An Asynchronous Communication Model for Distributed Concurrent Objects

Einar Broch Johnsen and Olaf Owe
Department of informatics, University of Oslo
PO Box 1080 Blindern, N-0316 Oslo, Norway
{einarj, olaf}@ifi.uio.no

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 seem less satisfactory; 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 a 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 an example of a peer-to-peer network.

1. Introduction

The importance of inter-process communication is rapidly increasing with the development of distributed computing, both over the Internet and over local networks. Object orientation appears as a promising framework for concurrent and distributed systems, and has been recommended by the RM-ODP [19], but object interaction by means of 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 with loosely coupled components. Here synchronous communication gives rise to undesired and uncontrolled waiting, and possibly deadlock. Asynchronous message passing gives better control and efficiency, but lacks the structure and discipline inherent in method calls. The integration of the message concept in the object-oriented setting is unsettled, especially with respect to inheritance and redefinition.

Intuitive high-level programming constructs are needed to unite object orientation and distribution in a natural way. In this paper programming constructs for concurrent objects are discussed, based on *processor release points* and a notion of *asynchronous method calls*. Processor release points are used to 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). These concepts are explored in a small object-oriented language Creol. The operational semantics of the language is defined in rewriting logic [26] and is executable as an interpreter in the tool Maude [11]. Our experiments suggest that rewriting logic and Maude provide a well-suited platform for experimentation with language constructs and concurrent environments. The suitability of the proposed language constructs 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 three basic interaction models for concurrent processes are shared variables, remote method calls, and message passing [5]. As objects encapsulate local states, we find inter-object communication most naturally modeled by (remote) method calls, avoiding shared variables. With the *remote method invocation* (RMI) model, an object is activated by a method call. Control is transferred with the call so there is a master-slave relationship between the caller and the callee. A similar approach is taken with the execution threads of e.g. Hybrid [29] and Java [18], where concurrency is achieved through multithreading. The interference problem related to shared variables reemerges when threads operate concurrently in the same object, which happens with non-serialized methods in Java. Reasoning about programs in this setting is a highly complex matter [1, 9]: Safety is by convention rather than by language design [7]. Verification considerations therefore suggest that all methods should be serialized as done in e.g. Hybrid. However, when restricting to serialized methods, the calling object must *wait* for the return of a call, blocking for any other activity in the object. In a distributed setting this limitation is severe; delays and instabilities may cause much unnecessary waiting. A nonterminating method will even block other method invocations, which makes it difficult to combine active and passive behavior in the same object.

In this paper, method calls are taken as the communication primitive for concurrent objects in the Creol language, and given an operational semantics reflected 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. In order to model real world systems in an object-oriented manner, asynchronously communicating concurrent objects appear as a much more natural approach.

The paper is organized as follows. Section 2 outlines the overall setting of the approach. Section 3 introduces the Creol language, including asynchronous method calls and processor release points. Section 4 gives an example of a peer-to-peer network in Creol. Section 5 illustrates how explicit synchronization may be obtained when needed. Section 6 defines the operational semantics of Creol by means of rewriting logic. Section 7 considers related work and Section 8 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 the implementation details of their design. Therefore, reasoning should be done relying on abstract specifications of the system's components. In this setting we find specification in terms of observable behavior particularly attractive, assuming that components come equipped with behavioral interfaces that instruct us on how to use them. Furthermore, a component may come equipped with multiple specifications, corresponding to different viewpoints.

Strong typing. We consider typing where two kinds of variables are declared; an object variable is typed by an interface and an ordinary variable is typed by a data type. Strong typing ensures that for each method invocation $o.m(inputs; outputs)$, where I is the declared interface of o, the actual object o (if not nil) will support I and the method m will be understood. 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.

In order to solve the conflict between unrestricted code reuse in subclasses, and behavioral subtyping and incremental reasoning control [24, 31], we use behavioral interfaces to type object variables, and allow multiple inheritance at both the interface and class level. Interface inheritance is restricted to a form of behavioral subtyping [24], whereas class inheritance may be used freely. Inherited class (re)declarations are resolved by disjoint union combined with an ordering of the super classes. A class may implement several interfaces, provided that it satisfies the syntactic and semantic requirements stated in the interfaces. An object of class C supports an interface I if the class 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 subinterface of I.* Subclassing is unrestricted, so method redefinition may violate semantic requirements of an interface. Therefore, implementation claims (as well as class invariants) are not in general inherited at the class level.

Observable behavior. An object's observable communication history, i.e. the trace of all communication events between the object and its environment, represents an abstract view of its state, readily available for reasoning about past and present behavior. By means of a local history variable, the behavior of an object is determined by its communication history up to present time. The approach emphasizes mathematically intuitive concepts such as generator inductive function definitions and finite sequences, avoiding fix-point semantics and infinite traces [21].

An interface may specify observable behavior in the form of an assumption guarantee specification [23] on the local history. The assumption is a requirement on the behavior of the 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 communicating systems. The semantic requirements of an interface rely on the present communication history of an object offering the interface, i.e. they are predicates on the finite traces. A compositional formalism for reasoning about behavioral interfaces, and an associated interface refinement relation, is given in previous work [20, 21]. At the imperative level, reasoning about class invariants in terms of class attributes and the local history can be done locally in each class.

3. The Creol language

This section proposes programming constructs for distributed concurrent objects, based on asynchronous method calls and processor release points. Concurrent objects are potentially active, encapsulating execution threads; consequently, elements of basic data types are not considered objects. In this sense, our objects resemble top-level objects in e.g. Hybrid. The objects considered have identity: communication takes place between named objects and object identifiers may be exchanged between objects. As motivated above, Creol objects are typed by interfaces, resembling CORBA's IDL, but extended with semantic requirements and mechanisms for type control in dynamically reconfigurable systems. As the language supports strong typing, invoked methods are supported by the called object (when not null), and formal and actual parameters match.

3.1. Interfaces

Interfaces describe viewpoints to objects and have the following general form:

```
interface F (⟨parameters⟩)
  inherits F_1, F_2, ..., F_m
begin with G
    op m_1(...)
       ⋮
    op m_n(...)
  asm <formula on local observable trace
          restricted to any calling object>
  inv <formula on local observable trace>
where <auxiliary function definitions>
end
```

where $F, F_1, \ldots, F_m$, and G are interfaces. Interfaces may have both value and object parameters, typed respectively by data types and interfaces. Interface parameters describe the minimal environment that any object offering the interface needs at the point of creation. Inheritance gives rise to concatenation of parameter lists.

For active objects we may want to restrict access to calling objects of a particular interface. This way, the active object can invoke methods of the caller and not only passively complete invocations of its own methods. Use of the **with** clause restricts the communication environment of an object, as considered through the interface, to external objects offering a given *cointerface* [20,21]. For some objects no such knowledge is required, which is captured by the keyword *Any* in the **with** clause. Mutual dependency is specified if two interfaces have each other as cointerface.

Example. We consider the interfaces of a node in a peer-to-peer file sharing network. A *Client* interface captures the client end of the node, available to any user of the system. It offers methods to list all files available in the network, and to request the download of a given file from a given server. A *Server* interface offers a method for obtaining a list of files available from the node, and a mechanism for downloading packs, i.e. parts of a target file. The Server interface is available to other servers in the network.

```
interface Client     interface Server
begin with Any       begin with Server
  op availFiles        op enquire
  op reqFile           op getLength
end                    op getPack
                     end
```

The **with**-construct allows the typing mechanism to deduce that any caller of a server request will understand the *enquire* and *getPack* methods. To save space, discussion of method parameters is postponed to Section 4. The two interfaces may be inherited by a third interface *Peer*

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

which describes nodes able to act according to both the client role and the server role.

3.2. Classes and objects

At the imperative level, attributes (object variables) and method declarations are organized in classes, which may have value and object parameters similar to interface parameters. Objects are dynamically created instances of classes. 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 *init* method is invoked at object creation to instantiate attributes and may not contain processor release points. After initialization, the *run* method, if provided, is started. 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* reflects active behavior. Methods need not terminate and, apart from *init*, all method instances may be temporarily *suspended*.

In order to focus the discussion on asynchronous method calls and processor release points in method bodies, other language aspects will not be 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. We denote by Var the set of program variables, by Mtd the set of method names, and by Label the set of method call identifiers. There is read-only access to labels, in-parameters, and the special object attribute *this*, which is used for self reference.

3.3. 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 interleaving 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. 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, an instantiated method is represented by a *process* $\langle S,L \rangle$ where S is a sequence of commands and $L : \mathsf{Var} \rightarrow \mathsf{Data}$ the local variable bindings. Consider an object o which offers the method

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

to the environment. Syntactically, method declarations end with a period. Accepting a call to m with argument 2 from another object o' creates a process $\langle S, \{caller \mapsto o', label \mapsto l, x \mapsto 2, y \mapsto nil, z \mapsto 0\}\rangle$ in the object o. An object can have several (suspended) instances of the same method, possibly with different values for local variables. The local variables *label* and *caller* are reserved to identify the call, allowing the proper reply to be emitted at method termination, i.e. when S is completed.

An asynchronous method call is made with the command $l!o.m(e)$, where $l \in \mathsf{Label}$ provides a locally unique 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 *this* object, otherwise *remote*. Labels are used to identify replies, and may be omitted if a reply is not explicitly requested. As no synchronization is involved, process execution can proceed after calling an external method until the return value is actually needed by the process.

To fetch the return values from the call, say in a variable list x, we ask for the reply to our call: $l?(x)$. This command treats x as a future variable [32]. If the reply to the call has not arrived when requested, process execution is blocked. In order to avoid blocking in the asynchronous case, processor release points are introduced for reply requests (Section 3.4): If the reply has arrived, return values may be assigned to x and execution continue without delay. Otherwise, execution is *suspended*.

Synchronous (RMI) method calls use the syntax $o.m(e;x)$, which is defined by $l!o.m(e);\ l?(x)$ for some fresh label l, immediately blocking the processor while waiting for the reply. In order to execute local calls, the calling method must eventually suspend its own execution. Therefore the reply command $l?(x)$ will enable execution of the call identified by l when this call is local. The language does not support monitor reentrance, mutual synchronous calls may therefore lead to deadlock.

Syntactic categories.	*Definitions.*
l in Label	
g in Guard	$g ::= wait \mid \phi \mid l? \mid g_1 \wedge g_2 \mid g_1 \vee g_2$
p in MtdCall	$p ::= o.m \mid m$
S in ComList	$S ::= C \mid C;S$
C in Com	$C ::= \textbf{skip} \mid (S) \mid S_1 \Box S_2 \mid S_1 \vert\vert\vert S_2$
x in VarList	$\mid x := e \mid x := \textbf{new}\ classname(e)$
e in ExprList	$\mid \textbf{if}\ \phi\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}$
m in Mtd	$\mid \textbf{while}\ \phi\ \textbf{do}\ S\ \textbf{od}$
o in ObjExpr	$\mid\ !p(e) \mid l!p(e) \mid l?(x) \mid p(e;x)$
ϕ in BoolExpr	$\mid \textbf{await}\ g \mid \textbf{await}\ l?(x) \mid \textbf{await}\ p(e;x)$

Figure 1. An outline of the language syntax.

3.4. A language with processor release points

Guarded commands g are used to explicitly declare potential release points **await** g for the object's processor. Guarded commands can be nested within the same local variable scope, corresponding to a series of processor release points. When an inner guard which evaluates to false is encountered during process execution, the process is suspended and the processor released. After processor release, any suspended process may be selected for execution.

The type Guard is constructed inductively:

- $wait \in \mathsf{Guard}$ (explicit release)
- $l? \in \mathsf{Guard}$, where $l \in \mathsf{Label}$
- $\phi \in \mathsf{Guard}$, where ϕ is a boolean expression over local and object variables

Here, *wait* is a construct for explicit release of the processor. The reply guard $l?$ succeeds if the reply to the method invocation with label l has arrived. We allow composition of guards: $g_1 \wedge g_2$ and $g_1 \vee g_2$ for guards g_1 and g_2. Evaluation of guards is done atomically. We define **await** $l?(x)$ as an abbreviation for **await** $l?; l?(x)$, and **await** $m(e;x)$ for $l!o.m(e)$; **await** $l?(x)$, a typical form of asynchronous communication.

Guarded commands can be *composed* in different ways, reflecting the requirements to the internal control flow in the objects. Let GS_1 and GS_2 denote the guarded commands **await** $g_1;S_1$ and **await** $g_2;S_2$. Nesting of guards is obtained by sequential composition; in a program statement $GS_1;GS_2$, the guard g_2 corresponds to a potential inner processor release point. Non-deterministic choice between guarded commands is expressed by $GS_1 \Box GS_2$, which may compute S_1 if g_1 evaluates to true or S_2 if g_2 evaluates to true. Non-deterministic merge is expressed by $GS_1 ||| GS_2$, defined as $(GS_1;GS_2)\Box(GS_2;GS_1)$. Control flow without potential processor release uses **if** and **while** constructs, and assignment to local and object variables is expressed as $x := e$ for (the same number of) program variables x and expressions e. Figure 1 summarizes the language syntax.

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 functions across a distributed network. Servers arrive and disappear dynamically. A client requests a file from a server in the network, and downloads it as a series of packet downloads until the file download is complete. The connection to the server may be blocked, in which case the download will automatically resume if the connection is reestablished. A client may run several downloads concurrently, at different speeds. We assume that every node in the network has an associated database with shared files. Downloaded files are stored in this database, which is not modeled here but implements the interface *DB*:

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

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

Nodes in the peer-to-peer network which implement the *Peer* interface can be modeled by a class *Node*. *Node* objects can have several interleaved activities: several downloads may be processed simultaneously as well as uploads to other servers, etc. All method calls are asynchronous: If a server temporarily becomes unavailable, the transaction is suspended and may resume at any time after the server becomes available again. Processor release points ensure that the processor is not blocked and transactions with other servers not affected.

```
class Node (db:DB)
  implements Peer
begin
with Server
  op enquire(out files: List[Str]) == await db.listFiles(;files) .
  op getLength(in fId:Str out lth:Nat) ==
    await db.getLength(fId;lth) .
  op getPack(in fId:Str, pNbr:Nat out pack:List[Data]) ==
    var f:List[Data]; await db.getFile(fId;f); pack:=f[pNbr] .

with Client
  op availFiles (in sList: List[Str] out files:List[Str×Str]) ==
    var s: Str, fList: List[Str];
    if (sList = empty) then files:= empty
    else await hd(sList).enquire(;fList);
      await this.availFiles(tl(sList);files);
      files:=(files; (hd(sList),fList)) fi .
  op reqFile(in sId:Str, 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:=(file; pack); lth:=lth - 1 od; !db.storeFile(sId, file) .
end
```

Here, *availFiles* returns a list of pairs where each pair contains a file identifier *fId* and the server identifier *sId* where *fId* may be found, *reqFile* the file associated with *fId*, *enquire* the list of files available from the server, and *getPack* a particular pack in the transmission of a file. The list constructor is represented by semicolon. For $x : T$ and $s : \mathrm{List}[T]$, we let $hd(x;s) = x$ and $tl(x;s)) = s$, and $s[i]$ denotes the i'th element of s, provided $i \leq length(s)$.

5. Example: explicit synchronization

Creol supports high-level implicit synchronization of interleaved method executions in concurrent objects. Consequently, objects may be regarded as abstract monitors, without the need for explicit signaling. Explicit signaling is often not necessary and adds an additional level of complexity for reasoning about monitors [13]. 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 [5], 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: Nat, d: Nat, q: Nat;
  op init == s:=0; d:=0; q:=0 .
with Any
  op wait == var myturn: Nat; d:=d+1; myturn:=d;
                 await (s>0 ∧ q=myturn); s:=s-1; q:=q+1 .
  op signal == if (d-q >0) then s:=s+1 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**.

6. An operational semantics for Creol

The operational semantics of Creol is defined using rewriting logic [26], emphasizing simplicity and abstraction while modeling the essential aspects of concurrency, distribution, and communication. A rewrite theory is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$. The signature Σ defines the function symbols of the language, E defines equations between terms, L is a set of labels, and R is a set of labeled rewrite rules. From a computational viewpoint, a rewrite rule $t \longrightarrow t'$ may be interpreted as a *local transition rule* allowing an instance of the pattern t to evolve into the corresponding instance of the pattern t'. Rewrite rules apply to fragments of a state configuration. If rewrite rules may be applied to non-overlapping fragments, the transitions may be performed in parallel. Consequently, rewriting logic (RL) is a logic which easily captures concurrent change. A number of concurrency models have been successfully represented in RL [11, 26], including Petri nets, CCS, Actors, and Unity, as well as the ODP computational model [28]. RL also offers its own model of object orientation [11].

Informally, a state configuration in RL is a multiset of terms of given types. These types are specified in (membership) equational logic (Σ, E), the functional sublanguage of RL which supports algebraic specification in the OBJ [17] 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. Each rule describes how a part of a configuration can evolve in one transition step. 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 RL.

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:

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

Rules in RL may be formulated at a high level of abstraction, closely resembling a compositional operational semantics. In fact, structural operational semantics can be uniformly mapped into RL specifications [15].

6.1. System configurations

In the semantics, an asynchronous method call will be reflected by a pair of messages and 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. In order to increase parallelism in the model, message queues will be external to object bodies. A state configuration is a multiset combining Creol objects, classes, messages, and queues. As usual in RL, the associative and commutative constructor for multisets is represented by whitespace.

In RL, objects are commonly represented by terms of the type $\langle O : C \,|\, a_1 : v_1, \ldots, a_n : v_n \rangle$ 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 [11]. We adopt this form of presentation and define Creol objects, classes, and external message queues as RL objects. Omitting RL types, a Creol object is represented by an RL object $\langle Ob \,|\, Cl, Pr, PrQ, Lvar, Att, Lab \rangle$, 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, and $Lvar$ and Att the local and object variables, respectively. Let τ be a type partially ordered by $<$, with least element 1, and let $Next : \tau \rightarrow \tau$ be such that $\forall x . x < Next(x)$. Lab is the method call identifier corresponding to labels in the language, of type τ. Thus, the object identifier and the generated label value provide a globally unique identifier for each method call. Message queues are RL objects $\langle Qu \,|\, Ev \rangle$, where Qu is the queue identifier and Ev a multiset of unprocessed messages. Each message queue is a distinct term in the state configuration, associated with one specific Creol object.

Creol classes are represented by RL objects $\langle Cl \mid Param, Att, Tok, init, Mtds \rangle$, where Cl is the class name, $Param$ the list of class parameters, Att a list of attributes, $init$ the initialization method, Tok is an unbounded set of tokens of some sort, and $Mtds$ a multiset of methods. When an object needs a method, it is loaded from the $Mtds$ multiset of its class (overloading and virtual binding issues connected to inheritance are ignored here.)

In RL's object model [11], 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. The command *new C(args)* creates a new object with a unique object identifier, object variables as listed in the class parameter list and in *Att*, and places the code from methods *init* and *run* in *Pr*.

6.2. Concurrent transitions

Concurrent change is achieved in the operational semantics by applying concurrent rewrite steps to state configurations. 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. For example, the assignment rule for the program $x := e$ binds the value of the expression list e to x within the lists of local and object variables.
- *Rules for suspension of the active process:* When an active process guard evaluates to false, the process and its local variables are suspended, leaving *Pr* empty.
- *Rules that activate suspended processes:* When *Pr* is empty, suspended processes may be activated. When this happens, the local variable bindings are replaced.
- *Transport rules:* These rules move messages into and out of the external message queue. Because the external message queue is represented as a separate RL object, it can belong to another subconfiguration than the object itself and it can therefore receive messages in parallel with other activity in the object.

When auxiliary functions are needed in the semantics, these are defined in equational logic, and are evaluated in between the state transitions [26]. The rules for the basic constructs concerning method calls, replies, guarded commands, and creation of new objects, are now considered in more detail.

Synchronous and Asynchronous Method Calls. In the operational semantics, objects communicate by sending messages. 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 *in*, and the execution results in the return values *out*, the call is reflected by two messages $invoc(o_2, m, o_1\ l\ in)$ and $comp(o_1, l, out)$, which represent the invocation and completion of the call, respectively. 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 l, generated by the caller.

The Creol semantics handles all invocation mechanisms by means of the primitives for asynchronous communication, i.e. asynchronous calls, reply commands, and reply guards, and represents the synchronous call $o.m(in; out)$ as $l!o.m(in); l?(out)$ for some fresh label identifier l.

When an object calls an external method, a message is placed in the configuration. The rewrite rule for this transition can be expressed as follows, ignoring irrelevant attributes in the style of Full Maude [11]:

$$\begin{array}{l}\langle O : Ob | Pr : (Q!X.Y(I); P), Lvar : L, Att : A, Lab : N\rangle \\ \quad\longrightarrow \\ \langle O : Ob | Pr : (Q := N; P), Lvar : L, Att : A, Lab : Next(N)\rangle \\ invoc(eval(X, (A; L)), Y, (O\ N\ eval(I, (A; L))))\end{array}$$

Here, *eval* is a function which evaluates an expression list in the context of a list of variable bindings. 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 external message queue. After method execution, a completion message is emitted into the configuration, eventually arriving at the caller's external message queue.

If an invocation is found in the external message queue of an object, the corresponding method code can be loaded from the object's class into its internal process queue.

$$\begin{array}{l}\langle O : Ob | Cl : C, PrQ : W\rangle\ \langle O : Qu | Ev : Q\ invoc(O, M, I)\rangle \\ \langle C : Cl | Mtds : MT\rangle \\ \quad\longrightarrow \\ \langle O : Ob | Cl : C, PrQ : (W\ bind(M, MT, I))\rangle\ \langle O : Qu | Ev : Q\rangle \\ \langle C : Cl | Mtds : MT\rangle\end{array}$$

The auxiliary function *bind* fetches method M in the method multiset MT of the class, and returns the method's code and local variables. It ensures that a completion message will be emitted upon method termination.

In case of a remote call, the reply command blocks until the appropriate reply message has arrived in the external message queue.

$$\begin{array}{l}\langle O : Ob | Pr : (X?(J); P), Lvar : L\rangle\ \langle O : Qu | Ev : Q\ comp(O, N, K)\rangle \\ \quad\longrightarrow \\ \langle O : Ob | Pr : (J := K; P), Lvar : L\rangle\ \langle O : Qu | Ev : Q\rangle \\ \textbf{\textit{if}}\ N = eval(X, L)\end{array}$$

In case of a local call, the reply command allows the call to be loaded in *Pr*:

$$\begin{array}{l}\langle O : Ob | Pr : (X?(J); P), PrQ : (P', L')\ W, Lvar : L\rangle \\ \quad\longrightarrow \\ \langle O : Ob | Pr : P'; cont(label), PrQ : (\text{await}\ X?(J); P, L)\ W, Lvar : L'\rangle \\ \textbf{\textit{if}}\ eval(caller, L') = O \wedge eval(label, L') = eval(X, L)\end{array}$$

Note that the language primitive $cont(label)$ is appended to the method code, thereby causing a LIFO discipline on *PrQ* for local synchronous calls. When evaluation of the new call is completed, the return values are placed in the external event queue as usual and the continuation primitive is evaluated:

$$\begin{array}{l}\langle O : Ob | Pr : cont(X), PrQ : \langle (X'?; P), L'\rangle\ W, Lvar : L\rangle \\ \quad\longrightarrow \\ \langle O : Ob | Pr : P, PrQ : W, Lvar : L'\rangle \\ \textbf{if}\ eval(X, L) = eval(X', L')\end{array}$$

Guarded Commands. There are three types of guards representing potential processor release points: The regular boolean expression, a wait guard, and a return guard. Here we will look closer at rules for evaluation of return guards in the active process.

Return guards allow process suspension when waiting for method completions, so the object may attend to other tasks while waiting. A return guard evaluates to true if the external message queue contains the completion of the method call, and execution of the process continues.

$$\begin{array}{l}\langle O : Ob|Pr : (await\, X?;P), Lvar : L\rangle\ \langle O : Qu|Ev : Q\rangle \\ \quad\longrightarrow \\ \langle O : Ob|Pr : P, Lvar : L\rangle\ \langle O : Qu|Ev : Q\rangle \\ \textbf{\textit{if}}\ inqueue(eval(X,L),Q)\end{array}$$

where the function *inqueue* checks whether the completion with the given label value is in the message queue Q.

If the message is not in the queue, the active process is suspended. The object can then compute other enabled processes while it waits for the completion of the method call.

$$\begin{array}{l}\langle O : Ob|Pr : (await\, X?;P), PrQ : W, Lvar : L\rangle\ \langle O : Qu|Ev : Q\rangle \\ \quad\longrightarrow \\ \langle O : Ob|Pr : empty, PrQ : (W\ \langle(await\, X?;P),L\rangle), Lvar : empty\rangle \\ \langle O : Qu|Ev : Q\rangle \\ \textbf{\textit{if}}\ not\ inqueue(eval(X,L),Q)\end{array}$$

If there is no active process, the suspended process with the return guard can be tested against the external message queue again. If the completion message is present, the process is reactivated.

$$\begin{array}{l}\langle O : Ob|Pr : empty, PrQ : \langle await\, X?;P,L'\rangle\ W, Lvar : L\rangle \\ \langle O : Qu|Ev : Q\rangle \\ \quad\longrightarrow \\ \langle O : Ob|Pr : P, PrQ : W, Lvar : L'\rangle\ \langle O : Qu|Ev : Q\rangle \\ \textbf{\textit{if}}\ inqueue(eval(X,L),Q)\end{array}$$

Otherwise, another suspended process from the process queue PrQ may be loaded into Pr.

Object Creation. Finally, the creation of new objects is considered. Class parameters are stored among object attributes. A new object with a unique identifier and an associated event queue are created. New object identifiers are created by concatenating tokens from the unbounded set *Tok* to the class name. The new object identifier is returned to the object initiating the object creation.

$$\begin{array}{l}\langle O : Ob|Pr : (X := new\, C(I);P), Lvar : L, Att : A\rangle \\ \langle C : Cl|Param : V, Att : A', Init : (P',L'), Tok : N\rangle \\ \quad\longrightarrow \\ \langle O : Ob|Pr : (X := (C;N);P), Lvar : L, Att : A\rangle \\ \langle (C;N) : Ob|Cl : C, Pr : (V := eval(I,(A;L));P';run), PrQ : empty, \\ \qquad Lvar : L', Att : (V;A'), Lab : 1\rangle\ \langle (C;N) : Qu|Ev : empty\rangle \\ \langle C : Cl|Param : V, Att : A', Init : (P',L'), Tok : Next(N)\rangle\end{array}$$

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

6.3. Testing specifications in the Creol interpreter

Specifications in RL are executable on the Maude modeling and analysis tool [11]. 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 [22]. The interpreter consists of 700 lines of code, including auxiliary functions and equational specifications, and it has 29 rewrite rules.

Although the proposed operational semantics is highly non-deterministic, Maude rewriting is deterministic in its choice of which rule to apply to a given configuration. For the evaluation of specifications of non-deterministic 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 [10], which allows execution strategies for Maude programs to be written in RL. A strategy based on a pseudo-random number generator is proposed in [22]. 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.

7. Related and future work

Related work. The Creol process commands are inspired by notions from process algebra [27]. However, Creol differs in its integration of processes in an object-oriented setting using methods, including active and passive object behavior, and self reference rather than channels. Further, Creol's high-level integration of asynchronous and synchronous communication and the organization of pending processes and interleaving at suspension points within class objects seem hard to capture naturally in process algebra. UML offers asynchronous event communication and synchronous method invocation but does not integrate these, resulting in significantly more complex formalizations [14].

Many object-oriented languages offer constructs for concurrency. A common approach has been to keep activity (threads) and objects distinct, as done in Hybrid [29] and Java [18]. 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 [7], 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 [5] and POOL-T [4].

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. Asynchronous method calls can be implemented in e.g. Java by explicitly creating new threads to handle calls [12]. To facilitate the programmer's task and reduce the risk of errors, implicit control structures based on asynchronous method calls seem more attractive, allowing a higher level of abstraction in the language.

Languages based on the Actor model [2, 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. Representing method calls by asynchronous messages has lead to the notion of future variables which may be found in languages such as ABCL [32], Eiffel// [8], CJava [12], and Polyphonic C$^\sharp$ [6]. The proposed reply guards further extend this view of asynchrony.

Maude's inherent object concept [11,26] also represents an object's state as a subconfiguration, 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.

Future work. The long term goal of our research is to study openness in distributed object systems. While this paper has focused on communication aspects in the asynchronous setting, we believe the language presented here offers interesting possibilities for reasoning in the presence of dynamic change. An obvious way to provide some openness is to allow dynamic addition of new (sub)classes and new (sub)interfaces. In our setting, this mechanism in itself does not violate reasoning control, in the sense that old proved results still hold. Also, additional implementation claims may be stated and proved. 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. 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. In order to avoid circular inheritance graphs, C may not inherit from a subclass of C. Unlike standard subclassing, all existing objects of class C or a subclass of C become renewed in this case and support the new interfaces. Reasoning control is maintained when the dynamic class construct is restricted to a form of behavioral subtyping, which can be ensured by verification conditions associated with the class modification [30]. Unrestricted use of the dynamic class construct implies that objects of class C may violate behavior specified earlier, and constraints on objects of class C must be reproved or weakened. Furthermore, as a special case of class modification, one may posteriorly add super-classes to an established class hierarchy. This would be an answer to a major criticism against object-oriented design [16], namely that the class hierarchy severely limits restructuring the system design.

Currently, reasoning control and inheritance in the setting of Creol are being investigated, along with a time-out mechanism. More elaborate case studies to test the mechanisms of the language are on the way. The next step in this research will be a detailed investigation of dynamic classes as outlined above. The run-time implementation of dynamic class constructs is non-trivial [25], even typing and virtual binding need special considerations. 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.

8. Conclusion

Whereas object orientation has been advocated as a promising framework for distributed systems, 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 design of distributed concurrent objects, high-level implicit control structures are needed to organize method invocations and internal object activity. This paper integrates remote and local asynchronous and synchronous method calls, and nested processor release points in method bodies for this purpose. 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.

References

[1] E. Ábrahám-Mumm, F. S. de Boer, W.-P. de Roever, and M. Steffen. Verification for Java's reentrant multithreading concept. In *Intl. Conf. on Foundations of Software Science and Computation Structures (FOSSACS'02)*, LNCS 2303, pages 5–20. Springer, Apr. 2002.

[2] G. A. Agha. Abstracting interaction patterns: A programming paradigm for open distributed systems. In E. Najm and J.-B. Stefani, editors, *Proc. Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'96)*, pages 135–153, Paris, 1996. Chapman & Hall.

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

[4] P. America. POOL-T: A parallel object-oriented language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. The MIT Press, Cambridge, Mass., 1987.

[5] G. R. Andrews. *Concurrent Programming: Principles and Practice*. Addison-Wesley, Reading, Mass., 1991.

[6] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C$^\sharp$. In B. Magnusson, editor, *Proc. of 16th European Conf. on Object-Oriented Programming (ECOOP 2002)*, LNCS 2374, pages 415–440. Springer, June 2002.

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

[8] D. Caromel and Y. Roudier. Reactive programming in Eiffel//. In J.-P. Briot, J. M. Geib, and A. Yonezawa, editors, *Proc. of the Conf. on Object-Based Parallel and Distributed Computation*, LNCS 1107, pages 125–147. Springer, 1996.

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

[10] M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Publications, Stanford, California, 2000.

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

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

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

[14] Werner Damm, Bernhard Josko, Amir Pnueli, Angelika Votintseva: *Understanding UML: A Formal Semantics of Concurrency and Communication in Real-Time UML.* FMCO 2002, LNCS 2852, pages 71-98. Springer, 2003.

[15] C. de Oliveira Braga. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics*. PhD thesis, Pontifícia Univ. Católica do Rio de Janeiro, 2001.

[16] C. Ghezzi and M. Jazayeri. *Programming Language Concepts*. John Wiley & Sons, 3rd edition, 1998.

[17] J. Goguen and J. Tardo. An introduction to OBJ: A language for writing and testing formal algebraic program specifications. In N. Gehani and A. McGettrick, editors, *Software Specification Techniques*. Addison-Wesley, 1986.

[18] J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java language specification*. Addison-Wesley, 2000.

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

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

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

[22] E. B. Johnsen, O. Owe, and E. W. Axelsen. A run-time environment for concurrent objects with asynchronous methods calls. In *Proc. 5th Intl. Workshop on Rewriting Logic and its Applications (WRLA'04)*, Mar. 2004. To appear in Electronic Notes in Theoretical Computer Science. Elsevier.

[23] C. B. Jones. *Development Methods for Computer Programmes Including a Notion of Interference*. PhD thesis, Oxford University, UK, June l981.

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

[25] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic Java classes. In E. Bertino, editor, *14th European Conf. on Object-Oriented Programming (ECOOP'00)*, LNCS 1850, pages 337–361. Springer, June 2000.

[26] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theor. Comp. Science*, 96:73–155, 1992.

[27] R. Milner. *Communication and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

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

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

[30] O. Owe and I. Ryl. A notation for combining formal reasoning, object orientation and openness. Research Report 278, Dept. of informatics, Univ. of Oslo, Norway, Nov. 1999.

[31] N. Soundarajan and S. Fridella. Inheritance: From code reuse to reasoning reuse. In P. Devanbu and J. Poulin, editors, *Proc. 5th Intl. Conf. on Software Reuse (ICSR5)*, pages 206–215. IEEE CS Press, 1998.

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