Available online at www.sciencedirect.com

ScienceDirect

ELSEVIER

Electronic Notes in Theoretical Computer Science 203 (2008) 19–34

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Observable Behavior of Dynamic Systems: Component Reasoning for Concurrent Objects[⋆]

Johan Dovland, Einar Broch Johnsen, and Olaf Owe[1]

Department of Informatics, University of Oslo, Norway

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. Models based on concurrent objects communicating by asynchronous method calls, have been proposed to combine object orientation and distribution in a more satisfactory way. In this paper, a high-level language and proof system are developed for such a model, emphasizing simplicity and modularity. In particular, the proof system is used to derive external specifications of observable behavior for objects, encapsulating their state. 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 a semantic encoding preserving soundness and relative completeness. Thus, the paper demonstrates that these models not only address the first criticism above, but also the second.

Keywords: observable behavior, concurrent objects, dynamic systems, interaction histories

1 Introduction

In order to facilitate reasoning about interacting components in nonterminating and distributed systems, specifications of component behavior should focus on the potential observable interactions between a component and its environment, rather than on internal, low-level implementation details such as the component's internal state variables. However for verification purposes it is then necessary to bridge the gap between a component's internal code and its observable behavior. This paper presents a proof system for deriving specifications of observable behavior from the internal code of components in the setting of distributed concurrent objects.

Object orientation is the leading framework for concurrent and distributed systems, recommended by the RM-ODP [17]. Object interaction is usually synchronous,

[⋆] This research is partially funded by the EU project IST-33826 CREDO: Modeling and analysis of evolutionary structures for distributed services (`http://credo.cwi.nl`).

[1] Email: `johand@ifi.uio.no`, `einarj@ifi.uio.no`, `olaf@ifi.uio.no`

doi:10.1016/j.entcs.2008.04.084

through method calls and changes to shared state variables. Such interaction, derived from sequential systems, is well-suited for tightly coupled systems. It is less suitable in a distributed setting with loosely coupled or externally coordinated components. Here synchronous communication results in undesired and uncontrolled waiting, and possibly deadlock. In addition these interaction mechanisms severely complicate reasoning. With the *remote method invocation* (RMI) model, control is transferred with the call. There is a master-slave relationship between the caller and the callee. Concurrency is achieved through multithreading. Shared variable interference occurs when threads operate concurrently in an object, which happens with, e.g., nonserialized methods in Java. Reasoning about programs in this setting is a highly complex matter [1, 9]: Safety is by convention rather than by language design [6]. Verification considerations suggest that all methods should be serialized. However, when restricting to serialized methods, the caller must *wait* for the return of a call, blocking all activity in the caller. 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 an object. Also, separating execution threads from objects breaks the modularity and encapsulation of object orientation, leading to a very low-level style of programming.

In order to better capture the setting of interacting distributed components we work with the concurrency and communication model of the Creol language [18], based on concurrent objects, *asynchronous method calls*, and so-called *processor release points*. There is no access to the internal state variables of other objects. A concurrent object has its own execution thread. Processor release points influence the implicit internal control flow in objects. This reduces the 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).

This paper presents a simple programming language and reasoning framework based on Creol's concurrency and communication model, and considers the problem of formal reasoning about dynamic systems of concurrent objects communicating by asynchronous method calls. 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 much simpler than for languages with thread concurrency. The approach of this paper is modular, as invariants expressing observable behavior may be established independently for each class and composed at need, resulting in behavioral specifications of dynamic systems in an open environment.

Paper overview. Section 2 introduces and informally explains the language. Section 3 describes class invariants for observable behavior, Section 4 the language semantics, and Section 5 the derived proof rules. Section 6 gives an example, Section 7 discusses related work, and Section 8 concludes the paper.

$$\begin{array}{lll} \mathit{Class} & ::= & \textbf{class}\ C\ [(\mathit{Param})]^{?}\ \mathit{Vdecl}^{?}\ \mathit{Mdecl}^{*} \\ \mathit{Param} & ::= & [v : T]^{+}_{,} \\ \mathit{Vdecl} & ::= & \textbf{var}\ [v : T[= e]^{?}]^{+}_{,} \\ \mathit{Mdecl} & ::= & \textbf{op}\ m\ ([\textbf{in}\ \mathit{Param}]^{?}\ [\textbf{out}\ \mathit{Param}]^{?}) == [\mathit{Vdecl};]^{?}\ [s]^{+}_{;} \end{array}$$

Figure 1. Syntax outline for classes, excluding expressions e and statements s. Here C is a class and m a method name. M^{+}_{d} denotes one or more repetitions of M with delimiter d, * indicates zero or more repetitions, and $^{?}$ indicates an optional part.

2 The Programming Language

We introduce a programming language based on the communication and concurrency aspects of Creol [18], which are now briefly explained. Concurrent objects are potentially active, encapsulating execution threads. Objects have explicit identifiers: communication is between named objects and object identifiers may be exchanged between objects. All object interaction is by means of method calls. We refer to the method activations on an object as the object's *processes*. At most one process in an object may be active at a time; the other processes are *suspended*. *Processor release points* influence the internal control flow in an object. These release points are declared as guarded commands [12], but adapted to the following semantics: When a guard evaluates to false during process execution, the *continuation* of the process is suspended on the process queue and the processor is released. After a processor release, an *enabled* and suspended processes is selected for execution.

A class declaration consists of a list of class parameters *cp*, class attributes W, and method declarations, as described in Fig. 1. Objects are dynamically created instances of classes. The state of an object is constructed from the parameters and attributes of its class. There is read-only access to the class parameters, including the implicit parameter *this*, used for self reference. The state of an object is encapsulated and can only be externally accessed via the object's methods. In particular, remote access to attributes is not allowed. For simplicity, all methods are assumed to be available to the environment, except the special methods *init* and *run*. The *init* method is used for object initialization and is invoked immediately after the object is created. After initialization, the *run* method, if provided, is invoked. The remaining methods reflect passive, or reactive, object behavior, whereas *run* initiates active behavior. Programs are assumed to be type-safe.

Methods are implemented by imperative statements, using the syntax of Fig. 2. A processor release point is written **await** g, for a guard g. A Boolean guard is enabled when the Boolean expression evaluates to *true*. Execution of an *asynchronous method call* **await** $x.m(\mathrm{E};\mathrm{V})$ invokes the method m in x with the input values E. The continuation of the calling process is then suspended and becomes enabled when the call returns. Consequently, other processes may be evaluated while waiting for the reply. Return values are assigned to the list V when the continuation gets processor control. Execution of the statement **await** *wait* explicitly releases the processor, similar to the method *yield* in Java. The syntax $x.m(\mathrm{E};\mathrm{V})$ is adopted for *synchronous method calls* (RPC), blocking the processor while waiting for the reply. The syntax $\mathit{this}.m(\mathrm{E};\mathrm{V})$ is used for local calls. Synchronous local calls are loaded directly into the active code.

While the execution of a method activation is suspended, the object's attributes may be changed by other processes on the object, but not the local variables and parameters of the method. There is read-only access to in-parameters of methods, including the implicit parameter *caller*. Due to nondeterminism in the distributed setting, overtaking of method calls is considered possible. If several invocations are made by a caller to the same method of an object, with identical actual values for the method's formal parameters, the caller cannot precisely identify the return to each of these method calls.

Object creation is written $x := \mathbf{new}\ C(\mathrm{E})$, where x is a variable and E a list of values for the class parameters of a class C. A reference to the new object is assigned to x and the *init* method is executed on the new object. Synchronous remote method calls are allowed in the body of *init*, but no processor release points nor local calls. Uniqueness of object identifiers is ensured by combining the identity of the creating object with local uniqueness. For this purpose we use a function $parent : Obj \rightarrow Obj$ such that $parent(o)$ denotes the creator of o. We assume that $parent(null) = null$ and that parent chains are cycle free; i.e., $o \notin anc(o)$, where the ancestor function $anc : Obj \rightarrow Set[Obj]$ is defined by $anc(o) \triangleq \mathbf{if}\ parent(o) = null\ \mathbf{then}\ \emptyset\ \mathbf{else}\ parent(o) \cup anc(parent(o))\ \mathbf{fi}$. Equality is the only executable basic operation on object identifiers.

3 Class Invariants and Observable Behavior

The execution of a distributed system can be represented by its *communication history*; i.e., the sequence of observable communication between system components [8, 15]. At any point in time the communication history abstractly captures the system state. Therefore a system may be specified by the finite initial segments of its communication histories. A *history invariant* holds for all finite sequences in the prefix-closure of the set of histories, expressing safety properties [2]. To observe and reason about object creation using histories, we let the history reveal relevant information about object creation.

Notation. Sequences are constructed by the empty sequence ε and the right append function $\_ \vdash \_ : Seq[T] \times T \rightarrow Seq[T]$ (where "_" indicates an argument position). Let $a, b : Seq[T]$, $x, y : T$, and $s : Set[T]$. The projection function $\_/\_ : Seq[T] \times Set[T] \rightarrow Seq[T]$ is defined inductively by $\varepsilon/s \triangleq \varepsilon$ and $(a \vdash x)/s \triangleq \mathbf{if}\ x \in s\ \mathbf{then}\ (a/s) \vdash x\ \mathbf{else}\ a/s\ \mathbf{fi}$. The "ends with" and "begins with" predicates $\_\mathbf{ew}\_ : Seq[T] \times T \rightarrow Bool$ and $\_\mathbf{bw}\_ : Seq[T] \times T \rightarrow Bool$ are defined inductively by $\varepsilon\ \mathbf{ew}\ x \triangleq false$, $(a \vdash x)\ \mathbf{ew}\ y \triangleq x = y$, $\varepsilon\ \mathbf{bw}\ x \triangleq false$, and $(a \vdash x)\ \mathbf{bw}\ y \triangleq y = first(a \vdash x)$. Furthermore, let a **is** $b \| c$ denote that a is an arbitrary interleaving of b and c, let $a \vdash\!\dashv b$ concatenate a with b, $a \leq b$ denote that a is a prefix of b, and $\# a$ denote the length of a. Finally, define $first((\varepsilon \vdash x) \vdash\!\dashv a) \triangleq x$ and $rest((\varepsilon \vdash x) \vdash\!\dashv a) \triangleq a$.

A call to a method of an object o' by an object o is modeled as passing an invocation message from o to o', and the reply as passing a completion message

Syntactic categories		*Definitions*
g in *Guard*	s in *Com*	$g ::= \mathit{wait} \mid b \mid x.m(\textsc{e}; \textsc{v})$
m in *Mtd*	$\textsc{v}$ in $\mathit{Var}^*_,$	$s ::= \textbf{skip} \mid \textsc{v} := \textsc{e} \mid \textbf{if } b \textbf{ then } \textsc{s}_1 \textbf{ else } \textsc{s}_2 \textbf{ fi}$
$\textsc{e}$ in $\mathit{Expr}^*_,$	x in *ObjExpr*	$\mid x := \textbf{new } C(\textsc{e})$
b in *Bool*	C in *ClassName*	$\mid x.m(\textsc{e}; \textsc{v}) \mid \textbf{await } g$

Figure 2. An outline of the imperative language syntax, with typical terms for each category. We let capitalized terms denote lists of a given category; for example, $\textsc{e}$ denotes an expression list while e denotes an expression.

from o' to o. Similarly, object creation is captured by a message from the parent object to the generated object. The *communication history* of a (sub)system up to present time is a finite sequence of type *Seq*[*Msg*], where *Msg* are the messages corresponding to method invocation, method completion, and object creation:

Definition 3.1 (Messages). Define the following sets of communication messages:

- the set *IMsg* of *invocation messages* $\langle \mathit{caller}, \mathit{callee}, \mathit{mtd}, \mathit{in} \rangle$,
- the set *CMsg* of *completion messages* $\langle \mathit{caller}, \mathit{callee}, \mathit{mtd}, \mathit{in}, \mathit{out} \rangle$,
- the set *NMsg* of *object creation messages* $\langle \mathit{caller}, \mathit{callee}, \mathit{class}, \mathit{in} \rangle$, and
- the set *Msg* consists of all messages; i.e., $\mathit{Msg} = \mathit{IMsg} \cup \mathit{CMsg} \cup \mathit{NMsg}$,

where $\mathit{caller}, \mathit{callee} : \mathit{Obj}$, $\mathit{mtd} : \mathit{Mtd}$, $\mathit{class} : \mathit{Cid}$, and $\mathit{in}, \mathit{out} : \mathit{List}[\mathit{Data}]$. *Obj*, *Mtd*, and *Cid* are the types of object, method, and class names, respectively, and *Data* the type of values occurring as actual parameters to method calls, including *Obj*.

Graphical representations of messages are given by $\mathit{caller} \rightarrow \mathit{callee}.\mathit{mtd}(\mathit{in})$, $\mathit{caller} \leftarrow \mathit{callee}.\mathit{mtd}(\mathit{in}; \mathit{out})$, and $\mathit{caller} \rightarrow \mathit{callee}.\textbf{new } \mathit{class}(\mathit{in})$, where the arrow suggests the direction of the message. When an object calls a method, the history h is extended with a message of type *IMsg*. When a reply is emitted, h is extended with a message of type *CMsg*. The message $o \rightarrow o'.\textbf{new } C(\textsc{e})$ corresponds to the execution of a **new** C statement in an object o, with $\textsc{e}$ as the actual values of the class parameters and o' as the new object identity.

Messages may be decomposed by functions $\_.\mathit{caller}, \_.\mathit{callee} : \mathit{Msg} \rightarrow \mathit{Obj}$; e.g., $\langle o, o', m, e \rangle.\mathit{callee} \triangleq o'$. The function $\_.\mathit{in} : \mathit{Msg} \rightarrow \mathit{List}[\mathit{Data}]$ returns the list of in-parameters. In addition, completion messages may be decomposed by the function $\_.\mathit{out}$, returning the list of out-parameters.

The messages potentially sent by an object o are defined as $\mathrm{OUT}_o \triangleq \{\mathit{msg} : \mathit{IMsg} \cup \mathit{NMsg} \mid \mathit{msg}.\mathit{caller} = o\} \cup \{\mathit{msg} : \mathit{CMsg} \mid \mathit{msg}.\mathit{callee} = o\}$, and those potentially received by o as $\mathrm{IN}_o \triangleq \{\mathit{msg} : \mathit{IMsg} \cup \mathit{NMsg} \mid \mathit{msg}.\mathit{callee} = o\} \cup \{\mathit{msg} : \mathit{CMsg} \mid \mathit{msg}.\mathit{caller} = o\}$. The intersection of OUT_o and IN_o, with o as both caller and callee, corresponds to internal messages. The *local history* $h/(\mathrm{IN}_o \cup \mathrm{OUT}_o)$, denoted h/o, contains the messages involving o and allows local reasoning about o. The object creation message involves both the new object and its parent. It is the first message in the new object's history, and allows compositional reasoning about dynamically created objects.

Functions may extract information from the history. In particular, we define

$pending : Seq[Msg] \times IMsg \rightarrow Bool$ and $oid : Msg \rightarrow Set[Obj]$ as follows:

$$pending(h, o \rightarrow o'.m(\textsc{e})) \triangleq \#(h/o \rightarrow o'.m(\textsc{e})) > \#(h/o \leftarrow o'.m(\textsc{e}; \_))$$

$$\begin{array}{llll} oid(\varepsilon) & \triangleq \{null\} & oid(o \rightarrow o'.m(\textsc{e})) & \triangleq \{o, o'\} \cup oid(\textsc{e}) \\ oid(h \vdash msg) & \triangleq oid(h) \cup oid(msg) & oid(o \leftarrow o'.m(\textsc{e}; \textsc{e}')) & \triangleq \{o, o'\} \cup oid(\textsc{e}, \textsc{e}') \\ & & oid(o \rightarrow o'.\mathbf{new}\ C(\textsc{e})) & \triangleq \{o, o'\} \cup oid(\textsc{e}) \end{array}$$

where $msg : Msg$, and $oid(\textsc{e})$ returns the set of object identifiers occurring in the list $\textsc{e}$. The function $ob : Seq[Msg] \rightarrow Set[Obj \times Cid \times List[Data]]$ returns the set of created objects in a history: $ob(h \vdash o \rightarrow o'.\mathbf{new}\ C(\textsc{e})) \triangleq ob(h) \cup \{o' : C(\textsc{e})\}$, and $ob(h \vdash msg) \triangleq ob(h)$ for all other messages msg. For a local history h/o, the projection $ob(h/o)$ returns o and all objects *created* by o.

3.1 Well-formed histories

In the asynchronous setting, objects may send messages at any time. Type checking ensures that only available methods are invoked for objects of given types. The run-time system ensures that generated objects will have unique identifiers. Assuming type correctness, well-formed histories satisfy a well-formedness predicate, as a completion message may only occur after the corresponding invocation message in the history:

Definition 3.2 (Well-formedness). Let $h : Seq[Msg]$, $\textsc{e}, \textsc{e}' : List[Data]$, $o, o' : Obj$, and $m : Mtd$. The *well-formedness predicate* $wf : Seq[Msg] \rightarrow Bool$ is defined by

$$\begin{array}{ll} wf(\varepsilon) & \triangleq true \\ wf(h \vdash o \rightarrow o'.m(\textsc{e})) & \triangleq wf(h) \wedge o \neq null \wedge o' \neq null \\ wf(h \vdash o \leftarrow o'.m(\textsc{e}; \textsc{e}')) & \triangleq wf(h) \wedge pending(h, o \rightarrow o'.m(\textsc{e})) \\ wf(h \vdash o \rightarrow o'.\mathbf{new}\ C(\textsc{e})) & \triangleq wf(h) \wedge parent(o') = o \wedge o' \notin oid(h) \end{array}$$

This definition ensures the local uniqueness of created identifiers, while *null* may create objects. Whenever an object identifier o' occurs in an output message in h/o, o' must either be a child of o, or occur in a previous input message to o. This leads to a notion of closure for histories.

Definition 3.3 (Closed histories). Let $h : Seq[Msg]$ and $o : Obj$. Define

$$closed(h \vdash x, o) \triangleq oid((h \vdash x)/\mathrm{OUT}_o) \subseteq oid(h/\mathrm{IN}_o) \cup (ob((h \vdash x)/o)/Obj),$$

where $\_/Obj : Set[Obj \times Cid \times List[Data]] \rightarrow Set[Obj]$ returns the identifiers of the created objects.

The following lemma holds for well-formed histories.

Lemma 3.4 *A history h is well-formed if the local projection h/o is well-formed and closed for each object $o \in oid(h)$, including null.*

Proof By induction over h, with $wf(h/o)$ and $closed(h, o)$ for all $o \in oid(h)$. □

3.2 Compositional Reasoning about Concurrent Objects in Dynamic Systems

In interactive and nonterminating systems, it is difficult to specify and reason compositionally about object behavior in terms of pre- and postconditions. Instead, pre- and postconditions to method declarations are used to establish a so-called *class invariant.* The class invariant must hold after initialization in all the instances of the class, be maintained by all methods, and hold at all processor release points. The class invariant serves as a *contract* between the different processes of the object instance: A method implements its part of the contract by ensuring that the invariant holds upon termination and when the method is suspended, assuming that the invariant holds initially and after suspensions. To facilitate compositional and component-based reasoning about programs, the class invariant is 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, whereas the observable behavior is expressed as a set of potential communication histories.

A *user-provided invariant* $I(\mathrm{W}, h)$ for a class C ranges over the class variables W of C and the local history h, as well as the class parameters cp and $this$, which are constant (read-only) variables. The *class invariant* $I_C(\mathrm{W}, h)$ is obtained by strengthening $I(\mathrm{W}, h)$ with the well-formedness property and knowledge about the initial object creation message on the local history:

$$I_C(\mathrm{W}, h) \triangleq I(\mathrm{W}, h) \wedge wf(h) \wedge h \ \mathbf{bw} \ (parent(this) \rightarrow this.\mathbf{new}\ C(cp)).$$

By organizing the state space in terms of only locally accessible variables, including a local history variable recording local communication messages, we obtain a compositional reasoning system. Let P_e^x denote the substitution of every free occurrence of x in P by e, and let $P_{\mathrm{E}}^{\mathrm{X}}$ denote simultaneous substitution. By hiding the internal state variables of an object o of class C, an *external invariant* $I_{o:C(\mathrm{E})}$ defining its observable behavior may be obtained:

$$I_{o:C(\mathrm{E})}(h) \triangleq \exists \mathrm{W} \mid (I_C(\mathrm{W}, h))^{this,cp}_{o,\mathrm{E}}.$$

Substitutions replace free occurrences of *this* with o and instantiate class parameters with actual values. The existential quantifier hides the local state variables.

To assert that objects compose we compare their local histories, adapting the composition method of Soundarajan [23,24]: Local histories must agree on common messages, expressed by projections from a common *global history.*

Consider a system with an initial object o created by an initial invocation message of the form $null \rightarrow o.\mathbf{new}\ C(...)$, such that all other objects are dynamically generated by o or generated objects. The global invariant of such a system of dynamically created objects may be constructed from the local invariants of the involved objects: The global invariant I^* of a system with global history H is

$$I^*(H) \triangleq \bigwedge_{(o:C(\mathrm{E}))\in ob(H)} I_{o:C(\mathrm{E})}(H/o).$$

The quantification ranges over all objects in the system, which is a finite number at any execution point. The global invariant is obtained directly from the external invariants of the composed objects, without any restrictions on the local reasoning. This ensures compositional reasoning. Note that we consider dynamic systems; the number and identifiers of the composed objects are nondeterministic. Lemma 4.2 below shows that $I^*(H)$ ensures well-formedness of the global history H.

4 Semantics: An Encoding into a Sequential Language

The semantics is expressed as an encoding into a sequential language without shared variables, but with a nondeterministic assignment operator [13]. Nondeterministic history extensions capture arbitrary activity of environment objects. The semantics describes a single object of a given class placed in an arbitrary environment. The semantics of a dynamically created system with several concurrent objects is given by the composition rule above. The compatibility requirement, implicit in the composition rule, reduces the amount of nondeterminism of the objects seen in isolation. This semantics suffices for partial correctness reasoning, but it is not suited as an operational semantics.

In order to simplify the semantics, we assume that an object may not control its environment. This means that, for all objects o of class C and $h_{in} \in \mathit{Seq}[\mathrm{IN}_o]$, the class invariant $I_C(\mathrm{W}, h)$ of C satisfies the following *asynchronous input property*:

$$\forall h' \mid (\mathit{wf}(h') \wedge h' \textbf{ is } h \| h_{in} \wedge I_C(\mathrm{W}, h)) \Rightarrow I_C(\mathrm{W}, h').$$

In the asynchronous setting objects may independently decide to send messages and these may arrive in a different order than sent, due to overtaking. The invariant should therefore restrict messages seen by an object, but allow the existence of additional unprocessed input. Therefore, we find the asynchronous input property natural for asynchronous systems: Note that invariants on $h/\mathrm{OUT}_{\mathit{this}}$ are guaranteed to have this property. Since completion messages give explicit information about the corresponding invocation messages, such invariants are often sufficient.

4.1 The Encoding

Consider a simple *sequential* language with the syntax

$$\textbf{skip} \mid \mathrm{V} := \mathrm{E} \mid s; \mathrm{S} \mid \textbf{if } b \textbf{ then } \mathrm{S}_1 \textbf{ else } \mathrm{S}_2 \textbf{ fi}.$$

This language has a well-established semantics and proof system. In particular, Apt shows that this proof system is sound and relative complete [3,4]. Let the language SEQ additionally include a statement for *nondeterministic assignment*, assigning to Y some value X satisfying a predicate P:

$$\mathrm{Y} := \textbf{some } \mathrm{X} \mid P(\mathrm{X}).$$

For partial correctness, we may assume that the statement does not terminate if no such X can be found.

A process with release points and asynchronous method calls is interpreted as a SEQ program *without* shared variables and release points, by the mapping $\langle\!\langle\ \rangle\!\rangle$. Expressions and types are mapped by the identity function. For classes, the list of class attributes is augmented with *this* : *Obj* and $\mathcal{H}$: $Seq[Msg]$, representing self reference and the history, respectively. The implicit parameter *caller* : *Obj* is added to each method. As before, there is read-only access to in-parameters and class parameters, including the additional variables.

The semantics of a method is defined from the local perspective of processes. A SEQ process executes on a state $\mathrm{W} \cup \{\mathcal{H}\}$ extended with local variables. The local effect of executing an invocation or a release statement is that W and $\mathcal{H}$ may be updated due to the execution of other processes. In the encoding, these updates are captured by nondeterministic assignments to W and $\mathcal{H}$. When the process executes an invocation statement, the history is extended by an output message: $\mathcal{H} := \mathcal{H} \vdash \mathit{this} \rightarrow x.m(\mathrm{E})$. When a process is suspended, a nondeterministic extension of $\mathcal{H}$ captures execution by the environment and by other processes in the same object. The termination of a local process extends $\mathcal{H}$ with a completion message: $\mathcal{H} := \mathcal{H} \vdash \mathit{caller} \leftarrow \mathit{this}.m(\mathrm{E};\mathrm{V})$. For partial correctness reasoning, we may assume that processes are not suspended infinitely long. Consequently, nondeterministic assignment abstractly captures the possible process interleaving.

When reasoning about a method m in a class C we may assume that it has been invoked, which is reflected in the local history by a pending invocation message. Thus, the *method invariant* I_m strengthens the class invariant:

$$I_m(\mathrm{W}, h) \triangleq I_C(\mathrm{W}, h) \land \mathit{pending}(h, \mathit{caller} \rightarrow \mathit{this}.m(\mathrm{X})),$$

where X are the formal in-parameters. A completion message is appended to the history upon method termination, establishing I_C. The interpretation of methods is defined in Fig. 3.

In the encoding of *object creation*, nondeterministic assignments are used to construct unique identifiers. The parent relationship is captured by updating the history with a creation message, which also ensures that the values of the class parameters are visible on the local history of the new object.

Synchronous invocations $x.m(\mathrm{E};\mathrm{V})$ of a method in the environment block the caller's internal activity. Except for the invocation message, there is no output from *this*. The execution of the method is modeled by a nondeterministic assignment to the out-parameters V. Since V might overlap with the values of E, W, and x, a list of fresh pseudo-variables V' captures the execution of the remote method. The completion message is appended to the history and the reply values assigned to V.

The statement $\mathit{this}.m(\mathrm{E};\mathrm{V})$ synchronously invokes the local method m. We may assume that the invariant is preserved by m. By adaptation, the execution of m is captured by a nondeterministic assignment to W and $\mathcal{H}$: if the invariant holds before the assignment, then the invariant also holds for the extended history and the new values of W. Since the invocation is synchronous, the extended history ends with

$$\langle\!\langle\, \textbf{op}\ m(\textbf{in}\ \mathrm{X}\ \textbf{out}\ \mathrm{Y}) == \textbf{var}\ \mathrm{W}_m := \mathrm{E};\ \mathit{body}_m \,\rangle\!\rangle \triangleq \textbf{op}\ m(\textbf{in}\ \mathrm{X},\ \mathit{caller}\ \textbf{out}\ \mathrm{Y}) ==$$
$$\quad \textbf{var}\ \mathrm{W}_m := \mathrm{E}; \langle\!\langle\, \mathit{body}_m \,\rangle\!\rangle;\ \mathcal{H} := \mathcal{H} \vdash \mathit{caller} \leftarrow \mathit{this}.m(\mathrm{X};\mathrm{Y})$$
$$\langle\!\langle\, \textbf{op}\ \mathit{init} == \mathit{body}_{\mathit{init}} \,\rangle\!\rangle \triangleq \textbf{op}\ \mathit{init} == \langle\!\langle\, \mathit{body}_{\mathit{init}} \,\rangle\!\rangle$$
$$\langle\!\langle\, s; \mathrm{S} \,\rangle\!\rangle \triangleq \langle\!\langle\, s \,\rangle\!\rangle; \langle\!\langle\, \mathrm{S} \,\rangle\!\rangle$$
$$\langle\!\langle\, \textbf{skip} \,\rangle\!\rangle \triangleq \textbf{skip}$$
$$\langle\!\langle\, \mathrm{X} := \mathrm{E} \,\rangle\!\rangle \triangleq \mathrm{X} := \mathrm{E}$$
$$\langle\!\langle\, x := \textbf{new}\ C(\mathrm{E}) \,\rangle\!\rangle \triangleq x' := \textbf{some}\ x' \mid \mathit{parent}(x') = \mathit{this} \wedge x' \notin \mathit{oid}(\mathcal{H});$$
$$\quad \mathcal{H} := \mathcal{H} \vdash \mathit{this} \rightarrow x'.\textbf{new}\ C(\mathrm{E});\ x := x'$$
$$\langle\!\langle\, \textbf{if}\ b\ \textbf{then}\ \mathrm{S}_1\ \textbf{else}\ \mathrm{S}_2\ \textbf{fi} \,\rangle\!\rangle \triangleq \textbf{if}\ b\ \textbf{then}\ \langle\!\langle\, \mathrm{S}_1 \,\rangle\!\rangle\ \textbf{else}\ \langle\!\langle\, \mathrm{S}_2 \,\rangle\!\rangle\ \textbf{fi}$$
$$\langle\!\langle\, x.m(\mathrm{E};\mathrm{V}) \,\rangle\!\rangle \triangleq \mathcal{H} := \mathcal{H} \vdash \mathit{this} \rightarrow x.m(\mathrm{E}); \mathrm{V}' := \textbf{some}\ \ \mathrm{V}' \mid\ \mathit{true};$$
$$\quad \mathcal{H} := \mathcal{H} \vdash \mathit{this} \leftarrow x.m(\mathrm{E};\mathrm{V}'); \mathrm{V} := \mathrm{V}',\ \text{where } x \neq \mathit{this}$$
$$\langle\!\langle\, \mathit{this}.m(\mathrm{E};\mathrm{V}) \,\rangle\!\rangle \triangleq \mathcal{H} := \mathcal{H} \vdash \mathit{this} \rightarrow \mathit{this}.m(\mathrm{E}); (\mathrm{W}, h', \mathrm{V}') := \textbf{some}\ (\mathrm{W}', h', \mathrm{V}') \mid$$
$$\quad (I_{m'}(\mathrm{W},\mathcal{H}) \Rightarrow I_{m'}(\mathrm{W}',\mathcal{H} \vdash\!\dashv h')) \wedge h'\ \textbf{ew}\ \mathit{this} \leftarrow \mathit{this}.m(\mathrm{E};\mathrm{V}'); \mathcal{H} := \mathcal{H} \vdash\!\dashv h';$$
$$\qquad \mathrm{V} := \mathrm{V}'$$
$$\langle\!\langle\, \textbf{await}\ \mathit{wait} \,\rangle\!\rangle \triangleq (\mathrm{W}, h') := \textbf{some}\ (\mathrm{W}', h') \mid (I_{m'}(\mathrm{W},\mathcal{H}) \Rightarrow I_{m'}(\mathrm{W}',\mathcal{H} \vdash\!\dashv h'));$$
$$\quad \mathcal{H} := \mathcal{H} \vdash\!\dashv h'$$
$$\langle\!\langle\, \textbf{await}\ b \,\rangle\!\rangle \triangleq \textbf{if}\ b\ \textbf{then}\ \textbf{skip}\ \textbf{else}\ (\mathrm{W}, h') := \textbf{some}\ (\mathrm{W}', h') \mid$$
$$\quad (I_{m'}(\mathrm{W},\mathcal{H}) \Rightarrow I_{m'}(\mathrm{W}',\mathcal{H} \vdash\!\dashv h')) \wedge b^{\mathrm{W}}_{\mathrm{W}'};\ \mathcal{H} := \mathcal{H} \vdash\!\dashv h'\ \textbf{fi}$$
$$\langle\!\langle\, \textbf{await}\ x.m(\mathrm{E};\mathrm{V}) \,\rangle\!\rangle \triangleq \mathcal{H} := \mathcal{H} \vdash \mathit{this} \rightarrow x.m(\mathrm{E}); (\mathrm{W}, h', \mathrm{V}') := \textbf{some}\ (\mathrm{W}', h', \mathrm{V}') \mid$$
$$\quad (I_{m'}(\mathrm{W},\mathcal{H}) \Rightarrow I_{m'}(\mathrm{W}',\mathcal{H} \vdash\!\dashv h')) \wedge \mathit{this} \leftarrow x.m(\mathrm{E};\mathrm{V}') \in h'; \mathcal{H} := \mathcal{H} \vdash\!\dashv h'; \mathrm{V} := \mathrm{V}'$$

Figure 3. The encoding of method declarations in SEQ, where m' denotes the enclosing method of the different statements.

$$(1)\ \mathcal{H} = \langle \mathit{parent}(\mathit{this}) \rightarrow \mathit{this}.\textbf{new}\ C(\mathit{cp}) \rangle \Rightarrow \mathit{wlp}(\mathit{body}_{\mathit{init}},\ \mathit{wf}(\mathcal{H}) \Rightarrow I_C)$$
$$(2)\ I_m \Rightarrow \mathit{wlp}(\textbf{var}\ \mathrm{W}_m := \mathrm{E}; \mathit{body}_m; \mathcal{H} := \mathcal{H} \vdash \mathit{caller} \leftarrow \mathit{this}.m(\mathrm{X};\mathrm{Y}),\ \mathit{wf}(\mathcal{H}) \Rightarrow I_C)$$

Figure 4. Verification conditions for Creol methods. Condition 1 is for *init* to establish the invariant. All remaining methods must preserve the invariant as described by Condition 2.

the completion message of the call. For local calls, the encoding does not rely on properties about the invoked method such as the absence of processor release points or its call structure. (For local calls, use of pre/postconditions strengthens the proof system. For further details, see the extended version of this paper [14].)

Release points are encoded using the same technique; the execution of other processes is modeled by a nondeterministic assignment to W and $\mathcal{H}$ which maintains the invariant. The implications $I_{m'}(\mathrm{W},\mathcal{H}) \Rightarrow I_{m'}(\mathrm{W}',\mathcal{H} \vdash\!\dashv h')$ in Fig. 3 capture this assumption on other processes. The invariant is assumed to hold after a suspension provided that it holds at processor release. The encoding of **await** b has two cases; if

b holds the statement is reduced to **skip**, otherwise the process is suspended. When the process continues after a suspension, b must hold for the current values of W. The encoding of **await** $x.m(\mathrm{E};\mathrm{V})$ resembles that of synchronous invocations and consists of several parts. The initiation message is appended to the history before the processor is released. After the suspension, the actual parameter list is assigned the values found in the completion message. Since there is no transfer of control between caller and callee, the history need not end with the completion message corresponding to the method activation, but this message must be somewhere on the history extension. We end this section with two lemmas.

Lemma 4.1 *The local histories of encoded objects are well-formed and closed.*

Proof By induction over method bodies. □

Lemma 4.2 *The global history is well-formed for dynamic systems initiated by a creation message* $null \rightarrow o.\mathbf{new}\ C(\mathrm{E})$.

Proof By Lemmas 3.4 and 4.1, since the local histories are derived from a global history H by projection, and since $ob(H)$ includes all objects in H. □

5 Class Verification

Proof rules for the language are derived from those of Apt [3, 4] by translation into SEQ. The weakest liberal precondition for nondeterministic assignment is

$$wlp(\mathrm{Y} := (\mathbf{some}\ \mathrm{X} \mid P(\mathrm{X})),\ Q) = \forall \mathrm{X} \mid (P(\mathrm{X}) \Rightarrow Q^{\mathrm{Y}}_{\mathrm{X}}),$$

assuming that X is disjoint from the free variables of Q other than $\{\mathrm{Y}\}$. The rule maintains soundness and relative completeness of Apt's proof system, and the side condition is satisfied by variable renaming. The language has object pointers but no dot notation for attribute access, so pointer reasoning follows standard rules [19]. Fig. 4 presents the verification conditions for methods, based on the weakest liberal preconditions for the different statements. The invariant is assumed as a method precondition and must hold at method termination as part of the contract between processes, *init* is treated separately.

Figure 5 presents the weakest liberal preconditions for the language statements, derived from the encoding in Fig. 3 by requiring that the invariant holds at processor release. The postcondition Q of a statement may range over the local variables W_m of a method m, as well as W, cp, and $\mathcal{H}$. For Boolean guards, $\{I\}$ **await** $b\,\{I \wedge b\}$ and $\{P \wedge b\}$ **await** $b\,\{P \wedge b\}$ follow directly, where P need not imply the invariant. Thus, **await** *true* is identical to **skip**. By backward construction, a sound and relative complete proof system is obtained for method invocations, processor release points, and object creation. For release points the proposed semantics depends on the given invariant, which means that the invariant must be a sufficiently strong precondition to ensure the invariant at the next suspension point (assuming well-formedness).

Theorem 5.1 *The proof system (Fig. 5) for the concurrent object language is sound and relative complete with respect to the semantic encoding (Fig. 3).*

$$wlp(s;\mathrm{S},\ Q) \triangleq wlp(s,\ wlp(\mathrm{S},\ Q))$$

$$wlp(\mathbf{skip},\ Q) \triangleq Q$$

$$wlp(\mathrm{V} := \mathrm{E},\ Q) \triangleq Q^{\mathrm{V}}_{\mathrm{E}}$$

$$wlp(\mathbf{if}\ b\ \mathbf{then}\ \mathrm{S}_1\ \mathbf{else}\ \mathrm{S}_2\ \mathbf{fi},\ Q) \triangleq \mathbf{if}\ b\ \mathbf{then}\ wlp(\mathrm{S}_1,\ Q)\ \mathbf{else}\ wlp(\mathrm{S}_2,\ Q)\ \mathbf{fi}$$

$$wlp(x := \mathbf{new}\ C(\mathrm{E}),\ Q) \triangleq \\ \quad \forall x' \mid (parent(x') = this \wedge x' \notin oid(\mathcal{H})) \Rightarrow Q^{x,\mathcal{H}}_{x',\mathcal{H}\vdash this\rightarrow x'.\mathbf{new}\ C(\mathrm{E})}$$

$$wlp(x.m(\mathrm{E};\mathrm{V}),\ Q) \triangleq \forall \mathrm{V}' \mid Q^{\mathrm{V},\mathcal{H}}_{\mathrm{V}',\mathcal{H}\vdash\dashv this\leftrightarrow x.m(\mathrm{E};\mathrm{V}')},\ \text{where } x \neq this$$

$$wlp(this.m(\mathrm{E};\mathrm{V}),\ Q) \triangleq \forall h', \mathrm{W}', \mathrm{V}' \mid (h'\ \mathbf{ew}\ this \leftarrow this.m(\mathrm{E};\mathrm{V}') \\ \quad \wedge (I_{m'}(\mathrm{W}, h) \Rightarrow I_{m'}(\mathrm{W}', h \vdash\dashv h'))) \Rightarrow Q^{\mathrm{V},\mathrm{W},\mathcal{H}}_{\mathrm{V}',\mathrm{W}',h\vdash\dashv h'}, \\ \quad \text{where } h \triangleq \mathcal{H} \vdash this \rightarrow this.m(\mathrm{E})$$

$$wlp(\mathbf{await}\ wait,\ Q) \triangleq I_{m'}(\mathrm{W}, \mathcal{H}) \wedge \forall \mathrm{W}', h' \mid I_{m'}(\mathrm{W}', \mathcal{H} \vdash\dashv h') \Rightarrow Q^{\mathrm{W},\mathcal{H}}_{\mathrm{W}',\mathcal{H}\vdash\dashv h'}$$

$$wlp(\mathbf{await}\ b,\ Q) \triangleq \mathbf{if}\ b\ \mathbf{then}\ Q \\ \quad \mathbf{else}\ I_{m'}(\mathrm{W}, \mathcal{H}) \wedge \forall \mathrm{W}', h' \mid (I_{m'}(\mathrm{W}', \mathcal{H} \vdash\dashv h') \wedge b^{\mathrm{W}}_{\mathrm{W}'}) \Rightarrow Q^{\mathrm{W},\mathcal{H}}_{\mathrm{W}',\mathcal{H}\vdash\dashv h'}\ \mathbf{fi}$$

$$wlp(\mathbf{await}\ x.m(\mathrm{E};\mathrm{V}),\ Q) \triangleq I_{m'}(\mathrm{W}, h) \wedge \\ \quad \forall \mathrm{W}', h', \mathrm{V}' \mid (I_{m'}(\mathrm{W}', h \vdash\dashv h') \wedge this \leftarrow x.m(\mathrm{E};\mathrm{V}') \in h') \Rightarrow Q^{\mathrm{V},\mathrm{W},\mathcal{H}}_{\mathrm{V}',\mathrm{W}',h\vdash\dashv h'}, \\ \quad \text{where } h \triangleq \mathcal{H} \vdash this \rightarrow x.m(\mathrm{E})$$

Figure 5. Weakest liberal preconditions for the language. The syntax $o \leftrightarrow o'.m(\mathrm{E};\mathrm{V})$ abbreviates $o \rightarrow o'.m(\mathrm{E}) \vdash o \leftarrow o'.m(\mathrm{E};\mathrm{V})$.

Proof Weakest liberal preconditions are derived via the encoding from the weakest liberal preconditions for SEQ. Soundness and relative completeness then follow from the soundness and relative completeness of the proof system for SEQ, as shown in [11,20]. □

6 Example

Consider a class *Buffer* with *put* and *get* methods, a single memory cell, and a link to another buffer object, given in Fig 6. If the buffer receives a call to *put* with argument *e*, it stores *e* in its cell if the buffer is empty. Otherwise, the *put* call is passed on to the *next* buffer (which is dynamically created if *nil*). With this behavior, a buffer instance as seen from the outside appears to be unbounded: there is always room to store an additional element. Similarly, if the buffer receives a call to *get* and there is an element in its cell, this element is returned. Otherwise, the call is passed to the *next* buffer. With this behavior, a buffer instance as seen from the outside implements a FIFO ordering. In order to let a *Buffer* object know the total number of elements in the buffer, it contains an additional counter.

The variable *cnt* counts the elements stored in the linked *Buffer* list, corresponding to the difference between the number of completed *put* and *get* operations. We

```
class Buffer
begin var cell : Object = nil, cnt : Nat = 0, next : Buffer = nil
  op put(in x : Object) == if cnt = 0 then cell := x
         else (if next = nil then next := new Buffer fi); next.put(x) fi;
           cnt := cnt + 1
  op get(out x : Object) == await (cnt > 0); cnt := cnt − 1;
              if cell = nil then next.get(;x) else x := cell; cell := nil fi
end
```

Figure 6. The code for the *Buffer* class.

get the invariant $cnt = \#(\mathcal{H}/\leftarrow this.put) - \#(\mathcal{H}/\leftarrow this.get)$. The proof of the invariant is straightforward; *cnt* only increases before a completion message of *put* and decreases before a completion message of *get*.

Consider the communication order of *Buffer* instances. The *get* operation of a *Buffer* object x will return elements in the same order as they where inserted by the *put* operation. Using history projections we can denote this FIFO property by

$$fifo(x,h) \triangleq (h/\leftarrow x.get).out \leq (h/\leftarrow x.put).in.$$

The FIFO property of *this* object relies on the FIFO property of the successor object *next*. Thus, in order to verify the FIFO property for *this*, we need an assumption on *next*. For this purpose, we reconstruct the buffer content of *next* from the history:

$$\begin{array}{lll}
buf(x,\varepsilon) \triangleq \varepsilon & & \\
buf(x, h \vdash this \rightarrow x.put(in)) & \triangleq & buf(x,h) \vdash in \\
buf(x, h \vdash this \leftarrow x.get(;out)) & \triangleq & rest(buf(x,h)) \\
buf(x, h \vdash msg) & \triangleq & buf(x,h) \qquad \text{for other messages } msg
\end{array}$$

We may now verify the following invariant for the *Buffer* class:

$$fifo(next,\mathcal{H}) \Rightarrow (\mathcal{H}/\leftarrow this.put).in = ((\mathcal{H}/\leftarrow this.get).out\ + cell) \vdash\!\dashv buf(next,\mathcal{H}),$$

where $h+x$ is h for $x = nil$ otherwise $h \vdash x$. The class is implemented using synchronous call statements, which means that the correspondence between invocation messages to and completion messages from the *next* object is tight. An implementation of the methods using asynchronous calls could break the FIFO structure of the buffer. (A different example using asynchronous calls is given in [18].) Notice that the *next* object can easily be identified from the history of a buffer object by a function $next(x,h)$. Focusing on this property, we express the *external* invariant of an instance o of the class *Buffer* as follows:

$$fifo(next(o,h),h) \Rightarrow fifo(o,h).$$

For a dynamic buffer system, one may prove by induction that each buffer object o satisfies the FIFO property $fifo(o,H)$ where H is the global history, using the fact that for finite H there may only be finitely many objects, and that cyclic buffer structures are impossible due to the *parent* assumptions.

7 Related and Future Work

Related work. This paper adapts communication histories [8, 15] to model object communication in a distributed setting. History sequences which reflect message passing have been used to specify and reason about CSP-like languages [10, 23]. Recent papers have addressed reasoning for sequential object-oriented languages [16, 21, 22], covering aspects such as inheritance, subtyping, and dynamic binding. However, reasoning about multithreaded object-oriented languages is more challenging [1, 7, 9]. 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. A sound and complete compositional Hoare logic for asynchronously communicating processes (objects) running in parallel is presented by de Boer [10]. The objects communicate asynchronously by message passing, but in contrast to our work they communicate through FIFO channels, disallowing message overtaking. Object creation in [10] uses the sequence of previously created objects identifiers. In our framework, this sequence is captured by restricting the local history to object creation messages. Olderog and Apt consider transformation of program statements preserving semantical equivalence [20]. This approach is extended in [11] to a general methodology for transformation of language constructions resulting in sound and complete proof systems. The approach resembles our encoding into SEQ, but it is noncompositional in contrast to our work. In particular, extending the transformational approach of [11] to multithreaded systems seems to require interference freedom tests.

Compared to previous work on Creol reasoning [13], this paper considers dynamic systems and presents a more general framework where class semantics and reasoning are significantly simplified by using external invariants based on the observed part of the local history, by label-free primitives for method interaction, and by capturing object creation as part of the observable behavior. Whereas labeled messages provide a unique correspondence between invocation and completion messages, relevant for an operational semantics, it is not oriented towards abstract specification, as needed for compositional component-based reasoning.

Future Work. Creol supports multiple inheritance. We intend to extend the approach of this paper to combine processor release points, multiple inheritance, and history-based compositionality. The combination of nondeterministic assignment and inherited class invariants challenges the transformational approach. For larger programs, tool supported proof systems are needed. For this purpose, we plan to adapt the KeY tool [5] to proof systems for active objects communicating by means of asynchronous method calls.

8 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 consider a small kernel of generalized Creol constructs, and 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. Weakest liberal preconditions are derived based on the encoding. Given sufficiently strong class invariants, the encoding yields sound and relative complete Hoare rules for partial correctness reasoning about Creol classes. The approach allows external specifications of observable behavior to be derived, expressing possible component interaction. In the approach of this paper, compositional reasoning about dynamic systems can be performed without imposing local compatibility restrictions. By making object creation visible in the local histories, compatibility is fully captured by the composition rule. Global welldefinedness follows from local welldefinedness through the formalization of the *parent* notion and local uniqueness of generated objects. The simplicity of the reasoning rules depends on the *asynchronous input property*, the encapsulation of the local state, process synchronization by guards, including guarded asynchronous method calls, and the use of the local history for local reasoning purposes.

Acknowledgement

The authors gratefully acknowledge helpful comments by Willem-Paul de Roever, Wolfgang Ahrendt, and Jasmin Blanchette on previous versions of this paper.

References

[1] E. Ábrahám, F. S. de Boer, W. P. de Roever, and M. Steffen. An assertion-based proof system for multithreaded Java. *Theoretical Computer Science*, 331(2–3):251–290, 2005.

[2] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, Oct. 1985.

[3] 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.

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

[5] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software. The KeY Approach*, volume 4334 of *Lecture Notes in Artificial Intelligence*. Springer, 2007.

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

[7] M. Broy. Distributed concurrent object-oriented software. In O. Owe, S. Krogdahl, and T. Lyche, editors, *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, volume 2635 of *Lecture Notes in Computer Science*, pages 83–96. Springer, 2004.

[8] M. Broy and K. Stølen. *Specification and Development of Interactive Systems.* Monographs in Computer Science. Springer, 2001.

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

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

[11] 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)*, volume 3188 of *Lecture Notes in Computer Science*, pages 111–133. Springer, 2004.

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

[13] J. Dovland, E. B. Johnsen, and O. Owe. Verification of concurrent objects with asynchronous method calls. In *Proceedings of the IEEE International Conference on Software - Science, Technology & Engineering (SwSTE'05)*, pages 141–150. IEEE Computer Society Press, Feb. 2005.

[14] J. Dovland, E. B. Johnsen, and O. Owe. A compositional proof system for dynamic object systems. Research Report 351, Department of Informatics, University of Oslo, Norway, Feb. 2007.

[15] C. A. R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice Hall, Englewood Cliffs, NJ., 1985.

[16] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. S. E. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE 2000)*, volume 1783 of *Lecture Notes in Computer Science*, pages 284–303. Springer, 2000.

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

[18] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.

[19] 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.

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

[21] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *European Symposium un Programming (ESOP '99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 1999.

[22] 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 2001)*, volume 2029 of *Lecture Notes in Computer Science*, pages 300–317. Springer, 2001.

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

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