Universitetet i Oslo
Institutt for informatikk

A Compositional Proof System for Dynamic Object Systems

Johan Dovland,
Einar B. Johnsen,
and Olaf Owe

Research Report 351
(Revised version)

February 2007
Revised March 2008

A Compositional Proof System for Dynamic Object Systems

Johan Dovland, Einar Broch Johnsen, and Olaf Owe

Department of Informatics, University of Oslo
PO Box 1080 Blindern, NO-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. Object-oriented 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 report, 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 report demonstrates that these models not only address the first criticism above, but also the second.

1 Introduction

In order to facilitate reasoning about interacting components in nonterminating and distributed systems, the specification of component behavior should focus on the potential observable interactions between a component and its environment, rather than on internal and low-level implementation details such as the component's internal state variables. However for verification purposes it then becomes necessary to bridge the gap between the internal code of a component and its observable behavior. In this report, we develop a proof system which allows a specification of observable behavior to be derived from the internal code of components in the setting of distributed concurrent objects.

Object orientation is the leading framework for concurrent and distributed systems and is recommended by the RM-ODP [17], but object interaction happens through method calls and changes to shared state variables, which are usually synchronous operations. These interaction mechanisms, derived from the setting of sequential systems, are well suited for tightly coupled systems. They are less suitable in a distributed setting with loosely coupled or externally coordinated components. Here synchronous communication gives rise to undesired and uncontrolled waiting, and possibly deadlock. Furthermore 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. The interference problem related to shared variables reemerges when threads operate concurrently in the same 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 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.

In order to better capture the setting of interacting distributed components we work with the concurrency and communication model of the Creol language [20], 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 report 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 report 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.

Report 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.

2 The Programming Language

We introduce a programming language based on the communication and concurrency aspects of Creol [20], 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 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 [13], 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, followed by a release of the processor. 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

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

Fig. 1. A syntax outline for classes, excluding expressions e and statements s, where C denotes a class name and m a method name. The notation $[M]^{+}_{d}$ denotes one or more repetitions of M with d as delimiter, * indicates zero or more repetitions, and $^{?}$ indicates an optional part.

enabled when the completion message arrives. 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.

During suspension of the execution of a method instance, the object's attributes may be changed by other processes on the same object, but not the variables and parameters local to the method instance. There is read-only access to in-parameters of methods, including the implicit parameter *caller*. Due to nondeterminism in the distributed setting, overtaking of invocation and completion messages is considered possible. Completion messages are identified by the combination of method name, callee, and input values. If several invocations are made by a caller to the same method of an object, the caller cannot distinguish completion messages corresponding to invocation messages with identical values for the formal input parameters. In order to have tighter control, invocation messages may be tagged with unique values by the run-time system, identifying completion messages, as in Creol [20].

Conventional control flow is expressed by an **if** construct, and assignment is expressed by $\mathrm{X} := \mathrm{E}$, where X is a list of disjoint variables to which there is write access, and E is a list of expressions of matching length and type of X.

Object creation is written $x := \textbf{new}\ C(\mathrm{E})$, 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 in 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. Let the function $\mathit{parent} : \mathit{Obj} \rightarrow \mathit{Obj}$ be such that $\mathit{parent}(o)$ is the creator of o, such that parent chains are cycle free, i.e. such that $\mathit{parent}(o) = o \Leftrightarrow o = \mathit{null}$, and such that the property $o \notin \mathit{anc}(o)$ holds, where the function $\mathit{anc} : \mathit{Obj} \rightarrow \mathit{Set}[\mathit{Obj}]$, denoting ancestors, is defined by

$$\mathit{anc}(o) \triangleq \textbf{if}\ \mathit{parent}(o) = \mathit{null}\ \textbf{then}\ \emptyset\ \textbf{else}\ \mathit{parent}(o) \cup \mathit{anc}(\mathit{parent}(o))\ \textbf{fi}$$

Equality is the only executable basic operations on object identifiers.

3 Class Invariants and Observable Behavior

The execution of a distributed system can be represented by the sequence of observable communication messages between system components, a so-called communication history or trace [8, 10, 15]. At any point in time this sequence abstractly captures the system state. Therefore a system may be specified in terms of the finite initial segments of these histories. A *history invariant* is a predicate which holds for all finite sequences in the prefix-closure of the set of traces and consequently for all abstract

Syntactic categories	*Definitions*	*Comments*
g in *Guard*	$g ::= \mathit{wait} \mid b \mid x.m(\mathrm{E};\mathrm{V}) \mid g_1 \wedge g_2$	bool and call
s in *Com*	$s ::= \mathbf{skip} \mid \mathrm{V} := \mathrm{E}$	statement
m in *Mtd*	$\mid x := \mathbf{new}\ C(\mathrm{E})$	object creation
V in $\mathit{Var}^*_{,}$	$\mid \mathbf{if}\ b\ \mathbf{then}\ \mathrm{S}_1\ \mathbf{else}\ \mathrm{S}_2\ \mathbf{fi}$	if-statement
E in $\mathit{Expr}^*_{,}$	$\mid\ !x.m(\mathrm{E})$	async. call
x in *ObjExpr*	$\mid x.m(\mathrm{E};\mathrm{V})$	sync. call
b in *Bool*	$\mid \mathbf{await}\ g$	release point
C in *ClassName*		

Fig. 2. An outline of the imperative language syntax, with typical terms for each category. Capitalized terms denote lists of the given syntactic categories. Thus, E denotes a list of expressions.

system states, expressing safety properties [2]. In order to observe and reason about object creation by means of histories, we will let the history reveal relevant information about object creation.

Sequences are constructed by the empty sequence ε and right append $\_ \vdash \_$. Let $a, b, c : Seq[T]$, $x, y : T$, and $s : Set[T]$. Define projection $\_/\_ : Seq[T] \times Set[T] \rightarrow Seq[T]$ by $\varepsilon/s \triangleq \varepsilon$ and $a \vdash x/s \triangleq$ $\mathbf{if}\ x \in s\ \mathbf{then}\ (a/s) \vdash x\ \mathbf{else}\ a/s\ \mathbf{fi}$. Let $\mathit{first}((\varepsilon \vdash x) \vdash\!\dashv a) = x$ and $\mathit{rest}((\varepsilon \vdash x) \vdash\!\dashv a) = a$. Define the "ends-with" function $\_\mathbf{ew}\_ : Seq[T] \times T \rightarrow Bool$ by $\varepsilon\ \mathbf{ew}\ x \triangleq \mathsf{false}$ and $(a \vdash x)\ \mathbf{ew}\ y \triangleq x = y$, and the "begins-with" function $\_\mathbf{bw}\_ : Seq[T] \times T \rightarrow Bool$ by $\varepsilon\ \mathbf{bw}\ x \triangleq \mathsf{false}$ and $(a \vdash x)\ \mathbf{bw}\ y \triangleq y = \mathit{first}(a \vdash x)$. Let $a\ \mathbf{is}\ b|||c$ denote that a is an arbitrary interleaving of b and c, let $a \vdash\!\dashv b$ concatenate a with b, let $a \leq b$ denote that a is a prefix of b, and let $\#a$ denote the length of 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 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 given by a finite sequence of type $Seq[Msg]$, where Msg consists of messages corresponding to method invocation, method completion and object creation:

Definition 1 (Messages). *Let Obj, Mtd, and Cid be the types of object, method, and class names, respectively, and Data the type of values occurring as actual parameters to method calls, including Obj. Define the following sets of communication messages:*

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

where caller, callee $: Obj$*, mtd* $: Mtd$*, class* $: Cid$*, and* $in, out : List[Data]$.

Graphical representations may be introduced for invocation, completion, and creation messages, respectively, letting the arrow suggest the direction of the message: $caller \rightarrow callee.mtd(in)$, $caller \leftarrow callee.mtd(in; out)$, and $caller \rightarrow callee.\mathbf{new}\ class(in)$. Messages may be decomposed by the functions $\_.caller, \_.callee : Msg \rightarrow Obj$, e.g., $\langle o, o', m, e \rangle.callee \triangleq o'$. The function $\_.in : Msg \rightarrow List[Data]$ returns the list of in-parameters. Completion messages may in addition be decomposed by the function $\_.out$, returning the list of out-parameters.

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 $o \rightarrow o'.\mathbf{new}\ C(\mathrm{E})$ corresponds to the execution of a $\mathbf{new}\ C$ statement in an object o, where E is the actual values of the class parameters and o' is the identity of the new object.

The messages potentially sent or received by an object o are defined as $\mathrm{OUT}_o \triangleq \{msg : IMsg \cup NMsg \mid msg.caller{=}o\} \cup \{msg : CMsg \mid msg.callee{=}o\}$, and $\mathrm{IN}_o \triangleq \{msg : IMsg \cup NMsg \mid msg.callee{=}o\} \cup \{msg : CMsg \mid msg.caller{=}o\}$. 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 the object o. The object creation message is visible to both the new object and its parent, respectively as input and output. It is the first message in the history of the new object, and allows compositional reasoning about dynamically created objects.

Functions are used to extract information from the history. In particular, we define $pending : Seq[Msg] \times IMsg \to Bool$ and $oid : Msg \to Set[Obj]$ as follows:

$$pending(h, o \to o'.m(\mathrm{E})) \triangleq \#(h/o \to o'.m(\mathrm{E})) > \#(h/o \leftarrow o'.m(\mathrm{E};\_))$$

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

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

Well-formed histories. In the asynchronous setting, object 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 object 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 2 (Well-formedness). *Let $h : Seq[Msg]$, $\mathrm{E},\mathrm{E}' : List[Data]$, $o,o' : Obj$, and $m : Mtd$. The* well-formedness predicate *$wf : Seq[Msg] \to Bool$ is inductively defined:*

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

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

The following lemma holds for well-formed histories.

Lemma 1. *A history h is well-formed if the local projection h/o is well-formed and closed for each object $o \in oid(h)$.*

Proof. By induction over h, assuming $wf(h/o)$ and $closed(h,o)$ for each $o \in oid(h)$.

3.1 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 whenever the method suspends itself, assuming that the invariant holds after suspensions and at the beginning of a method. In order 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 [18].

A *user-provided invariant* $I(\mathrm{W},h)$ for a class C ranges over the class variables W of C and the local history sequence h. It may also refer to 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:

$$I_C(\mathrm{W},h) \triangleq I(\mathrm{W},h) \wedge wf(h) \wedge h\ \mathbf{bw}\ (parent(this) \to this.\mathbf{new}\ C(cp)) \tag{1}$$

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 $P^{\mathrm{X}}_{\mathrm{E}}$ denote the substitution of every free occurrence of X in P by E. 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:

$$I_{o:C(\mathrm{E})}(h) \triangleq \exists \mathrm{W} \mid (I_C(\mathrm{W},h))^{this,cp}_{o,\mathrm{E}} \tag{2}$$

The substitutions replace the free occurrences of *this* with o and instantiates the class parameters with the actual ones, and the existential quantifier hides the local state variables.

In order to assert that objects compose, it suffices to compare the local histories. For this purpose, we adapt a composition method introduced by Soundarajan [25, 26]. Local histories must agree on common messages when composed, expressed by projections from the common *global history*.

Consider a system with an initial object o created by an initial invocation message of the form $null \to o.\mathbf{new}\ C(\ldots)$, 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

$$I^*(H) \triangleq \bigwedge_{(o:C(\mathrm{E}))\in ob(H)} I_{o:C(\mathrm{E})}(H/o) \tag{3}$$

The quantification ranges over all objects in the composition, which is a finite number at any execution point. Note that 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. Notice also that we consider dynamic systems where the number and identities of the composed objects are nondeterministic. Lemma 3 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 nondeterministic assignment [14]. Nondeterministic history extensions capture arbitrary activity of environment objects. Thus, 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, which is 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 $h_{in} \in Seq[\mathrm{IN}_o]$, the class invariant $I_C(\mathrm{W}, h)$ of C satisfies the following *asynchronous input property*:

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

In the asynchronous setting, an object may independently decide to send a message and, due to overtaking, messages may arrive in a different order than sent. The invariant of an object should therefore restrict messages seen by the object, but allow the existence of additional input not processed yet; If the invariant holds for h, it should also hold for h merged with h_{in}. Therefore, we find the asynchronous input property natural for asynchronous systems. Note that invariants on h/OUT_{this} are guaranteed to respect the asynchronous input property. Since completion messages give explicit information about the corresponding invocation messages, such invariants are often sufficient.

The Encoding. Consider a simple *sequential* language which consists of the standard syntax

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

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:

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

For partial correctness, we 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. At the class level, 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 processes 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 $\mathcal{H}$ and W. When the process executes an invocation statement, the history is extended by an output message: $\mathcal{H} := \mathcal{H} \vdash this \rightarrow x.m(\mathrm{E})$. When a process is suspended waiting for a reply, 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: $\mathcal{H} := \mathcal{H} \vdash caller \leftarrow this.m(\mathrm{E}; \mathrm{V})$. For partial correctness reasoning, we may assume that processes are not suspended infinitely long. Consequently, nondeterministic assignment captures the possible interleaving of processes in an abstract manner.

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 associated with $\mathbf{op}\ m(\mathbf{in}\ \mathrm{X}\ \mathbf{out}\ \mathrm{Y}) == \mathbf{var}\ \mathrm{W}_m := \mathrm{E}; body_m$, strengthens the class invariant:

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

$$
\begin{array}{l}
\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 \\
\qquad \textbf{op}\ m(\textbf{in}\ \mathrm{X},\ \mathit{caller}\ \textbf{out}\ \mathrm{Y}) == \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 \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 := \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});\ \mathcal{H} := \mathcal{H} \vdash \mathit{this} \rightarrow x'.\textbf{new}\ C(\mathrm{E});\ x := x' \\
\langle\langle\, !x.m(\mathrm{E}) \rangle\rangle \triangleq \mathcal{H} := \mathcal{H} \vdash \mathit{this} \rightarrow x.m(\mathrm{E}) \\
\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 \mathsf{true}; \\
\qquad \mathcal{H} := \mathcal{H} \vdash \mathit{this} \leftarrow x.m(\mathrm{E};\mathrm{V}'); \mathrm{V} := \mathrm{V}', \qquad\qquad\qquad\qquad \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}); \\
\qquad (\mathrm{W}, h', \mathrm{V}') := \textbf{some}\ (\mathrm{W}', h', \mathrm{V}') \mid h'\ \textbf{ew}\ \mathit{this} \leftarrow \mathit{this}.m(\mathrm{E};\mathrm{V}') \wedge \\
\qquad\qquad (I_{m'}(\mathrm{W},\mathcal{H}) \Rightarrow I_{m'}(\mathrm{W}', \mathcal{H} \vdash\!\dashv h')) \wedge\ (\forall \mathrm{Z} \mid S \Rightarrow R^{\mathrm{W},\mathcal{H}}_{\mathrm{W}',\mathcal{H} \vdash\!\dashv h'}); \mathcal{H} := \mathcal{H} \vdash\!\dashv h'; \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'));\ \mathcal{H} := \mathcal{H} \vdash\!\dashv h' \\
\langle\langle \textbf{await}\ b \rangle\rangle \triangleq \textbf{if}\ b\ \textbf{then skip} \\
\qquad \textbf{else}\ (\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')) \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}); \\
\qquad (\mathrm{W}, h', \mathrm{V}') := \textbf{some}\ (\mathrm{W}', h', \mathrm{V}') \mid (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'; \\
\qquad \mathcal{H} := \mathcal{H} \vdash\!\dashv h'; \mathrm{V} := \mathrm{V}'
\end{array}
$$

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

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 internal activity in the caller. Except for the invocation message, there is no output from *this*. The execution of the invoked 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 $\mathit{this}.m(\mathrm{E};\mathrm{V})$ invokes the local method m in a synchronous manner. The caller may rely on the invariant to be preserved by the call, in addition to a possible pre/post specification S/R of the method. In Fig.3, Z denotes $FV[S,R] \setminus \{\mathrm{W}, cp, \mathcal{H}\}$, where $FV[P]$ denotes the set of free variables in a predicate (list) P. By adaptation, execution of an invocation of m is then modeled by a nondeterministic assignment to W and $\mathcal{H}$ such that if the precondition and/or the invariant holds immediately before the assignment, the postcondition and/or the invariant holds for the extended history and the new values of W. Since the invocation is synchronous, the extended history must end with 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.

Release points are encoded using the same technique; the execution of other processes is modeled by a nondeterministic assignment to W and $\mathcal{H}$ such that the invariant holds for the new values. 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. In the encoding of **await** b there are two cases for b; 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

$$
\begin{array}{ll}
(1) & \mathcal{H} = \langle parent(this) \rightarrow this.\mathbf{new}\, C(cp)\rangle \Rightarrow wlp(body_{init},\ wf(\mathcal{H}) \Rightarrow I_C) \\
(2) & I_m \Rightarrow wlp(\mathbf{var}\ \mathrm{W}_m := \mathrm{E}; body_m; \mathcal{H} := \mathcal{H} \vdash caller \leftarrow this.m(\mathrm{X};\mathrm{Y}),\ wf(\mathcal{H}) \Rightarrow I_C) \\
(3) & S \wedge pending(\mathcal{H}, this \rightarrow this.m(\mathrm{X})) \Rightarrow \\
 & \qquad wlp(\mathbf{var}\ \mathrm{W}_m := \mathrm{E}; body_m; \mathcal{H} := \mathcal{H} \vdash this \leftarrow this.m(\mathrm{X};\mathrm{Y}),\ wf(\mathcal{H}) \Rightarrow R)
\end{array}
$$

Fig. 4. Verification conditions for Creol methods. Condition 1 is for *init* methods, ensuring that the invariant holds upon termination. All remaining methods must preserve the invariant as described by Condition 2. The third condition is for each method m with a precondition S and a postcondition R, providing additional knowledge for for local synchronous calls.

values of W. The encoding of **await** $x.m(\mathrm{E};\mathrm{V})$ resembles that of synchronous invocation 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. There is no explicit transfer of control between caller and callee, which means that the history need not end with the completion message corresponding to the method activation. The message is only known to exist somewhere on the history extension. A release point with a conjunction of guards may be separated into a sequence of release point with simple guards by the equations

$$
\begin{aligned}
\mathbf{await}\ wait \wedge wait \quad &= \mathbf{await}\ wait \\
\mathbf{await}\ x.m(\mathrm{E};\mathrm{V}) \wedge g &= \mathbf{await}\ x.m(\mathrm{E};\mathrm{V}); \mathbf{await}\ g
\end{aligned}
$$

together with associativity and commutativity of conjunction. We conclude this section with two lemmas.

Lemma 2. *The local histories of encoded objects are well-formed and closed.*

Proof. By induction over method bodies. All statements preserve the properties.

Using Lemma 1 and Lemma 2, well-formedness of the global history can be established.

Lemma 3. *For dynamic systems initiated by an initial invocation message* $null \rightarrow o.\mathbf{new}\ C(\mathrm{E})$*, the global history is well-formed.*

Proof. By Lemma 1 and 2 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 the proof system of Apt [3, 4] by the 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 may easily be satisfied by variable renaming. The language has object pointers but no dot notation for accessing attributes, thus pointer reasoning can be done according to standard rules [21]. Fig. 4 presents the verification conditions for methods, based on the weakest liberal preconditions for the different language statements. The invariant is assumed as a precondition to methods. As a part of the contract between processes, the invariant must be established at method termination. Local method calls may in addition be specified in terms of pre- and postconditions. The *init* method is treated separately.

$$
\begin{aligned}
&\mathit{wlp}(s;\textsc{s},\ Q) \triangleq \mathit{wlp}(s,\ \mathit{wlp}(\textsc{s},\ Q))\\
&\mathit{wlp}(\mathbf{skip},\ Q) \triangleq Q\\
&\mathit{wlp}(\textsc{v} := \textsc{e},\ Q) \triangleq Q^{\textsc{v}}_{\textsc{e}}\\
&\mathit{wlp}(\mathbf{if}\ b\ \mathbf{then}\ \textsc{s}_1\ \mathbf{else}\ \textsc{s}_2\ \mathbf{fi},\ Q) \triangleq \mathbf{if}\ b\ \mathbf{then}\ \mathit{wlp}(\textsc{s}_1,\ Q)\ \mathbf{else}\ \mathit{wlp}(\textsc{s}_2,\ Q)\ \mathbf{fi}\\
&\mathit{wlp}(x := \mathbf{new}\ C(\textsc{e}),\ Q) \triangleq \forall x' \mid (\mathit{parent}(x') = \mathit{this} \wedge x' \notin \mathit{oid}(\mathcal{H})) \Rightarrow Q^{x,\mathcal{H}}_{x',\mathcal{H}\vdash \mathit{this}\rightarrow x'.\mathbf{new}\ C(\textsc{e})}\\
&\mathit{wlp}(!x.m(\textsc{e}),\ Q) \triangleq Q^{\mathcal{H}}_{\mathcal{H}\vdash \mathit{this}\rightarrow x.m(\textsc{e})}\\
&\mathit{wlp}(x.m(\textsc{e};\textsc{v}),\ Q) \triangleq \forall \textsc{v}' \mid Q^{\textsc{v},\mathcal{H}}_{\textsc{v}',\mathcal{H}\vdash\!\dashv \mathit{this}\leftrightarrow x.m(\textsc{e};\textsc{v}')}, \qquad \text{where } x \neq \mathit{this}\\
&\mathit{wlp}(\mathit{this}.m(\textsc{e};\textsc{v}),\ Q) \triangleq \forall h', \textsc{w}', \textsc{v}' \mid (h'\ \mathbf{ew}\ \mathit{this}\leftarrow \mathit{this}.m(\textsc{e};\textsc{v}') \wedge\\
&\qquad (\forall \textsc{z} \mid (S^{\mathcal{H}}_{h} \Rightarrow R^{\textsc{w},\mathcal{H}}_{\textsc{w}',h\vdash\!\dashv h'})) \wedge (I_{m'}(\textsc{w},h) \Rightarrow I_{m'}(\textsc{w}',h\vdash\!\dashv h'))) \Rightarrow Q^{\textsc{v},\textsc{w},\mathcal{H}}_{\textsc{v}',\textsc{w}',h\vdash\!\dashv h'},\\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{where } h \triangleq \mathcal{H} \vdash \mathit{this}\rightarrow \mathit{this}.m(\textsc{e})\\
&\mathit{wlp}(\mathbf{await}\ \mathit{wait},\ Q) \triangleq I_{m'}(\textsc{w},\mathcal{H}) \wedge \forall \textsc{w}', h' \mid I_{m'}(\textsc{w}',\mathcal{H}\vdash\!\dashv h') \Rightarrow Q^{\textsc{w},\mathcal{H}}_{\textsc{w}',\mathcal{H}\vdash\!\dashv h'}\\
&\mathit{wlp}(\mathbf{await}\ b,\ Q) \triangleq\\
&\qquad \mathbf{if}\ b\ \mathbf{then}\ Q\ \mathbf{else}\ I_{m'}(\textsc{w},\mathcal{H}) \wedge \forall \textsc{w}', h' \mid (I_{m'}(\textsc{w}',\mathcal{H}\vdash\!\dashv h') \wedge b^{\textsc{w}}_{\textsc{w}'}) \Rightarrow Q^{\textsc{w},\mathcal{H}}_{\textsc{w}',\mathcal{H}\vdash\!\dashv h'}\ \mathbf{fi}\\
&\mathit{wlp}(\mathbf{await}\ x.m(\textsc{e};\textsc{v}),\ Q) \triangleq I_{m'}(\textsc{w},h) \wedge\\
&\qquad \forall \textsc{w}', h', \textsc{v}' \mid (I_{m'}(\textsc{w}',h\vdash\!\dashv h') \wedge \mathit{this}\leftarrow x.m(\textsc{e};\textsc{v}') \in h') \Rightarrow Q^{\textsc{v},\textsc{w},\mathcal{H}}_{\textsc{v}',\textsc{w}',h\vdash\!\dashv h'},\\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{where } h \triangleq \mathcal{H} \vdash \mathit{this}\rightarrow x.m(\textsc{e})
\end{aligned}
$$

Fig. 5. Weakest liberal preconditions for the language. The syntax $o \leftrightarrow o'.m(\textsc{e};\textsc{v})$ abbreviates $o \rightarrow o'.m(\textsc{e}) \vdash o \leftarrow o'.m(\textsc{e};\textsc{v})$.

Fig. 5 presents the weakest liberal preconditions for the different language statements, derived from the encoding in Fig. 3 by requiring the invariant to hold when the processor is released. The postcondition Q of the different statements may range over the local variables $\textsc{w}_m$ of a method m, in addition to $\textsc{w}$, *cp*, and $\mathcal{H}$. For boolean guards, the triples $\{I\}\,\mathbf{await}\ b\,\{I \wedge b\}$ and $\{P \wedge b\}\,\mathbf{await}\ b\,\{P \wedge b\}$ follow directly, where P need not imply the invariant. Thus, **await** `true` is identical to **skip**. By backward construction, a sound and relative complete reasoning 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 1. *The proof system (Fig. 5) for the concurrent object language is sound and relative complete with respect to the semantic encoding (Fig. 3).*

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 [12, 22].

We conclude this section with some examples illustrating different aspects of the reasoning system. The first one illustrates the use of pre/post specifications in combination with history based invariants. The second example shows how non-compatible histories possibly leading to deadlock can be ruled out by composition. The last example illustrates how message delay and method overtaking are captured by the reasoning system.

Example. This example illustrates how pre/post specifications of methods can be used in combination with the class invariant when we reason about local synchronous calls. The example also illustrates how inductive functions over the history can be used to establish a connection between the inner state of an object and the observable communication. Consider the following class providing two fetch-and-add methods *inc* and *inc2* which increment a counter by 1 and 2, respectively.

```
class Inc
   var $val : Nat = 0$
   op $inc($out $res : Nat) == res := val;\ val := val + 1$
   op $inc2($out $res : Nat) ==$ var $tmp : Nat = 0;\ this.inc(;res);\ this.inc(;tmp)$
end
```

The declaration of *init* is omitted for brevity. The method *inc* increases *val* by one and returns the initial value of *val*. The method *inc2* also returns the initial value of *val*, but increases the value of *val* by two. The value is increased by two local calls to *inc*. We can define an invariant for this class in terms of the two functions $V : Obj \times Seq[Msg] \rightarrow Nat$ and $Out : Obj \times Nat \times Seq[Msg] \rightarrow Bool$ defined inductively over the history:

$$\begin{array}{llll}
V(x,\varepsilon) & \triangleq 0 & Out(x,w,\varepsilon) & \triangleq \mathsf{true} \\
V(x,h \vdash \leftarrow x.inc(;v)) & \triangleq V(x,h)+1 & Out(x,w,h \vdash \leftarrow x.inc(;v)) & \triangleq Out(x,w,h) \wedge v = w-1 \\
V(x,h \vdash msg) & \triangleq V(x,h) & Out(x,w,h \vdash \leftarrow x.inc2(;v)) & \triangleq Out(x,w,h) \wedge v = w-2 \\
 & & Out(x,w,h \vdash msg) & \triangleq Out(x,w,h)
\end{array}$$

Here, $\leftarrow x.m(;\mathrm{E})$ matches all completion messages of the specified method. An occurrence of *msg* in an inductive function definition matches all messages not matching any of the other cases. An invariant can then be defined over *val* and $\mathcal{H}$ by:

$$I(val,\mathcal{H}) \triangleq val = V(this,\mathcal{H}) \wedge Out(this,val,\mathcal{H})$$

The invariant only restricts $\mathcal{H}/\mathrm{OUT}_{this}$ and satisfies therefore the asynchronous input property. For an instance of the class, the invariant will relate the value of *val* to the observable object communication. The class invariant I_{Inc} can then be defined as a strengthening of I:

$$I_{Inc}(val,\mathcal{H}) \triangleq I(val,\mathcal{H}) \wedge wf(\mathcal{H}) \wedge \mathcal{H}\ \mathbf{bw}(parent(this) \rightarrow this.\mathbf{new}\ Inc())$$

An external history invariant $I(h)$ can be defined by hiding the class attribute:

$$I(h) \triangleq I_{Inc}(V(this,h),h)$$

The external invariant of an instance o of *Inc* is $I(h)$ with every occurrence of *this* replaced by o. Next, we consider some internal verification details of the class invariant. The method *inc* is verified using Condition 2 of Fig 4:

$$I_{inc} \Rightarrow wlp(res := val;\ val := val+1;\ \mathcal{H} := \mathcal{H} \vdash caller \leftarrow this.inc(;res),\ wf(\mathcal{H}) \Rightarrow I_{Inc})$$

where I_{inc} is the method invariant of the *inc* method:

$$I_{inc}(val,\mathcal{H}) \triangleq I_{Inc}(val,\mathcal{H}) \wedge pending(\mathcal{H}, caller \rightarrow this.inc)$$

The verification condition follows by standard reasoning about assignment. Correspondingly, an invariant maintenance proof for *inc2* is based on the following verification condition:

$$\begin{array}{l}
I_{inc2} \Rightarrow wlp(\mathbf{var}\ tmp : Nat = 0;\ this.inc(;res);\ this.inc(;tmp); \\
\qquad\qquad\qquad\qquad \mathcal{H} := \mathcal{H} \vdash caller \leftarrow this.inc2(;res), wf(\mathcal{H}) \Rightarrow I_{Inc})
\end{array}$$

However, this verification condition is not provable without further knowledge about local invocations of *inc*. The invariant does not express that *val* is increased by one due to a local synchronous invocation of *inc*. The needed information can be expressed as a pre/post specification of *inc* by using a logical variable v_0: $\{val = v_0\}\, inc(\mathbf{out}\; res : Nat)\, \{val = v_0 + 1\}$. Ignoring the analysis of $\mathcal{H}$, this specification can be verified by an instance of Condition 3 in Fig. 4:

$$val = v_0 \Rightarrow wlp(res := val;\; val = val + 1,\; val = v_0 + 1)$$

By the weakest liberal preconditions of Fig 5, we can then assume the following implication in the weakest liberal preconditions of the local calls: $\forall v_0 \,|\, val = v_0 \Rightarrow val' = v_0 + 1$, where val' represents the value of *val* after the call. This assumption relates the states before and after a local synchronous call to *inc*, and a proof of the verification condition of *inc2* can thereby be derived.

Example. Execution of a synchronous invocation statement will block until the reply arrives. Deadlock may then occur if a cycle of synchronous method calls arises during execution. If the object invariants requires a certain call structure, histories leading to deadlock situations may be ruled out by composition due to non-compatibility between the different local requirements. Consider the following classes:

$\mathbf{class}\; A \quad \mathbf{op}\; m \;==\; caller.n()\; \mathbf{end}$

$\mathbf{class}\; B(o : A) \quad \mathbf{op}\; init \;==\; o.m() \quad \mathbf{op}\; n \;==\; \mathbf{skip\; end}$

The structure of a sequence may described by regular expressions. Let $h\; \mathbf{bel}\; a; [b \;[]\; c]^*$ denote that the sequence h is build up by a followed by zero or more repetitions of b or c, where a, b, and c are semicolon separated lists of messages. Thus, $h\; \mathbf{bel}\; [a \;[]\; b]^*$ implies that $(h \vdash\!\dashv a)\; \mathbf{bel}\; [a \;[]\; b]^*$ and that $(h \vdash\!\dashv b)\; \mathbf{bel}\; [a \;[]\; b]^*$. Define $RA : Obj \times Seq[Msg] \rightarrow Bool$ and $RB : Obj \times Obj \times Seq[Msg] \rightarrow Bool$ by

$$\begin{aligned} RA(x,h) &\triangleq h\; \mathbf{bel}\; [\exists y \mid x \rightarrow y.n; x \leftarrow y.n; y \leftarrow x.m]^* \\ RB(x,y,h) &\triangleq h\; \mathbf{bel}\; x \rightarrow y.m; x \leftarrow y.m; [\leftarrow x.n]^* \end{aligned}$$

Using these functions, we can define invariants I_A and I_B for the classes A and B, respectively, by:

$$\begin{aligned} I_A &\triangleq RA(this, \mathcal{H} \backslash \rightarrow this) \\ I_B &\triangleq RB(this, o, \mathcal{H} \backslash \rightarrow this) \end{aligned}$$

where $h \backslash \mathrm{S}$ denotes the sequence h except messages belonging to the set S. Thus, $\mathcal{H} \backslash \rightarrow this$ will ignore invocation messages to *this*, ensuring the asynchronous input property for the two invariants. For two instances $a : A$ and $b : B(a)$, the composed invariant is:

$$I^*(H) \triangleq RA(a, (H/a) \backslash \rightarrow a) \wedge RB(b, a, (H/b) \backslash \rightarrow b)$$

The object b will block after invoking $a.m$ whereas a will block after invoking $b.n$. This is reflected on the possible global histories; the only prefix that is compatible with both a and b is the history consisting of the two invocation messages. After that, the the requirements of the two invariants are no longer compatible.

Example. In the distributed setting with asynchronous method calls, messages can be delayed, and message overtaking is possible. The *pending* assumption of the method invariant reflects the loose connection between the caller and the callee. When a method instance starts execution, we only assume that there is a pending invocation message of the method on the local history. For local reasoning, no assumptions are made about the order of this message relative relative to other messages on the local history. (If an invariant restricted the order of input messages, the invariant would not satisfy the asynchronous input property.) Therefore, local invariants are verified without assumptions on the relative order of input messages. A *caller* may on the other hand impose a certain order of two emitted invocation messages. Local histories are required to be derivable from the global history by projection, which means that the global history will reflect the relative sending order of messages. The following example illustrates that a particular order of two invocation messages on the global history will not imply a particular execution order by the callee. The execution order may result in different observable behavior of the callee, and possible behavior is reflected by the global invariant. Consider the class:

```
class Rec
  var $a, b : Nat, Nat = 0, 0$
  op $n$ == await $(a = 0)$; $a := 1$; if $b = 1$ then $!caller.do1$ fi
  op $m$ == await $(b = 0)$; $b := 1$; if $a = 1$ then $!caller.do2$ fi
end
```

The attributes can be interpreted as two flags, the flag a is set by n, and the flag b is set by m. When n is executed, it checks whether the flag of m is set, and method *do1* provided by the caller is called if that flag is set. Likewise, an execution of m will lead to a callback to *do2* if the flag of n is set. The observable behavior of an instance of *Rec* therefore depends on the execution order of the two methods. Execution of n before m will lead to an invocation of *do2*, and execution of m before n will lead to an invocation of *do1*. For an instance o of the class, a possible invariant becomes:

$$\begin{aligned} I_{o:Rec}(h) \triangleq\ & \#(h/\leftarrow o.n) \leq 1 \wedge \#(h/\leftarrow o.m) \leq 1 \wedge \\ & (\#(h/\leftarrow o.n) + \#(h/\leftarrow o.m) = 2) \Rightarrow \\ & \qquad \exists o' \mid h/\mathrm{OUT}_o \ \mathbf{ew}\ [o \rightarrow o'.do1; o' \leftarrow o.n \,[]\, o \rightarrow o'.do2; o' \leftarrow o.m] \end{aligned}$$

where h **ew** $[a \,[]\, b]$ denotes that h ends with either a or b. Notice that the invariant of o does not assume any execution order or any order of the invocation messages of m and n. Even under assumptions about the order of input messages, the invariant of o allows both *do1* and *do2*. This is illustrated by the following class calling the different methods of o:

```
class Clr(o : Rec)
  op init == !o.n; !o.m
  op op1 == skip
  op op2 == skip
end
```

For an instance o' of *Clr*, we have $I_{o':Clr(o)}(h) \triangleq h/\mathrm{OUT}_{o'}$ **bel** $o' \rightarrow o.n; o' \rightarrow o.m; [\leftarrow o'.do1 \,[]\, \leftarrow o'.do2]^*$. On possible global histories, the invocation message of n must therefore occur before the invocation message of m. By composition, the only possible (terminated) global histories H satisfying both $I_{o:Rec}(H/o)$ and $I_{o':Clr(o)}(H/o')$ are:

$$\begin{aligned} a)\ & o' \rightarrow o.n;\ o' \rightarrow o.m;\ o' \leftarrow o.n;\ o \rightarrow o'.do2;\ o' \leftarrow o.m;\ o \leftarrow o'.do2 \\ b)\ & o' \rightarrow o.n;\ o' \rightarrow o.m;\ o' \leftarrow o.m;\ o \rightarrow o'.do1;\ o' \leftarrow o.n;\ o \leftarrow o'.do1 \\ c)\ & o' \rightarrow o.n;\ o' \leftarrow o.n;\ o' \rightarrow o.m;\ o \rightarrow o'.do2;\ o' \leftarrow o.m;\ o \leftarrow o'.do2 \end{aligned}$$

In addition, there are three more histories corresponding to the three above, but with the last two messages in reverse order. Consequently, the global invariant allows both *do1* and *do2* as answers to the two invocations made by o'.

In order to ensure a specific execution order of n and m, the caller must wait for reply to the first method before calling the other. This can be done by using a synchronous invocation statement in *init*:

$$\textbf{op}\ init\ ==\ o.n;\ !o.m$$

The above invariant $I_{o':Clr(o)}(h)$ can then be strengthened by $(h \backslash \rightarrow o')$ **bw** $o' \rightarrow o.n;\ o' \leftarrow o.n;\ o' \rightarrow o.m$. This requirement restricts the possible global histories, and only alternative $c)$ allows the required message order. The global invariant thereby expresses that *do2* will be invoked.

6 Unbounded Buffer Example

Consider a class *Buffer* with *put* and *get* methods, a single memory cell, and a link to another buffer object. 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 code for the *Buffer* class is as follows:

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

Starting at the initial *Buffer* object, the variable *cnt* represents the number of elements stored in the linked list. The value of *cnt* equals the number of completed *put* operations, minus the number of completed *get* operations, expressed by the invariant $cnt = \#(\mathcal{H}/\leftarrow this.put) - \#(\mathcal{H}/\leftarrow this.get)$. The proof of this invariant is straightforward; *cnt* is only increased before a completion message of *put* and decreased before a completion message of *get*.

Next we 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:

$$\begin{array}{lll}
Buf(x,y,\varepsilon) & \triangleq \varepsilon & \\
Buf(x,y,h \vdash x \leftarrow y.put(v)) & \triangleq Buf(x,y,h) \vdash v & \\
Buf(x,y,h \vdash x \leftarrow y.get(;v)) & \triangleq rest(Buf(x,y,h)) & \\
Buf(x,y,h \vdash msg) & \triangleq Buf(x,y,h) & \text{for other messages } msg
\end{array}$$

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

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

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. 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* assumption.

Using the two invariants above, we can express that *cnt* equals the number of elements buffered in *next* plus one if *cell* is not *nil*: $cnt = \#(Buf(this,next,\mathcal{H})+cell)$. Furthermore, the length of *Buf* must be non-negative: $\#(Buf(this,next,\mathcal{H})) \geq 0$. From the latter, it follows that an invocation of *next.get* always is proceeded by at least one invocation of *next.put*. It follows directly from the body of *put* that *next* is different from *nil* whenever *next.put* is called.

The function $Next : Obj \times Seq[Msg] \rightarrow Obj$ is now defined, together with a function $F : Obj \times Seq[Msg] \rightarrow Bool$ expressing that *next* is only instantiated once:

$$\begin{array}{llll}
Next(x,\varepsilon) & \triangleq nil & F(x,\varepsilon) & \triangleq \mathsf{true} \\
Next(x,h \vdash x \rightarrow y.\mathbf{new}\ Buffer) & \triangleq y & F(x,h \vdash x \rightarrow y.\mathbf{new}\ Buffer) & \triangleq Next(x,h) = nil \\
Next(x,h \vdash msg) & \triangleq Next(x,h) & F(x,h \vdash msg) & \triangleq F(x,h)
\end{array}$$

Now we may define the invariant $next = Next(this,\mathcal{H}) \wedge F(this,\mathcal{H})$, which is maintained by both methods.

7 Related and Future Work

Related work. In this report we have adapted communication histories [8, 10, 15] 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 [11, 25]. Recent work has addressed reasoning about sequential object-oriented languages [16,23,24], covering various 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. In [11], 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. Object creation in [11] is described using the sequence of objects identities already created by the considered object. In our framework, this sequence is captured by restricting the local history to object creation messages. Olderog and Apt [22] consider transformation of program statements preserving semantical equivalence. This approach is further developed in [12], 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 into SEQ, but it is non-compositional in contrast to our work. In particular, extending the transformational approach of [12] to multithreaded systems seems to require interference freedom tests.

In contrast to previous work on Creol reasoning [14], we here consider dynamic systems, and present a more general framework where class semantics and reasoning are significantly simplified by the notion of external invariants based on the observed part of the local history, by label-free primitives for method based interaction, and by capturing object creation as part of the observable behavior. Whereas labeled messages provide a unique correspondence between invocation and completion messages, which is relevant for an operational semantics, it is not oriented towards abstract specification, as needed for compositional component-based reasoning.

Future Work. Creol has been extended with constructs for multiple inheritance [19]. It is our present research goal to extend the approach to compositional verification presented in this report 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 verify larger programs, tool support to discharge proof conditions is necessary. In the context of the EU project Credo, an adaptation of the KeY tool [5] is investigated for this purpose.

The long term goal of our research is to study openness in distributed systems, taking an object-oriented approach. While this report 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. A natural way to provide some openness 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. 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. The work presented in this report 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.

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 report, 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. 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. The approach allows external specifications of observable behavior to be derived, expressing possible component interaction. 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, 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. 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, France, Dec. 1977.
11. F. S. de Boer. A Hoare logic for dynamic networks of asynchronously communicating deterministic processes. *Theoretical Computer Science*, 274:3–41, 2002.
12. 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.
13. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, Aug. 1975.
14. 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.
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. 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*, volume 2635 of *Lecture Notes in Computer Science*, pages 137–164. Springer-Verlag, 2004.
19. E. B. Johnsen and O. Owe. Inheritance in the presence of asynchronous method calls. In *Proc. 38th Hawaii International Conference on System Sciences (HICSS'05)*. IEEE Computer Society Press, Jan. 2005.
20. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
21. 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.
22. 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.
23. 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.
24. 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.
25. N. Soundararajan. Axiomatic semantics of communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 6(4):647–662, Oct. 1984.
26. N. Soundararajan. A proof technique for parallel programs. *Theoretical Computer Science*, 31(1-2):13–29, May 1984.