Stefani, J.-B.: A formal semantics for the ODP computational model. Comput. Netw. ISDN Syst. **27**, 1305–1329 (1995)
53. Nierstrasz, O.: A tour of Hybrid – a language for programming with active objects. In: Mandrioli, D., Meyer, B. (eds.) Advances in Object-Oriented Software Engineering, pp. 167–182. Prentice-Hall, Englewood Cliffs (1992)
54. Notkin, D., Garlan, D., Griswold, W.G., Sullivan, K.: Adding implicit invocation to languages: three approaches. In: Nishio, S., Yonezawa, A. (eds.) Proceedings of the 1st JSSST International Symposium on Object Technologies for Advanced Software. Lecture Notes in Computer Science, vol. 742, pp. 489–510. Springer, Berlin Heidelberg New York (1993)
55. Philippsen, M.: A survey on concurrent object-oriented languages. Concurrency: Pract. Exp. **12**(10), 917–980 (2000)
56. Pierce, B.C., Turner, D.N.: Pict: a programming language based on the pi-calculus. In: Plotkin, G., Stirling, C., Tofte, M.

(eds.) Proof, Language and Interaction: Essays in Honour of Robin Milner. The MIT Press, Cambridge (1998)
57. Sangiorgi, D., Walker, D.: The $\pi$-Calculus: A Theory of Mobile Processes. Cambridge University Press, Cambridge (2001)
58. Shaw, M.: Procedure calls are the assembly language of software interconnection: connectors deserve first-class status. In: Lamb, D.A. (ed.) Workshop on Studies of Software Design. Lecture Notes in Computer Science, vol. 1078, pp. 17–32. Springer, Berlin Heidelberg New York (1994)
59. Smith, G., Derrick, J.: Specification, refinement and verification of concurrent systems – an integration of Object-Z and CSP. Formal Methods Syst. Des. **18**(3), 249–284 (2001)
60. Vasconcelos, V.T.: Typed concurrent objects. In: Tokoro, M., Pareschi, R. (eds.) Proceedings of the 8th European Conference on Object-Oriented Programming (ECOOP'94). Lecture Notes in Computer Science, vol. 821, pp. 100–117. Springer, Berlin Heidelberg New York (1994)
61. Vinoski, S.: CORBA: integrating diverse applications within distributed heterogeneous environments. IEEE Commun. Mag. **14**(2), (1997)
62. Yokote, Y., Tokoro, M.: Concurrent programming in ConcurrentSmalltalk. In: Yonezawa, A., Tokoro, M. (eds.) Object-Oriented Concurrent Programming, pp. 129–158. The MIT Press, Cambridge (1987)
63. Yonezawa, A.: ABCL: An Object-Oriented Concurrent System. Series in Computer Systems. The MIT Press, Cambridge (1990)

## Author Biography

**Einar Broch Johnsen** is an associate professor at the Department of Informatics, University of Oslo and has previously worked at the University of Bremen. His doctoral thesis (University of Oslo, 2002) addressed the object-oriented specification of open distributed systems. His current research interests are in program specification and modelling, and related formal systems.

**Olaf Owe** (Dr. Scient. from University of Oslo) has been active for more than 25 years in research on specification and programming languages and formal methods. He has been a full professor at University of Oslo since 1993 and has earlier worked two years at Stanford (USA) and two years at UCSD (USA) as an assistant professor, and part time at the Norwegian Computing Centre. In the recent Adapt-FT and Creol projects, led by Owe, he has focused on open, distributed, and object oriented systems.