Combining Active and Reactive Behavior in Concurrent Objects

Einar Broch Johnsen, Olaf Owe, and Marte Arnestad
Department of Informatics, University of Oslo

Abstract

A distributed system can 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; at best resulting in inefficient use of processor capacity, at worst resulting in deadlock. Furthermore, it is difficult to combine active and passive object behavior without defining explicit control loops. This paper proposes a solution to these problems by means of asynchronous method calls and conditional processor release points in program code. Although at the cost of additional internal non-determinism 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.

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 [16], 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, 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). The operational semantics for these constructs is defined in rewriting logic [19] and executable as an interpreter in the tool Maude [11]. Experiments show that rewriting logic and Maude provide a well-suited platform for experimentation with language constructs and concurrent environments.

There are three basic interaction models for concurrent processes: shared variables, remote method calls, and message passing [6]. Rather than sharing variables, objects encapsulate local variables, and inter-object communication is more naturally modeled by (remote) method calls. With the *remote procedure call* (RPC) model, an object is brought to life by a procedure 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 [21] and Java [15], where concurrency is

achieved through multithreading. However, the interference problem related to shared variables reemerges when threads operate concurrently in the same object. Hybrid avoids this because all methods are serialized. In contrast, non-serialized methods in Java can access attributes simultaneously. Reasoning about programs in this setting is a highly complex matter [1, 10]: Safety is by convention rather than by language design [8]. Verification considerations therefore suggest that objects should have serialized methods. However, when restricting to serialized methods, the caller must *wait* for the return of a call, blocking for any activity in the object. In a distributed setting this limitation is severe; delays and instability due to distribution may cause a lot of unnecessary waiting. Furthermore, a nonterminating method will block access to the object monitor, which leads to difficulties for combining active and passive behavior in the same object.

Consequently we consider message passing, a communication form without any transfer of control between concurrent objects. A method call can here be modeled by an invocation and a reply message. Synchronous message passing, as in Ada's Rendezvous mechanism, requires that both sender and receiver are ready before communication can occur. Hence, the objects synchronize on message transmission. For method calls, the calling object must wait between the synchronized messages [6]. For distributed systems, even such synchronization must necessarily result in much waiting. In the asynchronous setting message emission is always possible, regardless of when the receiver accepts the message. Communication by asynchronous message passing is well-known from e.g. the Actor model [2, 3]. However, method calls imply an ordering on communication not easily captured in the Actor model. Actors do not distinguish replies from invocations, so modeling method calls with Actors quickly becomes unwieldy [2].

In this paper, method calls are taken as the communication primitive for concurrent objects in the language, and given an operational semantics reflected by pairs of asynchronous messages [17, 18]. The result resembles programming with so-called future variables [9, 12, 22, 23]; the caller can continue its computation until the return value of the call is explicitly needed in the code. If the reply has not arrived at this point, the object must wait — unless it can proceed with other pending computations. We extend the future variable programming style by introducing interleaved method evaluations in objects in a controlled manner, using nested guarded commands in method bodies. Guarded commands were introduced by Dijkstra [13] to capture non-deterministic choice as a basic programming construct. Here, guards will influence the control flow inside concurrent objects. In particular inner guards are used to release processor control, allowing the object's invoked and enabled methods to compete for the free processor.

The paper is organized as follows. Section 2 introduces Creol, a language with asynchronous method calls and processor release points. Section 3 gives an example of the dining philosophers in Creol. Section 4 presents rewriting logic and Maude, used in Section 5 to define the operational semantics of Creol. Section 6 discusses the language interpreter obtained by executing the rewriting logic semantics in the Maude tool, before we consider related work and conclude in Sections 7 and 8.

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 [21]. The objects considered have explicit identifiers: communication takes place between named objects and object identifiers may be exchanged between objects. Creol objects are typed by abstract interfaces [17,18]. 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), 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, including 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.

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 it exists, is invoked. 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 an 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. Each object has an associated *external message queue* which collects method invocations as they arrive. 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 created. 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 bindings. Consider an object o which offers the method

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

to the environment. Syntactically, method declarations end with a period. Accepting a call $invoc(l, o', o, m, 2)$ from another object o' adds the pair $\langle G \longrightarrow C, \{label \mapsto l, caller \mapsto o', x \mapsto 2, y \mapsto nil, z \mapsto 0\}\rangle$ to the *internal process queue* of the 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, i.e. when the remaining sequence of guarded commands is empty.

An asynchronous method call is made with the command $l!o.m(e)$, where $l \in \mathsf{Label}$ is a locally unique reference to the call, o an object identifier, m a method name, and the expression list e contains the actual parameters supplied to the method. Labels are used to identify replies, and may be omitted if a reply is not explicitly requested. Semantically, this corresponds to emitting an *invocation message* including sufficient information to identify the return message. 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 queue, say in a variable list x, we ask for the reply to our call: $l?(x)$. Replies to method calls are stored in the external message queue. If the reply to the message has not arrived when requested, the process must 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. If the reply has arrived, return values are assigned to x and execution continues without delay. Otherwise, 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 method must eventually suspend its own execution. In particular, synchronous local calls are loaded directly into the active code. The syntax $o.m(e;x)$ is adopted for synchronous (RPC) method calls, blocking the processor while waiting for the reply. The language does not presently support monitor reentrance, mutual synchronous calls may therefore lead to deadlock. Local calls need not be prefixed by an object identifier, in which case they may then be identified syntactically, otherwise equality between caller and callee can be determined at runtime.

2.3 A Language with Processor Release Points

Guarded commands are used to explicitly declare potential release points 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 remaining process code is placed in 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 1 The type Guard is constructed inductively as follows:

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

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

Figure 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.

Here, *wait* is a construct for explicit release of the processor. The intuition behind the reply guard $l?(x)$ is as follows. Unprocessed communication messages in the object's external message queue Ev represent either invocations of methods offered by the object or replies to methods invoked by the object. In the latter case, further execution of a process will often depend on the arrival of a certain reply. The guard $l?(x)$ matches the reply with the same label l in Ev, in which case $l?(x)$ returns true and instantiates x. Evaluation of guards is done atomically.

Guarded commands can be *composed* in different ways, reflecting the requirements to the internal control flow in the 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 processor release point. Exclusive non-deterministic disjunction between guarded commands is expressed by $GC_1 \Box GC_2$, which may compute C_1 if G_1 evaluates to true or C_2 if G_2 evaluates to true. Non-deterministic order is expressed by $GC_1 || GC_2$, which can be defined by $(GC_1; GC_2) \Box (GC_2; GC_1)$. Control flow without potential processor release is expressed by **if** and **while** constructs, and assignment to local and object variables is expressed as $x := e$ for a program variable x and expression e. 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: Because the object processor is not blocked, pending processes or new method calls may be evaluated in the meantime. However, when the reply has arrived, the *continuation* of the original process must compete with other enabled pending processes in the internal process queue.

3 Example: The Dining Philosophers

The dining philosophers example is now considered in Creol. The example will later be used to experiment with the language interpreter. A butler object is used to inform a philosopher of the identity of its 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 *cointerfaces*, which are declared in the interfaces by means of a **with** construct. Strong typing and cointerfaces guarantee that only other philosophers can call the methods *borrowStick* and *returnStick*.

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

3.1 Implementing the Philosophers

The philosophers are active objects, which implies that the Philosopher class will include a *run* method. The processor release point construct is illustrated by defining *run* in terms of several non-terminating internal methods representing different activities within a philosopher; *think*, *eat*, and *digest*. 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. The philosopher has to wait until both chopsticks are available. In order to avoid blocking the object processor, the *eat* method is therefore suspended after asking for the neighbor's chopstick; further processing of the method can happen once the guard is satisfied. The last method represents the action of becoming hungry. 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 [9]. 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 communication introduces so much non-determinism that explicit control structures quickly lead to program over-specification and possibly to unnecessary active waiting. Section 6 discusses the interpreter using this program.

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

4 Rewriting Logic and Maude

The operational semantics of Creol has been defined in rewriting logic [19], which is supported by the executable modeling and analysis tool Maude [11]. Rewriting logic is a logic of concurrent change. A number of concurrency models have been successfully represented in rewriting logic and Maude, including the ODP computational model [20].

For our purposes, a configuration in rewriting logic is a multiset of terms of given types. These types are specified in equational logic, the functional sublanguage of rewriting logic which supports algebraic specification in the OBJ [14] style. The terms of a configuration are expected to be reducible to normal form, which means that the equations defining the equivalence classes of the algebraic specifications must be confluent and terminating. Configurations may include the substates of computer systems, where different parts of the system are modeled by terms of the different types defined in the equational logic.

Rewriting logic extends algebraic specification 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 their type. Each rule describes how a part of a configuration can change in one step. If several rules can be applied to distinct subconfigurations, they can be executed in a concurrent rewrite step. Consequently, concurrency is implicit in rewriting logic.

Conditional rewrite rules are allowed, where the conditions can be formulated as either rewrites or equations which must hold for the main rule to apply:

```
crl [label] subconfiguration => subconfiguration if condition
```

Unconditional rules have the keyword rl. Rules in rewriting logic may be formulated at a high level of abstraction, closely resembling a compositional operational semantics [5].

5 An Operational Semantics for Creol

The operational semantics of the proposed language constructs is given in rewriting logic (RL), with a set of rewrite rules defining transition steps. These operate on RL configurations, which are multisets combining Creol objects, classes, messages, and queues. Auxiliary functions are defined in equational logic, and are therefore evaluated in between transitions [19]. As customary in RL, the associative and commutative constructor for multisets is represented by whitespace.

An RL object is a term of the type $< O : C \,|\, a_1 : v_1, \ldots, a_n : v_n >$ 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 $< Id \,|\, Cl, Pr, PrQ, Lvar, Att, Lcnt >$, 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), and $Lvar$ and Att the local and object variables, respectively. Finally, $Lcnt$ is the method call identifier corresponding to labels in the language. 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 $< Cl \,|\, Att, Ocnt, init, run, Mtds >$, where Cl is the class name, Att a list of attributes, $Ocnt$ the number of objects instantiated of the class, 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 in this paper).

In RL's object model [19], classes are not represented explicitly in the configuration. This leads to ad hoc mechanisms to handle object creation, which we avoid by representing classes in the configuration. The command *new C* will create a new object with a unique object identifier, object variables as listed 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*.

Concurrent change is achieved by applying concurrent rewrite rules to the current configuration. There are three 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. There are also rules for suspension of the active process with its local variables.
- *Rules that activate pending processes:* When *Pr* is empty, pending processes may be activated. When a pending process is loaded, the local variables are changed.
- *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.

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.

5.1 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 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 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.

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 internal process queue *PrQ* of the object. The rewrite rule for this transition can be expressed as follows, ignoring irrelevant attributes in the style of Full Maude [11]:

```
rl [receivecall] :
< O : Id | Cl: C, PrQ: W >  < q( O ) : QId | Ev: Q invoc(N, O', O, M, I) > < C : Cl | Mtds: MT >
    =>
< O : Id | Cl: C, PrQ: ( get(M, MT, (N O' I)) ) W > < q( O ) : QId | Ev: Q > < C : Cl | > .
```

The auxiliary function *get* fetches method *M* in the method multiset *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 program statements *Pr*, since waiting actively in this case leads to deadlock.

5.2 Guarded Commands

Creol has 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. For a single return value the rule becomes:

```
crl [returnguard] :
< O : Id | Pr: X ? ( J ) --> P, Lvar: L > < q( O ) : QId | Ev: Q comp(N, O, K) > =>
< O : Id | Pr: (J := K) ; P, Lvar: L > < q( O ) : QId | Ev: Q >
if (N == (val(X, L)) .
```

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

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.

```
crl [returnguard_notinqueue] :
< O : Id | Pr: X ? ( J ) --> P, PrQ: W, Lvar: L > < q( O ) : QId | Ev: Q > =>
< O : Id | Pr: empty, PrQ: W (X ? ( J ) --> P, L), Lvar: no > < q( O ) : QId | Ev: Q >
if not inqueue(val(X, L), Q) .
```

where the function *inqueue* checks whether the completion is in the message queue *Q*.

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 return value is matched to local or object attributes and the process is reactivated.

```
crl [return_guard_st] :
< O : Id | Pr: empty, PrQ: (X ? ( J ) --> P, L') W > < q( O ) : QId | Ev: Q comp(N, O, K) >
    =>
< O : Id | Pr: (J := K) ; P, PrQ: W, Lvar: L' > < q( O ) : QId | Ev: Q > if (N == (val(X, L') .
```

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

6 Program Execution in the Creol Interpreter

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 and search and model-checking abilities.

Although the operational semantics is highly non-deterministic, Maude is deterministic in its choice of which rule to apply to a given configuration. In order to improve the performance of the interpreter in this respect, we experiment with changes in the operational semantics. 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, the philosophers would only *think* after 10 000 rewrites. This means that although the suspended instance of *digest* is enabled, it is not executed.

In order to better control scheduling, we can change the representation of the internal process queue from a multiset to a list. This gives us explicit control of the order in which the interpreter selects processes for guard evaluation. The change is easily implemented by removing the commutative property of *PrQ*. The rules for activation of pending processes are modified either to support first-in-first-out (FIFO) scheduling or round robin scheduling. Experiments show that the philosophers will then *think*, *digest*, and *eat*, repeatedly and equally often, with only minor differences between the two scheduling methods (for this example).

Guards may be given different *priorities* for the selection of enabled processes from the internal process queue. A lower priority can for instance be assigned to *wait* than to the other guards by adding an extra process queue to the definition of Creol objects, changing the rule which deals with *wait* guards, and adding a rule which activates a process in the low priority queue if no high priority process is enabled. This allows a more fine-grained programming control of the object's activity (for further details, see [7]). Experimenting with priorities on the external queue is also most relevant, in order to allow e.g. system overrides.

By executing the operational semantics, Maude may be used as a tool for program analysis. Maude's search facilities can be employed to look for a specific configuration or a configuration which satisfies a given condition. For example a deadlock configuration of the example can be detected, where all philosophers are hungry and all chopsticks are taken. This occurs when each of the philosophers answers a request for a chopstick from the right neighbor, while waiting for the left neighbor to answer its request.

7 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 [21] and Java [15]. 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 [8], 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 are found in languages such as ABCL [23] and ConcurrentSmalltalk [22], as well as in Eiffel// [9] and CJava [12]. The proposed asynchronous method calls are similar to future variables and nested processor release points further extend this view of asynchrony.

Maude's inherent object concept [11,19] represents an object's state as a subconfiguration, as we have done in this paper, but 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.

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 the object model's organization of communication in terms of method calls (with return values). 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 proposes asynchronous method calls and nested processor release points in method bodies for this purpose. The approach improves on the efficiency gained by future variables and allows implicit control of interleaved intra-object concurrency between invoked methods. Active and reactive behavior in an object is thereby easily combined. The proposed interleaving of method executions is more flexible than serialized methods, while maintaining the ease of code verification lost for non-serialized methods. In fact, it suffices that class invariants hold at processor release points. A detailed investigation of reasoning control and of inheritance in this setting remains, along with further work with priorities on messages and guards, possibly combined with a time-out mechanism.

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, *1st Intl. Conf. on Formal Methods for Open Object-based Distributed Systems (FMOODS'96)*, pages 135–153, 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, 1987.

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

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

[7] M. Arnestad. En abstrakt maskin for Creol i Maude. Master's thesis, Department of informatics, University of Oslo, 2003. In Norwegian.

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

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

[10] 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.

[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 International Conference on Object Oriented Information Systems (OOIS'97)*, pages 504–514, London, 1997. Springer.

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

[14] 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.

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

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

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

[18] 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: Dedicated to the Memory of Ole-Johan Dahl*, LNCS 2635. Springer, 2003. To appear.

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

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

[21] 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.

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

[23] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System.* The MIT Press, 1990.