A run-time environment for concurrent objects with asynchronous method calls

Einar Broch Johnsen, Olaf Owe, and Eyvind W. Axelsen

Department of Informatics, University of Oslo
PO Box 1080 Blindern, N-0316 Oslo, Norway
Email: {einarj,olaf,eyvindwa}@ifi.uio.no

Abstract

A distributed system may be modeled by objects that run concurrently, each with its own processor, and communicate by remote method calls. However objects may have to wait for response to external calls; which can lead to inefficient use of processor capacity or even to deadlock. This paper addresses this limitation 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 distributed environments. The concepts are illustrated by the small object-oriented language Creol and its operational semantics, which is defined using rewriting logic as a semantic framework. Thus, Creol specifications may be executed with Maude as a language interpreter, which allows an incremental development of the language constructs and their operational semantics supported by testing in Maude. However, for prototyping of highly nondeterministic systems, Maude's deterministic engine may be a limitation to practical testing. To overcome this problem, a rewrite strategy based on a pseudo-random number generator is proposed, providing Maude with nondeterministic behavior.

Key words: Object orientation, asynchronous method calls, operational semantics, rewriting logic, nondeterministic rewrite strategies

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 [20], but object interaction by means of method calls is usually synchronous and therefore less suitable in a distributed setting. 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 proposed with an object-oriented design

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

language Creol, based on *processor release points* and *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).

We consider how object-oriented method calls, returning output values in response to input values, can be adapted to the distributed setting. With the *remote procedure call* (RPC) model, an object is brought to life by a procedure call [6]. Control is transferred with the call so there is a master-slave relationship between the caller and the callee. Concurrency is achieved by multiple execution threads, e.g. Hybrid [26] and Java [19]. In Java the interference problem related to shared variables reemerges when threads operate concurrently in the same object, and reasoning about programs in this setting is a highly complex matter [1,11]. Reasoning considerations suggest that all methods should be serialized [9], which is the approach taken by Hybrid. But with serialized methods, the caller 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 due to distribution may cause considerable waiting. In contrast, message passing is a communication form without transfer of control. For synchronous message passing, as in Ada's Rendezvous mechanism, both sender and receiver must be ready before communication can occur. Method calls may be modeled by pairs of messages, on which the two objects must synchronize [6]. For distributed systems, this synchronization still results in much waiting. In the asynchronous setting, messages may always be emitted regardless of when the receiver accepts the message. Communication by asynchronous message passing is well-known from e.g. the Actor model [2,3]. However, method calls imply an ordering on communication not easily captured in the Actor model.

In this paper, method calls are taken as the communication primitive for concurrent objects and given an operational semantics reflected by pairs of asynchronous messages, allowing message overtaking. The result resembles programming with so-called future variables [8,10,16,28,29]; computation may continue until the return value of the call is explicitly needed in the code. To avoid blocking the object at this point, we propose interleaved method evaluations in objects by defining potential processor release points in method bodies using inner guards. Hence, present activity may be suspended, allowing the object's invoked and enabled methods to compete for the free processor.

The operational semantics of Creol has been defined in rewriting logic [23], which is supported by the executable modeling and analysis tool Maude [13]. Rewriting logic is a logic of concurrent change. A number of concurrency models have been successfully represented in rewriting logic and Maude [23,24], including the ODP computational model [25] and structural operational semantics [17]. We have used rewriting logic and Maude as a tool for development of high-level programming constructs for distributed concurrent objects. As

our aim is to consider constructs for a traditional imperative setting, rewrite rules capture the behavior of the abstract machine, rather than method calls as in the object model of rewriting logic and Maude [13,23]. Our experiments suggest that rewriting logic and Maude provide a well-suited framework for experimentation with language constructs and concurrent environments. However, in order to capture the nondeterminism of distributed systems, Maude's deterministic engine may be a limitation. Therefore, the paper proposes a new rewrite strategy for Maude, based on a pseudo-random number generator. This allows nondeterministic executions, selecting not only the rewrite rule according to the random number, but also where it is applied. The strategy seems well-suited for testing any nondeterministic Maude specification, as several runs of the same specification give rise to different executions.

Paper overview. Section 2 introduces the imperative level of Creol. Section 3 provides an example. Section 4 defines Creol's operational semantics using rewriting logic. Section 5 presents a nondeterministic rewrite strategy for Maude. Section 6 considers related work and Section 7 concludes.

2 Programming Constructs

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 [26]. Objects have explicit identifiers: communication takes place between named objects and object identifiers may be exchanged between objects. All object interaction is by means of method calls. Creol objects are typed by abstract interfaces [21,22]. These resemble CORBA's IDL, but extended with semantic requirements and mechanisms for type control in dynamically reconfigurable systems. The language supports strong typing, e.g. invoked methods are supported by the called object (when not null), and formal and actual parameters match.

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 which may be passed as arguments to methods, including as subtypes the object identifiers Obj and data types such as Bool. 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.

2.1 Classes and Objects

At the programming level, attributes (object variables) and method declarations are organized in classes in a standard way. 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 by other objects of appropriate interfaces. These methods reflect passive or reactive behavior in the object, whereas *run* reflects active behavior. Object activity is organized around an *external message queue* and an *internal process queue* which contains *pending processes*. Methods need not terminate and, apart from *init*, all methods may be temporarily *suspended* on the internal process queue.

2.2 Asynchronous Methods

An object offers methods to its environment, specified through a number of interfaces. All interaction with the object happens through the methods of its interfaces. In the asynchronous setting method calls can always be emitted, because the receiving object cannot block communication. Method overtaking is allowed in the sense that if methods offered by an object are invoked in one order, the object may react to the invocations in another order. Methods are, roughly speaking, implemented by nested guarded commands $G \longrightarrow C$, to be evaluated in the context of locally bound variables. Guarded commands are treated in detail in Section 2.3.

Due to the possible interleaving of different method executions, the values of an object's instance variables are not entirely controlled by a method instance if it suspends itself before completion. However, a method may create local variables supplementing the object variables. In particular, the values of formal parameters are stored locally, but other local variables may also be declared. Semantically, an instantiated method is a *process*, represented as a pair $\langle GC, L\rangle$ where GC is a (guarded) sequence of commands and $L : \mathsf{Var} \to \mathsf{Data}$ the local variable bindings. Consider an object o which offers the method

$$\mathbf{op}\ m(\mathbf{in}\ x : \mathsf{Data}\ \mathbf{out}\ y : \mathsf{Data}) == \mathbf{var}\ z : \mathsf{Data} := 0; G \longrightarrow C\ .$$

to the environment. Accepting a call $\mathit{invoc}(l, o', o, m, 2)$ from an object o' adds the pair $\langle G \longrightarrow C, \{\mathit{label} \mapsto l, \mathit{caller} \mapsto o', x \mapsto 2, y \mapsto \mathit{nil}, z \mapsto 0\}\rangle$ to the internal process queue of object o, where pending processes wait for the object processor. An object can have several pending calls to the same method, possibly with different values for local variables. The local variables *label* and *caller* are reserved to identify the call and the caller for the reply, which is emitted at method termination.

An asynchronous method call is made with the command $l!o.m(e)$, where $l \in \mathsf{Label}$ is a unique reference to the call, o an object identifier, m a method name, and e an expression list with the supplied actual parameters. Labels are used to identify replies, and may be omitted if a reply is not explicitly

requested. As no synchronization is involved, process execution may proceed after calling an external method until the return values are needed by the process. To fetch the return values from the queue, say in a variable list x, we ask for the reply to the call: $l?(x)$. If the reply has arrived, return values are assigned to x and execution continues without delay. If no reply to the call has been received, the process must now wait. This interpretation of $l?(x)$ gives the same effect as treating x as a future variable. However, waiting in the asynchronous case can be avoided altogether by introducing processor release points for reply requests. In the case without reply, execution is *suspended*, placing the active process and its local variables on the internal process queue.

Although remote and local calls can be handled operationally in the same way, it is clear that for execution of local calls the calling process must eventually suspend its own execution. In particular, synchronous local calls are given direct access to the object processor. The syntax $o.m(e; x)$ is adopted for synchronous (RPC) method calls, blocking the processor while waiting for the reply. Local calls need not be prefixed by an object identifier.

2.3 A Language with Processor Release Points

In Creol, the control flow inside concurrent objects may be influenced by potential processor release points. These are explicitly declared in method bodies using guarded commands, as introduced by Dijkstra [18], and may be nested within the same local variable scope. When an inner guard evaluates to false during process execution, the remaining process code is suspended to the internal process queue and the processor is released. After processor release, an enabled process from the internal process queue is selected for execution.

Definition 2.1 The type Guard is constructed inductively as follows:

- $wait \in$ Guard (explicit release)
- $l?(x) \in$ Guard, where $l \in$ Label and $x \in$ Var
- $\phi \in$ 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 $l?(x)$ checks whether the reply to a method call has been received, as further execution of a process will often depend on the arrival of a certain reply. If this is the case, $l?(x)$ returns true and instantiates x with the return values. Evaluation of guards is done atomically.

Guarded commands can be *composed* in different ways, reflecting the requirements to the internal control flow in objects. Let GC_1 and GC_2 denote the guarded commands $G_1 \longrightarrow C_1$ and $G_2 \longrightarrow C_2$. Nesting of guards is obtained by sequential composition; in a program statement $GC_1; G_2 \longrightarrow C_2$, the guard G_2 corresponds to a potential inner processor release point. Nondeterministic choice between guarded commands is expressed by $GC_1 \Box GC_2$, which may compute C_1 if G_1 evaluates to true, C_2 if G_2 evaluates to true, and is otherwise suspended. Nondeterministic merge is expressed by $GC_1 \| GC_2$,

Syntactic categories.	*Definitions.*
C in Com	$C ::= \varepsilon \mid x := e \mid GC \mid C_1; C_2 \mid \textbf{new}\ classname(e)$
GC in Gcom	$\mid \textbf{if}\ G\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{fi}$
G in Guard	$\mid \textbf{while}\ G\ \textbf{do}\ C\ \textbf{od}$
x in VarList	$\mid m(e;x) \mid l!m(e) \mid !m(e) \mid l?(x)$
e in ExprList	$\mid o.m(e;x) \mid l!o.m(e) \mid !o.m(e)$
m in Mtd	$GC ::= G \longrightarrow C$
o in Obj	$\mid GC_1 \Box GC_2$
l in Label	$\mid GC_1 \lVert GC_2$

Fig. 1. An outline of the syntax for the proposed language Creol, focusing on the main syntactic categories Com of commands and Gcom of guarded commands.

which can be defined by $(GC_1; GC_2)\Box(GC_2; GC_1)$. Ordinary control flow is expressed by **if** and **while** constructs, and assignment to local and object variables is expressed as $x := e$ for a list x of program variables and a list e of expressions. Figure 1 summarizes the language syntax.

With nested processor release points, the processor need not wait actively for replies. This approach is more flexible than future variables: pending processes or new method calls may be evaluated instead of blocking the processor. However, when the reply has arrived, the *continuation* of the original process must compete with the other enabled and pending processes in the internal process queue.

3 Example: The Dining Philosophers

The well-known dining philosophers are now considered in Creol. The example will later be used to experiment with the language interpreter. A butler informs a philosopher of the identity of the philosopher's left neighbor. A philosopher may borrow and return its neighbor's chopstick. Interaction between the philosophers and the butler is restricted by interfaces. This results in a clear distinction between internal methods and methods externally available to other objects typed by so-called *cointerfaces* [21,22]. These express mutual dependency between interfaces, and are declared in the interfaces by means of a **with** construct. Strong typing and cointerfaces guarantee that only philosophers may call the methods *borrowStick* and *returnStick*.

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

```
interface Butler
begin
  with Phil
    op getNeighbor(out n:Phil)
end
```

In this approach, philosopher objects display both active and reactive behavior. Each philosopher controls one chopstick and must borrow its neighbor's chopstick in order to eat. Thus, philosophers have their internal activity as well as responding to calls from the environment. The standard configuration of the dining philosophers is most easily obtained by means of a single butler.

3.1 Implementing the Philosophers

Philosophers are active objects so the Philosopher class will include a *run* method. This 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, and will be interleaved in a nondeterministic and nonterminating manner, illustrating the processor release point construct. All three methods depend on the value of the internal variable *hungry*. The *think* method is a loop which suspends its own evaluation before each iteration, whereas *eat* attempts to grab the object's and the neighbor's chopsticks in order to satisfy the philosopher's hunger. In this case, the philosopher must wait until both chopsticks are available. In order to avoid blocking the object processor, the *eat* method is suspended after asking for the neighbor's chopstick; further processing of the method can first happen when the guard is satisfied. The *digest* method represents the action of becoming hungry. Classes may include class parameters, which become instance attributes bound at object creation, as in Simula. The Philosopher class is defined as follows:

```
class Philosopher(butler: Butler) implements Phil
begin
  var hungry: bool, chopstick: bool, neighbor: Phil
  op init == chopstick := true; hungry := false; butler.getNeighbor(;neighbor) .
  op run == true ⟶ !think ||| true ⟶ !eat ||| true ⟶ !digest .
  op think == not hungry ⟶ ⟨thinking...⟩; wait ⟶ !think .
  op eat == var l : label; hungry ⟶ l!neighbor.borrowStick;
              (chopstick ∧ l?()) ⟶ ⟨eating...⟩; hungry := false;
               !neighbor.returnStick; wait ⟶ !eat .
  op digest == not hungry ⟶ (hungry := true; wait ⟶ !digest) .

 with Phil
  op borrowStick == chopstick ⟶ chopstick := false .
  op returnStick == chopstick := true .
end
```

This implementation favors implicit control of the object's active behavior. Caromel and Rodier argue that facilities for both implicit and explicit control are needed in languages which address concurrent programming [10]. Explicit activity control can be programmed in Creol by using a **while** loop in the *run* method. However in asynchronous distributed systems, we believe that com-

munication introduces so much nondeterminism that explicit control structures quickly lead to program over-specification and possibly to unnecessary active waiting.

In contrast to the active philosophers, butlers are passive. After creating philosophers during initialization, a butler waits for philosophers to request the identity of their neighbors. The code of the butler class is straightforward and therefore omitted here.

4 A Rewriting Logic Semantics for Creol

The operational semantics of the proposed language constructs is given using the semantic framework provided by rewriting logic (RL). A rewrite theory is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$, where the signature Σ defines the function symbols of the language, E defines equations between terms, L is a set of labels, and R is a set of labeled rewrite rules. From a computational viewpoint, a rewrite rule $t \longrightarrow t'$ may be interpreted as a *local transition rule* allowing an instance of the pattern t to evolve into the corresponding instance of the pattern t'. Rewrite rules apply to local fragments of a state configuration. Rules may be applied in parallel to non-overlapping *subconfigurations*. We assume that RL is known to the reader and present the operational semantics in the syntax of Maude. Rewrite rules will capture the behavior of a Creol abstract machine, and not of the Creol objects as in Maude's object model. Hence, a configuration is a multiset combining Creol objects, messages, queues, and classes. Auxiliary functions are defined in equational logic, and evaluated in between rewrite steps [23]. As usual, the associative and commutative constructor for multisets and the associative constructor for lists are represented by whitespace.

An RL object is a term of the type $\langle O : C \mid a_1 : v_1, \ldots, a_n : v_n \rangle$, where O is the object's identifier, C is its class, the a_i's are the names of the object's attributes, and the v_i's are the corresponding values [13]. 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 $\langle Id \mid Cl, Pr, PrQ, Lvar, Att, Lcnt \rangle$, where Id is the object identifier, Cl the class name, Pr the active process code, PrQ a multiset of pending processes (see Section 2.2) allowing all kinds of queue orderings, and $Lvar$ and Att the local and object variables, respectively. Finally, $Lcnt$ is the method call identifier corresponding to labels in the language. Thus, the object identifier Id and the local label value provide a globally unique identifier for each method call. External message queues have a name and contain a multiset of unprocessed messages. Each external message queue is associated with one specific Creol object.

Creol classes are represented by RL objects $\langle Cl \mid Att, Ocnt, init, run, Mtds \rangle$, where Cl is the class name, Att a list of attributes, $Ocnt$ the number of objects instantiated of the class, and $Mtds$ a set of methods. When an object needs a method, it is loaded from the $Mtds$ set of its class (overloading and virtual

binding issues connected to inheritance are ignored in this paper).

In RL's object model [23], classes are not represented explicitly in the system configuration. This leads to ad hoc mechanisms to handle object creation, which we avoid by representing Creol classes in the configuration. The Creol command *new C(args)* will create a new object with a unique object identifier, object variables as listed in the class parameter list with values defined in *args* and in *Att*, and place the code from methods *init* and *run* in Pr. Uniqueness of the object identifier is ensured by appending the number $Ocnt$ to the class name, and increasing $Ocnt$.

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 E to X in either the list of local or 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 pending processes:* When Pr is empty, a pending process may be activated. When a process is loaded, its local variables are also loaded into memory.
- *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 a different subconfiguration from that of the object itself. Consequently, messages can be received in parallel with other activity in the object.

Specifications in RL are executable on the Maude tool, so Creol's operational semantics may be used as a language interpreter. The entire interpreter consists of 700 lines of code, including auxiliary functions and equational specifications, and it has 32 rewrite rules. A detailed presentation of the interpreter may be found in [7]. The rules for asynchronous method calls and guarded commands are now considered in more detail.

4.1 Asynchronous Method Calls

Objects communicate by 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 in, and the execution of $m(in)$ results in the return values out, the call is reflected by two messages $invoc(l, o_1, o_2, m, in)$ and $comp(l, o_1, out)$, 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 unique

label l, generated by the caller.

When an object calls an external method, a message is placed in the configuration. Transport rules eventually move the message to 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.

The interpreter checks the external message queue of a Creol object for method invocations, and loads the corresponding method code from the object's class into the object's internal process queue PrQ The rewrite rule for this transition can be expressed as follows, ignoring irrelevant attributes in the style of Full Maude [13]:

rl [receivecall] :
$\langle O : Id \mid Cl : C, PrQ : W \rangle\ \langle q(O) : QId \mid Ev : Q\ invoc(N, O', O, M, I) \rangle$
$\langle C : Cl \mid Mtds : MT \rangle\ \Rightarrow$
$\langle O : Id \mid Cl : C, PrQ : (get(M, MT, (N\ O'\ I)))\ W \rangle\ \langle q(O) : QId \mid Ev : Q \rangle$
$\langle C : Cl \mid \rangle$.

The auxiliary function *get* fetches method M in the method set MT of the class, and returns a process with the method's code and local variables. Values of actual parameters I, the caller O', and the message label N, are stored as local variables. (The label cannot be modified by the process.) The rule for a local asynchronous call is similar, but the call comes from the active process code Pr instead of the external message queue. For a synchronous local call the code is loaded directly into the active process code Pr, since waiting actively in this case leads to deadlock.

4.2 Guarded Commands

Creol has three types of guards representing potential processor release points: The standard boolean expression, a wait guard, and a return guard. Rules for evaluation of return guards in the active process are now considered. 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.

crl [returnguard] :
$\langle O : Id \mid Pr : X?(J) \longrightarrow P, Lvar : L \rangle\ \langle q(O) : QId \mid Ev : Q\ comp(N,\ O,\ K) \rangle\ \Rightarrow$
$\langle O : Id \mid Pr : (J := K); P, Lvar : L \rangle\ \langle q(O) : QId \mid Ev : Q \rangle$
if $N == val(X, L)$.

The condition ensures that the correct reply message is identified. The auxiliary function *val* fetches the value associated with the label variable X from the local variables L.

If the message is not in the queue, the active process is suspended. In this case other enabled processes may be activated while waiting for the method call completion.

crl [returnguard_notinqueue] :
$\langle O : Id \mid Pr : X?(J) \longrightarrow P, PrQ : W, Lvar : L\rangle\ \langle q(O) : QId \mid Ev : Q\rangle\ \Rightarrow$
$\langle O : Id \mid Pr : empty, PrQ : W(X?(J) \longrightarrow P, L), Lvar : no\rangle\ \langle q(O) : QId \mid Ev : Q\rangle$
if not $inqueue(val(X, L), Q)$.

where the function *inqueue* looks for the completion in the message queue Q.

If no process is active, the suspended process with the return guard can be tested against the external message queue again. If the completion message is present, the return value is matched to local or object attributes and the process is reactivated.

crl [return_guard_st] :
$\langle O : Id \mid Pr : empty, PrQ : (X?(J) \longrightarrow P, L')\ W\rangle$
$\langle q(O) : QId \mid Ev : Q\ comp(N,\ O,\ K)\rangle\ \Rightarrow$
$\langle O : Id \mid Pr : (J := K); P, PrQ : W, Lvar : L'\rangle\ \langle q(O) : QId \mid Ev : Q\rangle$
if $N == val(X, L')$.

Otherwise, another pending process from the multiset PrQ may be activated.

4.3 Execution of Creol Programs in Maude

The operational semantics of Creol is executable on the rewriting logic tool Maude, as an interpreter for Creol programs. This makes Maude well-suited for experimenting with programming constructs and language prototypes, combined with Maude's various rewrite strategies, search, and model-checking abilities.

Although the operational semantics is highly nondeterministic, Maude is deterministic in its choice of which rule to apply to a given configuration. The dining philosophers program of Section 3 is used to test the performance of the interpreter. Running the example on the interpreter, we observe that Maude selects processes from the internal process queue in an unfair manner. Even with the "fair" rewrite strategy [14], the philosophers would only *think* after 10 000 rewrites. This means that although the suspended instance of *digest* is enabled in each philosopher, it is not executed. In order to explore the full state space of the above program, Maude provides a breadth-first search facility. However, for highly nondeterministic systems, the naive use of this search mechanism will often become very resource demanding and hard to apply in practice.

5 Nondeterministic Execution

This section presents an approach to nondeterministic execution of Maude specifications. The approach is based on Maude's reflective capabilities in order to control the rewriting process. Informally, a configuration C and the set $\mathcal{R}$ of rewrite rules of a Maude specification may be represented as terms $\overline{C}$ and $\overline{\mathcal{R}}$ at the metalevel, and metalevel rewrite rules may be used to select which rule from $\mathcal{R}$ to apply to which subterm of C. This is done by a

sequential interpreter function which takes as arguments a finitely presented rewrite theory $\mathcal{R}$, a term C, and a deterministic strategy S. Details on the theory and use of reflection in rewriting logic and Maude may be found in [12,15]. A strategy for rule selection which employs a pseudo-random number generator, is now defined in Maude syntax.

There is a vast selection of algorithms for generating pseudo-random numbers. For simplicity, we select a simple "minimal" general purpose algorithm from *Press et al.* [27, p. 278]: $I_{j+1} = a\ I_j\ (mod\ m)$. In [27], the authors argue that choosing $a = 7^5$ and $m = 2^{31} - 1$ yields a generator that has passed all important theoretical tests, and that has been put to successful use in a variety of practical applications. The algorithm is programmed in Maude as a functional module *RANDOM*:

```
fmod RANDOM is
  protecting NAT .
  op rand : Nat -> Nat .
  ops a m : -> Nat .
  eq a = 16807 . *** = 7^5
  eq m = 2147483647 . *** = 2^31 - 1
  var N : Nat .
  eq rand(N) = (a * N) rem m .
endfm
```

The nondeterministic rewriting strategy is defined as a Maude module *META-ENGINE* that protects the built-in module *META-LEVEL*. Metalevel rewriting is carried out by a conditional rule *exec*, described below. An object *Engine* keeps track of the current state, and is defined as follows:

```
op <Engine | curTerm : _, curModule : _, labels : _, failedRules : _,
  numRules : _, randomNum : _, randomNum2 : _> :
    Term Qid QidList QidList Int Int Int -> EngineObject .
```

The object contains several attributes, whose values are set at run-time; *curTerm* contains the metarepresentation of the current configuration, *curModule* is the metarepresentation of the name of the base-level module in which the rewrites will be performed, *labels* is a *QidList* of rule labels from the module *curModule*, *failedRules* contains a *QidList* of rule labels for rules that could not be applied to *curTerm*, *numRules* is the length of the *labels* list, included for performance reasons, and finally *randomNum* and *randomNum2* are numbers generated by the pseudo-random number generator defined above.

The actual metalevel rewrite steps are handled by Maude's built-in descent function *metaXapply*$(\overline{\mathcal{R}}, \overline{t}, \overline{l}, \sigma, n, b, m)$, where $\mathcal{R}$ is module, t a term, l a rule label, σ a (partial) substitution, n a match number, b a bound, and m a solution number [14]. The last argument, the solution number m, is of particular interest for nondeterministic execution. Our strategy for performing a rewrite is two-fold:

(i) The engine will select, using the pseudo-random number generator, a rule label l corresponding to a rule in $\mathcal{R}$.

(ii) A rule identified by l may be applicable to a term t at several different positions within the term, referred to in Maude as solutions. To allow for "deep" randomization, within objects as well as between them, we will also select the solution number pseudo-randomly.

The conditional rewrite rule *exec* implements this strategy, using *metaXapply* and an auxiliary function *chooseSolution*$(\overline{\mathcal{R}}, \overline{t}, \overline{l}, r)$. The latter takes care of part (ii) of the strategy, and chooses a valid solution number using the pseudo-random number r. It makes use of the fact that *metaXapply*$(\overline{\mathcal{R}}, \overline{t}, \overline{l}, \sigma, n, b, m)$ returns *failure* when there is no solution m. It is therefore easy to find the number s of possible solutions by repeatedly calling *metaXapply* with increasing values for m, until it returns failure. Utilizing this information, selecting a solution randomly becomes a matter of calculating $r \ mod \ s$. If no solution can be found (i.e. the rule is not applicable), *chooseSolution* returns -1. The code for *exec* is given below:

```
crl [exec] :
   ⟨Engine | curTerm : T, curModule : MOD, labels : L, failedRules : FR,
   numRules : NR, randomNum : R, randomNum2 : R2⟩
⇒
   if chooseSolution(MOD, T, findItem(L, R rem NR), R2) == −1
   then
      ⟨Engine | curTerm : T, curModule : MOD, labels : L,
      failedRules : if findItem(L, R rem NR) in FR then FR
      else FR findItem(L, R rem NR) fi,
      numRules : NR, randomNum : rand(R), randomNum2 : rand(R2)⟩
   else
      ⟨Engine | curTerm : getTerm(metaXapply([MOD], T,
      findItem(L, Rrem NR), none, 0, unbounded,
      chooseSolution(MOD, T, findItem(L, R rem NR), R2))),
      curModule : MOD, labels : L, failedRules : nil, numRules : NR,
      randomNum : rand(R), randomNum2 : rand(R2)⟩
   fi
if length(FR) < NR .
```

Performing one rewrite at the time, the engine selects a rule randomly from its list of rule labels, and tries to apply it to the current configuration. If rule application fails (i.e., the left side of the rule does not match the current configuration and *chooseSolution* returns -1), the rule label is added to the list *failedRules*. If the length of *failedRules* equals the number *numRules* of rules in the module, the execution terminates, as no rule is applicable.

Once an applicable rule has been selected, the list of failed rules is reset to *nil*, and the rule is applied. The resulting term is assigned to *curTerm*, and another rewrite may be performed in the same manner. For specifications

where a majority of the rules will be nonapplicable to any given configuration, the strategy given in *exec* will prove to be inefficient. To amend this, rules that have already failed may be temporarily removed from the *labels* list until a rule application succeeds.

The metarewriting engine introduced in this section makes it easy to simulate a series of different executions of any valid Maude specification, by supplying different seeds for the pseudo-random number generator. Therefore, the engine provides a rewriting strategy for testing specifications of nondeterministic systems, complementary to Maude's standard rewrite and search facilities. For the Creol example of the Dining Philosophers (Section 3), this strategy provides much more informative testing than Maude's internal fair rewrite strategy on a single run, distributing rewrite steps evenly between the different philosophers and their different enabled methods, and easily provides a series of different "fair" executions of the program.

6 Related Work

Many object-oriented languages offer constructs for concurrency. A common approach has been to keep activity (threads) and objects distinct, as done in Hybrid [26] and Java [19]. These languages rely on the tightly synchronized RPC 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 [9], 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 [4]. The latter is interesting because of its emphasis on reasoning control and compositional semantics, allowing inter-object concurrency [5].

For distributed systems, with potential delays and even loss of communication, the tight synchronization of the RPC model seems less desirable. Hybrid offers *delegation* as an explicit programming construct to (temporarily) branch an activity thread. Asynchronous method calls can be implemented in e.g. Java by explicit concurrency control, creating new threads to handle calls. In order 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 concurrency 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 is found in languages such as ABCL [29] and ConcurrentSmalltalk [28], as well as in Eiffel// [10], CJava [16], and Polyphonic C♯ [8]. The proposed asynchronous method calls are similar to future variables, but the proposed nested processor release points further extend this approach to asynchrony.

In Maude's standard object model [13,23], object behavior is captured directly by rewrite rules. Both Actor-style asynchronous messages and synchronous transitions (rewrite rules involving many objects) are allowed, which makes Maude's object model very flexible. However, asynchronous method calls and suspension points as proposed in this paper are hard to represent directly within this model.

7 Concluding Remarks

Whereas object orientation has been advocated as a promising framework for distributed systems, common approaches to combining object-oriented method invocations with distribution seem less satisfactory. High-level implicit control structures may facilitate the design of distributed concurrent objects. This paper proposes asynchronous method calls and nested processor release points in method bodies for this purpose. The approach is more flexible than serialized methods, while maintaining the ease of partial correctness reasoning about code which is lost for nonserialized methods. However liveness reasoning in our setting will require a fairness guarantee which is not provided by the RL semantics, suggesting the need for a fair scheduling strategy.

The executable semantic framework provided by rewriting logic and Maude offers valuable support for the development of program constructs, allowing experimentation with the language behavior during development. In order to simulate the highly nondeterministic environment targeted by the language, a nondeterministic rewrite strategy has been proposed, based on a pseudo-random number generator. We are currently extending the abstract machine with a metalayer which captures the semantic specifications of the abstract interfaces of the language [21,22]. This provides a prototyping environment for initial designs, allowing the observable behavior of objects to be controlled without explicit modelling, and a testing environment for imperative programs without explicit representation of the environment. Analysis techniques for Creol programs are under development. In future work, we plan to extend the operational semantics with class inheritance mechanisms, including method overloading, and use this model to experiment with dynamic features of open object systems such as run-time mechanisms for system upgrades.

References

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

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

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

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

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

[5] America, P., J. de Bakker, J. N. Kok and J. Rutten, *Operational semantics of a parallel object-oriented language*, in: *13th ACM Symposium on Principles of Programming Languages (POPL'86)* (1986), pp. 194–208.

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

[7] Arnestad, M., "En abstrakt maskin for Creol i Maude," Master's thesis, Department of informatics, University of Oslo, Norway (2003), in Norwegian. Available from http://heim.ifi.uio.no/~creol.

[8] Benton, N., L. Cardelli and C. Fournet, *Modern concurrency abstractions for C♯*, in: B. Magnusson, editor, *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, Lecture Notes in Computer Science **2374** (2002), pp. 415–440.

[9] Brinch Hansen, P., *Java's insecure parallelism*, ACM SIGPLAN Notices **34** (1999), pp. 38–45.

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

[11] Cenciarelli, P., 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*, Lecture Notes in Computer Science **1523**, Springer-Verlag, 1999 pp. 157–200.

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

[13] Clavel, M., 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** (2002), pp. 187–243.

[14] Clavel, M., F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer and C. Talcott, "Maude 2.0 Manual," Computer Science Laboratory, SRI International (2003).

[15] Clavel, M. and J. Meseguer, *Reflection in conditional rewriting logic*, Theoretical Computer Science **285** (2002), pp. 245–288.

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

[17] de Oliveira Braga, C., "Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics," Ph.D. thesis, Pontifícia Universidade Católica do Rio de Janeiro (2001).

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

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

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

[21] Johnsen, E. B. and O. Owe, *A compositional formalism for object viewpoints*, in: B. Jacobs and A. Rensink, editors, *Proc. 5th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'02)* (2002), pp. 45–60.

[22] Johnsen, E. B. 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*, Lecture Notes in Computer Science **2635**, Springer-Verlag, 2004 pp. 137–164.

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

[24] Meseguer, J., *Rewriting logic as a semantic framework for concurrency: A progress report*, in: U. Montanari and V. Sassone, editors, *7th International Conference on Concurrency Theory (CONCUR'96)*, Lecture Notes in Computer Science **1119** (1996), pp. 331–372.

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

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

[27] Press, W. H., S. A. Teukolsky, W. T. Vetterling and B. P. Flannery, "Numerical Recipes in C: The Art of Scientific Computing," Cambridge University Press, Cambridge, UK, 2002, second edition.

[28] Yokote, Y. and M. Tokoro, *Concurrent programming in ConcurrentSmalltalk*, in: A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, The MIT Press, Cambridge, Mass., 1987 pp. 129–158.

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