Verification of Concurrent Objects with Asynchronous Method Calls

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

Abstract

Current object-oriented approaches to distributed programs may be criticized in several respects. First, method calls are generally synchronous, which leads to much waiting in distributed and unstable networks. Second, the common model of thread concurrency makes reasoning about program behavior very challenging. A model based on concurrent objects communicating by means of asynchronous method calls has been proposed to combine object orientation and distribution in a more satisfactory way. This paper introduces a reasoning system for this model, focusing on simplicity and modularity. We believe that a simple and compositional proof system is paramount to allow verification of real programs. The proposed proof rules are derived from the Hoare rules of a standard sequential language by means of a semantic encoding preserving soundness and relative completeness.

1. Introduction

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

Three basic interaction models for concurrent processes are shared variables, remote method calls, and message passing [5]. As objects encapsulate local states, we find inter-object communication most naturally modeled by (remote) method calls, avoiding shared variables. With the *remote method invocation* (RMI) model, an object is activated by a method call. Control is transferred with the call so there is a master-slave relationship between the caller and the callee. A similar approach is taken with the execution threads of e.g. Hybrid [25] and Java [16], where concurrency is achieved through multithreading. The interference problem related to shared variables reemerges when threads operate concurrently in the same object, which happens with nonserialized methods in Java. Reasoning about programs in this setting is a highly complex matter [1, 10]: Safety is by convention rather than by language design [9]. Verification considerations therefore suggest that all methods should be serialized as done in e.g. Hybrid. However, when restricting to serialized methods, the calling object must *wait* for the return of a call, blocking for any other activity in the object. In a distributed setting this limitation is severe; delays and instabilities may cause much unnecessary waiting. A serialized nonterminating method will even block other method invocations, which makes it difficult to combine active and passive behavior in the same object. Also, separating execution threads from objects breaks the modularity and encapsulation of object orientation, leading to a very low-level style of programming.

Message passing is 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 [5]. 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]. Method calls imply an ordering on communication not easily captured in the Actor model. Actors do not distinguish replies from invocations, so capturing method calls with Actors quickly becomes unwieldy [2]. In addition, the abstraction mechanism provided by object-oriented methods is lost in languages where communication is expressed directly in terms of message passing.

Intuitive high-level programming constructs are needed to unite object orientation and distribution in a natural way. Recently, programming constructs for concurrent objects have been proposed in the Creol language [20], based on *processor release points* and a notion of *asynchronous method calls*. A concurrent object has its own execution thread. Processor release points are used to influence the implicit internal control flow in 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). In order to model real world systems in an object-oriented manner, asynchronously communicating concurrent objects appear as a natural approach.

This paper considers the problem of formal reasoning about concurrent objects communicating by asynchronous method calls, based on the approach of the Creol language. A partial correctness proof system is derived from that of a standard sequential language by means of a semantic encoding. This suggests that reasoning is significantly simpler than for languages based on thread concurrency. The approach of this paper is modular, as invariants for classes may be established independently and composed at need.

The paper is structured as follows. Section 2 introduces and informally explains the language syntax, Section 3 considers reasoning in terms of class invariants, Section 4 explains the language constructs in terms of a sequential language, Section 5 derives proof rules for the Creol language, Section 6 considers composition of class invariants, Section 7 provides an example, Section 8 discusses related work, and Section 9 concludes the paper.

2. The Creol Language

This section introduces the communication and concurrency aspects of Creol [20], a programming language for distributed concurrent objects, and in particular the notions of asynchronous method calls and processor release points. Concurrent objects are potentially active, encapsulating execution threads. Objects have explicit identifiers: communication takes place between named objects and object identifiers may be exchanged between objects. All object interaction is by means of method calls.

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 the method *run*, which is given special treatment operationally. After initialization of the object, the *run* method, if provided, is invoked. Apart from *run*, declared methods may be invoked by other objects. These methods reflect passive, or reactive, behavior in the object, whereas *run* reflects active behavior. We will refer to the invoked method instances as the object's *processes*. Object activity is organized around an external message queue, which contains incoming messages, and an internal process queue, which contains *pending* processes. Methods need not terminate and may be temporarily *suspended* on the internal process queue.

Objects are typed by interface. Let I be the declared interface of an object o and let m be a method declared in I. Creol is strongly typed, which ensures that for each method invocation $o.m$, the actual object o will support I and the method m is understood.

Asynchronous Method Calls. Methods in Creol may be invoked in an asynchronous way [20]. Methods are implemented by guarded commands to be evaluated in the context of locally bound variables. Due to possible processor release points, the values of an object's instance attributes are not entirely controlled by a method instance if it suspends itself before completion. However, a method may create local variables supplementing the attributes. In particular, the values of formal parameters are stored locally, but other local variables may also be created. An object can have several pending calls to the same method, possibly with different values for local variables. The local variables *label* and *caller* are reserved to identify the call and the caller for the reply, which is emitted at method termination.

An asynchronous method call is made with the command $l!o.m(e)$, where the label l is a unique reference to the call, o an object expression which reduces to an object identifier, m a method name, and e an expression list with the supplied actual parameters. Labels are used to identify replies, and may be omitted in the syntax if a reply is not explicitly requested. No synchronization is involved and process execution may proceed after calling an external method until the return value is needed by the process. To fetch the return values from the queue, say in a variable list x, we ask for the reply to the call: $l?(x)$. If the reply has arrived, return values are assigned to x and execution continues without delay. If no reply to the call has been received, process execution is blocked. This interpretation of $l?(x)$ gives the same effect as treating x as a *future variable*, e.g. [8, 31]. Local calls need

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

Figure 1. An outline of Creol syntax.

not be prefixed by an object identifier. The syntax $o.m(e;x)$ is adopted for synchronous (RPC) method calls, blocking the processor while waiting for the reply. Synchronous local calls are loaded directly into the active code.

Processor Release Points. In Creol, the control flow inside concurrent objects may be influenced by potential processor release points. These are explicitly declared in method bodies using guarded commands [15], but adapted to the following semantics: When a guard evaluates to false during process execution, the remaining process code and the values of its local variables are *suspended* on 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 by*

- wait $\in$ *Guard (explicit release)*
- $l? \in$ *Guard, where l is a label*
- $\phi \in$ *Guard, where ϕ is a boolean expression over local and object variables.*

Here, *wait* is a construct for explicit release of the processor, resembling the method *yield* in Java. The reply guard $l?$ succeeds if the reply to the invocation labeled l has arrived. Guards are evaluated atomically, and may be combined: $g_1 \wedge g_2$ and $g_1 \vee g_2$ for guards g_1 and g_2.

Guarded commands can be *composed* in different ways, reflecting the requirements to the internal control flow in the objects. Let GS_1 and GS_2 denote the guarded commands $\textbf{await}\, g_1; S_1$ and $\textbf{await}\, g_2; S_2$. Nesting of guards is obtained by sequential composition; in a program statement $GS_1; GS_2$, the guard g_2 corresponds to a potential inner processor release point. Nondeterministic choice between guarded commands is expressed by $GS_1 \Box GS_2$, which may compute S_1 only when g_1 evaluates to true or S_2 only when g_2 evaluates to true. An unguarded statement may be considered as a statement guarded by true. Control flow without potential processor release is expressed by **if** and **while** constructs, and assignment to local and object variables is expressed by $x := e$, where x is a list of disjoint variables to which there is write access, and e is a list of expressions of equal length of x. There is read-only access to in-parameters of methods. Figure 1 summarizes the language syntax.

With nested processor release points, the processor need not wait actively for replies. Pending processes or new method calls may be evaluated instead of blocking the processor. However, when the reply has arrived, the *continuation* of the original process must compete with other enabled pending processes in the internal process queue.

3. Class Invariants with Mythical Histories

The execution of a distributed system can be represented by the sequence of observable communication events between system components. At any given point in time this finite sequence, called a communication history [11] or trace [17], abstractly captures the system state. Therefore, system specifications may be given in terms of the finite initial segments of these histories. A *history invariant* is a predicate on finite sequences which holds for all sequences in the prefix-closure of the set of traces, and consequently for all abstract system states, expressing safety properties in the sense of Alpern and Schneider [4].

In order to reason about distributed object systems, we use the assumption commitment (or rely-guarantee) paradigm [23], but adapted to input and output prefixes of the communication history [21], which allows compositional reasoning. For nonterminating systems, these predicates typically express invariant requirements on the (local) communication history.

Communication Events. In order to model object communication, a call to a method of an object o' by an object o can be seen as passing an invocation message from o to o', and the reply as passing a completion message from o' to o. The alphabet of communication events is restricted to these two kinds of messages, which are now formally defined.

Let *Obj*, *Mtd*, and *Label* denote the types of objects, methods, and labels. The latter is totally ordered. Let *Data* be the type of values occurring as actual parameters to method calls, and *Kind* the enumeration type $\{init, comp\}$. The set *Msg* of messages consists of tuples $\langle caller, label, kind, callee, mtd, par\rangle$ where $par : List[Data]$, $caller, callee : Obj$, $label : Label$, $kind : Kind$, and $mtd : Mtd$. The set *IMsg* of invocation messages is obtained by restricting *Msg* to messages of kind *init*, represented graphically as $caller \xrightarrow{label} callee.mtd(par)$, and the set *CMsg* of completion messages by restricting *Msg* to messages of kind *comp*, represented graphically as $caller \xleftarrow{label} callee.mtd(par)$. In the graphical notation, the arrow illustrates which way the message is sent. Messages may be decomposed by the functions *caller*, *callee*, *label*, *kind*, *mtd*, and *par*: For instance, $\langle o,l,k,o',m,e\rangle.label == l$.

3.1. The Communication History

The communication history of a system up to present time is represented as a finite sequence of type $Seq[Msg]$. Finite sequences are defined by the empty (ε) and right append ($\vdash$) constructors. Initially, the history sequence is empty. Whenever an object in the system calls a method, the history is extended by means of right append with a message of type *IMsg*. When a reply is emitted, the history is similarly extended with a message of type *CMsg*.

Preliminaries. Decomposition functions for messages are lifted to sequences, returning a sequence of the specified message element. For instance *label*: $Seq[Msg] \to Seq[Label]$ constructs the sequence of labels from the history and is inductively defined. Restriction of the history to a set of messages is now defined.

Definition 2 (Projection). *Let* $h : Seq[Msg]$ *and* $S : Set[Msg]$. *Define* $\_/\_ : Seq[Msg] \times Set[Msg] \to Seq[Msg]$ *by:*

$$\begin{aligned} \varepsilon/S &== \varepsilon \\ h \vdash m/S &== \textbf{\textit{if}}\ m \in S\ \textbf{\textit{then}}\ (h/S) \vdash m\ \textbf{\textit{else}}\ h/S\ \textbf{\textit{fi}} \end{aligned}$$

For $o : Obj$, let $o\!\to$ denote the set $\{m : IMsg \mid m.caller = o\}$ and $o\!\leftarrow$ the set $\{m : CMsg \mid m.caller = o\}$. We now define the functions *init* and *comp*.

Definition 3 (Init, Comp). *Let* $h : Seq[Msg]$, $l : Label$, *and* $o : Obj$. *Define* $init, comp : Seq[Msg] \times Obj \times Label \to Bool$ *by:*

$$\begin{aligned} init(h,o,l) &== l \in (h/o\!\to).label \\ comp(h,o,l) &== l \in (h/o\!\leftarrow).label \end{aligned}$$

In a distributed system with asynchronous communication an object can in general emit an invocation message at any time, since no synchronization is involved. However, a completion message may only occur after the corresponding invocation message in the history. For simplicity, we assume that all invocation messages sent from a particular object are equipped with unique labels. Wellformed histories are now defined:

Definition 4 (Wellformed histories). *Let* $h : Seq[Msg]$, $o, o' : Obj$, $m : Mtd$, $l : Label$, *and* $e : List[Data]$. *Define* $wf : Seq[Msg] \to Bool$ *inductively by:*

$$\begin{aligned} &wf(\varepsilon) == \text{true} \\ &wf(h \vdash o \xrightarrow{l} o'.m(e)) == wf(h) \wedge \neg init(h,o,l) \\ &wf(h \vdash o \xleftarrow{l} o'.m(e)) == \\ &\qquad wf(h) \wedge init(h/(o'.m),o,l) \wedge \neg comp(h,o,l) \end{aligned}$$

In this definition, $h/(o'.m)$ denotes the restriction of h to messages involving the method m provided by an object o'. In a wellformed history, every invocation message is uniquely defined by its caller and label. This is because every object identifier is assumed to be unique and every invocation from a given caller has a unique label. Furthermore, every completion message must match exactly one invocation message. Define $\_ \leq \_ : Seq[T] \times Seq[T] \to Bool$ such that $h \leq h'$ iff h is a prefix of h'.

Local History Projections. In order to reason locally about a particular object o, we will consider the restricted communication history h/o, defined as

$$h/o == h/\{m : Msg \mid (m.caller = o \vee m.callee = o)\}.$$

If h is a wellformed history, every local history projection h/o is also wellformed. Consequently, all properties of wellformed histories apply to local histories. The local history of an uninstantiated object is ε.

For a particular object o, define the set OUT_o of possible messages sent from o. These are either invocation messages sent from o or completion messages generated by o.

$$\begin{aligned} &\mathrm{OUT}_o == \\ &\quad \{m : IMsg \mid m.caller = o\} \cup \{m : CMsg \mid m.callee = o\} \end{aligned}$$

Invariant Reasoning. In a nonterminating system it is difficult to specify and reason compositionally about behavior in terms of pre- and postconditions. Instead, pre- and postconditions to method declarations are used to establish a *class invariant*. In order to facilitate compositional reasoning about Creol programs, the class invariant will be used to establish a *relationship between the internal state and the observable behavior* of class instances. The internal state reflects the values of class attributes and the observable behavior is expressed by a set of potential communication histories [21]. For this purpose the class attributes are extended with a mythical variable $\mathcal{H}$, reflecting the local history, and the code is extended with (mythical) statements to update $\mathcal{H}$ for every *output message* generated by the program code. At the imperative level execution of $l!o'.m(e)$ by an object o is is reflected by a history extension $\mathcal{H} := \mathcal{H} \vdash o \xrightarrow{l} o'.m(e)$, where $\langle o,l\rangle$ forms a unique pair of values. The corresponding completion message is recorded on the history when the invocation of m finishes execution $\mathcal{H} := \mathcal{H} \vdash o \xleftarrow{l} o'.m(y)$, where y is the list of return parameters. Mythical statements are introduced for reasoning purposes only, and need not be included in the final program code [12]. Let $FV[P]$ denote the set of variables which occur free in a predicate P and let P^x_e denote the substitution of every free occurrence of x in P by the expression e. For a class C we want to establish a class invariant I_C, ranging over class attributes (w) and the history sequence $\mathcal{H}$, i.e. $FV[I_C] \subseteq w \cup \{\mathcal{H}\}$.

Class invariants are established in the following way. For an arbitrary object of class C, we must prove that I_C holds initially. Consequently, I_C may be assumed when the object processor *starts* or *resumes* execution of an invoked method, if we can show that I_C holds every time processor control is *released*. Due to Creol's nested processor release points, processor control can be released either because method execution is completed or because guards are not satisfied.

Let w_m denote the variables local to a method m and assume disjointness between the names of local and class variables, i.e. $w_m \cap w = \emptyset$. Ignoring type information, a method m may then be declared as

$$\textbf{op}\ m(\textbf{in}\ x\ \textbf{out}\ y) == \textbf{var}\ w_m := e; body,$$

where **var** $w_m := e$ denotes concurrent assignment of initial values to local variable declarations. For reasoning purposes we assume that method m of object *this* has been invoked, which is reflected in the history by a pending invocation message. Upon method termination a corresponding completion message is appended to the history, preserving wellformedness. For reasoning purposes, the history update is explicitly represented. This leads to the following verification condition:

$$\begin{array}{l}\{\, I_C \wedge init(\mathcal{H}/(this.m), caller, label) \\ \quad \wedge \neg comp(\mathcal{H}, caller, label) \\ \quad \wedge x = (find(\mathcal{H}, caller, label, init)).par\} \\ w_m := e;\ body; \mathcal{H} := \mathcal{H} \vdash caller \xleftarrow{label} this.m(y) \\ \{\, I_C \,\}\end{array}$$

Here *find* returns the message on $\mathcal{H}$ with the specified caller label and kind. The precondition accounts for the assignment of actual parameters to the formal parameter list x.

4. Semantics

To define the semantics of Creol programs, we consider a *sequential sublanguage* of Creol, excluding constructs for asynchronous method calls and processor release points:

$$\begin{array}{l}\textbf{skip} \mid x := e \mid S_1 \Box S_2 \mid m(e;x) \mid (\ S\ ) \mid S_1;S_2 \\ \mid \textbf{if}\ \phi\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi} \mid \textbf{while}\ \phi\ \textbf{do}\ S\ \textbf{od}\end{array}$$

This sequential sublanguage SEQ consists of standard syntax with a well-established semantics and proof system. In particular, Apt [6, 7] shows that this proof system is sound and relatively complete. In this section we will give a semantic encoding of the remaining Creol statements in terms of SEQ. To do this, we emphasize the encoding of $l!o.m(e), l?(x)$, and **await** g for $g \in Guard$. The remaining language constructs may be defined in terms of these and the sequential language as shown in Figure 2, where L denotes some fresh label value. Note that synchronous calls to remote objects are simulated by asynchronous communication, whereas synchronous local calls are performed directly (without involving any communication).

$$\begin{array}{lcl} l!m(e) & == & l!this.m(e) \\ !o.m(e) & == & L!o.m(e) \\ !m(e) & == & L!this.m(e) \\ o.m(e;x) & == & \textbf{if}\ o = this\ \textbf{then}\ m(e;x) \\ & & \textbf{else}\ L!o.m(e); L?(x)\ \textbf{fi} \\ \textbf{await}\ l?(x) & == & \textbf{await}\ l?;\ l?(x) \\ \textbf{await}\ o.m(e;x) & == & L!o.m(e); \textbf{await}\ L?(x) \\ \textbf{if}\ \phi\ \textbf{then}\ S\ \textbf{fi} & == & \textbf{if}\ \phi\ \textbf{then}\ S\ \textbf{else skip fi} \end{array}$$

Figure 2. Language abbreviations

A Creol process with release points and asynchronous method calls is interpreted in SEQ as a nondeterministic program *without* shared variables, release points, and asynchronous method calls. The local history is captured by a variable $\mathcal{H}$ in each class, using nondeterministic updates on $\mathcal{H}$ to mimic the current state of the local interaction history of the original Creol program. To obtain an interleaving semantics for Creol, each atomic statement is proceeded by a nondeterministic extension of $\mathcal{H}$, mimicking asynchronous interaction with external objects.

4.1. Encoding Creol in SEQ

This section defines a mapping $\langle\langle\ \rangle\rangle$ which translates Creol programs into SEQ. All expressions and types are translated by the identity function. Creol classes, with methods and attributes, are translated directly to SEQ, with some implicit parameters added to the methods, and with $this : Obj$ added as a class attribute. We consider a given class C with variable attributes w, and a given method m in this class with local variables and in- and out-parameters. Each method gets two implicit in-parameters, $caller : Obj$ and $label : Label$, which store the object identifier of the initiator and label value of the call, respectively. As in Creol there is only read access to in-parameters.

$$\begin{array}{l}\langle\langle\, \textbf{op}\ m(\textbf{in}\ x\ \textbf{out}\ y) == \textbf{var}\ w_m := Initval;\ body \rangle\rangle = \\ \quad \textbf{op}\ m(\textbf{in}\ x,\ caller,\ label\ \textbf{out}\ y) == \\ \qquad \textbf{var}\ w_m := Initval; \\ \qquad \langle\langle body \rangle\rangle;\ \mathcal{H} := \mathcal{H} \vdash caller \xleftarrow{label} this.m(y)\end{array}$$

The additional class variable $\mathcal{H}$ is a sequence of messages involving *this*, initialized to empty.

In SEQ we introduce a *nondeterministic assignment*

$$y := \textbf{some}\ x \mid P(x),$$

which assigns to y arbitrary values satisfying the predicate P. (The variable lists x and y have equal length and type.)

Capturing the Environment. Reasoning about a Creol process cannot alone capture the concurrent activity of the asynchronously communicating object. In particular, the object may receive arbitrary input and other processes in the object may send output. We shall mimic this activity by nondeterministic extensions to the history variable $\mathcal{H}$. For this purpose, we introduce two particular nondeterministic assignments: interleave represents activity by the environment when the object is not active, and release represents activity by the environment and also by other processes in the object, capturing release points. Define interleave as

$$\text{interleave} == \mathcal{H} := \textbf{some } h \mid IntReq(h),$$

where $IntReq(h) == \mathcal{H} \leq h \wedge h/\mathrm{OUT}_{this} = \mathcal{H}/\mathrm{OUT}_{this} \wedge wf(h)$, i.e. $\mathcal{H}$ is extended in a nondeterministic manner preserving wellformedness, and without output from *this*. One could model the parallel execution of the environment by inserting interleave before each Creol statement, however, for simpler partial correctness it suffices to insert interleave before each statement accessing $\mathcal{H}$.

The definition of interleave expresses that *this* object does not control the environment. Predicates not restricting input to *this* or not concerned with input to *this* at all are not affected by interleave, but predicates may however relate output events to input events. Especially, class invariants must fulfill the criteria

$$\{I_C(w,\mathcal{H})\}\ \text{interleave}\ \{I_C(w,\mathcal{H})\}.$$

Consequently, we may omit interleave when the invariant is required to hold, i.e. before and after methods bodies.

In contrast, release denotes a simultaneous assignment to the class attributes and to $\mathcal{H}$, defined as

$$\text{release} == w, \mathcal{H} := \textbf{some } w', h \mid RelReq(w',h),$$

where $RelReq(w',h) == \mathcal{H} \leq h \wedge \neg comp(h, caller, label) \wedge wf(h) \wedge (I_C(w,\mathcal{H}) \Rightarrow I_C(w',h))$. This assignment updates the history and class attributes nondeterministically with values satisfying the class invariant, extending the history in a wellformed manner. Although output from *this* may occur, the event representing completion of the current method invocation cannot occur. It follows that the following Hoare triple holds:

$$\{I_C(w,\mathcal{H})\}\ \text{release}\ \{I_C(w,\mathcal{H})\}.$$

For reasoning, two subsequent interleave statements, as well as two release statements, may be replaced by one. In addition, an interleave preceding a release may be omitted.

The Encoding. Asynchronous method calls give rise to events recorded in $\mathcal{H}$:

$$\langle\langle\, l!o.m(e)\,\rangle\rangle == \text{interleave}; l := \textbf{some } l' \mid \neg init(\mathcal{H}, this, l');$$
$$\mathcal{H} := \mathcal{H} \vdash this \xrightarrow{l} o.m(e)$$

Reply statements block the object's internal activity. Therefore, input to the object may occur but output from the object is not allowed. Reply statements are therefore modeled by a loop doing interleave as long as the reply message has not arrived. However, restricting ourself to partial correctness we may assume termination of this loop, giving:

$$\langle\langle\, l?(y)\,\rangle\rangle == \mathcal{H} := \textbf{some } h \mid IntReq(h) \wedge comp(h, this, l);$$
$$y := find(\mathcal{H}, this, l, comp).par$$

Processor release points allow output from *this* object, except for the reply to the current method invocation. Therefore, await statements are modeled by means of release.

$\langle\langle\, \textbf{await } wait\,\rangle\rangle$ == release
$\langle\langle\, \textbf{await } l?\,\rangle\rangle$ ==
$\quad w, \mathcal{H} := \textbf{some } w', h \mid RelReq(w',h) \wedge comp(h, this, l)$
$\langle\langle\, \textbf{await } \phi\,\rangle\rangle == \textbf{if } \phi \textbf{ then skip else}$
$\quad w, \mathcal{H} := \textbf{some } w', h \mid RelReq(w',h) \wedge \phi^{w}_{w'} \textbf{ fi}$

Again, the encoding is restricted to partial correctness. For **await** $l?$ statements there is a nondeterministic (finite) delay between sending and receiving of messages, modeled by release. Other Creol statements are translated directly.

$\langle\langle\, \textbf{skip}\,\rangle\rangle$	$== \textbf{skip}$
$\langle\langle x := e\rangle\rangle$	$== x := e$
$\langle\langle\, (\ S\ )\,\rangle\rangle$	$== (\ \langle\langle S\rangle\rangle\ )$
$\langle\langle\, \textbf{while } \phi \textbf{ do } S \textbf{ od}\rangle\rangle$	$== \textbf{while } \phi \textbf{ do } \langle\langle S\rangle\rangle \textbf{ od}$
$\langle\langle\, \textbf{if } \phi \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}\rangle\rangle$	$== \textbf{if } \phi \textbf{ then } \langle\langle S_1\rangle\rangle \textbf{ else } \langle\langle S_2\rangle\rangle \textbf{ fi}$
$\langle\langle S_1 \Box S_2\rangle\rangle$	$== \langle\langle S_1\rangle\rangle \Box \langle\langle S_2\rangle\rangle$
$\langle\langle\, m(e;y)\,\rangle\rangle$	$== m(e;y)$

We conclude this section with a lemma.

Lemma (Preservation of wellformedness). *The SEQ encoding of Creol programs preserves history wellformedness.*

The proof goes by induction over method bodies. Every statement preserves $wf(\mathcal{H})$, in particular interleave, release, and the encoding of $l!o.m(e)$ fulfill this criteria. Since every invocation message is unique and assumed to create exactly one process, read-only access to *caller*, *label*, and *this* combined with the encoding ensure that the completion message recorded at process termination is unique and corresponds to the invocation message.

5. Verification of Creol Classes

The sequential language SEQ has a well-established proof system [6, 7], from which we may derive proof rules for Creol via the presented encoding. Due to the abbreviations introduced in Figure 2 it suffices to consider the statements $l!o.m(e), l?(x)$, and **await** g for a basic guard g. Rules for combined guards may be derived from these.

The weakest liberal precondition for nondeterministic assignment is

$$wlp(y := \textbf{some } x \mid P,\ Q) = \forall x \mid (P \Rightarrow Q^{y}_{x}),$$

assuming that x is disjoint from $FV[Q]-\{y\}$. The side condition may easily be satisfied, since variable names in **some** expressions may be renamed to avoid name capture.

Creol has object pointers but no dot notation for accessing attributes, thus Hoare reasoning about pointers can be done according to standard rules [24]. The rules for nondeterministic assignment and local procedure calls maintain soundness and relative completeness of the proof system.

We first consider invocation and reply statements, and then processor release points. Backward construction over the encoding of the invocation statement $l!o.m(e)$ leads to:

$$\begin{aligned}&wlp(l!o.m(e),\ Q) =\\&\quad \forall l',h \mid IntReq(h) \wedge \neg init(h,this,l') \Rightarrow Q^{l,\mathcal{H}}_{l',h\vdash this \overset{l'}{\rightarrow} o.m(e)}\end{aligned}$$

This statement includes an assignment to $\mathcal{H}$ so the precondition captures a nondeterministic update on $\mathcal{H}$, expressed by interleave, preceding the nondeterministic assignment. By backward construction over the encoding of $l?(y)$, the weakest liberal precondition for this statement becomes:

$$\begin{aligned}&wlp(l?(y),\ Q) =\\&\quad \forall h \mid IntReq(h) \wedge comp(h,this,l) \Rightarrow Q^{\mathcal{H},y}_{h,find(h,this,l,comp).par}\end{aligned}$$

The precondition captures the nondeterministic extension of $\mathcal{H}$, while leaving $\mathcal{H}/\mathrm{OUT}_{this}$ unaltered.

Release Points. The **await** *wait* sentence is modeled by a release, which results in the following precondition:

$$wlp(\textbf{await}\ wait,\ Q) = \forall w',h \mid RelReq(w',h) \Rightarrow Q^{w,\mathcal{H}}_{w',h}$$

For the weakest precondition of reply guards, we may assume the existence of a completion message:

$$\begin{aligned}&wlp(\textbf{await}\ l?,\ Q) =\\&\quad \forall w',h \mid RelReq(w',h) \wedge comp(h,this,l) \Rightarrow Q^{w,\mathcal{H}}_{w',h}\end{aligned}$$

For boolean guards, the postcondition must be satisfied directly if the guard is true:

$$\begin{aligned}&wlp(\textbf{await}\ \phi, Q) =\\&\quad \textbf{if}\ \phi\ \textbf{then}\ Q\ \textbf{else}\ \forall w',h \mid RelReq(w',h) \wedge \phi^{w}_{w'} \Rightarrow Q^{w,\mathcal{H}}_{w',h}\ \textbf{fi}\end{aligned}$$

By backward construction, we have obtained a sound and complete reasoning system for asynchronous method calls and processor release points. For release, the proposed semantics depends on the given invariant I. In order to establish completeness of the proof system relative to Creol, we therefore require $I \Rightarrow wlp(S,I)$ for execution paths S between suspension points. Consequently, the invariant I must be a sufficient precondition to ensure that I holds at the next suspension point. Weakest liberal preconditions for the remaining statements may be derived from the abbreviations given in Figure 2.

Both wlp and Hoare reasoning may be used in the same proof, since proving $\{P\}S\{Q\}$ is the same as proving $P \Rightarrow wlp(S,Q)$. To further emphasize invariant reasoning we may set up a theorem for processor release points.

Theorem. *Given an invariant $I(w,\mathcal{H})$, a predicate $L(w_m)$ over local variables w_m such that $FV[L] \cap FV[I] = \emptyset$ and a predicate $\phi : Data \times \ldots \times Data \rightarrow Bool$ on local and object variables, then:*

$$\{I \wedge L\}\ \textbf{await}\ wait\ \{I \wedge L\} \quad (1)$$
$$\{I \wedge L\}\ \textbf{await}\ \phi\ \{I \wedge L \wedge \phi(w,w_m)\} \quad (2)$$
$$\{I \wedge L\}\ \textbf{await}\ l?\ \{I \wedge L \wedge comp(\mathcal{H},this,l)\} \quad (3)$$

The proof goes by showing that the weakest liberal preconditions in the three cases follow from the assumptions.

Given a predicate P where $FV[P] \subseteq w \cup w_m \cup \{\mathcal{H}\}$, we may prove $\{P \wedge \phi\}\ \textbf{await}\ \phi\ \{P \wedge \phi\}$, where P need not imply the invariant. This implies $\{P\}\ \textbf{await}\ \mathsf{true}\ \{P\}$, which is in accordance with the intuitive understanding of the sentence **await** true as being identical to **skip**.

6. Parallel Composition

The organization of the state space in locally accessible variables and communication by messages mimicked by *local* communication history variables allows a compositional reasoning style. In order to check that objects compose, it is sufficient to compare the local histories. For this purpose, we adapt a composition method introduced by Soundarajan [29, 30] and require that local histories are *compatible*. In our approach, compatibility between two histories is checked directly by projection from a common history. The local history variable $\mathcal{H}_o$ in a SEQ class represents the local history of an arbitrary object o of class C, denoted $o:C$. For two objects o and o', the local histories $\mathcal{H}_o$ and $\mathcal{H}_{o'}$ are composable if $\exists \mathcal{H} \mid \mathcal{H}/o = \mathcal{H}_o \wedge \mathcal{H}/o' = \mathcal{H}_{o'}$. The invariant of the object o must satisfy the class invariant I_C for some appropriate values of the class attributes:

$$I_{o:C}(\mathcal{H}) == \exists w \mid (I_C(w,\mathcal{H}))^{this}_{o}$$

The substitution replaces the free occurrences of *this* with o and the existential quantifier hides the local state variables.

Two histories must agree on common events when composed, which is expressed by projection from the common history. An invariant for two composed objects o and o' may consequently be derived by conjunction from the invariants for the two objects:

$$I_{o:C||o':C'}(\mathcal{H}) == I_{o:C}(\mathcal{H}/o) \wedge I_{o':C'}(\mathcal{H}/o')$$

Similarly, invariants may be derived for sets of objects. The compatibility requirement reduces the amount of nondeterminism of the object seen in isolation. Consequently, the encoding leads to many noncomposable histories and is therefore not convenient for an operational semantics. However, it suffices for partial correctness reasoning.

7. Example: Readers and Writers

This section considers a class *RWController*, which implements a version of the readers and writers problem. We assume given a shared database *db*, which provides two basic operations *read* and *write*. Through interface specifications, these are assumed to be accessible for *RWController* objects only. Clients will communicate with an *RWController* object to obtain read and write access to the database. *RWController* provides *read* and *write* operations to clients and in addition four methods used to synchronize reading and writing activity: *OR* (OpenRead), *CR* (CloseRead), *OW* (OpenWrite) and *CW* (CloseWrite). A reading session happens between invocations of *OR* and *CR* and writing between invocations of *OW* and *CW*. A clients is assumed not to terminate unless it has invoked *CR* and *CW* at least as many times as *OR* and *OW*, respectively. To ensure fair competition between readers and writers, invocations of *OR* and *OW* compete on equal terms for a guard *free*. If the condition for reading or writing is unsatisfied, *free* is set to false and the process is suspended. Let *ObjSet* be a set over the type *Obj* of object identifiers.

```
class  RWController(db: DataBase)
begin
var free: Bool = true,  readers: ObjSet = ∅,  writer: Obj = null
var pr, pw: Nat = 0, 0     // pending calls to db.read and db.write
with RWClient
  op OR() == await free; if writer ≠ null then free := false;
    await (writer = null); free := true fi;
    readers := readers ∪ {caller}
  op CR() == await (caller ∈ readers); readers := readers \ {caller}
  op OW() == await free; free := false;
   await (readers = ∅ ∧ pr = 0 ∧ writer = null);
    free := true; writer := caller
  op CW() == await (pw = 0 ∧ writer = caller); writer := null
  op read(in k: Key out x: Data) == await (caller ∈ readers);
    pr := pr + 1; await db.read(k; x); pr := pr – 1
  op write(in k: Key, x: Data) == await (writer = caller);
    pw := pw + 1; await db.write(k,x); pw := pw – 1
end
```

For reasoning purposes, a mythical variable α is introduced to count the number of processes waiting on the inner guard of *OR*: $\alpha := \alpha + 1$; **await** $(writer = null \wedge \alpha > 0)$; $\alpha := \alpha - 1$. Similarly, the variable β is used for the inner guard of *OW*. We may then prove the invariant

$$I_{\text{fair}} : (free = (\alpha = 0 \wedge \beta = 0)) \wedge (\alpha + \beta \leq 1),$$

i.e. at most one process is waiting on the inner guards in *OR* and *OW*, and *free* is false iff a process is waiting. Such a process has priority over other invocations of *OR* and *OW*.

In order to express a safety invariant, we define the functions $Readers, Writers : Seq[Msg] \rightarrow Set[Obj]$ by:

$$
\begin{aligned}
&Readers(\varepsilon) == \emptyset \\
&Readers(\mathcal{H} \vdash caller \leftarrow OR) == Readers(\mathcal{H}) \cup \{caller\} \\
&Readers(\mathcal{H} \vdash caller \leftarrow CR) == Readers(\mathcal{H}) \setminus \{caller\} \\
&Readers(\mathcal{H} \vdash \textbf{others}) == Readers(\mathcal{H})
\end{aligned}
$$

where **others** matches all ground terms not giving any match in the previous equations. The definition of *Writers* follows the same pattern. Also define $Reading, Writing : Seq[Msg] \rightarrow Nat$ by:

$$
\begin{aligned}
&Reading(\mathcal{H}) == \#(\mathcal{H}/this \rightarrow db.read) - \#(\mathcal{H}/\leftarrow this.read) \\
&Writing(\mathcal{H}) == \#(\mathcal{H}/this \rightarrow db.write) - \#(\mathcal{H}/\leftarrow this.write)
\end{aligned}
$$

The safety invariant I relates the internal object state and the externally observable behavior:

$$
\begin{aligned}
&I_1 : \{writer\} = Writers(\mathcal{H}) \\
&I_2 : \wedge\; readers = Readers(\mathcal{H}) \\
&I_3 : \wedge\; pr = Reading(\mathcal{H}) \\
&I_4 : \wedge\; pw = Writing(\mathcal{H}) \\
&I_5 : \wedge\; (\#readers = 0 \vee writer = null) \wedge \#\{writer\} \leq 1 \\
&I_6 : \wedge\; writer = null \Rightarrow pw = 0 \\
&I_7 : \wedge\; (writer \neq null \Rightarrow pr = 0)
\end{aligned}
$$

This invariant shows how the values of class attributes may be expressed in terms of observable communication. In addition, the invariant implies $pr = 0 \vee pw = 0$, i.e. no reading and writing *activity* happens simultaneously. To illustrate the proposed reasoning system, we indicate some verification details for the methods *OR* and *read*. Using the derived Hoare rule for boolean guards, *OR* leads to three verification conditions. However, since I is not concerned with assignments to *free* and α, only one condition is relevant:

$$
\begin{aligned}
&\{I \wedge writer = null\} readers := readers \cup \{caller\}; \\
&\quad \mathcal{H} := \mathcal{H} \vdash caller \leftarrow this.OR\{I\}
\end{aligned}
$$

Here, the code is extended with a mythical assignment to $\mathcal{H}$ at method termination. I_1, I_3, and I_4 are not affected by the assignments. For the other parts of I we have to prove

$$
\begin{aligned}
&readers = Readers(\mathcal{H}) \wedge writer = null \wedge pw = 0 \\
&\Rightarrow readers \cup \{caller\} = Readers(\mathcal{H} \vdash caller \leftarrow this.OR) \\
&\quad \wedge (writer = null \Rightarrow pw = 0) \wedge (writer \neq null \Rightarrow pr = 0),
\end{aligned}
$$

which follows directly from the definition of *Readers*.

Using definitions in Figure 2, the inner release point in *read* becomes *l!db.read(k);* ***await*** *l?; l?(x)* for some fresh *l*. Following the same line of argument as above, we may interpret *pr* as a process counter, and strengthen the inner guard to ***await*** *(l? ∧ pr > 0)*. Applying the derived Hoare rules for boolean and reply guards, this results in the following two verification conditions:

$$
\begin{aligned}
&i)\;\; \{I \wedge caller \in readers\}\, pr := pr + 1; l!db.read(k)\, \{I\} \\
&ii)\; \{I \wedge comp(\mathcal{H}, this, l) \wedge pr > 0\} \\
&\qquad l?(x); pr := pr - 1; \mathcal{H} := \mathcal{H} \vdash caller \leftarrow this.read(x)\{I\}
\end{aligned}
$$

For verification of *i*), we use backward proof construction, and arrive at the condition:

$$
\begin{aligned}
&(I \wedge caller \in readers \wedge \mathcal{H} \leq h \wedge \mathcal{H}/\mathrm{OUT}_{this} = h/\mathrm{OUT}_{this}) \\
&\Rightarrow I^{pr,\mathcal{H}}_{pr+1, h \vdash this \xrightarrow{l} db.read(k)}
\end{aligned}
$$

This is proved by using the assumptions on the history. The most interesting parts are I_3 and I_7:

$$I \wedge caller \in readers \wedge \mathcal{H} \leq h \wedge \mathcal{H}/\mathrm{OUT}_{this} = h/\mathrm{OUT}_{this}$$
$$\Rightarrow pr + 1 = Reading(h \vdash this \xrightarrow{l} db.read(k))$$
$$\wedge\; writer \neq null \Rightarrow pr + 1 = 0$$

This follows since output from *this* does not occur in the extension to $\mathcal{H}$. Verification of *ii*) is similar.

Abstraction. In order to give a composable specification of the *RWController* class, we formulate the following abstract invariant in terms of the communication history only:

$$(Readers(\mathcal{H}) = \emptyset \vee Writers(\mathcal{H}) = \emptyset) \wedge \#Writers(\mathcal{H}) \leq 1$$

It is easy to show that this follows from the class invariant.

8. Related and Future Work

Related work. In this paper we have adapted communication histories, as introduced in [11], to model object communication in the distributed setting. History sequences reflecting message passing have also been used for specification and reasoning about CSP-like languages [13, 29].

Much recent work has addressed reasoning about sequential object-oriented languages [18, 27, 28], covering various aspects such as inheritance, subtyping, and dynamic binding. However, reasoning about multithreaded object-oriented languages is more challenging [1, 10]. For example, the approach of [1] uses a global cooperation test to deal with object communication. In addition, interference freedom must be proved since several threads may execute concurrently in the same object. In [13], de Boer presents a sound and complete compositional Hoare logic for collections of processes (objects) running in parallel. The objects communicate asynchronously by message passing, but in contrast to our work they communicate through FIFO channels, disallowing message overtaking.

Olderog and Apt [26] consider transformation of program statements preserving semantical equivalence. This approach is further developed in [14], which introduces a general methodology for transformation of language constructions into subparts of the language resulting in sound and complete reasoning systems. The approach resembles our encoding of Creol into SEQ, but it is noncompositional in contrast to our work. In particular, extention of the transformational approach to multithreaded systems seem to require interference freedom tests.

Future Work. In a recent paper [22], Creol has been extended with constructs for multiple inheritance. It is our present research goal to extend the approach to compositional verification presented in this paper to capture the combination of processor release points, multiple inheritance, and history-based compositionality. The combination of nondeterministic assignment and inherited class invariants represents a challenge for the transformational approach, but may be solved by appropriate behavioral restrictions. In order to address the verification of larger programs, tool support to discharge proof conditions should be developed.

The long term goal of our research is to study openness in distributed systems, taking an object-oriented approach. While this paper has focused on reasoning about communication and concurrency aspects in the asynchronous setting, we believe the language presented here offers interesting possibilities for reasoning in the presence of dynamic change. An obvious way to provide some openness is to allow dynamic addition of new (sub)classes and new (sub)interfaces. In our setting, this mechanism in itself does not violate reasoning control, because established results still hold. Also, additional implementation claims may be stated and proved. However, old objects may not use new interfaces that require new methods.

A natural way to overcome this limitation is through a dynamic class construct, allowing a class to be *replaced* by a subclass. Thus a class may be modified by adding attributes and methods, redefining methods, as well as extending the inheritance and implements relationships. The formalization of an operational semantics for such dynamic updates is currently being developed. The work presented in this paper is part of a larger effort to understand how to formalize and verify the effect of runtime modifications to open distributed systems in a compositional way. We believe that reasoning about suitably restricted runtime class extensions can be done by combining compositional history-based reasoning and behavioral subtyping.

9. Conclusion

The Creol language proposes programming constructs which aim to unite object orientation and distribution in a high-level and natural way, by means of processor release points and a notion of asynchronous method calls. In this paper, we develop Hoare rules for local reasoning about these constructs. The reasoning rules are derived in a transformational manner from a standard sequential language with a well-known semantics and established reasoning system. The language constructs for asynchronous method calls and processor release points are encoded in the sequential sublanguage extended with nondeterministic assignment. Combined with local communication histories, this allows the highly nondeterministic nature of concurrent and distributed systems to be captured in the sequential language. Based on the encoding weakest liberal preconditions are derived, which given sufficiently strong class invariants yield sound and relative complete Hoare rules for Creol classes, expressing partial correctness. In contrast to related approaches, the proposed local proof system is compositional, based on a compatibility requirement on local history variables capturing observable communication.

References

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

[2] G. A. Agha. Abstracting interaction patterns: A programming paradigm for open distributed systems. In E. Najm and J.-B. Stefani, editors, *Proc. 1st IFIP 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] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, Oct. 1985.

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

[6] K. R. Apt. Ten years of Hoare's logic: A survey — Part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, Oct. 1981.

[7] K. R. Apt. Ten years of Hoare's logic: A survey — Part II: Nondeterminism. *Theoretical Computer Science*, 28(1–2):83–109, Jan. 1984.

[8] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C$^\sharp$. In B. Magnusson, editor, *Proc. 16th European Conf. on Object-Oriented Programming (ECOOP'02)*, LNCS 2374, pages 415–440. Springer, 2002.

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

[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] O.-J. Dahl. Can program proving be made practical? In M. Amirchahy and D. Néel, editors, *Les Fondements de la Programmation*, pages 57–114. Institut de Recherche d'Informatique et d'Automatique, Toulouse, Dec. 1977.

[12] O.-J. Dahl. *Verifiable Programming*. Prentice Hall, 1992.

[13] F. S. de Boer. A Hoare logic for dynamic networks of asynchronously communicating deterministic processes. *Theoretical Computer Science*, 274:3–41, 2002.

[14] F. S. de Boer and C. Pierik. How to Cook a Complete Hoare Logic for Your Pet OO Language. In *Formal Methods for Components and Objects (FMCO'03)*, LNCS 3188, pages 111–133. Springer, 2004.

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

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

[17] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[18] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE'00)*, LNCS 1783, pages 284–303. Springer, 2000.

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

[20] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. In *Proc. 2nd IEEE Intl. Conf. on Software Engineering and Formal Methods (SEFM'04)*, pages 188–197. IEEE Computer Society Press, Sept. 2004.

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

[22] E. B. Johnsen and O. Owe. Inheritance in the presence of asynchronous method calls. In *Proc. 38th Hawaii Intl. Conf. on System Sciences (HICSS'05)*. IEEE Computer Society Press, Jan. 2005.

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

[24] J. M. Morris. A general axiom of assigment. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 25–34. Reidel, 1982.

[25] 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.

[26] E.-R. Olderog and K. P. Apt. Fairness in parallel programs: The transformational approach. *ACM Transactions on Programming Languages*, 10(3):420–455, July 1988.

[27] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *European Symosium on Programming (ESOP'99)*, LNCS 1576, pages 162–176. Springer, 1999.

[28] B. Reus, M. Wirsing, and R. Hennicker. A hoare calculus for verifying java realizations of OCL-constrained design models. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE'01)*, LNCS 2029, pages 300–317. Springer, 2001.

[29] N. Soundararajan. Axiomatic semantics of communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 6(4):647–662, Oct. 1984.

[30] N. Soundararajan. A proof technique for parallel programs. *Theoretical Computer Science*, 31(1-2):13–29, May 1984.

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