Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 215 (2008) 151–169
www.elsevier.com/locate/entcs

An Open System Operational Semantics for an Object-Oriented and Component-Based Language

Jasmin Christian Blanchette and Olaf Owe [1]

Department of Informatics
University of Oslo
Oslo, Norway
{jasmincb,olaf}@ifi.uio.no

Abstract

Object orientation and component-based development have both proven useful for the elaboration of open distributed systems. These paradigms are offered by the Creol language. Creol objects are concurrent, each with its own virtual processor and internal process control, and communicate using asynchronous (non-blocking) method calls. This provides the efficiency of message passing systems, while keeping the structuring benefits of methods and object-oriented programming. Conditional processor release points provide a high-level synchronization mechanism based on passive waiting that allows us to combine active and reactive behavior. A Creol component can be a single (concurrent) object or a collection of objects, together with a number of interfaces, and cointerfaces, defining the provided and required interaction and semantic behavior. Creol's semantics is defined formally using operational semantics and Hoare logic. An operational semantics lets us simulate an entire system, where all components are known in advance; in contrast, Hoare logic, together with class invariants and communication histories, lets us reason locally about a method body, without needing access to the implementations of the other classes. To bridge the gap between these two semantics, we introduce a history-based operational semantics for open systems. This new semantics can be used as an intermediate step for proving that Creol's Hoare logic is sound and complete with respect to the language's operational semantics. The approach can easily be adapted to other component-based languages where communication is done by message passing or by method interaction.

Keywords: Operational semantics, open distributed systems, communication histories, object orientation.

[1] This work is in the context of the EU project IST-33826 CREDO (http://credo.cwi.nl/).

doi:10.1016/j.entcs.2008.06.026

1 Introduction

Component-based system design directly supports the role of autonomous objects in distributed architectures. This useful paradigm can be combined with the structuring mechanisms of object orientation, as done within the Creol framework [11,12,13,14,15]. Creol objects are concurrent, each with its own virtual processor and internal process control, and communicate using asynchronous (non-blocking) method calls. This provides the efficiency of message passing systems, while keeping the structuring benefits of methods and object-oriented programming, notably late binding and inheritance. A Creol object, together with its interfaces, constitutes an autonomous unit that can act and react in a distributed setting. More complex components are formed by combining several concurrent objects together and defining interfaces that describe and control the component's visible behavior. Creol's notion of cointerface allows the specification of required and provided interfaces.

The goal of Creol is to develop a formal framework for reasoning about dynamic and reflective modifications in open distributed systems, where objects may be dispersed geographically, ensuring reliability and correctness of the overall system. The Creol language is high-level, imperative, and object-oriented. The language's semantics is defined formally using a small-step operational semantics expressed in rewriting logic [12]. This semantics forms the core of the Creol interpreter, which is written in Maude [7]. Using Maude's extensible rewrite strategies, its search capabilities, and its model checker, we can test Creol programs in various ways [13,14].

The Creol interpreter allows us to simulate a closed distributed system, where all the initial components are known in advance. On the other hand, it does not let us execute a component without providing and implementing an environment with which it can interact. At the reasoning level, this limitation is addressed by the Hoare logic developed by Dovland et al. [11]. The Hoare logic allows us to prove that an invariant holds for a given class. A system-wide invariant can be constructed from the class invariants using a compositional rule. The invariants may refer to a mythical communication history that records the object creations and method calls that have taken place in the system [8].

In this paper, we introduce an "open system" operational semantics that incorporates the class invariants and the communication history that characterize Creol's Hoare logic. One benefit of this approach is that it moves these techniques from the syntax-driven world of Hoare logic to the more fundamental semantics level. We then have the full power of mathematics and of formal tools like Maude at our disposal to analyze individual Creol classes.

Once we have an open system operational semantics, we can use it as a stepping stone towards the development of a Hoare logic. For Creol, where a Hoare logic already exists, it could be used to prove that the Hoare logic is sound and complete with respect to Creol's reference interpreter. The proof would proceed in two steps: (1) Prove that the closed system semantics and the open system semantics are equivalent, modulo the way they represent the environment. (2) Prove that the Hoare logic is sound and complete with respect to the open system semantics.

We focus on the class aspect of the Creol language, and limit ourselves to the basic communication and synchronization model of Creol, omitting the notions of interface, inheritance, self-reentrance, and dynamic update, as well as typing and specification. The approach can then easily be adapted to other languages where communication is done by message passing or by method interaction. We assume throughout that Creol programs are syntactically correct and well-typed.

The rest of this paper is organized as follows: Section 2 reviews the syntax of the main Creol statements. Section 3 presents a closed system operational semantics for the core language. Section 4 introduces the open system semantics and connects it to the closed semantics and to Hoare logic. Section 5 considers related work. Finally, Section 6 summarizes the paper.

2 The Creol Language

Creol is a strongly-typed object-oriented language that supports interfaces, inheritance, and polymorphism. Classes are the fundamental structuring unit, and all interaction between objects occurs through method calls. Each object executes on its own virtual processor, leading to increased parallelism in a distributed system. Classes are equipped with class parameters, as in Simula.

Objects are uniquely identified and communicate using *asynchronous method calls*. When an object A calls a method m of an object B, it first sends an invocation message to B along with arguments. Method m executes on B's processor and sends a reply to A once it has finished executing, with return values. Object A may continue executing while waiting for B's reply. Object identities may be passed around, and thanks to Creol's interface concept, method calls are type-safe [15]. In an object-oriented system, asynchronous method calls arguably offer a more natural interaction model than shared variables and message passing, while avoiding the delays associated with synchronous method calls [12].

The other main distinguishing feature of Creol is its reliance on explicit *processor release points*. Since there is only one processor per object, at most

one method m may execute at a given time for a given object; any other method invocations must wait until m finishes or releases the processor using the **await** statement. This ensures that while a method is active, no other processes can access the object's attributes, leading to a programming and reasoning style reminiscent of monitors [5].

The syntax of the Creol statements relies on a few basic syntactic entities. The set *Ident*, with typical elements c, l, m, x, y, consists of alphanumeric tokens that start with a letter, excluding keywords. The set *BExp*, with typical element B, consists of Boolean expressions such as $i \geq n$. The set *Exp*, with typical element e, consists of expressions of any type, including Boolean expressions, arithmetic expressions, and object references. The keyword **self** refers to the current object, and the implicit parameter **caller** identifies the caller of a method, allowing type-safe call-backs. The set *Guard*, with typical element g, includes *BExp* and otherwise contains the reply guard $l?$, the release guard **wait**, and the conjunction $g_1 \,\&\, g_2$.

The set *Stmt* of statements, with typical element S, contains these constructs:

$x := e$	assignment
$x :=$ **new** $c(e_1, \ldots, e_n)$	object creation
$l!x.m(e_1, \ldots, e_n)$	asynchronous invocation
$l?(y_1, \ldots, y_p)$	asynchronous reply
await g	conditional wait
if B **then** S_1 **else** S_2 **fi**	conditional statement
return $e_1, \ldots, e_n$	return statement
$S_1; S_2$	sequential composition

The object creation statement creates a new instance of class c. The expressions $e_1, \ldots, e_n$ are assigned to class parameters. If the class has a parameterless *run* method, this method executes immediately.

Asynchronous method calls consist of an invocation and a reply. The invocation can be seen as a message from the caller to the called method, with arguments corresponding to the method's input parameters. The reply is a message from the called method, containing the return values. The label l uniquely identifies the method call. For convenience, Creol also provides the classic synchronous (blocking) method call $x.m(e_1, \ldots, e_n; y_1, \ldots, y_p)$ as an abbreviation for $t!x.m(e_1, \ldots, e_n); t?(y_1, \ldots, y_p)$, where t is a fresh label name. Here each e_i acts as an actual input parameter to the method, and each y_j acts as an actual output parameter.

The statement **await** g releases the processor if the guard g evaluates to false and reacquires it at some later time when g is true. The guard $l?$

evaluates to true if and only if a reply for the asynchronous call identified by l has arrived. The **wait** guard evaluates to false the first time it is encountered, resulting in a processor release, and evaluates to true from then on.

The conditional statement, the return statement, sequential composition, and assignment behave essentially like their Java equivalents.

Example 2.1 The following code initiates two asynchronous calls, releases the processor while waiting for the replies, and retrieves the return values:

var *result1* : **int**, *result2* : **int**;
l1!*server.request*(); *l2*!*server.request*();
await *l1*? & *l2*?;
l1?(*result1*); *l2*?(*result2*)

Without the **await** statement, the program would block on the reply statements *l1*?(*result1*) and *l2*?(*result2*) until the method invocations have terminated.

3 An Operational Semantics for Creol

The operational semantics of Creol is defined using rewriting logic (RL) [12], which can be seen as a generalization of structural operational semantics [17]. A rewrite theory is a triple $\mathcal{R} = (\Sigma, E, R)$, where the signature Σ defines the function symbols of the language, E defines equations between terms, and R is a set of rewrite rules. When modeling computational systems, we represent a state configuration by a multiset of terms of given types. These types are specified algebraically in the equational logic (Σ, E), the functional sublanguage of RL.

The dynamic behavior of a system is expressed by rewrite rules, which describe how a part of a configuration can evolve in one transition step. A rule $p \longrightarrow q \ [\textbf{if } c]$ allows an instance of the pattern p to evolve into the corresponding instance of the pattern q if the (optional) side condition c is met. Rewrite rules are applied modulo E to complete terms or to subterms. Rules can be applied simultaneously on non-overlapping subterms; as a result, RL is implicitly concurrent.

The operational semantics for Creol consists of 11 rewrite rules that model concurrent execution, object creation, and inter-object communication. It also relies on equations to perform auxiliary tasks. The rewrite rules have the form

$$\mathit{subconfiguration}_1 \longrightarrow \mathit{subconfiguration}_2 \ [\textbf{if } \mathit{condition}]$$

where $\mathit{subconfiguration}_1$ is a subset of the current configuration. In our setting, a configuration is a multiset of Creol objects, Creol classes, and messages

reflecting either method invocations or replies. Typically, each subconfiguration consists of a single object, and possibly a message, reflecting that Creol objects are autonomous.

In a system configuration, Creol objects are represented by terms of the form

$$\big\langle o : c \;\big|\; \text{Pr: } S,\ \text{LVar: } \beta,\ \text{Att: } \alpha,\ \text{PrQ: } P,\ \text{MsgQ: } Q,\ \text{LabCnt: } k \big\rangle,$$

where $o \in \mathit{OId}$ is a unique object identity, c is the object's class, S is the active process's code, $\beta \in \mathit{State}$ is the active process's local variables, $\alpha \in \mathit{State}$ is the current state of the object's attributes, $P \in \mathcal{P}(\mathit{State} \times \mathit{Stmt})$ is a queue of suspended processes, $Q \in \mathcal{P}(\mathit{Msg})$ is the incoming message queue, and $k \in \mathbb{N}$ is a counter used to generate unique label values for asynchronous calls.

The set State, with typical elements σ, α, β, consists of mappings from variables to values. For example, $[x \mapsto 1][y \mapsto 2]$ denotes the state in which $x = 1$ and $y = 2$. The concatenation $\alpha\beta$ of two states α and β gives precedence to β for variables defined by both. The function $\{\bar{e}\}_\sigma$ returns the value of an expression list $\bar{e}$ in a state σ. The notation $\bar{x}$ stands for the comma-separated list $x_1, \ldots, x_n$. The empty list is written ϵ. The set Value, with typical elements v, w, includes the Boolean constants **true** and **false**, numeric constants, and object identities.

Creol classes are represented by terms of the form

$$\big\langle c : \text{Class} \;\big|\; \text{Param: } \bar{x},\ \text{Att: } \alpha,\ \text{Mtd: } M,\ \text{ObjCnt: } n \big\rangle,$$

where c is the class name, $\bar{x}$ is the list of class parameters, α is the list of class attributes with initial values, $M \in \mathcal{P}(\mathit{Mtd})$ is a set of methods, and $n \in \mathbb{N}$ is a counter used to generate unique object identities.

Creol objects interact by exchanging messages. *Invocation messages* have the form $\mathit{Invoke}(o, k, m, \bar{v})$, where o is the calling object, k is the sequence number (label value) associated with the method call, m is the called method, and $\bar{v}$ is a list of input arguments to m. *Reply messages* have the form $\mathit{Reply}(k, \bar{v})$, where k is the sequence number for the method call and $\bar{v}$ is a list of return values. When messages are passed around, the receiver object o' is specified by appending **to** o'.

Rewrite Rule R1 (Assignment)

$$\begin{aligned}
&\big\langle o : c \;\big|\; \text{Pr: } x := e;\ S,\ \text{LVar: } \beta,\ \text{Att: } \alpha \big\rangle \\
&\quad\longrightarrow \\
&\textbf{if } x \in \beta \textbf{ then } \big\langle o : c \;\big|\; \text{Pr: } S,\ \text{LVar: } \beta[x \mapsto \{e\}_{\alpha\beta}],\ \text{Att: } \alpha \big\rangle \\
&\qquad\qquad \textbf{else } \big\langle o : c \;\big|\; \text{Pr: } S,\ \text{LVar: } \beta,\ \text{Att: } \alpha[x \mapsto \{e\}_{\alpha\beta}] \big\rangle \textbf{ fi}
\end{aligned}$$

The assignment statement evaluates the expression e in the compound state $\alpha\beta$ and stores that value in x. If x is a local variable, we update the local

state β; otherwise, we update the object state α. In the style of Full Maude [7], the fields that are not used by a rule are omitted in the rule.

Rewrite Rule R2 (If Statement)

$$\begin{array}{l}\langle o : c \mid \text{Pr: } \textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}; S,\ \text{LVar: } \beta,\ \text{Att: } \alpha\rangle \\ \longrightarrow \\ \textbf{if } \{B\}_{\alpha\beta} \textbf{ then } \langle o : c \mid \text{Pr: } S_1; S,\ \text{LVar: } \beta,\ \text{Att: } \alpha\rangle \\ \qquad\qquad \textbf{else } \langle o : c \mid \text{Pr: } S_2; S,\ \text{LVar: } \beta,\ \text{Att: } \alpha\rangle \textbf{ fi}\end{array}$$

If B evaluates to true, the Creol **if** statement expands to its **then** branch; otherwise, it expands to its **else** branch. Notice that the first **if** construct in the rule above is a Creol statement, while the second **if** is a conditional expression in RL.

Rewrite Rule R3 (Guard Crossing)

$$\begin{array}{l}\langle o : c \mid \text{Pr: } \textbf{await } g; S,\ \text{LVar: } \beta,\ \text{Att: } \alpha,\ \text{MsgQ: } Q\rangle \\ \longrightarrow \\ \langle o : c \mid \text{Pr: } S,\ \text{LVar: } \beta,\ \text{Att: } \alpha,\ \text{MsgQ: } Q\rangle \\ \textbf{if } \mathit{enabled}(g, \alpha\beta, Q)\end{array}$$

An **await** statement whose guard evaluates to true is simply skipped. The *enabled* predicate is defined recursively using equations:

$$\begin{array}{ll}\mathit{enabled}(B, \sigma, Q) & \triangleq \{B\}_\sigma \\ \mathit{enabled}(\textbf{wait}, \sigma, Q) & \triangleq \textbf{false} \\ \mathit{enabled}(l?, \sigma, \emptyset) & \triangleq \textbf{false} \\ \mathit{enabled}(l?, \sigma, Q \cup \{\mathit{Invoke}(\ldots)\}) & \triangleq \mathit{enabled}(l?, \sigma, Q) \\ \mathit{enabled}(l?, \sigma, Q \cup \{\mathit{Reply}(k, \bar{v})\}) & \triangleq k = \{l\}_\sigma \vee \mathit{enabled}(l?, \sigma, Q) \\ \mathit{enabled}(g_1 \,\&\, g_2, \sigma, Q) & \triangleq \mathit{enabled}(g_1, \sigma, Q) \wedge \mathit{enabled}(g_2, \sigma, Q)\end{array}$$

Rewrite Rule R4 (Process Suspension)

$$\begin{array}{l}\langle o : c \mid \text{Pr: } \textbf{await } g; S,\ \text{LVar: } \beta,\ \text{Att: } \alpha,\ \text{PrQ: } P,\ \text{MsgQ: } Q\rangle \\ \longrightarrow \\ \langle o : c \mid \text{Pr: } \epsilon,\ \text{LVar: } \emptyset,\ \text{Att: } \alpha,\ \text{PrQ: } P \cup \\ \{\langle \textbf{await } \mathit{clearWait}(g); S, \beta\rangle\},\ \text{MsgQ: } Q\rangle \\ \textbf{if } \neg\mathit{enabled}(g, \alpha\beta, Q)\end{array}$$

If the next statement to execute is an **await** statement whose guard is not enabled, the active process is put on the process queue, together with its local variables. The *clearWait* auxiliary function replaces any occurrence of **wait** in the guard with **true**, so that a process that was suspended because of **wait** may become active again.

Rewrite Rule R5 (Process Activation)

$$\begin{array}{l}\langle o : c \mid \text{Pr: } \epsilon,\ \text{LVar: } \beta,\ \text{Att: } \alpha,\ \text{PrQ: } \{\langle S', \beta'\rangle\} \cup P,\ \text{MsgQ: } Q\rangle \\ \longrightarrow \\ \langle o : c \mid \text{Pr: } S',\ \text{LVar: } \beta',\ \text{Att: } \alpha,\ \text{PrQ: } P,\ \text{MsgQ: } Q\rangle \\ \textbf{if } \mathit{ready}(S', \alpha\beta', Q)\end{array}$$

After a process has been suspended, other processes that are ready may be activated. A reply statement is *ready* only if the reply message has arrived, and an **await** statement only if the guard is enabled, while other statements are always ready. The list $S_1; \ldots; S_n$ is ready whenever S_1 is ready. Maude's facilities for associative, commutative, and identity (ACI) matching allow $\{\langle S', \beta'\rangle\}$ to match any process in PrQ.

Rewrite Rule R6 (Object Creation)

$$\begin{array}{l}\langle o : c \mid \text{Pr: } y := \textbf{new } c'(\bar{e});\ S,\ \text{LVar: } \beta,\ \text{Att: } \alpha\rangle \\ \langle c' : \text{Class} \mid \text{Param: } \bar{x},\ \text{Att: } \alpha',\ \text{ObjCnt: } n\rangle \\ \longrightarrow \\ \langle o : c \mid \text{Pr: } y := c'\#n;\ S,\ \text{LVar: } \beta,\ \text{Att: } \alpha\rangle \\ \langle c' : \text{Class} \mid \text{Param: } \bar{x},\ \text{Att: } \alpha',\ \text{ObjCnt: } n+1\rangle \\ \langle c'\#n : c' \mid \text{Pr: } \textbf{self}.run(),\ \text{LVar: } \emptyset,\ \text{Att: } \alpha'[\bar{x} \mapsto \{\bar{e}\}_{\alpha\beta}][\textbf{self} \mapsto c'\#n], \\ \qquad\qquad \text{PrQ: } \emptyset,\ \text{MsgQ: } \emptyset,\ \text{LabCnt: } 0\rangle\end{array}$$

A **new** statement creates an instance of a given class. The new object's identity is $c'\#n$, where c' is the class name and n a sequence number that identifies this object among c' instances. The new object is set up with the class parameters and attributes of class c'. The Pr field is initialized with a synchronous call to *run* to launch the object's active behavior. In the parent object, creating an object is viewed as an assignment of $c'\#n$ to a variable. In the instantiated class, the object counter is incremented to ensure that object identities remain unique.

Rewrite Rule R7 (Asynchronous Invocation)

$$\begin{array}{l}\langle o : c \mid \text{Pr: } l!x.m(\bar{e});\ S,\ \text{LVar: } \beta,\ \text{Att: } \alpha,\ \text{LabCnt: } k\rangle \\ \longrightarrow \\ \langle o : c \mid \text{Pr: } S,\ \text{LVar: } \beta[l \mapsto k],\ \text{Att: } \alpha,\ \text{LabCnt: } k+1\rangle \\ \mathit{Invoke}(o, k, m, \{\bar{e}\}_{\alpha\beta})\ \textbf{to}\ \{x\}_{\alpha\beta}\end{array}$$

Asynchronous method calls lead to the creation of an invocation message that is sent to the called object. Each method call originated by a given object is identified by a unique sequence number k. This number is assigned to the local variable l, which corresponds to the label l. A call is uniquely identified by the pair (o, k).

Rewrite Rule R8 (Transport of Message)

$$\begin{array}{l}\langle o : c \mid \text{MsgQ: } Q\rangle \\ \mu \textbf{ to } o \\ \longrightarrow \\ \langle o : c \mid \text{MsgQ: } Q \cup \{\mu\}\rangle\end{array}$$

At some unspecified point after an invocation or reply message μ has been sent, the recipient receives it. Rewrite Rules R7 and R8 allow *message overtaking*—messages might arrive in a different order than they were sent. Again, ACI matching applies.

Rewrite Rule R9 (Method Binding)

$$\begin{array}{l}\langle o : c \mid \text{PrQ: } P,\ \text{MsgQ: } \{\mathit{Invoke}(o', k, m, \bar{v})\} \cup Q\rangle \\ \langle c' : \text{Class} \mid \text{Mtd: } M\rangle \\ \longrightarrow \\ \langle o : c \mid \text{PrQ: } P \cup \{\mathit{bind}(o', k, m, \bar{v}, M)\},\ \text{MsgQ: } Q\rangle \\ \langle c' : \text{Class} \mid \text{Mtd: } M\rangle\end{array}$$

A pending invocation message gives rise to a new pending process. The *bind* function fetches method m from the method set M and returns a $\langle S, \beta\rangle$ pair storing the code and initial state of the process. The rule does not consider base classes; method binding with multiple inheritance in Creol is treated in Johnsen et al. [15].

Rewrite Rule R10 (Method Return)

$$\begin{array}{l}\langle o : c \mid \text{Pr: } \textbf{return } \bar{e};\ S,\ \text{LVar: } \beta\rangle \\ \longrightarrow \\ \langle o : c \mid \text{Pr: } \epsilon,\ \text{LVar: } \beta\rangle \\ \mathit{Reply}(\{\textbf{label}\}_\beta, \{\bar{e}\}_\beta) \textbf{ to } \{\textbf{caller}\}_\beta\end{array}$$

The **return** statement sends a reply message to the caller along with the values of the output parameters. Reply messages are eventually received by the calling object and put into its incoming message queue by Rewrite Rule R8.

Rewrite Rule R11 (Asynchronous Reply)

$$\begin{array}{l}\langle o : c \mid \text{Pr: } l?(\bar{y});\ S,\ \text{LVar: } \beta[l \mapsto k],\ \text{MsgQ: } \{\mathit{Reply}(k, \bar{v})\} \cup Q\rangle \\ \longrightarrow \\ \langle o : c \mid \text{Pr: } \bar{y} := \bar{v};\ S,\ \text{LVar: } \beta,\ \text{MsgQ: } Q\rangle\end{array}$$

A statement $l?(\bar{y})$ may proceed only if the corresponding reply message has arrived, which we can find out by looking for a reply message numbered k, where k is l's value. The output parameter values stored in the reply are assigned to $\bar{y}$.

4 An Alternative Semantics for Open Systems

While the operational semantics presented in the previous section correctly captures the behavior of a closed system, it doesn't directly cater for open systems, in which objects don't have access to each other's implementations. This means that we have no satisfactory way to simulate the activity of a single process taken in isolation once we abstract away the environment with which it communicates (the other processes executing in the same object and the other objects in the system). It also means that there's no direct way to derive a Hoare logic from the operational semantics.

4.1 Definition of the Open System Semantics

In this section, we will define an alternative version of Creol's operational semantics that focuses on the execution of a single process, mimicking a Hoare logic. The new "open system" operational semantics uses a communication history to abstract away the environment. This semantics reuses Rewrite Rules R1–R3 and R11 from the previous section, because these rules involve no interaction between objects or between processes within an object. Rewrite Rules R4–R10 are replaced with a new set of rules that operate on the history. The table below compares the closed system semantics of Section 3 with the open system semantics introduced here.

	Closed System	Open System
Process Suspension	R4	R4′
Process Activation	R5	
Object Creation	R6	R6′
Asynchronous Invocation	R7	R7′
Transport of Message	R8	–
Method Binding	R9	–
Method Return	R10	R10′
Environment Activity	–	R12′

From Hoare logic we borrow the concept of a *communication history* [8]. The communication history records the creation of objects and the messages that are exchanged between objects in a distributed system. More formally, a history is a finite sequence of *communication events*:

$[o{\rightarrow}o'.\mathbf{new}\ c(\bar{v})]$	object creation
$[o{\rightarrow}o'.m(\bar{v})]^k$	asynchronous invocation
$[o{\leftarrow}o'.m(\bar{v};\bar{w})]^k$	asynchronous reply

For invocation events, $\bar{v}$ stores the values passed to the method; for reply events, $\bar{w}$ stores the return values. For both types of event, k is the sequence number of the method call. For object creation events, $\bar{v}$ stores the actual class parameters. The history represents a snapshot of the system's execution at a given point and is therefore finite. When designing or analyzing a complex system, we often want to know the possible histories for that system, to deduce safety properties about it [11].

In the new semantics, Creol objects have the form

$$\langle o : c \mid \text{Pr: } S,\ \text{LVar: } \beta,\ \text{Att: } \alpha,\ \text{MsgQ: } Q,\ \text{LabCnt: } k \rangle.$$

Since we concentrate on one process's execution, we now omit the PrQ field. On the other hand, the object's attribute set now includes a distinguished $\mathcal{H}$ attribute that stores the system's communication history. The history could also have been stored in a separate field, or as a separate object, but making it an attribute will simplify the definition of Hoare logic formulas. The MsgQ field is redundant now that we record the history; we keep it because Rewrite Rules R3 and R11 rely on it. Also, some of the rewrite rules will refer to the class invariant I_c, which is expected to hold at startup and whenever the processor is released. This invariant is derived from the semantic specifications supplied by the programmer in the class declaration for c and in the interface declarations for the interfaces implemented by c.

Rewrite Rule R6′ (Object Creation)

$$\langle o : c \mid \text{Pr: } y := \textbf{new } c'(\bar{e});\ S,\ \text{LVar: } \beta,\ \text{Att: } \alpha \rangle$$
$$\longrightarrow$$
$$\langle o : c \mid \text{Pr: } y := o';\ S,\ \text{LVar: } \beta, \text{Att: } \alpha[\mathcal{H} \mapsto \{\mathcal{H}\}_\alpha \frown [o \rightarrow o'.\textbf{new } c'(\{\bar{e}\}_{\alpha\beta})]] \rangle$$
$$\textbf{if } o' \notin \mathit{objectIds}(\{\mathcal{H}\}_\alpha)$$

With the open system semantics, an object creation statement allocates a fresh object identity o' and extends the history $\mathcal{H}$ with an object creation event. The new object is now part of the implicit environment embodied by $\mathcal{H}$.

Rewrite Rule R4′ (Process Suspension and Reactivation)

$$\langle o : c \mid \text{Pr: } \textbf{await } g;\ S,\ \text{LVar: } \beta,\ \text{Att: } \alpha,\ \text{MsgQ: } Q,\ \text{LabCnt: } k \rangle$$
$$\longrightarrow$$
$$\langle o : c \mid \text{Pr: } S,\ \text{LVar: } \beta,\ \text{Att: } \alpha',\ \text{MsgQ: } \mathit{replies}(\{\mathcal{H}\}_{\alpha'}, o),\ \text{LabCnt: } \mathit{nextLabel}(\{\mathcal{H}\}_{\alpha'}, o) \rangle$$
$$\textbf{if } \neg \mathit{enabled}(g, \alpha\beta, Q) \wedge \mathit{release}(I_c, \alpha, \alpha', \beta) \wedge \mathit{enabled}(\mathit{clearWait}(g), \alpha'\beta, \mathit{replies}(\{\mathcal{H}\}_{\alpha'}, o))$$

If the next statement is **await** g and the guard g is not enabled, the process is suspended and wakes up in a different state in which the guard is enabled. The

class attributes, including the history $\mathcal{H}$, might have changed in the meantime; this is modeled by replacing α with α'. In addition, the MsgQ and LabCnt fields are updated to reflect the new history.

The function $\mathit{replies}(h, o)$ returns a set of pending reply messages corresponding to the pending replies encoded in the history h. The constraint $\mathit{release}(I_c, \alpha, \alpha', \beta)$ restricts the values that the attributes α' may take. It is defined as follows:

$$\begin{aligned}\mathit{release}(I_c, \alpha, \alpha', \beta) \triangleq\ & \{\mathcal{H}\}_\alpha \preceq \{\mathcal{H}\}_{\alpha'} \wedge \mathit{wf}(\{\mathcal{H}\}_{\alpha'}) \wedge \{I_c\}_\alpha \Rightarrow \{I\}_{\alpha'} \\ & \wedge \mathit{pending}(\{\mathcal{H}\}_{\alpha'}, \{\mathbf{caller}\}_\beta, \{\mathbf{self}\}_\alpha, \{\mathbf{label}\}_\beta)\end{aligned}$$

Informally, the new history $\{\mathcal{H}\}_{\alpha'}$ must be an extension of the original history $\{\mathcal{H}\}_\alpha$, it must be well-formed, the class invariant I_c should still hold if it held before the release, and the call that released the processor should still be pending after the processor release. The well-formedness predicate is defined below:

$$\begin{aligned}&\mathit{wf}(\epsilon) && \triangleq \mathbf{true} \\ &\mathit{wf}(h \frown [o{\rightarrow}o'.\mathbf{new}\ c(\bar{v})]) && \triangleq \mathit{wf}(h) \wedge o' \notin \mathit{objectIds}(h) \\ &\mathit{wf}(h \frown [o{\rightarrow}o'.m(\bar{v})]^k) && \triangleq \mathit{wf}(h) \wedge \forall o'', m', \bar{v}'.\, [o{\rightarrow}o''.m'(\bar{v}')]^k \notin h \\ &\mathit{wf}(h \frown [o{\leftarrow}o'.m(\bar{v}; \bar{w})]^k) && \triangleq \mathit{wf}(h) \wedge \mathit{pending}(h, o, o', k)\end{aligned}$$

A communication history is well-formed if new objects have unique identifiers and if method invocations and replies match. We also require that a pair (o, k) uniquely identifies a method call originating from an object o. Well-formedness expresses program-independent properties of the history. If we omitted it, the programmer could compensate by embedding well-formedness in the class invariant I_c.

Rewrite Rule R7′ (Asynchronous Invocation)

$$\begin{aligned}&\big\langle o : c \;\big|\; \text{Pr: } l!x.m(\bar{e});\ S,\ \text{LVar: } \beta,\ \text{Att: } \alpha,\ \text{LabCnt: } k \big\rangle \\ &\xrightarrow{\qquad} \\ &\big\langle o : c \;\big|\; \text{Pr: } S,\ \text{LVar: } \beta[l \mapsto k],\ \text{Att: } \alpha[\mathcal{H} \mapsto \{\mathcal{H}\}_\alpha \frown [o{\rightarrow}\{x\}_{\alpha\beta}.m(\{\bar{e}\}_{\alpha\beta})]^k], \\ &\qquad \text{LabCnt: } k+1 \big\rangle\end{aligned}$$

Asynchronous method calls lead to an extension of the history with a new invocation event. Similarly, returning from a method extends the history with a reply event:

Rewrite Rule R10′ (Method Return)

$$\begin{aligned}&\big\langle o : c \;\big|\; \text{Pr: } \mathbf{return}\ \bar{e};\ S,\ \text{LVar: } \beta,\ \text{Att: } \alpha \big\rangle \\ &\xrightarrow{\qquad} \\ &\big\langle o : c \;\big|\; \text{Pr: } \epsilon,\ \text{LVar: } \beta, \\ &\qquad \text{Att: } \alpha[\mathcal{H} \mapsto \{\mathcal{H}\}_\alpha \frown \mathit{replyEvent}(\{\mathcal{H}\}_\alpha, o, \{\mathbf{caller}\}_\beta, \{\mathbf{label}\}_\beta, \{\bar{e}\}_{\alpha\beta}) \big\rangle\end{aligned}$$

The auxiliary function *replyEvent* determines the reply event by inspecting the history. If $[o' \rightarrow o.m(\bar{v})]^k \in h$, then $replyEvent(h, o, o', k, \bar{w})$ is $[o' \leftarrow o.m(\bar{v}; \bar{w})]^k$.

Rewrite Rule R12′ (Environment Activity)

$$\langle o : c \mid \text{Pr: } S,\ \text{Att: } \alpha,\ \text{MsgQ: } Q \rangle$$
$$\longrightarrow$$
$$\langle o : c \mid \text{Pr: } S,\ \text{Att: } \alpha[\mathcal{H} \mapsto h],\ \text{MsgQ: } replies(h, o) \rangle$$
if $interleave(o, \alpha, h)$

Rewrite Rule R12′ lets us extend the history in a nondeterministic way with events originating from the environment at any point during the execution of a process. The new history h must abide by the following rules, expressed by the *interleave* predicate: The environment may only append events to the history, it must preserve the well-formedness of the history, and it may not produce events that o can produce. This is formalized as follows:

$$interleave(o, \alpha, h) \triangleq \{\mathcal{H}\}_\alpha \preceq h \wedge wf(h) \wedge \{\mathcal{H}\}_\alpha / out_o = h / out_o$$

In the above, out_o denotes the set of events that originate from o, and h/E denotes the longest subsequence of h that consists exclusively of events belonging to E.

Because some of the rules presented here use variables that do not occur in their left-hand side, they cannot be used directly to test or simulate a Creol component. One solution would be to alter the rewrite rules so that they accept user-supplied data along with the Creol program. An alternative is to define a custom evaluation strategy in Maude that instantiates the unbound variables using random data [14].

4.2 Example: An Internet Bank Account

We will consider a *NetBankAccount* class that models a simplistic Internet bank account. In a real-world scenario, the user would log into the Internet bank, perform some deposits and payments, and log out. The deposits and payments take place during the night, and if there is not enough money in the account, the payment is delayed. In Creol, this would be modeled using asynchronous calls:

$account :=$ **new** $NetBankAccount$;
$l1\,!\,account.deposit(50)$;
$l2\,!\,account.payBill(80)$;
$l3\,!\,account.deposit(50)$

Because method overtaking is allowed, the bank might receive the deposit

and payment requests in any order. Furthermore, to prevent the user from going overdrawn, the bank would first process the deposits, then pay the bill. The *NetBankAccount* class achieves synchronization using **await** and relies on Creol's implicit mutual exclusion for processes in the same object. The class declaration follows:

class *NetBankAccount*
begin
  var *balance* : **int** := 0
  op *deposit*(*amount* : **nat**) **is**
    balance := *balance* + *amount*;
    return
  op *payBill*(*amount* : **nat**) **is**
    await *balance* ≥ *amount*;
    balance := *balance* − *amount*;
    return
  spec $balance \geq 0 \wedge balance = sumDeposits(\mathcal{H}) - sumPayments(\mathcal{H})$
end

In the class declaration, the **spec** clause specifies an invariant that should hold initially and whenever the processor is released. Intuitively, *NetBankAccount* guarantees that the balance will always be nonnegative and that it always equals the difference between the deposits and the payments that have been performed so far. The *sumDeposits* and *sumPayments* functions are defined recursively on histories, by inspection of reply events. Here is the definition of *sumDeposits*:

$$\begin{array}{lll} sumDeposits(\epsilon) & \triangleq & 0 \\ sumDeposits(h \frown [o{\leftarrow}\mathbf{self}.deposit(a)]^k) & \triangleq & sumDeposits(h) + a \\ sumDeposits(h \frown \upsilon) & \triangleq & sumDeposits(h) \qquad \text{[otherwise]} \end{array}$$

Using the open system semantics, we can verify the class invariant. The invariant holds initially, because at that point the balance is 0 and $sumDeposits(\mathcal{H}) = sumPayments(\mathcal{H}) = 0$. We must prove that *deposit* and *payBill* preserve the invariant. Let us first verify *deposit*. We must consider an arbitrary *NetBankAccount* object in a state where the class invariant holds just before executing *deposit*'s body, and show that the invariant still holds when the method is finished. Let h_0 and b_0 be the initial values of $\mathcal{H}$ and *balance*, respectively, such that the invariant holds. Ignoring Rewrite Rule R12′ (Environment Activity), which has no impact on the invariant, we only need to consider one execution:

$$\begin{aligned}
&\langle o : c \mid \text{Pr: } balance := balance + amount;\ \mathbf{return}\ \epsilon,\\
&\qquad\quad \text{Att: } \alpha[\mathcal{H} \mapsto h_0][balance \mapsto b_0]\rangle\\
\xrightarrow{\text{R1}}\ &\langle o : c \mid \text{Pr: } \mathbf{return}\ \epsilon,\ \text{Att: } \alpha[\mathcal{H} \mapsto h_0][balance \mapsto b_0 + a_0]\rangle\\
\xrightarrow{\text{R10}'}\ &\langle o : c \mid \text{Pr: } \epsilon,\ \text{Att: } \alpha[\mathcal{H} \mapsto h_0 \frown [\mathbf{caller}{\leftarrow}o.deposit(a_0)]][balance \mapsto b_0 + a_0]\rangle
\end{aligned}$$

Clearly, if the invariant holds for $\mathcal{H} = h_0$ and $balance = b_0$, it also holds for $\mathcal{H} = h_0 \frown [\mathbf{caller}{\leftarrow}o.deposit(a_0)]$ and $balance = b_0 + a_0$.

Let's now turn to *payBill*. If there is enough money to perform the payment, the **await** is skipped and the reasoning is similar to what we did for *deposit*. Otherwise, there is too little money and the payment must wait, leading to this execution:

$$\begin{aligned}
&\langle o : c \mid \text{Pr: } \mathbf{await}\ balance \geq amount; balance := balance - amount; \mathbf{return}\ \epsilon,\\
&\qquad\quad \text{Att: } \alpha[\mathcal{H} \mapsto h_0][balance \mapsto b_0]\rangle\\
\\
\xrightarrow{\text{R4}'}\ &\langle o : c \mid \text{Pr: } \mathbf{await}\ balance \geq amount; balance := balance - amount; \mathbf{return}\ \epsilon,\\
&\qquad\quad \text{Att: } \alpha[\mathcal{H} \mapsto h_1][balance \mapsto b_1]\rangle\\
\\
\xrightarrow{\text{R3}}\ &\langle o : c \mid \text{Pr: } balance := balance - amount;\ \mathbf{return}\ \epsilon,\\
&\qquad\quad \text{Att: } \alpha[\mathcal{H} \mapsto h_1][balance \mapsto b_1]\rangle\\
\xrightarrow{\text{R1}}\ &\langle o : c \mid \text{Pr: } \mathbf{return}\ \epsilon,\ \text{Att: } \alpha[\mathcal{H} \mapsto h_1][balance \mapsto b_1 - a_0]\rangle\\
\xrightarrow{\text{R10}'}\ &\langle o : c \mid \text{Pr: } \epsilon, \text{Att: } \alpha[\mathcal{H} \mapsto h_1 \frown [\mathbf{caller}{\leftarrow}o.payBill(a_0)]][balance \mapsto\\
&b_1 - a_0]\rangle
\end{aligned}$$

Rewrite Rule R4′ suspends and reactivates the process. When the process is reactivated, $\mathcal{H}$ and *balance* might have changed; their new value is denoted h_1 and b_1, respectively. Furthermore, we may assume that the invariant holds, and from the **await** guard, we know that $balance \geq amount$, that is, $b_1 \geq a_0$. From there, it's easy to prove that the invariant holds at the end of the method's execution.

Because the open system semantics focuses on a single process executing in an unspecified environment, an open system configuration will always contain exactly one object executing one process. Dovland et al. [11] describe a method for composing objects, including restrictions on the class invariants to account for asynchronous communication, that can be used unchanged for our semantic setting.

4.3 Connection to the Closed System Semantics

The closed system and the open system operational semantics are fairly similar: Some rewrite rules are common to both semantics, and for the others

there is an almost one-to-one correspondence between the rules of the two semantics. This makes it easy to detect inconsistencies between them.

If we wanted to prove that the open system semantics is a safe approximation of the closed system semantics, we could proceed as follows: We assume that we have valid class invariants (with respect to the closed system semantics augmented by an implicit history [13]) for all the classes appearing in an arbitrary Creol program, and show that each possible closed system behavior is also possible in the open system semantics, proceeding by cases on the Creol statements.

To illustrate this, we will sketch the proof for **await** g. If g is enabled, R3 applies for both semantics, so there is nothing to prove. Otherwise, R4 moves the active process Π to PrQ, then other processes are allowed to run in the object, and finally R5 reactivates Π and removes the **await** statement from Pr. Activity in other objects can be interwoven into this sequence of rewrite rules. In the open semantics, the behavior of **await** g with g disabled is captured by R4′ alone. Like R4/R5, it removes **await** g from the beginning of the statement list, and it simulates activities in other objects by allowing nondeterministic extension of the history variable $\mathcal{H}$. The attributes are assigned random values to reflect activity within the object while the process was suspended, and the MsgQ and LabCnt fields are updated based on $\mathcal{H}$ to the values they would have had in the corresponding closed system configuration.

We must now show that the side conditions of R4′ are weak enough to model any possible behavior of R4/R5. (i) R4′ requires that g is initially disabled but that $\mathit{clearWait}(g)$ is enabled after Π has been reactivated. By inspecting R4 and R5, we can prove that this will always be the case in the closed semantics. (ii) R4′ specifies that the new history is an extension of the old history. This obviously holds for the implicit history of the closed semantics. (iii) R4′ requires the new history to be well-formed. This can be proved by induction on the length of a closed system computation. (iv) R4′ requires the invariant to hold when Π is reactivated if it held when it was suspended. This holds by hypothesis. (v) R4′ requires the call that initiated Π to be still pending. It suffices to observe that only Π could have sent the missing reply message, which cannot have happened since Π was suspended.

4.4 Connection to Hoare Logic

With the open system semantics in place, we can interpret Hoare logic formulas as follows: A partial correctness formulas $\{P\}\, S\, \{Q\}$ is valid if and only if the state $\alpha'\beta'$ satisfies the postcondition Q for all executions of the form

$$\langle o : c \mid \text{Pr: } S;\ S',\ \text{LVar: } \beta,\ \text{Att: } \alpha\rangle \xrightarrow{*} \langle o : c \mid \text{Pr: } S',\ \text{LVar: } \beta',\ \text{Att: } \alpha'\rangle$$

where the initial state $\alpha\beta$ satisfies the precondition P. Since the history $\mathcal{H}$ is stored in the object as an attribute, P and Q may refer to $\mathcal{H}$.

Consider the following Hoare axiom schema for **await wait**:

$$\{\forall h, \bar{a}.\, \mathit{releaseReq}(h, \bar{a}) \Rightarrow Q[h/\mathcal{H}][\bar{a}/\bar{\mathcal{A}}]\}\ \mathbf{await\ wait}\ \{Q\}$$

The *releaseReq* assertion is modeled after the *release* predicate from Section 4.1:

$$\begin{aligned}\mathit{releaseReq}(h, \bar{a}) \triangleq\ & \mathcal{H} \preceq h \wedge \mathit{wf}(h) \wedge I_c(\mathcal{H}, \bar{\mathcal{A}}) \Rightarrow I_c(h, \bar{a}) \\ & \wedge \mathit{pending}(h, \mathbf{caller}, \mathbf{self}, \mathbf{label})\end{aligned}$$

The relationship between the axiom schema for **await wait** and Rewrite Rule R4′ can be made more obvious by encoding the semantics of the **await wait** statement in terms of the following simultaneous random assignment statement [11]:

$$\mathcal{H}, \bar{\mathcal{A}} := \mathbf{some}\ h, \bar{a} : \mathit{releaseReq}(h, \bar{a})$$

This statement assigns arbitrary values $h, \bar{a}$ to the history $\mathcal{H}$ and the other mutable attributes $\bar{\mathcal{A}}$, such that the condition $\mathit{releaseReq}(h, \bar{a})$ is true. Clearly, the above random assignment is equivalent to Rewrite Rule R4′ (with **wait** as the guard).

The Hoare axiom schema for random assignment is $\{\forall \bar{y}.\, P \Rightarrow Q[\bar{y}/\bar{x}]\}$ $\bar{x} := \mathbf{some}\ \bar{y} : P\ \{Q\}$, for fresh $\bar{y}$. Using it, we derive the axiom schema given above for **await wait**. In general, to develop a history-based Hoare logic from a traditional closed system operational semantics, we follow these steps: (1) Specify an open system semantics that abstracts away the environment using a history. (2) Develop a Hoare logic for the language's sequential subset. (3) Reformulate the open semantics as an encoding in terms of the language's sequential subset augmented with random assignment. (4) Mechanically derive a Hoare logic from this encoding.

The Hoare logic is essentially a reformulation of the open system semantics at the syntactic level. The strength of the reasoning system depends on the strength of the class invariant, since the open system semantics relies on class invariants to determine the possible results of release points.

5 Related Work

The two main interaction models for distributed processes are remote method invocation (RMI) and message passing [3]. RMI is the approach adopted by Java and typically leads to unnecessary waiting in a distributed setting; moreover, Java's thread concept forces the programmer to choose between reduced parallelism (using `synchronized`) and shared-variable interference,

and makes reasoning highly complex [1]. Synchronous message passing also results in unnecessary delays. Asynchronous message passing, as popularized by the actor model [2], is very flexible but lacks the structure and discipline of object-oriented method calls; moreover, actors have no direct notion of inheritance or hierarchy. Creol's release points improve on the efficiency of future variables, found in several languages [6,18]. Johnsen et al. [12] provide a more thorough review of alternative communication models.

The open semantics introduced here is inspired by Dovland et al. [11], who devised an encoding of the Creol language in a nondeterministic sequential language called SEQ, from which they derived a Hoare logic, following the approach advocated by de Boer and Pierik [9]. Our presentation retains the history flavor of SEQ. Histories have been used before both to define the denotational semantics of a concurrent language [16] and to facilitate program verification [8]. The idea of recasting the notion of history from the syntactic level to the semantic level is inspired by de Roever et al. [10], who conduct their soundness and completeness proofs at the semantic level (in their case, on Floyd inductive assertion networks) and carry these proofs over to the syntactic level of Hoare logic.

6 Conclusion

The Creol language supports component and object orientation in a high-level and natural way by means of concurrent objects with processor release points and asynchronous methods calls. The language's operational semantics is defined using rewrite rules, which form the core of the language's interpreter.

In this paper, we introduced an "open system" operational semantics for Creol that defines the behavior of a single method execution seen in isolation, using a communication history to abstract away the environment. The semantics can be seen as the missing link between the Creol interpreter and Hoare logic, bringing the concept of a communication history to the semantic level. Because the open system semantics is expressed in rewriting logic, it is straightforward to detect inconsistencies with the interpreter, by comparing the rewrite rules. From the new semantics, we can easily derive a history-based Hoare logic that is sound and complete by construction. The construction of such a proof system would be significantly simpler than in Dovland et al. [11], since the communication history is explicitly captured by the semantics. The open semantics can also serve as a semantic foundation for studying language extensions, including different network models such as Reo [4].

We have shown in this paper how to construct an "open system" semantics from a more conventional "closed system" operational semantics. This

approach can easily be adapted to other component-based languages where communication is done by messages passing, method interaction, or both.

Acknowledgment

We want to thank Willem-Paul de Roever and Martin Steffen for valuable insights into the nature of semantics, as well as Marcel Kyas, Mark Summerfield, and the anonymous reviewers for suggesting textual improvements.

References

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

[2] Agha, G., I. A. Mason, S. F. Smith, and C. L. Talcott, *A foundation for actor computation*, J. Funct. Program. **7** (1997), 1–72.

[3] Andrews, G. R., "Foundations of Multithreaded, Parallel, and Distributed Programming," Addison-Wesley, 2000.

[4] Arbab, F., *Reo: A channel-based coordination model for component composition*, Math. Struct. Comp. Sci. **14** (2004), 329–366.

[5] Brinch Hansen, P., *The nucleus of a multiprogramming system*, Comm. ACM **13** (1970), 238–242.

[6] Caromel, D. and L. Henrio, "A Theory of Distributed Objects," Springer, 2005.

[7] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada, *Maude: Specification and programming in rewriting logic*, Theor. Comput. Sci. **285** (2002), 187–243.

[8] Dahl, O.-J., *Can program proving be made practical?*, "Les fondements de la programmation," 57–114, Institut de Recherche d'Informatique et d'Automatique, Toulouse, 1977.

[9] de Boer, F. S. and C. Pierik, *How to cook a complete Hoare logic for your pet OO language*, "Formal Methods for Components and Objects (FMCO 2003)," LNCS 3188, 111–133, Springer, 2004.

[10] de Roever, W.-P., F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers, "Concurrency Verification," Cambridge University Press, 2001.

[11] Dovland, J., E. B. Johnsen, and O. Owe, *Verification of concurrent objects with asynchronous method calls*, "Proceedings of the 2005 International Conference on Software," 141–150, IEEE Press, 2005.

[12] Johnsen, E. B. and O. Owe, *An asynchronous communication model for distributed concurrent objects*, Softw. Sys. Model. **6** (2007), 39–58.

[13] Johnsen, E. B., O. Owe, and E. W. Axelsen, *A run-time environment for concurrent objects with asynchronous method calls*, Electr. Notes Theor. Comput. Sci. **117** (2005), 375–392.

[14] Johnsen, E. B., O. Owe, and A. B. Torjusen, *Validating behavioral component interfaces in rewriting logic*, Electr. Notes Theor. Comput. Sci. **159** (2006), 187–204.

[15] Johnsen, E. B., O. Owe, and I. C. Yu, *A type-safe object-oriented model for distributed concurrent systems*, Theor. Comput. Sci. **365** (2006), 23–66.

[16] Kahn, G., *The semantics of a simple language for parallel programming*, "Information Processing '74: Proceedings of the IFIP Congress," 471–475, North-Holland, 1974.

[17] Verdejo, A. and N. Martí-Oliet, *Executable structural operational semantics in Maude*, J. Log. Algebr. Program. **67** (2006), 226–293.

[18] Yonezawa, A., "ABCL: An Object-Oriented Concurrent System," MIT Press, 1990.