Observable Behavior of Distributed Systems: Component Reasoning for Concurrent Objects

Crystal Chang Din, Johan Dovland, Einar Broch Johnsen, and Olaf Owe

Dept. of Informatics – Univ. of Oslo,
P.O. Box 1080 Blindern, N-0316 Oslo, Norway.
E-mails: {crystald,johand,einarj,olaf}@ifi.uio.no

Abstract

Distributed and concurrent object-oriented systems are difficult to analyze due to the complexity of their concurrency, communication, and synchronization mechanisms. Rather than performing analysis at the level of code in, e.g., Java or C++, we consider the analysis of such systems at the level of an abstract, executable modeling language. This language, based on concurrent objects communicating by asynchronous method calls, avoids some difficulties of mainstream object-oriented programming languages related to compositionality and aliasing. To facilitate system analysis, *compositional verification systems* are needed, which allow components to be analyzed independently of their environment. In this paper, a proof system for partial correctness reasoning is established based on communication histories and class invariants. A particular feature of our approach is that the alphabets of different objects are completely disjoint. Compared to related work, this allows the formulation of a much simpler Hoare-style proof system and reduces reasoning complexity by significantly simplifying formulas in terms of the number of needed quantifiers. The soundness and relative completeness of this proof system are shown using a transformational approach from a sequential language with a non-deterministic assignment operator.

Key words: Distributed systems, Object-orientation, Compositional reasoning, Hoare Logic

1. Introduction

Distributed systems play an essential role in society today. For example, distributed systems form the basis for critical infrastructure in different domains such as finance, medicine, aeronautics, telephony, and Internet services. The quality of such distributed systems is often crucial. However, quality assurance is non-trivial since the systems depend on unpredictable factors including different processing speeds of independent components and network transmission speeds. It is highly challenging to test such distributed systems after deployment under different relevant conditions.

Object orientation is the leading framework for concurrent and distributed systems, recommended by the RM-ODP [1]. Many distributed systems are today programmed in

[☆]This work was done in the context of the EU project FP7-231620 *HATS: Highly Adaptable and Trustworthy Software using Formal Models* (`http://www.hats-project.eu`).

object-oriented, imperative languages such as Java and C++. Programs written in these languages are in general difficult to analyze due to composition and alias problems, and due to the complexity of their concurrency, communication, and synchronization mechanisms. Therefore, it may be easier to consider a *model* of the program at a suitable level, using an idealized object-oriented language which is easier to analyze. This motivates frameworks combining precise modeling and analysis, with suitable tool support. In particular, *compositional verification systems* are needed, which allow the different components to be analyzed independently from their surrounding components.

In this paper, we consider *ABS*, a high-level imperative object-oriented modeling language, based on the concurrency and synchronization model of *Creol* [2], but ignoring other aspects of *Creol* such as inheritance, which are not in the focus of this paper. *ABS* supports concurrent objects with an asynchronous communication model that is suitable for loosely coupled objects in a distributed setting. The language avoids some of the aforementioned difficulties of analyzing distributed systems at the level of, e.g., Java and C++. In particular, the concurrent object model of *ABS* is *inherently compositional* [3]: In *ABS*, there is *no direct access* to the internal state variables of other objects and object communication is by means of *asynchronous method calls* only.

A concurrent object has its own execution thread. Asynchronous method calls do not transfer control between the caller and the callee. This way undesirable waiting is avoided in the distributed setting, because execution in one object need not depend on the responsiveness of other objects. Asynchronous method calls resemble the spawning of new threads in the multi-thread concurrency model. A consequence of the asynchronous communication model is that an object can have several processes to execute, stemming from different method activations. Internally in an *ABS* object, there is at most one process executing at a time, and intra-object synchronization is programmed explicitly by *processor release points*. Concurrency problems inside the object are controlled since each region from a release point to another release point is performed as a critical region. Together, these mechanisms provide high-level constructs for process control, and in particular allow an object to change dynamically between active and reactive behavior by means of *cooperative* scheduling. The operational semantics of *ABS* has been worked out in [4]. Recently, this notion of cooperative scheduling and asynchronous method calls has been integrated in Java by means of concurrent object groups [5]. A Java code generator for *ABS* model is available. Programmers can model and verify distributed systems in the *ABS* language and transform them into Java programs.

The execution of a distributed system can be represented by its *communication history* or *trace*; i.e., the sequence of observable communication events between system components [6, 7]. At any point in time the communication history abstractly captures the system state [8, 9]. In fact communication traces are used in semantics for full abstraction results (e.g., [10, 11]). A system may be specified by the finite initial segments of its communication histories. Let the *local history* of an object reflect the communication between the object and its surroundings. A *history invariant* is a predicate over the communication history, which holds for all finite sequences in the prefix-closure of the set of possible histories, expressing safety properties [12].

In this paper, we develop a partial correctness proof system for the *ABS* language (ignoring object groups, interfaces, data types, and futures). A class is specified by a *class invariant* over the class attributes and the local communication history. Thus the class invariant directly relates the internal implementation of the class to its observable behavior. The proof system is derived from a standard sequential language by means of a syntactic

P	::=	$Dd^*\ F^*\ In^*\ Cl^*\ [s]^?$	program
In	::=	**interface** I [**extends** $I^+]^?\{S^*\}$	interface declaration
Cl	::=	**class** $C([T\ cp]^*)$ [**implements** I^+] $\{[T\ w]^*\ [s]^?\ M^*\}$	class definition
M	::=	$S\ B$	method definition
S	::=	$T\ m([T\ x]^*)$	method signature
B	::=	$\{$[**var** $[T\ x]^*;]^?\ [s;]^?$ **return** $e\}$	method blocks
T	::=	$I \mid D \mid Void$	types(interface or data type)
v	::=	$x \mid w$	local variables or attributes
e	::=	**null** $\mid$ **this** $\mid v \mid cp \mid t$	pure expressions
e_s	::=	**new** $C(e^*) \mid e.m(e^*)$	expressions with side-effects
s	::=	$v := e \mid v := e_s \mid e!m(e^*) \mid$ **await** $g \mid$ **suspend**	statements
		$\mid e_s \mid$ **skip** $\mid$ **abort** $\mid$ **if** e **then** s [**else** $s]^?$ **fi** $\mid s; s$	
g	::=	$v := e.m(e^*) \mid e.m(e^*) \mid e$	guards

Figure 1: The BNF syntax of the *ABS* language with the imperative sublanguage s, with I: interface name, C: class name, D: data type name, cp: formal class parameter, w: class attribute, m: method name, x: method parameter or local variable. We use [ ] as meta parenthesis and let $^?$ denote optional parts, * repeated parts, $^+$ parts repeated at least once. Thus e^* denotes a (possibly empty) expression list. Futures and object groups are omitted. The functional sublanguage for terms t can be found in A.

encoding, extending a transformational technique originally proposed by Olderog and Apt [13] to use non-deterministic assignments to the local history to reflect the activity of other processes at processor release points. This way, the reasoning inside a class is comparable to reasoning about a simple sequential while-language extended with non-deterministic assignment, and amounts to proving that the class invariant is maintained from one release point to another. By hiding the internal state, an external specification of an object may be obtained as an invariant over the local history. In order to derive a global specification of a system composed of several objects, one may compose the specifications of different objects. Modularity is achieved since history invariants can be established independently for each object and composed at need.

This paper extends and improves previous work by the authors [14, 15] as well as recent work by Ahrendt and Dylla [16, 17]. Technically, we here develop a system based on a four-event semantics for asynchronous method calls, which introduces *disjoint alphabets* for the local histories of different objects. Compared to previous work, this allows us to formulate a much simpler Hoare-style proof system for object-oriented languages based on concurrent objects with asynchronous method calls, and to reduce the complexity of reasoning about such concurrent programs by significantly simplifying the formulas in terms of the number of needed quantifiers.

Paper overview. Section 2 introduces and explains the *ABS* language syntax, Section 3 formalizes the observable behavior in the distributed systems, and Section 4 defines the proof system for *ABS* programs and considers object composition. A reader/writer example is presented in Section 2 through 4; and Section 5 considers an unbounded buffer example. Section 6 discusses related and future work, and Section 7 concludes the paper.

2. Syntax for the *ABS* Language

The syntax of the *ABS* language (slightly simplified) can be found in Fig. 1. An interface I may extend a number of superinterfaces, and defines a set of method signatures S^*. We say that I *provides* a method m if a signature for m can be found in S^* or among the signatures defined by a superinterface. A class C takes a list of formal class parameters $\overline{cp}$, defines class attributes $\overline{w}$, methods M^*, and may implement a number of interfaces. Remark that there is no class inheritance in the language, and the optional code block s of a class denotes object initialization, we will refer to this code block by the name *init*. There is read-only access to the formal class parameters $\overline{cp}$. For each method m provided by an implemented interface I, an implementation of m must be found in M^*. We then say that instances of C *support* I. Object references are typed by interfaces, and only the methods provided by some supported interface are available for external invocation on an object. The class may in addition implement auxiliary methods, used for internal purposes. Among the auxiliary methods we distinguish the special method `run` which is used to define the local activity of objects. If defined, this method is assumed to be invoked on newly created objects after initialization. In this paper, we focus on the internal verification of classes where interfaces play no role, and where programs are assumed to be type correct. Therefore types and interfaces are not considered in our reasoning system (but appear in the *ABS* examples). We say that a method m is *available* on an instance of class C if an implementation of m is found in C.

A method definition has the form $m(\overline{x})\{\textbf{var}\ \overline{y};\ s;\ \textbf{return}\ e\}$, ignoring type information, where $\overline{x}$ is the list of parameters, $\overline{y}$ an optional list of *method-local variables*, s is a sequence of statements and the value of the expression e is returned to the caller upon method termination. To simplify the presentation without loss of generality, we assume that all methods return a value; methods declared with return type *Void* are assumed to end with a **return** *void* statement, where *void* is the only value of type *Void*. Compound return types can be defined by means of data types.

Each concurrent object o encapsulates its own processor, and a method invocation on o leads to a new *process* on o. At most one process is executing in o at a time. *Processor release points* influence the internal control flow in an object. An **await** statement causes a release point, which suspends the executing process, releasing the processor and allowing an *enabled* and suspended process to be selected for execution. The continuation of a process suspended by **await g** is enabled when the guard g evaluates to true. The **suspend** statement is equivalent to **await** *true*. (An alternative semantic definition of **await** *true* as **skip** is presented in [18].) Object communication in *ABS* is *asynchronous*, as there is no explicit transfer of execution control between the caller and the callee. However, there are different statements for calling the method m in x with input values $\overline{e}$, allowing the caller to wait for the reply in various manners:

- **await** $x.m(\overline{e})$ / **await** $v := x.m(\overline{e})$: The continuation of the calling process is here suspended and becomes enabled when the call returns. Other processes of the caller may thereby execute while waiting for the reply from x. The return value is assigned to v when the continuation gets processor control.
- $x!m(\overline{e})$: Here the calling process continues without waiting for the reply
- $x.m(\overline{e})$ / $v := x.m(\overline{e})$: If x is different from `this`, the method is invoked without releasing the processor of the calling object; the calling process *blocks* the processor while waiting for the reply from x. For the caller, the statement thereby appears to

be synchronous which may potentially lead to deadlock, and should be used with care; however, they are often used on local objects generated and controlled by this object, as illustrated in the examples.

If x evaluates to this, the statement corresponds to standard synchronous invocation where m is loaded directly for execution and the calling process continues after termination of m.

The language additionally contains statements for assignment, object creation, **skip**, **abort**, conditionals, loops, and sequential composition. Concurrent object groups are not considered; however, our reasoning system allows reasoning about subsystems formed by (sub)sets of concurrent objects. The execution of a system is assumed to be initialized by a system generated root object main. Object main is allowed to generate objects, but can otherwise not participate in the execution. Especially, main provides no methods and invokes no methods on the generated objects.

2.1. Reader/Writer Example

To illustrate the *ABS* language, and in particular the different call constructs, we consider an implementation of the Reader/Writer problem. We use this implementation later in the paper to illustrate our reasoning techniques: we will define class invariants and illustrate the proof system by verification of these invariants. We assume given a shared database db, which provides two basic operations read and write. In order to synchronize reading and writing activity on the database, we consider the class RWController as implemented in Fig. 2, where caller is an implicit method parameter. All client activity on the database is assumed to go through a single RWController object. The RWController provides read and write operations to clients and in addition four methods used to synchronize reading and writing activity: openR (OpenRead), closeR (CloseRead), openW (OpenWrite) and closeW (CloseWrite). A reading session happens between invocations of openR and closeR and writing between invocations of openW and closeW. Several clients may read the database at the same time, but writing requires exclusive access. A client with write access may also perfom read operations during a writing session. Clients starting a session are responsible for closing the session.

Internally in the class, the attribute readers contains a set of clients currently with read access and writer contains the client with write access. Additionally, the attribute pr counts the number of pending calls to method db.read. (A corresponding counter for writing is not needed since db.write is called synchronously.) In order to ensure *fair* competition between readers and writers, invocations of openR and openW compete on equal terms for a guard writer = null. The set of readers is extended by execution of openR or openW, and the guards in both methods ensure that there is no writer. If there is no writer, a client gains write access by execution of openW. A client may thereby become the writer even if readers is non-empty. The guard in openR will then be *false*, which means that new invocations openR will be delayed, and the write operations initiated by the writer will be delayed until the current reading activities are completed. The client with write access will eventually be allowed to perform write operations since all active readers (other than itself) are assumed to end their sessions at some point. Thus even though readers may be non-empty while writer contains a client, the controller ensures that reading and writing *activity* cannot happen simultaneously on the database. The complete implementation of the example can be found in B. For simplicity we have omitted **return** *void* statements.

```
interface DB{
   Data read(Int key);
   Void write(Int key, Data element) }

interface RW{
   Void openR();
   Void closeR();
   Void openW();
   Void closeW();
   Data read(Int key);
   Void write(Int key, Data element) }

class RWController() implements RW{

   DB db; DataSet readers; Obj writer; Int pr;

   {db := new DataBase(); readers := Empty; writer := null; pr := 0 }

   Void openR(){await writer = null; readers := Add(caller, readers) }

   Void closeR(){readers := delete(caller, readers) }

   Void openW(){await writer = null; writer := caller; readers := Add(caller, readers) }

   Void closeW(){await writer = caller; writer := null; readers := delete(caller, readers) }

   Data read(Int key){ Data result;
      await isElement(caller, readers);
      pr := pr +1; await result := db.read(key); pr := pr −1;
      return result }

   Void write(Int key, Data value){
      await caller = writer and pr = 0 and
        (readers = Empty or (isElement(writer, readers) and size(readers) = 1));
      db.write(key, value) }
}
```

Figure 2: Implementation of the fair reader/writer controller. See B for full implementation including data type definitions and implementation of the DataBase class.

3. Observable Behavior

The execution of a distributed system can be represented by its *communication history* or *trace*; i.e., the sequence of observable communication events between system components [6, 7]. At any point in time the communication history abstractly captures the system state [8, 9]. Therefore a system may be specified by the finite initial segments of its communication histories. A *history invariant* is a predicate over the communication history, which holds for all finite sequences in the prefix-closure of the set of possible histories, expressing safety properties [12]. To deal with concurrent objects interacting by method calls we let the history reflect invocation events and completion events of the called methods. 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 $\_ \vdash \_ : \mathit{Seq}[T] \times T \to \mathit{Seq}[T]$ (where "_" indicates an argument position). For communication histories, this choice of constructors gives rise to generate inductive function definitions where one characterizes the new state in terms of the old state and the last event, in the style of [8]. Let $a, b : \mathit{Seq}[T]$, $x, y, z : T$, and $s : \mathit{Set}[T]$. Projection $\_/\_ : \mathit{Seq}[T] \times \mathit{Set}[T] \to \mathit{Seq}[T]$ 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 $\_\mathbf{ew}\_ : \mathit{Seq}[T] \times T \to \mathit{Bool}$ and $\_\mathbf{bw}\_ : \mathit{Seq}[T] \times T \to \mathit{Bool}$ are defined inductively by $\varepsilon\ \mathbf{ew}\ x \triangleq \mathit{false}$, $(a \vdash y)\ \mathbf{ew}\ x \triangleq x = y$, $\varepsilon\ \mathbf{bw}\ x \triangleq \mathit{false}$, $(\varepsilon \vdash y)\ \mathbf{bw}\ x \triangleq x = y$, and $(a \vdash z \vdash y)\ \mathbf{bw}\ x \triangleq (a \vdash z)\ \mathbf{bw}\ x$. The left-rest operation is defined by the partial function $\mathit{pop} : \mathit{Seq}[T] \to \mathit{Seq}[T]$, such that $\mathit{pop}(a \vdash x) \triangleq a$. Furthermore, let $a \leqslant b$ denote that a is a prefix of b, $a \vdash\!\!\dashv b$ denote the concatenation of a and b, and $\# a$ denote the length of a. Let *Arrow* be the enumeration type ranging over $\{\to, \twoheadrightarrow, \leftarrow, \twoheadleftarrow\}$, and let *Data* be the supertype of all kinds of data. Communication events are defined next.

Definition 1. ***(Communication events)*** *Let $o, o' : \mathit{Obj}, m : \mathit{Mtd}$, $c : \mathit{Cls}$, $\overline{e} : \mathit{List}[\mathit{Data}]$, and $v : \mathit{Data}$. We define the following sets of communication events:*

- *the set IEv of invocation events $\langle o, \to, o', m, \overline{e}\rangle$,*
- *the set IREv of invocation reaction events $\langle o, \twoheadrightarrow, o', m, \overline{e}\rangle$,*
- *the set CEv of completion events $\langle o, \leftarrow, o', m, v\rangle$,*
- *the set CREv of completion reaction events $\langle o, \twoheadleftarrow, o', m, v\rangle$,*
- *the set NEv of object creation events $\langle o, \to, o', C, \overline{e}\rangle$,*
- *the set NREv of object creation reaction events $\langle o, \twoheadrightarrow, o', C, \overline{e}\rangle$, and*
- *the set Ev of all events; i.e, $\mathit{Ev} = \mathit{IEv} \cup \mathit{IREv} \cup \mathit{CEv} \cup \mathit{CREv} \cup \mathit{NEv} \cup \mathit{NREv}$.*

Graphical representation of the events are given by $o \to o'.m(\overline{e})$, $o \twoheadrightarrow o'.m(\overline{e})$, $o \leftarrow o'.m(v)$, $o \twoheadleftarrow o'.m(v)$, $o \to o'.\mathbf{new}\ C(\overline{e})$ and $o \twoheadrightarrow o'.\mathbf{new}\ C(\overline{e})$. Events may be decomposed by the functions $\_.\mathit{caller}, \_.\mathit{callee} : \mathit{Ev} \to \mathit{Obj}$, $\_.\mathit{mtd} : \mathit{Ev} \to \mathit{Mtd}$, $\_.\mathit{cls} : \mathit{Ev} \to \mathit{Cls}$, and $\_.\mathit{data} : \mathit{Ev} \to \mathit{Data}$. For each of the events listed above, the function $\_.\mathit{caller}$ returns o and the function $\_.\mathit{callee}$ returns o'. The decomposition functions are lifted to sequences in the standard way. We assume a total function $\mathit{parent} : \mathit{Obj} \to \mathit{Obj}$ where $\mathit{parent}(o)$ denotes the creator of o, such that $\mathit{parent}(\mathsf{main}) = \mathsf{main}$ and $\mathit{parent}(o) = \mathsf{null} \Leftrightarrow o = \mathsf{null}$. Equality is the only executable operation on object identities. Given the *parent* function, we may define an *ancestor* function $\mathit{anc} : \mathit{Obj} \to \mathit{Set}[\mathit{Obj}]$ by $\mathit{anc}(\mathsf{main}) \triangleq \{\mathsf{main}\}$ and $\mathit{anc}(o) = \mathit{parent}(o) \cup \mathit{anc}(\mathit{parent}(o))$ (where $o \neq \mathsf{main}$). We say that parent chains are *cycle free* if $o \notin \mathit{anc}(o)$ for all generated objects o, i.e., for $o \neq \mathsf{main}$.

A method call is in our model reflected by four communication events, as illustrated in Fig. 3 where object o calls a method m on object o'. An invocation message is sent from o to o' when the method is called, which is reflected by the invocation event $o \to o'.m(\overline{e})$ where $\overline{e}$ is the list of actual parameters. The event $o \twoheadrightarrow o'.m(\overline{e})$ reflects that o' starts execution of the method, and the event $o \leftarrow o'.m(v)$ reflects method termination. Reading the reply in object o is reflected by the event $o \twoheadleftarrow o'.m(v)$. The creation of an object o' by an object o is reflected by the events $o \to o'.\mathbf{new}\ C(\overline{e})$ and $o \twoheadrightarrow o'.\mathbf{new}\ C(\overline{e})$, where o' is an instance of class C and $\overline{e}$ are the actual values for the class parameters. The event $o \to o'.\mathbf{new}\ C(\overline{e})$

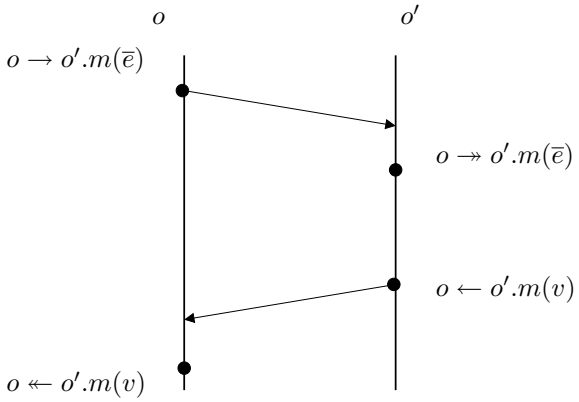

Figure 3: A method call cycle, where object o calls a method m on object o'. The arrows indicate message passing, and the bullets indicates events. The events on the left hand side are visible to o, whereas the events on the right hand side are visible to o'. Remark that there is an arbitrary delay between message receiving and reaction.

reflects that o initiates the creation, whereas $o \twoheadrightarrow o'.\mathbf{new}\ C(\overline{e})$ reflects that o' is created. Next we define *communication histories* as a sequence of events. When restricted to a set of objects, the communication history contains only events that are generated by the considered objects.

Definition 2. ***(Communication histories)*** *The* communication history *of a (sub)system up to given time is a finite sequence of type Seq[Ev].*

The communication history for a set O of objects is a finite sequence of type $Seq[Ev_O]$ where

$$IEv_O \triangleq \{e : IEv \mid e.caller \in O\}$$
$$IREv_O \triangleq \{e : IREv \mid e.callee \in O\}$$
$$CEv_O \triangleq \{e : CEv \mid e.callee \in O\}$$
$$CREv_O \triangleq \{e : CREv \mid e.caller \in O\}$$
$$NEv_O \triangleq \{e : NEv \mid e.caller \in O\}$$
$$NREv_O \triangleq \{e : NREv \mid e.callee \in O\}$$
$$Ev_O \triangleq IEv_O \cup IREv_O \cup CEv_O \cup CREv_O \cup NEv_O \cup NREv_O$$

Given a communication history h, we let $h/\{o\}$ abbreviate $h/Ev_{\{o\}}$, i.e., the projection restricts h to the events that are *generated* by o. The *local* communication history of an object contains only events that are generated by that object.

Definition 3. ***(Local communication histories)*** *The* local communication history *of an object o is a finite sequence of type $Seq[Ev_{\{o\}}]$.*

In this manner, the local communication history reflects the local activity of each object. For the method call $o'.m(\overline{e})$ made by object o as explained above, the events $o \rightarrow o'.m(\overline{e})$ and $o \twoheadleftarrow o'.m(v)$ are local to o. Correspondingly, the events $o \twoheadrightarrow o'.m(\overline{e})$ and $o \leftarrow o'.m(v)$ are local to o'. For object creation, the event $o \rightarrow o'.\mathbf{new}\ C(\overline{e})$ is local to o whereas $o \twoheadrightarrow o'.\mathbf{new}\ C(\overline{e})$ is local to o'. Let h_o denote that h is a local history of object o, i.e.,

$h_o : Seq[Ev_{\{o\}}]$. It follows by the definitions above that $Ev_{\{o\}} \cap Ev_{\{o'\}} = \emptyset$ for $o \neq o'$, i.e., the two local histories h_o and $h_{o'}$ have *no common events*.

Functions may extract information from the history. In particular, we define $oid : Seq[Ev] \to Set[Obj]$ extracting all object identities occurring in a history, as follows:

$$
\begin{array}{ll}
oid(\varepsilon) \triangleq \{\mathsf{null}\} & oid(h \vdash \gamma) \triangleq oid(h) \cup oid(\gamma) \\
oid(o \to o'.m(\overline{e})) \triangleq \{o, o'\} \cup oid(\overline{e}) & oid(o \twoheadrightarrow o'.m(\overline{e})) \triangleq \{o, o'\} \cup oid(\overline{e}) \\
oid(o \leftarrow o'.m(v)) \triangleq \{o, o'\} \cup oid(v) & oid(o \twoheadleftarrow o'.m(v)) \triangleq \{o, o'\} \cup oid(v) \\
oid(o \to o'.\mathbf{new}\ C(\overline{e})) \triangleq \{o, o'\} \cup oid(\overline{e}) & oid(o \twoheadrightarrow o'.\mathbf{new}\ C(\overline{e})) \triangleq \{o, o'\} \cup oid(\overline{e})
\end{array}
$$

where $\gamma : Ev$, and $oid(\overline{e})$ returns the set of object identifiers occurring in the expression list $\overline{e}$. The function $new_{ob} : Seq[Ev] \to Set[Obj \times Cls \times List[Data]]$ returns the set of created objects (each given by its object identity, associated class and class parameters) in a history:

$$
\begin{array}{l}
new_{ob}(\varepsilon) \triangleq \emptyset \\
new_{ob}(h \vdash o \to o'.\mathbf{new}\ C(\overline{e})) \triangleq new_{ob}(h) \cup \{o' : C(\overline{e})\} \\
new_{ob}(h \vdash \boldsymbol{others}) \triangleq new_{ob}(h)
\end{array}
$$

(where ***others*** matches all other events). The function $new_{id} : Set[Obj \times Cls \times List[Data]] \to Set[Obj]$ extracts object identities from the output of function new_{ob}. For a local history h_o, all objects *created by* o are returned by $new_{ob}(h_o)$.

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. Assuming type correctness, we define the following wellformedness predicate over communication histories, ensuring freshness of identities of created objects, non-nullness of communicating objects, and ordering of communication events according to Fig. 3:

Definition 4. ***(Wellformed histories)*** *Let $O : Set[Obj]$ and $h : Seq[Ev_O]$, the wellformedness predicate $wf : Seq[Ev_O] \times Set[Obj] \to Bool$ for the (sub)set of objects O is defined by:*

$$
\begin{array}{l}
wf(\varepsilon, O) \triangleq true \\
wf(h \vdash o \to o'.m(\overline{e}), O) \triangleq wf(h, O) \wedge o \neq \mathsf{null} \wedge o' \neq \mathsf{null} \\
wf(h \vdash o \twoheadrightarrow o'.m(\overline{e}), O) \triangleq wf(h, O) \wedge o \neq \mathsf{null} \wedge o' \neq \mathsf{null} \\
\qquad\qquad \wedge\ (o \in O \Rightarrow dom(h, o \to o'.m(\overline{e}), o \twoheadrightarrow o'.m(\overline{e}))) \\
wf(h \vdash o \leftarrow o'.m(v), O) \triangleq wf(h, O) \wedge dom(h, o \twoheadrightarrow o'.m(\_), o \leftarrow o'.m(\_)) \\
wf(h \vdash o \twoheadleftarrow o'.m(v), O) \triangleq wf(h, O) \wedge dom(h, o \to o'.m(\_), o \twoheadleftarrow o'.m(\_))) \\
\qquad\qquad \wedge\ (o' \in O \Rightarrow dom(h, o \leftarrow o'.m(v), o \twoheadleftarrow o'.m(v))) \\
wf(h \vdash o \to o'.\mathbf{new}\ C(\overline{e}), O) \triangleq wf(h, O) \wedge o \neq \mathsf{null} \wedge parent(o') = o \wedge o' \notin oid(h) \cup oid(\overline{e}) \\
wf(h \vdash o \twoheadrightarrow o'.\mathbf{new}\ C(\overline{e}), O) \triangleq wf(h, O) \wedge o \neq \mathsf{null} \wedge parent(o') = o \wedge o' \notin oid(\overline{e}) \\
\qquad\qquad \wedge\ (o \in O) \Rightarrow h/\{o\}\ \mathbf{ew}\ o \to o'.\mathbf{new}\ C(\overline{e})
\end{array}
$$

where the domination function $dom : Seq[Ev] \times Ev \times Ev \to Bool$ is defined by:

$$dom(h, \gamma_1, \gamma_2) \triangleq \#(h/\{\gamma_1\}) > \#(h/\{\gamma_2\})$$

The domination checks of wellformedness ensure that method call cycles correspond to Fig. 3: For invocation reaction events, if the caller is in O, the method must have been called more times than the number of started method executions. In other words, there must be more invocations events than invocation reaction events. When sending completion events, there

must be more invocation reaction events than completion events. For completion reaction events there must be more invocation events than completion reaction events, and if the callee is in O, there must be more completion events than completion reaction events.

For a set O of objects such that $o, o' \in O$, a history h such that $wf(h, O)$, and a method m available on o', we have the following relationship:

$$\#(h/\{o \rightarrow o'.m(\_)\}) \geq \#(h/\{o \twoheadrightarrow o'.m(\_)\}) \geq \#(h/\{o \leftarrow o'.m(\_)\}) \geq \#(h/\{o \twoheadleftarrow o'.m(\_)\})$$

Remark that for object creation, the parent object and the created object synchronize, i.e., if o is the parent of o' and h ends with the creation event $o \twoheadrightarrow o'.\mathbf{new}\ C(\overline{e})$, then the last event generated by o is $o \rightarrow o'.\mathbf{new}\ C(\overline{e})$.

Consider next wellformedness $wf(h_o, \{o\})$ of the local history h_o of an object o. For an event $o' \twoheadrightarrow o.m$, we observe that the domination constraint for $wf(h_o \vdash o' \twoheadrightarrow o.m, \{o\})$ is trivially satisfied if $o \neq o'$. Consequently, in this case the domination constraint only applies to local calls (i.e., where $o = o'$). For a completion event $o' \leftarrow o.m$, the domination constraint must be satisfied unconditionally since both $o' \leftarrow o.m$ and $o' \twoheadrightarrow o.m$ are local to o. For completion reaction events $o \twoheadleftarrow o'.m$ on h_o, the first domination constraint must be satisfied unconditionally, whereas the second constraint only applies to local calls. For a global system, i.e., where O contains all objects in the system, all the domination constraints must be satisfied since both the caller and the callee of each event must be in O.

Note that a history with the event $o \rightarrow \mathsf{null}.m(\overline{e})$ is considered non-wellformed, reflecting that a remote call statement aborts if the callee is null and therefore needs not be considered for partial correctness reasoning. Alternatively such histories could be considered wellformed (as long as there are no events caused by null). An advantage would be that one could express and reason about absence of invocations to null objects. However, such properties are trivial for the examples we consider and could be guaranteed by simple static checks.

3.1. Invariant Reasoning

In interactive and non-terminating systems, it is difficult to specify and reason compositionally about object behavior in terms of pre- and postconditions of the defined methods. Also, the highly non-deterministic behavior of *ABS* objects due to internal suspension points complicates reasoning in terms of pre- and postconditions. Instead, pre- and postconditions to method definitions are in our setting 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: 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. By hiding the internal state, class invariants form a suitable basis for compositional reasoning about object systems.

A *user-provided invariant* $I_C(\overline{w}, h_{\mathsf{this}})$ for a class C is a predicate over the attributes $\overline{w}$ and the local history h_{this}, as well as the formal class parameters $\overline{cp}$ and this, which are constant (read-only) variables.

3.2. Specification of Reader/Writer Example

For the RWController class in Fig. 2, we may define a class invariant expressing a relation between the internal state of class instances and observable communication. The internal state is given by the values of the class attributes. Functions are defined to extract relevant information from the local communication history. We define $Readers : Seq[Ev] \to Set[Obj]$:

$$
\begin{aligned}
&Readers(\varepsilon) \triangleq \emptyset \\
&Readers(h \vdash o \leftarrow \mathsf{this.openR}) \triangleq Readers(h) \cup \{o\} \\
&Readers(h \vdash o \leftarrow \mathsf{this.openW}) \triangleq Readers(h) \cup \{o\} \\
&Readers(h \vdash o \leftarrow \mathsf{this.closeR}) \triangleq Readers(h) \setminus \{o\} \\
&Readers(h \vdash o \leftarrow \mathsf{this.closeW}) \triangleq Readers(h) \setminus \{o\} \\
&Readers(h \vdash \boldsymbol{others}) \triangleq Readers(h)
\end{aligned}
$$

where ***others*** matches all events not matching any of the above cases. The caller is added to the set of readers upon termination of openR or openW, and the caller is removed from the set upon termination of closeR or closeW. We furthermore assume a function *Writers*, defined over completions of openW and closeW in a corresponding manner, see C. Next we define $Reading : Seq[Ev] \to Nat$ by:

$$Reading(h) \triangleq \#(h/\{\mathsf{this} \to \mathsf{db.read}\}) - \#(h/\{\mathsf{this} \twoheadleftarrow \mathsf{db.read}\})$$

Thus the function $Reading(h)$ computes the difference between the number of initiated calls to db.read and reaction event from this method. The function *Writing(h)* follows the same pattern over calls to db.write, the definition can be found in D.

The class invariant I is defined over the class attributes and the local history by:

$$I \triangleq I_1 \wedge I_2 \wedge I_3 \wedge I_4$$

where

$$
\begin{aligned}
I_1 &\triangleq Readers(\mathcal{H}) = \mathsf{readers} \\
I_2 &\triangleq Writers(\mathcal{H}) = \{\mathsf{writer}\} \\
I_3 &\triangleq Reading(\mathcal{H}) = \mathsf{pr} \\
I_4 &\triangleq OK(\mathcal{H})
\end{aligned}
$$

where $\{\mathsf{writer}\} = \emptyset$ if $\mathsf{writer} = \mathsf{null}$. The invariants I_1, I_2, and I_3, illustrate how the values of class attributes may be expressed in terms of observable communication, e.g. $Readers(\mathcal{H})$ has the same value as readers. The predicate $OK : Seq[Ev] \to Bool$ is defined inductively over the history by:

$$
\begin{aligned}
&OK(\varepsilon) \triangleq true \\
&OK(h \vdash \_ \leftarrow \mathsf{this.openR}) \triangleq OK(h) \wedge \#Writers(h) = 0 && (1)\\
&OK(h \vdash \_ \leftarrow \mathsf{this.openW}) \triangleq OK(h) \wedge \#Writers(h) = 0 && (2)\\
&OK(h \vdash \mathsf{this} \to \mathsf{db.write}) \triangleq OK(h) \wedge Reading(h) = 0 \wedge \#Writers(h) = 1 && (3)\\
&OK(h \vdash \mathsf{this} \twoheadleftarrow \mathsf{db.write}) \triangleq OK(h) \wedge h \ \mathbf{ew}\ \mathsf{this} \to \mathsf{db.write} && (4)\\
&OK(h \vdash \mathsf{this} \to \mathsf{db.read}) \triangleq OK(h) \wedge Writing(h) = 0 && (5)\\
&OK(h \vdash \boldsymbol{others}) \triangleq OK(h)
\end{aligned}
$$

Here, conditions (1) and (2) reflect the *fairness condition*: invocations of openR and openW compete on equal terms for the guard $\mathsf{writer} = \mathsf{null}$, which equals $Writers(\mathcal{H}) = \emptyset$ by I_2. If

writer is different from null, conditions (1) and (2) additionally ensure that no clients can be included in the readers set or be assigned to writer. Conditions (3) and (4) presentes the synchronization of db.write and captures the guard in write: when invoking db.write, there cannot be any pending calls to db.read. Correspondingly, Condition (5) expresses that when invoking db.read, there is no incomplete writing operation. The invariant I implies that no reading and writing *activity* happens simultaneously:

$$\mathit{Reading}(\mathcal{H}) = 0 \vee \mathit{Writing}(\mathcal{H}) = 0$$

4. Analysis of *ABS* Programs

The semantics of *ABS* statements is expressed as an encoding into a sequential sublanguage without shared variables, but with a non-deterministic assignment operator [14]. Non-deterministic history extensions capture arbitrary activity of other processes in the object during suspension. The semantics describes a single object of a given class placed in an arbitrary environment. The encoding is defined in Section 4.1, and weakest liberal preconditions are derived in Section 4.2. In Section 4.3 we consider Hoare rules derived from the weakest liberal preconditions. The semantics of a dynamically created system with several concurrent objects is given by the composition rule in Section 4.5.

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. This communication is captured by four *events* on the communication history, as illustrated in Fig. 3. For a local call (i.e., $o = o'$), all four events are visible on the local history of o. Similarly, object creation is captured by a message from the parent object to the generated object.

4.1. Semantic Definition by a Syntactic Encoding

We consider a simple *sequential* language where statements have the syntax

$$s ::= \mathbf{skip} \mid \mathbf{abort} \mid \overline{v} := \overline{e} \mid s; s \mid \mathbf{if}\ b\ \mathbf{then}\ s\ \mathbf{else}\ s\ \mathbf{fi} \mid v := m(\overline{e})$$

This language has a well-established semantics and proof system. In particular, soundness and relative completeness are discussed in [19, 20, 21]. Let the language *SEQ* additionally include a statement for non-deterministic assignment, assigning to $\overline{y}$ some (type correct) values:

$$\overline{y} := \mathbf{some}$$

In addition we include *assert* statements in order to state required conditions. The statement

$$\mathbf{assert}\ b$$

means that one is obliged to verify the condition b for the current state, and has otherwise no effect. Similarly, *assume* statements are used to encode known facts. Semantically the statement

$$\mathbf{assume}\ b$$

is understood as **if** b **then skip else abort fi**. To summarize, we have the following syntax for *SEQ* statements:

$$\begin{aligned} s ::= \quad & \mathbf{skip} \mid \mathbf{abort} \mid \overline{v} := \overline{e} \mid s; s \mid \mathbf{if}\ b\ \mathbf{then}\ s\ \mathbf{else}\ s\ \mathbf{fi} \mid v := m(\overline{e}) \\ & \mid \overline{y} := \mathbf{some} \mid \mathbf{assert}\ b \mid \mathbf{assume}\ b \end{aligned}$$

$$
\begin{array}{l}
\langle\!\langle\, m(\overline{x})\ \{\mathbf{var}\ \overline{y}; s\}\,\rangle\!\rangle \triangleq m'(\overline{x}, \mathsf{caller})\ \{\mathbf{var}\ \overline{y}; \mathcal{H} := \mathcal{H} \vdash \mathsf{caller} \twoheadrightarrow \mathsf{this}.m(\overline{x}); \langle\!\langle\, s\,\rangle\!\rangle; \\
\quad \mathcal{H} := \mathcal{H} \vdash \mathsf{caller} \leftarrow \mathsf{this}.m(\mathsf{return}); \mathbf{assume}\ \ \mathit{wf}(\mathcal{H})\} \\
\langle\!\langle\, \mathbf{suspend}\,\rangle\!\rangle \triangleq \mathbf{assert}\ \ I_C(\overline{w}, \mathcal{H}) \wedge \mathit{wf}(\mathcal{H});\ \overline{w}, h' := \mathbf{some}; \mathcal{H} := \mathcal{H} \vdash h'; \\
\quad \mathbf{assume}\ \ I_C(\overline{w}, \mathcal{H}) \wedge \mathit{wf}(\mathcal{H}) \\
\langle\!\langle\, \mathbf{await}\ \ b\,\rangle\!\rangle \triangleq \langle\!\langle\, \mathbf{suspend}\,\rangle\!\rangle; \mathbf{assume}\ \ b \\
\langle\!\langle\, \mathbf{await}\ \ o.m(\overline{e})\,\rangle\!\rangle \triangleq \mathbf{assume}\ o \neq \mathsf{null}; \mathcal{H} := \mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e}); o' := o; \langle\!\langle\, \mathbf{suspend}\,\rangle\!\rangle; \\
\quad v' := \mathbf{some}\ ; \mathcal{H} := \mathcal{H} \vdash \mathsf{this} \twoheadleftarrow o'.m(v'); \mathbf{assume}\ \ \mathit{wf}(\mathcal{H}) \\
\langle\!\langle\, \mathbf{await}\ \ v := o.m(\overline{e})\,\rangle\!\rangle \triangleq \langle\!\langle\, \mathbf{await}\ \ o.m(\overline{e})\,\rangle\!\rangle; v := v' \\
\langle\!\langle\, o.m(\overline{e})\,\rangle\!\rangle \triangleq \mathbf{assume}\ o \neq \mathsf{null}; \mathcal{H} := \mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e}); \\
\quad \mathbf{if}\ \ o = \mathsf{this}\ \ \mathbf{then}\ \ v' := m'(\overline{e}, \mathsf{this}); \mathcal{H} := \mathcal{H} \vdash \mathsf{this} \twoheadleftarrow \mathsf{this}.m(v') \\
\quad \mathbf{else}\ \ v' := \mathbf{some};\ \mathcal{H} := \mathcal{H} \vdash \mathsf{this} \twoheadleftarrow o.m(v')\ \ \mathbf{fi} \\
\langle\!\langle\, v := o.m(\overline{e})\,\rangle\!\rangle \triangleq \langle\!\langle\, o.m(\overline{e})\,\rangle\!\rangle; v := v' \\
\langle\!\langle\, o!m(\overline{e})\,\rangle\!\rangle \triangleq \mathbf{assume}\ o \neq \mathsf{null}; \mathcal{H} := \mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e}) \\
\langle\!\langle\, x := \mathbf{new}\ \ C(\overline{e})\,\rangle\!\rangle \triangleq x' := \mathbf{some};\ \mathcal{H} := \mathcal{H} \vdash \mathsf{this} \rightarrow x'.\mathbf{new}\ C(\overline{e}); x := x'; \mathbf{assume}\ \ \mathit{wf}(\mathcal{H}) \\
\langle\!\langle\, \mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{fi}\,\rangle\!\rangle \triangleq \mathbf{if}\ \ b\ \ \mathbf{then}\ \ \langle\!\langle\, s_1\,\rangle\!\rangle\ \ \mathbf{else}\ \ \langle\!\langle\, s_2\,\rangle\!\rangle\ \ \mathbf{fi} \\
\langle\!\langle\, s_1; s_2\,\rangle\!\rangle \triangleq \langle\!\langle\, s_1\,\rangle\!\rangle; \langle\!\langle\, s_2\,\rangle\!\rangle \\
\langle\!\langle\, \mathbf{skip}\,\rangle\!\rangle \triangleq \mathbf{skip} \\
\langle\!\langle\, \mathbf{abort}\,\rangle\!\rangle \triangleq \mathbf{abort} \\
\langle\!\langle\, v := e\,\rangle\!\rangle \triangleq v := e \\
\langle\!\langle\, \mathbf{return}\ e\,\rangle\!\rangle \triangleq \mathsf{return} := e
\end{array}
$$

Figure 4: *ABS* syntactic equations. Here, C is the class enclosing the encoded statements, and I_C is the class invariant. The assumptions reflect that the history in an execution is wellformed, that suspension maintains the local invariant, and that a waiting condition holds when control returns. Note that for each method m, m' is the corresponding method with encoded body and with the caller as an extra parameter.

Method definitions are of the form $m(\overline{x}, \mathsf{caller})\ body$, where $body$ is of the form $\{\mathbf{var}\ \overline{y}; s\}$. Thus a body contains declaration of method-local variables followed by a sequence of statements. For simplicity we use the same $body$ notation as in *ABS*. However, in *ABS* the body must end with a final **return** statement, whereas *SEQ* uses a return variable.

At the class level, the list of class attributes is augmented with $\mathsf{this} : Obj$ and $\mathcal{H} : Seq[Ev_{\{\mathsf{this}\}}]$, representing self reference and the local communication history, respectively. The semantics of a method is defined from the local perspective of processes. An *ABS* process with release points and asynchronous method calls is interpreted as a non-deterministic *SEQ* process *without* shared variables and release points, by the mapping $\langle\!\langle\ \rangle\!\rangle$, as defined in Fig. 4. Expressions and types are mapped by the identity function. A *SEQ* process executes on a state $\overline{w} \cup \mathcal{H}$ extended with local variables and auxiliary variables introduced by the encoding. As in *ABS*, there is read-only access to the formal class parameters. We let $\mathit{wf}(\mathcal{H})$ abbreviate $\mathit{wf}(\mathcal{H}, \{\mathsf{this}\})$.

When an instance of $m(\overline{x})$ starts execution, the history $\mathcal{H}$ is extended by an invocation reaction event: $\mathcal{H} := \mathcal{H} \vdash \mathsf{caller} \twoheadrightarrow \mathsf{this}.m(\overline{x})$. Process termination is reflected by appending a completion event: $\mathcal{H} := \mathcal{H} \vdash \mathsf{caller} \leftarrow \mathsf{this}.m(\mathsf{return})$, where return is the return value of m. When invoking some method $o.m(\overline{e})$, the history is extended with an invocation event: $\mathcal{H} := \mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e})$, and fetching the reply is encoded by $\mathcal{H} := \mathcal{H} \vdash \mathsf{this} \twoheadleftarrow o.m(v)$.

The local effect of executing a release statement is that $\overline{w}$ and $\mathcal{H}$ may be updated due to the execution of other processes. In the encoding, these updates are captured by non-deterministic assignments to $\overline{w}$ and $\mathcal{H}$, as reflected by the encoding of the **suspend** statement. Here, the **assume** and **assert** statements reflect that the class invariant for-

(1) $\mathcal{H} = \langle parent(\mathsf{this}) \twoheadrightarrow \mathsf{this}.\mathbf{new}\ C(\overline{cp})\rangle \Rightarrow wlp(init_C, I_C(\overline{w}, \mathcal{H}))$
(2) $wf(\mathcal{H}) \wedge \mathcal{H}\,\mathbf{bw}\,\langle parent(\mathsf{this}) \twoheadrightarrow \mathsf{this}.\mathbf{new}\ C(\overline{cp})\rangle \wedge I_C(\overline{w}, \mathcal{H}) \Rightarrow wlp(m(\overline{x})\ body, I_C(\overline{w}, \mathcal{H}))$
(3) $wf(\mathcal{H}) \wedge \mathcal{H}\,\mathbf{bw}\,\langle parent(\mathsf{this}) \twoheadrightarrow \mathsf{this}.\mathbf{new}\ C(\overline{cp})\rangle \wedge S(\overline{w}, \mathcal{H}) \Rightarrow wlp(m(\overline{x})\ body, R(\overline{w}, \mathcal{H}))$

Figure 5: Verification conditions for *ABS* methods. Condition (1) ensures that the class invariant is established by the class initialization block *init*. Condition (2) ensures that each method $m(\overline{x})$ *body* maintains the class invariant. Condition (3) is used to verify additional properties for a method $m(\overline{x})$ *body*, verifying the pre/post specification S/R for the implementation. Notice that $\mathsf{this} \neq \mathsf{null}$ follows from each premise.

malizes a contract between the different processes in the object. The class invariant must be established before releasing processor control, and may be assumed when the process continues. For partial correctness reasoning, we may assume that processes are not suspended infinitely long. Consequently, non-deterministic assignment captures the possible interleaving of processes in an abstract manner.

For method call statements, fresh auxiliary variables are used as temporary placeholders for the return value. In the encoding, a method call statement $o.m(\overline{e})$ is treated as $v' := o.m(\overline{e})$, where v' is a fresh auxiliary variable. The call statement $v := o.m(\overline{e})$, where the return value is assigned to v, is encoded as $v' := o.m(\overline{e}); v := v'$. The encoding of $v := o.m(\overline{e})$ can thereby be defined in terms of the encoding of $o.m(\overline{e})$. Correspondingly, a guarded call statement **await** $v := o.m(\overline{e})$ can be defined in terms of **await** $o.m(\overline{e})$ by using fresh auxiliary variables.

In the encoding of *object creation*, non-deterministic assignment is used to construct object identifiers, and the history is then extended with the creation event. The final wellformedness assumption ensures the parent relationships and uniqueness of the generated identifiers. Remark that the history extension ensures that the values of the class parameters are visible on the communication history.

Lemma 1. *The local history of an object is wellformed for any legal execution.*

Proof. Preservation of wellformedness is trivial for statements that do not extend the local history $\mathcal{H}$, and we need to ensure wellformedness after extensions of $\mathcal{H}$. Wellformedness is maintained by processor release points. Extending the history with invocation or invocation reaction events maintains wellformedness of the local history. It follows straightforwardly that $wf(\mathcal{H})$ is preserved by the encoding of statement $o.m(\overline{e})$. For the remaining extensions, i.e., completion and completion reaction events, wellformedness is guaranteed by the **assume** statements following the different extensions.

4.2. Weakest Liberal Preconditions

We may define *weakest liberal preconditions* for the different *ABS* statements, reflecting that we consider partial correctness. The definitions are based on the encoding from *ABS* to *SEQ*. The verification conditions of a class C with invariant $I_C(\overline{w}, \mathcal{H})$ are summarized in Fig. 5. Condition (1) applies to the initialization block *init* of C, ensuring that the invariant is established upon termination. We may reason about possible processor release points in *init* by the weakest liberal preconditions given below. Condition (2) applies to each method $m(\overline{x})$ *body* defined in C; ensuring that each method maintains the class invariant. Condition (3) is used in order to prove additional knowledge for local synchronous calls, as described below, where S is the precondition and R is the postcondition (given by a user specification).

$$
\begin{array}{l}
wlp(m(\overline{x})\ \{\mathbf{var}\ \overline{y}; s\}, Q) \triangleq wlp(m'(\overline{x}, \mathsf{caller})\ \{\mathbf{var}\ \overline{y}; \mathcal{H} := \mathcal{H} \vdash \mathsf{caller} \twoheadrightarrow \mathsf{this}.m(\overline{x}); s; \\
\quad \mathcal{H} := \mathcal{H} \vdash \mathsf{caller} \leftarrow \mathsf{this}.m(\mathsf{return})\}, wf(\mathcal{H}) \Rightarrow Q) \qquad \text{for } \overline{y} \notin FV[Q] \\
wlp(\mathbf{suspend}, Q) \triangleq I_C(\overline{w}, \mathcal{H}) \wedge wf(\mathcal{H}) \wedge \forall \overline{w}, h' \,.\, (I_C(\overline{w}, \mathcal{H} \vdash h') \wedge wf(\mathcal{H} \vdash h') \Rightarrow Q^{\mathcal{H}}_{\mathcal{H} \vdash h'}) \\
wlp(\mathbf{await}\ \ b, Q) \triangleq I_C(\overline{w}, \mathcal{H}) \wedge wf(\mathcal{H}) \wedge \forall \overline{w}, h' \,.\, (I_C(\overline{w}, \mathcal{H} \vdash h') \wedge wf(\mathcal{H} \vdash h') \wedge b \Rightarrow Q^{\mathcal{H}}_{\mathcal{H} \vdash h'}) \\
wlp(\mathbf{await}\ \ o.m(\overline{e}), Q) \triangleq o \neq \mathsf{null} \Rightarrow I_C(\overline{w}, \mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e})) \wedge wf(\mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e})) \wedge \\
\quad \forall v', \overline{w}, h' \,.\, (((h'\ \mathbf{bw}\ \mathsf{this} \rightarrow o.m(\overline{e})) \wedge I_C(\overline{w}, \mathcal{H} \vdash h') \wedge wf(\mathcal{H} \vdash h' \vdash \mathsf{this} \twoheadleftarrow o.m(v'))) \\
\quad \Rightarrow Q^{\mathcal{H}}_{\mathcal{H} \vdash h' \vdash \mathsf{this} \twoheadleftarrow o.m(v')}) \\
wlp(\mathbf{await}\ \ v := o.m(\overline{e}), Q) \triangleq wlp(\mathbf{await}\ \ o.m(\overline{e}), Q^{v}_{v'}) \\
wlp(o.m(\overline{e}), Q) \triangleq o \neq \mathsf{null} \Rightarrow\ \forall v' \,.\, \mathbf{if}\ \ o = \mathsf{this}\ \ \mathbf{then}\ \ wlp(v' := m'(\overline{e}, \mathsf{this}), Q^{\mathcal{H}}_{\mathcal{H} \vdash \mathsf{this} \twoheadleftarrow \mathsf{this}.m(v')})^{\mathcal{H}}_{\mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e})} \\
\qquad \mathbf{else}\ \ Q^{\mathcal{H}}_{\mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e}) \vdash \mathsf{this} \twoheadleftarrow o.m(v')} \\
wlp(v := o.m(\overline{e}), Q) \triangleq wlp(o.m(\overline{e}), Q^{v}_{v'}) \\
wlp(o!m(\overline{e}), Q) \triangleq o \neq \mathsf{null} \Rightarrow Q^{\mathcal{H}}_{\mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e})} \\
wlp(v' := m'(\overline{e}, \mathsf{this}), Q) \triangleq (wlp(m'(\overline{x}, \mathsf{caller})\ body, Q^{\overline{y}, v'}_{\overline{y}', \mathsf{return}}))^{\overline{x}, \mathsf{caller}, \overline{y}'}_{\overline{e}, \mathsf{this}, \overline{y}} \\
\qquad \text{where } \overline{y} \text{ are the local variables of the caller (including } \mathsf{caller}), \text{ and } \overline{y}' \text{ are fresh logical variables} \\
wlp(m'(\overline{x}, \mathsf{caller})\ body, Q) \triangleq wlp(body, Q) \\
wlp(x := \mathbf{new}\ \ C(\overline{e}), Q) \triangleq \forall x' \,.\, wf(\mathcal{H} \vdash \mathsf{this} \rightarrow x'.\mathbf{new}\ C(\overline{e})) \Rightarrow Q^{x, \mathcal{H}}_{x', \mathcal{H} \vdash \mathsf{this} \rightarrow x'.\mathbf{new}\ C(\overline{e})} \\
wlp(\mathbf{var}\ \overline{y}, Q) \triangleq \forall \overline{y} \,.\, Q \\
wlp(\mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{fi}, Q) \triangleq \mathbf{if}\ b\ \mathbf{then}\ wlp(s_1, Q)\ \mathbf{else}\ wlp(s_2, Q) \\
wlp(s_1; s_2, Q) \triangleq wlp(s_1, wlp(s_2, Q)) \\
wlp(\mathbf{skip}, Q) \triangleq Q \\
wlp(\mathbf{abort}, Q) \triangleq true \\
wlp(v := e, Q) \triangleq Q^{v}_{e} \\
wlp(\mathbf{return}\ e, Q) \triangleq Q^{\mathsf{return}}_{e}
\end{array}
$$

Figure 6: Weakest liberal preconditions for *ABS* statements.

Let $P^{\overline{x}}_{\overline{e}}$, where $\overline{x}$ and $\overline{e}$ are of the same length, denote P where every free occurrence of each $x_i \in \overline{x}$ is replaced by e_i.

The weakest liberal precondition for non-deterministic assignment is given by:

$$wlp(\overline{y} := \mathbf{some}\ , Q) = \forall \overline{y} \,.\, Q$$

where the universal quantifier reflects that the chosen value of y is not known in the prestate. The weakest liberal preconditions for **assert** and **assume** statements are given by:

$$wlp(\mathbf{assert}\ \ b, Q) \triangleq b \wedge Q \qquad \text{and} \qquad wlp(\mathbf{assume}\ \ b, Q) \triangleq b \Rightarrow Q$$

Weakest liberal preconditions for the different *ABS* statements are summarized in Fig. 6. These are straightforwardly derived from the encoding in Fig. 4, where the quantifiers are introduced by the non-deterministic assignments in the encoding. The execution control is explicitly transferred by local synchronous calls, which allows the called method to be executed from a state where the invariant does not hold. The weakest liberal precondition of the local synchronous call statement is defined in terms of the weakest liberal precondition of the called method.

4.3. Hoare Logic

The central feature of Hoare Logic is the Hoare triple, of the form $\{P\}\ s\ \{Q\}$. Triples $\{P\}\ s\ \{Q\}$ have the standard partial correctness semantics: If s is executed in a state where

$$(\textsc{skip}) \quad \{P\}\ \texttt{skip}\ \{P\}$$

$$(\textsc{abort}) \quad \{\mathit{true}\}\ \texttt{abort}\ \{\mathit{false}\}$$

$$(\textsc{assert}) \quad \{P \wedge Q\}\ \texttt{assert}\ \ P\ \{Q\}$$

$$(\textsc{assume}) \quad \{P \Rightarrow Q\}\ \texttt{assume}\ \ P\ \{Q\}$$

$$(\textsc{some}) \quad \{\forall \overline{y}\,.\,Q\}\ \overline{y} := \texttt{some}\ \{Q\}$$

$$(\textsc{assign}) \quad \{Q^v_e\}\ v := e\ \{Q\}$$

$$(\textsc{method}) \quad \frac{\{S\}\ s\ \{R\}}{\{\forall \overline{y}.S\}\ (m(\overline{x})\{\texttt{var}\ \overline{y}; s\})\ \{\exists \overline{y}.R\}}$$

$$(\textsc{call}) \quad \frac{\{S\}\ (m(\overline{x})\ \mathit{body})\ \{R\}}{\{S^{\overline{x},\mathsf{caller}}_{\overline{e},\mathsf{this}}\}\ v := m(\overline{e})\ \{R^{\mathsf{caller},\mathsf{return}}_{\mathsf{this},v}\}} \quad \text{for } v, \overline{x} \notin FV[R] \text{ and } \mathsf{return} \notin FV[S]$$

$$(\textsc{seq}) \quad \frac{\{P\}\ s_1\ \{R\} \qquad \{R\}\ s_2\ \{Q\}}{\{P\}\ s_1; s_2\ \{Q\}}$$

$$(\textsc{if}) \quad \frac{\{P \wedge b\}\ s_1\ \{Q\} \qquad \{P \wedge \neg b\}\ s_2\ \{Q\}}{\{P\}\ \texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \{Q\}}$$

$$(\textsc{cons}) \quad \frac{P' \Rightarrow P \qquad \{P\}\ s\ \{Q\} \qquad Q \Rightarrow Q'}{\{P'\}\ s\ \{Q'\}}$$

$$(\textsc{conj}) \quad \frac{\{P_1\}\ s\ \{Q_1\} \qquad \{P_2\}\ s\ \{Q_2\}}{\{P_1 \wedge P_2\}\ s\ \{Q_1 \wedge Q_2\}}$$

$$(\textsc{disj}) \quad \frac{\{P_1\}\ s\ \{Q_1\} \qquad \{P_2\}\ s\ \{Q_2\}}{\{P_1 \vee P_2\}\ s\ \{Q_1 \vee Q_2\}}$$

$$(\textsc{adap}) \quad \frac{\{P\}\ s\ \{R\}}{\{\forall \overline{x}'\,.\,(\forall \overline{z}\,.\,P \Rightarrow R^{\overline{x}}_{\overline{x}'}) \Rightarrow Q^{\overline{x}}_{\overline{x}'}\}\ s\ \{Q\}} \quad \text{for } \overline{x} = W[s],\ \overline{z} = FV[P,R] \setminus FV[s], \{\overline{x}'\} \not\subseteq FV[P,s,Q]$$

Figure 7: Hoare Rules for the underlying language *SEQ*. As above, $FV[P]$ returns the variables occurring free in the assertion P. $FV[s]$ is the set of (non-local) variables used in the statement s. In addition, we let $W[s]$ return the variables that may be written to by s.

P holds and the execution terminates, then Q holds after s has terminated. Weakest liberal preconditions and Hoare reasoning are closely related since $\{P\}\ s\ \{Q\}$ is the same as $P \Rightarrow \mathit{wlp}(s, Q)$. Hoare rules for *SEQ* is given in Fig. 7, including rules for the subset of *ABS* that is included in *SEQ*. The adaption rule (ADAP)[8] is a right-to-left constructive rule forming a new pre/post condition pair from the premise. The first quantifier reflects that updated program variables are unknown in the prestate, and the second that logical variables in the premise pre/post condition pair can be instantiated to any values.

In Figs. 8 and 9 we extend this rule set with *ABS* specific Hoare rules. Remark that since there is no remote access the internal state of other objects, we may reason about assignments by the standard assignment axiom (ASSIGN). Application of the *ABS* Hoare rules instead of *wlp* may simplify proofs since quantifiers are not used for **suspend** and **await** statements. In order to avoid the problem of undefined right-hand-side expressions, we assume defined default values for all types and that partial functions are applied only when defined, e.g., writing **if** $y \neq 0$ **then** $x := 1/y$ **else abort fi** instead of $x := 1/y$.

$$(\textsc{wf}) \quad \{wf(\mathcal{H})\}\ s\ \{wf(\mathcal{H})\}$$

$$(\textsc{his}) \quad \{\mathcal{H}_0 \ = \ \mathcal{H}\}\ s\ \{\mathcal{H}_0 \ \leqslant \ \mathcal{H}\}$$

$$(\textsc{notNull}) \quad \{o = \mathsf{null}\}\ s'\ \{false\}$$

$$(\textsc{return}) \quad \{Q_e^{\mathsf{return}}\}\ \mathbf{return}\ e\ \{Q\}$$

$$(\textsc{suspend}) \quad \{I_C\}\ \mathbf{suspend}\ \{I_C\}$$

$$(\textsc{await}) \quad \{I_C\}\ \mathbf{await}\ b\ \{I_C \wedge b\}$$

$$(\textsc{callAsync}) \quad \{Q^{\mathcal{H}}_{\mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e})}\}\ o!m(\overline{e})\ \{Q\}$$

$$(\textsc{method})$$
$$\frac{\{S\}\ \mathcal{H} := \mathcal{H} \vdash \mathsf{caller} \twoheadrightarrow \mathsf{this}.m(\overline{x}); s; \mathcal{H} := \mathcal{H} \vdash \mathsf{caller} \leftarrow \mathsf{this}.m(\mathsf{return})\ \{wf(\mathcal{H}) \Rightarrow R\}}{\{\forall \overline{y}\,.\,S\}\ (m(\overline{x})\{\mathbf{var}\ \overline{y}; s\})\ \{\exists \overline{y}\,.\,R\}}$$

$$(\textsc{callSync}1)$$
$$\{\forall v'.Q^{v,\mathcal{H}}_{v',\mathcal{H}\ \vdash\ \mathsf{this} \rightarrow o.m(\overline{e}) \vdash \mathsf{this} \twoheadleftarrow o.m(v')} \wedge o \neq \mathsf{this}\}\ v := o.m(\overline{e})\ \{Q\}$$

$$(\textsc{callSync}2)$$
$$\frac{\{S \wedge \mathsf{caller} = \mathsf{this}\}\ (m(\overline{x})\ body)\ \{R \wedge \mathsf{caller} = \mathsf{this}\} \qquad \text{for } z, \overline{x} \notin FV[R], \mathsf{return} \notin FV[S]}{\{S^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e})} \wedge o = \mathsf{this}\}\ v := o.m(\overline{e})\ \{\exists z\,.\,R^{v,\mathsf{return},\mathsf{caller},\mathcal{H}}_{z,v,\mathsf{this},pop(\mathcal{H})} \wedge \mathcal{H}\ \mathbf{ew}\ \mathsf{this} \twoheadleftarrow \mathsf{this}.m(v)\}}$$

$$(\textsc{awaitCall}1)$$
$$\{h_0 = \mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e}) \wedge I_C{}^{\mathcal{H}}_{h_0} \wedge o = o_0\}\mathbf{await}\ v \ := \ o.m(\overline{e})$$
$$\{h_0 \leq \mathcal{H} \wedge \mathcal{H}\ \mathbf{ew}\ \mathsf{this} \twoheadleftarrow o_0.m(v) \wedge \exists v\,.\,I_C{}^{\mathcal{H}}_{pop(\mathcal{H})}\}$$

$$(\textsc{new})$$
$$\{\forall x'\,.\,(parent(x') = \mathsf{this} \ \wedge \ x' \notin oid(\mathcal{H}) \cup oid(\overline{e})) \Rightarrow Q^{x,\mathcal{H}}_{x',\mathcal{H} \vdash \mathsf{this} \rightarrow x'.\mathbf{new}\ C(\overline{e})}\}\ x := \mathbf{new}\ C(\overline{e})\ \{Q\}$$

Figure 8: Derived Hoare Rules for ABS. For Rule (notNull), s' is a statement calling some method on the object referred to by o. I_C denotes the class invariant, primed variables are logical variables, and s ranges over ABS statements (which cannot use $\mathcal{H}$ as a program variable). Remark that the Rule (callSync1) does not make any assumptions on the callee. In addition rules for assignment, **skip**, **abort**, **if**, sequential composition, as well as CONS, CONJ, DISJ, and ADAP are as given for the SEQ language.

$$(\textsc{callSync}1\text{-}1)$$
$$\{\forall v'.Q^{\mathcal{H}}_{\mathcal{H}\ \vdash\ \mathsf{this} \rightarrow o.m(\overline{e}) \vdash \mathsf{this} \twoheadleftarrow o.m(v')} \wedge o \neq \mathsf{this}\}\ o.m(\overline{e})\ \{Q\}$$

$$(\textsc{callSync}2\text{-}1)$$
$$\frac{\{S \wedge \mathsf{this} = \mathsf{caller}\}\ (m(\overline{x})\ body)\ \{R \wedge \mathsf{this} = \mathsf{caller}\} \qquad \text{for } v', \overline{x} \notin FV[R], \mathsf{return} \notin FV[S]}{\{S^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e})} \wedge o = \mathsf{this}\}\ o.m(\overline{e})\ \{\exists v'\,.\,R^{\mathsf{caller},\mathsf{return},\mathcal{H}}_{\mathsf{this},v',pop(\mathcal{H})} \wedge \mathcal{H}\ \mathbf{ew}\ \mathsf{this} \twoheadleftarrow \mathsf{this}.m(v')\}}$$

$$(\textsc{awaitCall}2)$$
$$\{h_0 = \mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e}) \wedge I_C{}^{\mathcal{H}}_{h_0} \wedge o = o_0\}\mathbf{await}\ o.m(\overline{e})$$
$$\{h_0 \leq \mathcal{H} \wedge I_C{}^{\mathcal{H}}_{pop(\mathcal{H})} \wedge \exists v\,.\,\mathcal{H}\ \mathbf{ew}\ \mathsf{this} \twoheadleftarrow o_0.m(v)\}$$

Figure 9: ABS Hoare rules for the method calls without explicit assignment of the return value.

The following lemma establishes soundness and relative completeness of the proposed Hoare Rules. The proof relies on the weakest liberal preconditions in Fig. 6. For each rule of

$$\{wf(pop(\mathcal{H})) \wedge \mathcal{H}\ \mathbf{ew}\ \mathsf{caller} \twoheadrightarrow \mathsf{this}.m(\overline{x}) \wedge {I_C}^{\mathcal{H}}_{pop(\mathcal{H})}\} body \{wf(\mathcal{H} \vdash \mathsf{caller} \leftarrow \mathsf{this}.m(\mathsf{return})) \Rightarrow {I_C}^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{caller} \leftarrow \mathsf{this}.m(\mathsf{return})}\}$$

Figure 10: Hoare triple formulation of verification condition (2) in Fig. 5 for the method $m(\overline{x})$ *body*.

$\{\mathcal{H}\ \mathbf{ew}\ \mathsf{caller} \twoheadrightarrow \mathsf{this.openR} \wedge \mathit{Readers}(pop(\mathcal{H})) = \mathsf{readers}\}$
$\{\mathit{Readers}(\mathcal{H}) = \mathsf{readers}\}$
await $\mathsf{writer} = \mathsf{null}$;
$\{\mathit{Readers}(\mathcal{H}) = \mathsf{readers} \wedge \mathsf{writer} = \mathsf{null}\}$
$\{\mathit{Readers}(\mathcal{H}) \cup \{\mathsf{caller}\} = \mathsf{Add}(\mathsf{caller}, \mathsf{readers})\}$
$\mathsf{readers} := \mathsf{Add}(\mathsf{caller}, \mathsf{readers})$
$\{\mathit{Readers}(\mathcal{H} \vdash \mathsf{caller} \leftarrow \mathsf{this.openR}) = \mathsf{readers}\}$

Figure 11: Verification details for the body of method openR with respect to the invariant I_1 : $\mathit{Readers}(\mathcal{H}) = \mathsf{readers}$. Here, two consecutive predicates $\{P\}\{Q\}$ resolves to the verification condition $P \Rightarrow Q$. Remark that the ordering between readers is not concerned. Add is the constructor of the set data structure. The verification condition follows from $\mathsf{Add}(x, s) = s \cup \{x\}$.

the form $\{P\}\,\mathrm{s}\,\{Q\}$, soundness follows by $P \Rightarrow \mathit{wlp}(s, Q)$. For instance, for a Boolean guard b, the triple $\{I_C\}$ **await** $b\,\{I_C \wedge b\}$ follows directly since $I_C \Rightarrow \mathit{wlp}(\mathbf{await}\ b, I_C \wedge b)$ with the assumption of preserved well-formedness, see Lemma 1. Given a Hoare triple $\{P\}\,\mathrm{s}\,\{Q\}$, we say that the triple is *relative complete* with respect to the semantical encoding of statement s if $\mathit{wlp}(s, Q) \Rightarrow P$. Thus this completeness result ensures that any $\{P\}\,\mathrm{s}\,\{Q\}$ may be proved if $\{P\}\,\mathrm{s}\,\{Q\}$ is valid by the *ABS* semantics. The proof of Lemma 2 can be found in G.

Lemma 2. *The Hoare rules in Figs. 8 and 9 are sound and relative complete with respect to the semantical encoding, assuming the standard Hoare rules in Fig. 7.*

Remark that by the rules for processor release points and local method calls, adaption is needed whenever the postcondition ranges over method-local variables. However, the values of variables declared local to the method are not changed during method suspension, since processor release points are encoded as non-deterministic assignments to $\overline{w}$ and $\mathcal{H}$. As an alternative to adaptation, we could have accounted for local variables directly in the Hoare rules, e.g. the triple $\{I_C \wedge L\}$ **suspend** $\{I_C \wedge L\}$ follows directly from $\mathit{wlp}(\mathbf{suspend}, I_C \wedge L)$ for $FV[L] \cap \{\overline{w}, \mathcal{H}\} = \emptyset$.

The syntactic encoding of a method $m(\overline{x})$ in Fig. 4 reveals the invocation reaction event $(\mathcal{H} := \mathcal{H} \vdash \mathsf{caller} \twoheadrightarrow \mathsf{this}.m(\overline{x}))$ and completion event$(\mathcal{H} := \mathcal{H} \vdash \mathsf{caller} \leftarrow \mathsf{this}.m(\mathsf{return}))$. Verification condition (2) in Fig. 5 may then be formulated as the Hoare triple given in Fig. 10, where the pre- and postconditions to the method body are derived by standard reasoning.

4.4. Verification of Reader/Writer Example

As a verification example, the successful verification of method openR with respect to the invariant $I_1 : \mathit{Readers}(\mathcal{H}) = \mathsf{readers}$ is shown by the proof outline presented in Fig. 11. The body of openR is analyzed following the pre/post specification outlined in Fig. 10, ignoring the unneeded well-formedness assumptions, and the **await** statement is analyzed by Rule (AWAIT). The complete verification of this case study can be found in E.

4.5. Object Composition

By organizing the state space in terms of only locally accessible variables, including a local history variable recording communication messages local to the object, we obtain a compositional reasoning system, where it suffices to compare the local histories of the composed objects. For this purpose, we adapt a composition method introduced by Soundarajan [22, 23]. When composing objects, the local histories of the composed objects are merged to a common history containing all the events of the composed objects. Local histories must agree with a common wellformed history when composed. Thus for a set O of objects with wellformed history H, we require that the projection of H on each object, e.g. o, is the same as the *local history* h_o of object o:

$$H/\{o\} = h_o$$

The observable behavior of an object $o : C(\overline{e})$ can be captured by a prefix-closed *history invariant*, $I_{o:C(\overline{e})}(h_o)$. If only a subset of the methods should be visible, the history invariant should be restricted to the desired external alphabet. As discussed above, reasoning inside a class is based on the class invariant, which must be satisfied at release points and after method termination and need not be prefix-closed. For instance, 'the history has equally many calls to o_1 and o_2' can be a possible class invariant, but not a history invariant. Therefore the history invariant is in general weaker than the class invariant, i.e.,

$$I_C(\overline{w}, \mathcal{H}) \Rightarrow I_{\mathsf{this}:C(\overline{cp})}(\mathcal{H})$$

By hiding the internal state variables of an object o of class C, an external, prefix-closed *history invariant* $I_{o:C(\overline{e})}(h_o)$ defining its observable behavior on its local history h_o may be obtained from the class invariant of C:

$$I_{o:C(\overline{e})}(h_o) \triangleq \exists h', \overline{w} \,.\; h_o \leqslant h' \wedge (I_C(\overline{w}, h'))^{\mathsf{this},\overline{cp}}_{o,\overline{e}}$$

The substitution replaces the free occurrence of this with o and instantiates the formal class parameters with the actual ones, and the existential quantifier on the attributes hides the local state variables, whereas the existential quantifier on h' ensures that the history invariant is prefix-closed. Note that if the class invariant already is prefix-closed, the history invariant reduces to $\exists \overline{w} \,.\, (I_C(\overline{w}, h_o))^{\mathsf{this},\overline{cp}}_{o,\overline{e}}$. Also observe that a prefix-closed property $P(h_o)$ is the same as the property $\forall h \leq h_o \,.\, P(h)$. Alternatively, a history invariant can be verified by showing that it is maintained by each statement s affecting the local history, i.e., one must prove $\{I_{\mathsf{this}:C(\overline{cp})}(\mathcal{H}) \wedge P\}\; s\; \{Q \Rightarrow I_{\mathsf{this}:C(\overline{cp})}(\mathcal{H})\}$ where P and Q are the pre- and postconditions of s used in the proof outline of the class.

We next consider a composition rule for a (sub)system O of objects $o : C(\overline{e})$ together with dynamically generated objects. The invariant $I_O(H)$ of such a subsystem is given by

$$I_O(H) \triangleq \; wf(H, new_{id}(O \cup new_{ob}(H))) \bigwedge_{(o:C(\overline{e}))\in O\cup new_{ob}(H)} I_{o:C(\overline{e})}(H/\{o\})$$

where H is the history of the subsystem. The wellformedness property serves as a connection between the local histories, which are by definition over disjoint alphabets. The quantification ranges over all objects in O as well as all generated objects in the composition, which is a finite number at any execution point. Note that the system invariant is obtained directly from the external history 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 non-deterministic. When considering a closed subsystem, one may add the assumption

$$(oid(H) \setminus \{\mathsf{null}\}) \subseteq new_{id}(O \cup new_{ob}(H))$$

Reasoning about a *global system* can be done as above assuming the existence of an initial object main of some class *Main*, such that all objects are created by main or generated objects. Thus main is an ancestor of all objects. The global invariant of a total system of dynamically created objects may be constructed from the history invariants of the composed objects, requiring wellformedness of global history. According to the rule above, the global invariant $I_{\{\mathsf{main}:Main\}}(H)$ of a global system with history H is

$$\begin{array}{l} wf(H, new_{id}(new_{ob}(H)) \cup \{\mathsf{main}\}) \wedge \\ \qquad (oid(H) \setminus \{\mathsf{null}, \mathsf{main}\}) \subseteq new_{id}(new_{ob}(H)) \bigwedge_{(o:C(\overline{e}))\in new_{ob}(H)} I_{o:C(\overline{e})}(H/\{o\}) \end{array}$$

assuming *true* as the class invariant for main. Since main is the initial root object, the creation of main is not reflected on the global history H, i.e., $\mathsf{main} \notin new_{id}(new_{ob}(H))$. The following lemma expresses that parent chains are cycle free for global systems.

Lemma 3. *Given a global system with history H and invariant $I(H)$, then*

$$\forall o \in new_{id}(new_{ob}(H)) \,.\, o \notin anc(o) \wedge \mathsf{main} \in anc(o) \wedge (anc(o) \setminus \{\mathsf{main}\}) \subseteq new_{id}(new_{ob}(H))$$

Proof. By induction over the length of H. The base case $H = \varepsilon$ is trivial. For the induction step, we consider a history of the form $H \vdash \gamma$, for $\gamma : Ev$, and prove

$$\begin{array}{l} \forall o \in new_{id}(new_{ob}(H \vdash \gamma)) \,. \\ \qquad o \notin anc(o) \wedge \mathsf{main} \in anc(o) \wedge (anc(o) \setminus \{\mathsf{main}\}) \subseteq new_{id}(new_{ob}(H \vdash \gamma)) \end{array}$$

under induction hypothesis IH: $\forall o \in new_{id}(new_{ob}(H)) \,.\, o \notin anc(o) \wedge \mathsf{main} \in anc(o) \wedge (anc(o) \setminus \{\mathsf{main}\}) \subseteq new_{id}(new_{ob}(H))$. The conclusion follows from IH for all γ except $\gamma : NEv$ (object creation events), since we then have $new_{id}(new_{ob}(H \vdash \gamma)) = new_{id}(new_{ob}(H))$.

For the case $H \vdash o \rightarrow o'.\mathbf{new}$ (ignoring the class of o'), the conclusion follows from IH and the proof obligation:

$$o' \notin anc(o') \wedge \mathsf{main} \in anc(o') \wedge (anc(o') \setminus \{\mathsf{main}\}) \subseteq new_{id}(new_{ob}(H))$$

By wellformedness we have $parent(o') = o \wedge o' \notin oid(H)$, and by the definition of *anc*, the proof obligation can then be written as:

$$o' \notin \{o\} \cup anc(o) \wedge \mathsf{main} \in \{o\} \cup anc(o) \wedge ((\{o\} \cup anc(o)) \setminus \{\mathsf{main}\}) \subseteq new_{id}(new_{ob}(H))$$

We distinguish two cases, $o = \mathsf{main}$ and $o \neq \mathsf{main}$.
Case $o = \mathsf{main}$: The conclusion follows directly by $anc(\mathsf{main}) = \mathsf{main}$ and $o' \neq \mathsf{main}$.
Case $o \neq \mathsf{main}$: Since $o \in oid(H \vdash o \rightarrow o'.\mathbf{new})$, we have $o \in new_{id}(new_{ob}(H \vdash o \rightarrow o'.\mathbf{new}))$ since H is global, which gives $o \in new_{id}(new_{ob}(H))$. Since $o' \notin oid(H)$, we then have $o \neq o'$, and the proof obligation reduces to

$$o' \notin anc(o) \wedge \mathsf{main} \in anc(o) \wedge (anc(o) \setminus \{\mathsf{main}\}) \subseteq new_{id}(new_{ob}(H))$$

Since $o \in new_{id}(new_{ob}(H))$, we have $\mathsf{main} \in anc(o)$ and $(anc(o) \setminus \{\mathsf{main}\}) \subseteq new_{id}(new_{ob}(H))$ by IH. Since $o' \neq \mathsf{main}$, the remaining proof obligation $o' \notin anc(o)$ can be rewritten as $o' \notin (anc(o) \setminus \{\mathsf{main}\})$, which by IH is satisfied if $o' \notin new_{id}(new_{ob}(H))$. Since $new_{id}(new_{ob}(H)) \subseteq oid(H)$, we prove $o' \notin oid(H)$ which follows by the wellformedness assumptions above.

```
class Buffer {
  Obj cell, Nat cnt, Buffer next;
  {cell := null; cnt := 0; next := null }

  Void put(Obj x) { if (cnt = 0) then cell := x
    else if (next=null) then next := new Buffer fi; next.put(x) fi;
    cnt := cnt + 1 }

  Obj get() { var Obj r; await (cnt > 0); cnt := cnt − 1;
    if (cell = null) then r := next.get() else r := cell; cell := null fi;
    return r }
}
```

Figure 12: Implementation of the Buffer class.

4.6. Final Remark of Reader/Writer Example

The invariant $OK(\mathcal{H})$ is prefix-closed and may be used as a composable history invariant. Remark that the property *Writing* $(\mathcal{H}) = 0$ can be verified as a part of the class invariant since *db.write* is only called synchronously. This property is however not contributing to the history invariant for RWcontroller objects since it is not prefix-closed.

5. Unbounded Buffer Example

Different from the Reader/Writer example where *class* is the scope of verification, here we present how to achieve compositional verification among objects. In this example we consider a class Buffer with put and get operations. The class contains a single memory cell and a link to another buffer object. If the buffer receives a call to put with argument x, it stores x in its cell if the buffer is empty. Otherwise, the put call is passed on to the next buffer (which is dynamically created if null). 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 object. Thus a buffer instance as seen from the outside implements a FIFO ordering. For simplicity we assume that the arguments of put operations are not null (this could have been ensured by an additional check in method put). And we omit **return** *void* statements. The code for the Buffer class can be found in Fig. 12. Notice that the Buffer class is implemented using synchronous call statements, which means that the correspondence between invocation events to and completion reaction events from the next object is tight. An implementation using asynchronous calls could break the FIFO structure of the buffer.

The desired property of a buffer object is the FIFO property, i.e., that the get operation of this Buffer object will return elements in the same order as they were inserted by the put operation. Using the prefix relation we specify this property by

$$\mathit{fifo}(\mathsf{this}, h) \triangleq \mathit{out}(\mathsf{this}, h) \leq \mathit{in}(\mathsf{this}, h)$$

where $\mathit{in}(\mathsf{this}, h)$ is the sequence of elements inserted by the *put* operations and $\mathit{out}(\mathsf{this}, h)$ is the ones returned by *get* operations, defining the following auxiliary functions:

$$\begin{array}{lcl} \mathit{in}(\mathsf{this}, h) & \triangleq & (h/\{\_ \twoheadrightarrow \mathsf{this}.\mathit{put}\}).\mathit{data} \\ \mathit{out}(\mathsf{this}, h) & \triangleq & (h/\{\_ \leftarrow \mathsf{this}.\mathit{get}\}).\mathit{data} \end{array}$$

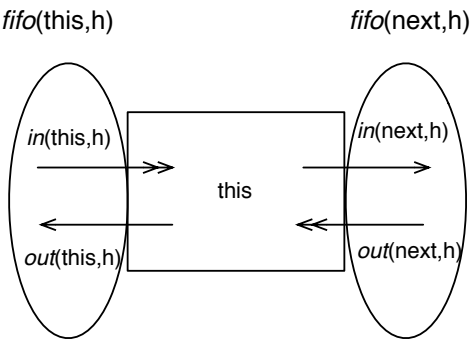

Figure 13: First in first out property of a buffer object.

However, the FIFO property $\mathit{fifo}(\mathsf{this}, \mathcal{H})$ cannot be proved as a local invariant; it depends on the FIFO property of the next object, which may be expressed as

$$\mathit{fifo}(\mathsf{next}, h) \triangleq \mathit{out}(\mathsf{next}, h) \leq \mathit{in}(\mathsf{next}, h)$$

where

$$\begin{array}{lcl} \mathit{in}(\mathsf{next}, h) & \triangleq & (h/\{\_ \rightarrow \mathsf{next}.\mathit{put}\}).\mathit{data} \\ \mathit{out}(\mathsf{next}, h) & \triangleq & (h/\{\_ \twoheadleftarrow \mathsf{next}.\mathit{get}\}).\mathit{data} \end{array}$$

Thus $\mathit{fifo}(\mathsf{next}, \mathcal{H})$ expresses that the sequence of elements returned from next is a prefix of the ones insert to next. For local reasoning we therefore consider a conditional FIFO property with an assumption on next:

$$\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \mathit{fifo}(\mathsf{this}, \mathcal{H})$$

Fig. 13 gives a visualized illustration. We will later show that the assumption, $\mathit{fifo}(\mathsf{next}, \mathcal{H})$, of a buffer object is satisfied among object composition. For simplicity we will assume $\mathsf{null} \notin \mathit{in}(\mathsf{next}, \mathcal{H})$, and do not include this as an explicit assumption below.

5.1. Local Reasoning

In order to prove the conditional FIFO property, we need stronger class invariants for the Buffer class, involving the attributes used in the program. First, we define *buf* such that $\mathit{buf}(\mathsf{next}, \mathcal{H})$ returns the buffer content of next:

$$\mathit{buf}(o, h) \triangleq \mathit{in}(o, h) \ \mathbf{after} \ \#\mathit{out}(o, h)$$

where h **after** n denotes the rest of h (if any) after the n first elements. We may then formulate the following two class invariants and prove them in F.1 and F.2:

$$\mathsf{cnt} = \#(\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H})) \tag{6}$$

$$\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \mathit{in}(\mathsf{this}, \mathcal{H}) = \mathit{out}(\mathsf{this}, \mathcal{H}) \vdash (\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H})) \tag{7}$$

where $v + h$ is h for $v = \mathsf{null}$ otherwise h with v first. Accordingly, we proved the conditional FIFO property, $\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \mathit{fifo}(\mathsf{this}, \mathcal{H})$ as a prefix-closed invariant in F.3 through F.4.

Since there are no local calls in the code, the additional pre- and postconditions to the methods are not needed.

The value of next is related to the history by the condition

$$\mathsf{next} \sim (\mathcal{H}/\{\mathsf{this} \rightarrow .\mathbf{new}\ \mathsf{Buffer}\}).\mathit{callee}$$

where $\sim$ is defined by $\mathsf{null} \sim \varepsilon$ and $x \sim \varepsilon \vdash x$ and $x \sim q = \mathit{false}$ otherwise. A composable *history invariant* for the Buffer class may then be formulated as:

$$I_{\mathsf{this:Buffer}}(\mathcal{H}) \triangleq \exists\, \mathsf{next}\,.\,\mathsf{next} \sim (\mathcal{H}/\{\mathsf{this} \rightarrow .\mathbf{new}\mathsf{Buffer}\}).\mathit{callee} \wedge (\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \mathit{fifo}(\mathsf{this}, \mathcal{H}))$$

hiding the attribute next by an existential quantifier. Remark that if next is null, the second conjunct of the invariant reduces to: $\mathit{fifo}(\mathsf{this}, \mathcal{H})$.

5.2. Object Composition

The external history invariant of a Buffer object o is obtained by substitution of this by o in the above history invariant:

$$I_{o:\mathsf{Buffer}}(h_o) \triangleq \exists n\,.\,n \sim (h_o/\{o \rightarrow .\mathbf{new}\ \mathsf{Buffer}\}).\mathit{callee} \wedge (\mathit{fifo}(n, h_o) \Rightarrow \mathit{fifo}(o, h_o))$$

Consider a buffer subsystem with history H including an (outermost) object $o : \mathsf{Buffer}$ and $new_{ob}(H)$, i.e., a closed system containing o and all objects in the next-chain of o. For this system, we have the invariant:

$$I_{\{o\}}(H) \triangleq \mathit{wf}(H, new_{id}(new_{ob}(H)) \cup \{o\}) \wedge I_{o:\mathsf{Buffer}}(H/\{o\}) \bigwedge_{(n:\mathsf{Buffer}) \in new_{ob}(H)} I_{n:\mathsf{Buffer}}(H/\{n\})$$

One may prove by induction that the subsystem generated by a buffer object o satisfies the FIFO property $\mathit{fifo}(o, H)$, 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. The induction can be done over next-closed subsystems of Buffer objects, with the Buffer object o with no next object (i.e. where $new_{ob}(H/\{o\})$ is empty) as the base case, and a Buffer object o with a next object n given by $new_{ob}(H/\{o\})$ as the induction step, assuming the induction hypothesis for n. In order to ensure $\mathit{fifo}(o, H)$, the crucial step is to ensure the implication $\mathit{fifo}(n, H/\{n\}) \Rightarrow \mathit{fifo}(n, H/\{o\})$. We observe that history invariants

$$\#(H/\{o \rightarrow n.\mathit{put}\}) - \#(H/\{o \twoheadleftarrow n.\mathit{put}\}) \leq 1$$
$$\#(H/\{o \rightarrow n.\mathit{get}\}) - \#(H/\{o \twoheadleftarrow n.\mathit{get}\}) \leq 1$$

are trivially satisfied, since the communication with next is synchronous. By assuming that there is no external interaction with the created objects, i.e., $\{(H/new_{id}(new_{ob}(H))).\mathit{caller}\} \subseteq new_{id}(new_{ob}(H)) \cup \{o\}$, we have by well-formedness of H that $(H/\{\_ \twoheadrightarrow n.\mathit{put}\}).\mathit{caller} = o$ and $(H/\{\_ \twoheadrightarrow n.\mathit{get}\}).\mathit{caller} = o$. Given a wellformed history H as above, we then have $\mathit{fifo}(n, H/\{n\}) \Rightarrow \mathit{fifo}(n, H/\{o\})$. We thereby have the FIFO property $\mathit{fifo}(o, H)$ for the subsystem of $new_{id}(new_{ob}(H)) \cup \{o\}$ (under the two assumptions on the subsystem: no external interaction with generated objects and no external interaction with null as argument to *put*).

The proof could be simplified by using rely/guarantee style reasoning. Then $\mathit{fifo}(\mathsf{next}, \mathcal{H})$ could be taken as the rely part and $\mathit{fifo}(\mathsf{this}, \mathcal{H})$ as the guarantee part. Only the guarantee part needs to be verified for the class, under the assumption of the rely part. The rely part must be discharged in the composition step. For a closed system this would be possible since there is no other communication than the one generated by the next-chain.

6. Related and Future Work

Reasoning about distributed and object-oriented systems is challenging, due to the combination of concurrency, compositionality and object orientation. Moreover, the gap in reasoning complexity between sequential and distributed, object-oriented systems makes tool-based verification difficult in practice. A recent survey of these challenges can be found in [16]. The present approach follows the line of work based on communication histories to model object communication events in a distributed setting [6, 7, 26]. Objects are concurrent and interact solely by method calls, and remote access to object fields are forbidden. Object generation is reflected in the history by means of creation events. This enables compositional reasoning of concurrent systems with dynamic generation of objects and aliasing.

The *ABS* language provides a natural model for object-oriented distributed systems, with the advantage of explicit programming control of blocking and non-blocking calls. Other object-oriented features such as inheritance is not considered here; however, our approach may be combined with behavioral subtyping, as well as lazy behavioral subtyping which has been worked out for the same language setting [27]. History invariants can be naturally included in interface definitions, defining the external visible alphabet of an object and specifying the external behavior of the provided methods. Adding interfaces to our formalism would affect the composition rule in that events not observed through the interface must be hidden.

A Hoare Logic for concurrent processes (objects) is presented in [28]. The Hoare logic is compositional, and soundness and relative completeness are proven. In contrast to our work, communication is by message passing rather than by method interaction, and the objects communicate through FIFO channels. Olderog and Apt consider transformation of program statements preserving semantical equivalence [13]. This approach is extended in [29] to a general methodology for transformation of language constructions resulting in sound and relative 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 [29] to multi-threaded systems seems to require interference freedom tests.

The current work is based on earlier work reported in [14, 15]. Those works are based on a two-event semantics for method calls, where message sending is visible on the local history of the receiver. Message sending then leads to restrictions on the local history of the receiver that must be accounted for in the model. The four-event semantics suggested in the current paper however, leads to *disjoint alphabets* for different objects. This simplifies the model and the accompanied proof system, thereby reducing the gap between reasoning about sequential systems and distributed object-oriented systems. Especially, when reasoning about a class, it is not necessary to explicitly account for the activity of objects in the environment. The reasoning involves specifications given in terms of (internal) class invariants and (external) history invariants for single objects and (sub)systems of concurrent objects. A class invariant gives rise to an object history invariant describing the external behavior of the object, and a composition rule gives history invariants for a system or subsystem. The composition rule is similar to previous approaches [22, 23], and the paper focuses in particular on the reasoning system for class invariants. Related is also the work of Dylla and Ahrendt [16], which presents a compositional verification system for *Creol*. As in our work, the analysis of processor release points uses non-deterministic assignments in order to capture the activity of other processes; the denotational Creol semantics features the same four communication events, there called 'invoc', 'begin', 'end', and 'comp'. However, the reasoning system [16] is based on the two-event semantics of [14], which requires more complex rules than the present

one. A prototype of the verification system [16] has been implemented as part of the KeY [30] framework. We believe that also our verification system is suitable for implementation within the KeY framework, and a dynamic logic formulation of the reasoning system is currently being investigated. Having support for (semi-)automatic verification, such an implementation will be valuable when developing larger case studies. As a part of this work, we also intend to extend the four event semantics with *ABS* futures [4]. Our current framework is well suited for this extension; the history events for asynchronous method calls will be extended by a future identity, and history wellformedness must be relaxed in order to allow several readings of the same return value, possibly by different objects. Additionally, it is natural to investigate how our reasoning system would benefit by extending it with rely/guarantee style reasoning. We may for instance use callee interfaces as an assumption in order to express properties of the values returned by method calls. More sophisticated techniques may also be used, e.g., [24, 25] adapts rely/guarantee style reasoning to history invariants. The rely part may be expressed as properties over input events, whereas the guaranteed behavior is associated with output events. Such techniques however, requires more complex object composition rules, and are not considered here since the focus is on class invariants.

7. Conclusion

In this paper we present a compositional reasoning system for distributed objects based on the concurrency and communication model of the *ABS* language. Compositional reasoning is facilitated by expressing object properties in terms of observable interaction between the object and its environment, recorded on communication histories. A method call cycle is reflected by four events, which gives rise to disjoint communication alphabets for different objects. Specifications in terms of history invariants may then be derived independently for each object and composed in order to derive properties for object systems. At the class level, invariants define relations between the class attributes and the observable communication of class instances. By construction, the *wlp* system for class analysis is sound and complete relative to the given semantics, and the presented Hoare system is proven sound and relative complete. This system is easy to apply in the sense that class reasoning is similar to standard sequential reasoning, but with the addition of effects on the local history for statements involving method calls. The presented reasoning system is illustrated by two examples.

Acknowledgements. The authors gratefully acknowledge valuable comments from the anonymous reviewers. Their criticism has improved the paper.

References

[1] International Telecommunication Union, Open Distributed Processing - Reference Model parts 1–4, Tech. rep., ISO/IEC, Geneva (Jul. 1995).

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

[3] F. S. de Boer, D. Clarke, E. B. Johnsen, A complete guide to the future, in: R. de Nicola (Ed.), Proc. 16th European Symposium on Programming (ESOP'07), Vol. 4421 of Lecture Notes in Computer Science, Springer-Verlag, 2007, pp. 316–330.

[4] R. Hähnle, E. B. Johnsen, B. M. Østvold, J. Schäfer, M. Steffen, A. B. Torjusen, Deliverable D1.1A Report on the Core ABS Language and Methodology: Part A, http://www.cse.chalmers.se/research/hats/sites/default/files/Deliverable11a_rev2.pdf (2010).

[5] J. Schäfer, A. Poetzsch-Heffter, JCoBox: Generalizing active objects to concurrent components, in: T. D'Hondt (Ed.), European Conference on Object-Oriented Programming (ECOOP 2010), Vol. 6183 of Lecture Notes in Computer Science, Springer-Verlag, 2010, pp. 275–299.

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

[7] C. A. R. Hoare, Communicating Sequential Processes, International Series in Computer Science, Prentice Hall, 1985.

[8] O.-J. Dahl, Verifiable Programming, International Series in Computer Science, Prentice Hall, New York, N.Y., 1992.

[9] O.-J. Dahl, Object-oriented specifications, in: Research directions in object-oriented programming, MIT Press, Cambridge, MA, USA, 1987, pp. 561–576.

[10] A. S. A. Jeffrey, J. Rathke, Java Jr.: Fully abstract trace semantics for a core Java language, in: Proc. European Symposium on Programming, Vol. 3444 of Lecture Notes in Computer Science, Springer-Verlag, 2005, pp. 423–438.

[11] E. Ábrahám, I. Grabe, A. Grüner, M. Steffen, Behavioral interface description of an object-oriented language with futures and promises, Journal of Logic and Algebraic Programming 78 (7) (2009) 491–518.

[12] B. Alpern, F. B. Schneider, Defining liveness, Information Processing Letters 21 (4) (1985) 181–185.

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

[14] J. Dovland, E. B. Johnsen, O. Owe, Verification of concurrent objects with asynchronous method calls, in: Proceedings of the IEEE International Conference on Software Science, Technology & Engineering(SwSTE'05), IEEE Computer Society Press, 2005, pp. 141–150.

[15] J. Dovland, E. B. Johnsen, O. Owe, Observable Behavior of Dynamic Systems: Component Reasoning for Concurrent Objects, Electronic Notes in Theoretical Computer Science 203 (3) (2008) 19–34.

[16] W. Ahrendt, M. Dylla, A system for compositional verification of asynchronous objects, Science of Computer Programming In Press, Corrected Proof (2010) –. doi:DOI: 10.1016/j.scico.2010.08.003.
URL http://www.sciencedirect.com/science/article/B6V17-50TRX0X-1/2/80b594aa8b2596602fdbd0d6dad85849

[17] W. Ahrendt, M. Dylla, A verification system for distributed objects with asynchronous method calls, in: K. Breitman, A. Cavalcanti (Eds.), Proc. International Conference on Formal Engineering Methods (ICFEM'09), Vol. 5885 of Lecture Notes in Computer Science, Springer-Verlag, 2009, pp. 387–406.

[18] C. C. Din, J. Dovland, E. B. Johnsen, O. Owe, Observable behavior of distributed systems: Component reasoning for concurrent objects, Research Report 401, Department of Informatics, University of Oslo, Norway (Oct. 2010).

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

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

[21] C. Pierik, F. S. d. Boer, A Syntax-Directed Hoare Logic for Object-Oriented Programming Concepts, Tech. Rep. UU-CS-2003-010, Department of Information and Computing Sciences, Utrecht University (2003).

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

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

[24] O.-J. Dahl, O. Owe, Formal methods and the RM-ODP, Research Report 261, Department of Informatics, University of Oslo, Norway (May 1998).

[25] E. B. Johnsen, O. Owe, Object-oriented specification and open distributed systems, in: O. Owe, S. Krogdahl, T. Lyche (Eds.), From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl, Vol. 2635 of Lecture Notes in Computer Science, Springer-Verlag, 2004, pp. 137–164.

[26] O.-J. Dahl, Can program proving be made practical?, in: M. Amirchahy, D. Néel (Eds.), Les Fondements de la Programmation, Institut de Recherche d'Informatique et d'Automatique, Toulouse, France, 1977, pp. 57–114.

[27] J. Dovland, E. B. Johnsen, O. Owe, M. Steffen, Lazy behavioral subtyping, J. Log. Algebr. Program. 79 (7) (2010) 578–607.

[28] F. S. de Boer, A Hoare Logic for Dynamic Networks of Asynchronously Communicating Deterministic Processes, Theoretical Computer Science 274 (2002) 3–41.

[29] F. S. de Boer, C. Pierik, How to Cook a Complete Hoare Logic for Your Pet OO Language, in: Formal Methods for Components and Objects (FMCO'03), Vol. 3188 of Lecture Notes in Computer Science, Springer-Verlag, 2004, pp. 111–133.

[30] B. Beckert, R. Hähnle, P. H. Schmitt (Eds.), Verification of Object-Oriented Software. The KeY Approach, Vol. 4334 of Lecture Notes in Artificial Intelligence, Springer-Verlag, 2007.

A. Syntax of the *ABS* functional sublanguage

BNF syntax for the *ABS* functional sublanguage with terms t, data type definitions Dd, and function definitions F is given below:

Dd	::=	**data** D $\{[Co(T^*)]^*\}$	data type declaration
F	::=	**def** T $fn([T\ x]^*) == rhs$	function declaration
t	::=	$Co(e^*) \mid fn([e]^*)$	constructor and function application
		$\mid (e, e)$	pair constructor
p	::=	$v \mid Co(p^*) \mid (p, p)$	pattern
rhs	::=	e	pure expressions
		$\mid$ **case** $e\{b^*\}$	case expression
b	::=	$p \ \Rightarrow \ \ rhs$	branch

Data types are implicitly defined by declaring constructor functions Co. The right hand side of the definition of a function fn may be a nested case expression. Patterns include constructor terms and pairs over constructor terms. The functional if-then-else contruct and infix operator are not included in the syntax above. We use + and − for numbers, **and** and **or** for booleans, and = for equality.

B. Complete Code of Fairness Reader/Writer

```
data Data{int(Int) bool(Bool) string(String) obj(Obj) Nothing}
data Map{Empty Bind(Int, Data, Map)}
data DataSet{Empty Add(Data, DataSet)}

def Bool isElement(Data element, DataSet set) ==
  case set{Empty => False;
    Add(d, s) => element = d or isElement(element, s)}

def Data lookup(Int key, Map map) ==
  case map{Empty => Nothing;
    Bind(k, d, m) => if key = k then d else lookup(key, m) }

def DataSet delete(Data element, DataSet set) ==
  case set{Empty => Empty;
    Add(d, s) => if element = d then delete(element, s) else Add(d, delete(element, s))}

def Map modify(Int key, Data element, Map map) ==
  case map{Empty => Bind(key, element, Empty);
    Bind(k, d, m) => if key = k then Bind(k, element, m)
                     else Bind(k, d, modify(key, element, m))}

def Int size(DataSet set) ==
  case set{Empty => 0;
    Add(d, s) => 1 + size(s)}

interface RW{
  Void openR();
  Void closeR();
  Void openW();
  Void closeW();
  Data read(Int key);
  Void write(Int key, Data element) }
```

```
interface DB{
   Data read(Int key);
   Void write(Int key, Data element)}

class DataBase implements DB{
   Map map;
   {map := Empty;}
   Data read(Int key) {return lookup(key, map)}
   Void write(Int key, Data element) {map := modify(key, element, map)} }

class RWController() implements RW{
   DB db; DataSet readers; Obj writer; Int pr;
   {db := new DataBase(); readers := Empty; writer := null; pr := 0}
   Void openR(){await writer = null; readers := Add(caller, readers)}
   Void closeR(){readers := delete(caller, readers)}
   Void openW(){await writer = null; writer := caller; readers := Add(caller, readers)}
   Void closeW(){await writer = caller; writer := null; readers := delete(caller, readers)}
   Data read(Int key){ Data result;
      await isElement(caller, readers); pr := pr + 1;
      await result := db.read(key); pr := pr − 1; return result }
   Void write(Int key, Data value){
      await caller = writer and pr = 0 and
        (readers = Empty or (isElement(writer, readers) and size(readers) = 1));
      db.write(key, value) }}
```

C. Definition of Writers

$Writers : Seq[Ev] \to Set[Obj]$

$Writers(\varepsilon) \triangleq \emptyset$
$Writers(h \vdash o \leftarrow \mathsf{this.openW}) \triangleq Writers(h) \cup \{o\}$
$Writers(h \vdash o \leftarrow \mathsf{this.closeW}) \triangleq Writers(h) \setminus \{o\}$
$Writers(h \vdash \boldsymbol{others}) \triangleq Writers(h)$

D. Definition of Writing

$Writing : Seq[Ev] \to Nat$

$Writing(h) \triangleq \#(h/\{\mathsf{this} \to \mathsf{db.write}\}) - \#(h/\{\mathsf{this} \twoheadleftarrow \mathsf{db.write}\})$

E. Verification Details for RWController

E.1. Method: openR

$I_1 \wedge I_2 \wedge I_3 \wedge I_4$:

$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$
await writer = null;
$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge \mathsf{writer} = \mathsf{null}\}$
$\{Readers(\mathcal{H}) \cup \{\mathsf{caller}\} = \mathsf{Add}(\mathsf{caller}, \mathsf{readers}) \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge$
$Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge \#Writers(\mathcal{H}) = 0\}$
readers := Add(caller, readers);
$\{Readers(\mathcal{H}) \cup \{\mathsf{caller}\} = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge$
$Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge \#Writers(\mathcal{H}) = 0\}$

E.2. Method: openW

$I_1 \wedge I_2 \wedge I_3 \wedge I_4$:

$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$
await writer = null;
$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge \mathsf{writer} = \mathsf{null}\}$
$\{Readers(\mathcal{H}) \cup \{\mathsf{caller}\} = \mathsf{Add}(\mathsf{caller}, \mathsf{readers}) \wedge$
$Writers(\mathcal{H}) \cup \{\mathsf{caller}\} = \{\mathsf{caller}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge \#Writers(\mathcal{H}) = 0\}$
writer := caller;
$\{Readers(\mathcal{H}) \cup \{\mathsf{caller}\} = \mathsf{Add}(\mathsf{caller}, \mathsf{readers}) \wedge$
$Writers(\mathcal{H}) \cup \{\mathsf{caller}\} = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge \#Writers(\mathcal{H}) = 0\}$
readers := Add(caller, readers);
$\{Readers(\mathcal{H}) \cup \{\mathsf{caller}\} = \mathsf{readers} \wedge$
$Writers(\mathcal{H}) \cup \{\mathsf{caller}\} = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge \#Writers(\mathcal{H}) = 0\}$

E.3. Method: closeR

$I_1 \wedge I_2 \wedge I_3 \wedge I_4$:

$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$
$\{Readers(\mathcal{H}) \setminus \{\mathsf{caller}\} = delete(\mathsf{caller}, \mathsf{readers}) \wedge$
$Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$
readers := $delete$(caller, readers);
$\{Readers(\mathcal{H}) \setminus \{\mathsf{caller}\} = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$

E.4. Method: closeW

$I_1 \wedge I_2 \wedge I_3 \wedge I_4$:

$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$
await writer = caller;
$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge \mathsf{writer} = \mathsf{caller}\}$
$\{Readers(\mathcal{H}) \setminus \{\mathsf{caller}\} = delete(\mathsf{caller}, \mathsf{readers}) \wedge$
$Writers(\mathcal{H}) \setminus \{\mathsf{caller}\} = \{\mathsf{null}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$
writer := $null$;
$\{Readers(\mathcal{H}) \setminus \{\mathsf{caller}\} = delete(\mathsf{caller}, \mathsf{readers}) \wedge$
$Writers(\mathcal{H}) \setminus \{\mathsf{caller}\} = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$
readers := $delete$(caller, readers);
$\{Readers(\mathcal{H}) \setminus \{\mathsf{caller}\} = \mathsf{readers} \wedge$
$Writers(\mathcal{H}) \setminus \{\mathsf{caller}\} = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$

E.5. Method: read

$I_1 \wedge I_2 \wedge I_3 \wedge I_4$:

$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$
await isElement(caller, readers);
$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge$
$Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge \mathsf{isElement}(\mathsf{caller}, \mathsf{readers})\}$
$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge$

$\quad Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) + 1 = \mathsf{pr} + 1 \wedge OK(\mathcal{H}) \wedge Writing(\mathcal{H}) = 0\}$
$\mathsf{pr} := \mathsf{pr} + 1;$
$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge$
$\quad Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) + 1 = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge Writing(\mathcal{H}) = 0\}$
await result := db.read(key);
$\{(\exists \mathsf{result}\ .\ (I_1 \wedge I_2 \wedge I_3 \wedge I_4)^{\mathcal{H}}_{pop(\mathcal{H})}) \wedge \mathcal{H}$ **ew** this $\twoheadleftarrow db.\mathsf{read}(\mathsf{result})\}$
$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} - 1 \wedge OK(\mathcal{H})\}$
$\mathsf{pr} := \mathsf{pr} - 1;$
$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$
return result;
$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$

E.6. Method: write

$I_1 \wedge I_2 \wedge I_3 \wedge I_4$:

$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$
await caller = writer && pr = 0 &&
$\quad (\mathsf{readers} = Empty \vee (\mathsf{isElement}(\mathsf{writer}, \mathsf{readers}) \&\& \mathsf{size}(\mathsf{readers}) = 1));$
$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge$
$\quad \mathsf{caller} = \mathsf{writer} \wedge \mathsf{pr} = 0 \wedge (\mathsf{readers} = Empty \vee (\mathsf{isElement}(\mathsf{writer}, \mathsf{readers}) \wedge \mathsf{size}(\mathsf{readers}) = 1))\}$
$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge$
$\quad Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge Reading(\mathcal{H}) = 0 \wedge \#Writers(\mathcal{H}) = 1\}$
db.write(key, value);
$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})\}$

F. Verification Details for Unbounded Buffer

F.1. The put method

Proof outline:

$\{I\}$
$\{((\mathsf{cnt} = 0 \Rightarrow Q^{\mathsf{cell}}_x) \wedge (\mathsf{cnt} \neq 0 \Rightarrow Q^{\mathcal{H}}_{\mathcal{H} \vdash \mathsf{this} \rightarrow \mathsf{next}.put(x) \vdash \mathsf{this} \twoheadleftarrow \mathsf{next}.put}))^{\mathcal{H}}_{\mathcal{H} \vdash \mathsf{caller} \twoheadrightarrow \mathsf{this}.put(x)}\}$
$\mathcal{H} = \mathcal{H} \vdash \mathsf{caller} \twoheadrightarrow \mathsf{this}.put(\overline{x});$
$\{(\mathsf{cnt} = 0 \Rightarrow Q^{\mathsf{cell}}_x) \wedge (\mathsf{cnt} \neq 0 \Rightarrow Q^{\mathcal{H}}_{\mathcal{H} \vdash \mathsf{this} \rightarrow \mathsf{next}.put(x) \vdash \mathsf{this} \twoheadleftarrow \mathsf{next}.put})\}$
if $(\mathsf{cnt} = 0)$ **then** $\{Q^{\mathsf{cell}}_x\}$ cell := x
$\quad$ **else if** $(\mathsf{next} = \mathsf{null})$ **then** next := **new** Buffer **fi**;
$\qquad \{\mathsf{next} \neq \mathsf{null} \wedge Q^{\mathcal{H}}_{\mathcal{H} \vdash \mathsf{this} \rightarrow \mathsf{next}.put(x) \vdash \mathsf{this} \twoheadleftarrow \mathsf{next}.put}\}$
$\qquad \mathsf{next}.\mathsf{put}(x)$
fi;
$\{Q\}$
$\mathsf{cnt} := \mathsf{cnt} + 1;$
$\{I^{\mathcal{H}}_{\mathcal{H} \vdash \mathsf{caller} \twoheadleftarrow \mathsf{this}.put}\}$
$\mathcal{H} = \mathcal{H} \vdash \mathsf{caller} \twoheadleftarrow \mathsf{this}.put;$
$\{I\}$

where $Q \triangleq I^{\mathcal{H},\mathsf{cnt}}_{\mathcal{H} \vdash \mathsf{caller} \twoheadleftarrow \mathsf{this}.put,\mathsf{cnt}+1}$

The proof outline leads to two verification conditions:

$$(1)\quad I \wedge \mathsf{cnt} = 0 \Rightarrow (Q_x^{\mathsf{cell}})^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{caller}\twoheadrightarrow\mathsf{this}.put(x)}$$
$$(2)\quad I \wedge \mathsf{cnt} \neq 0 \Rightarrow (Q^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{this}\rightarrow\mathsf{next}.put(x)\vdash \mathsf{this}\twoheadleftarrow\mathsf{next}.put})^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{caller}\twoheadrightarrow\mathsf{this}.put(x)}$$

F.1.1. Invariant Analysis

The two class invariants are proved by the following verification conditions:

$I_1 : \mathsf{cnt} = \#(\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H}))$
(1) :
$\mathsf{cnt} = \#(\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H})) \wedge \mathsf{cnt} = 0$
$\Rightarrow$
$\mathsf{cnt} + 1 = \#(x + \mathit{buf}(\mathsf{next}, \mathcal{H}))$
(2) :
$\mathsf{cnt} = \#(\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H})) \wedge \mathsf{cnt} \neq 0$
$\Rightarrow$
$\mathsf{cnt} + 1 = \#(\mathsf{cell} + (\mathit{in}(\mathsf{next}, h) \vdash x \ \textbf{after}\ \#\mathit{out}(\mathsf{next}, h)))$

$I_2 : \mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \mathit{in}(\mathsf{this}, \mathcal{H}) = \mathit{out}(\mathsf{this}, \mathcal{H}) \ \vdash (\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H}))$
(1) :
$I_1 \wedge (\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \mathit{in}(\mathsf{this}, \mathcal{H}) = \mathit{out}(\mathsf{this}, \mathcal{H}) \ \vdash (\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H}))) \wedge \mathsf{cnt} = 0$
$\Rightarrow$
$\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \mathit{in}(\mathsf{this}, \mathcal{H}) \vdash x = \mathit{out}(\mathsf{this}, \mathcal{H}) \ \vdash (x + \mathit{buf}(\mathsf{next}, \mathcal{H}))$
(2) :
$I_1 \wedge (\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \mathit{in}(\mathsf{this}, \mathcal{H}) = \mathit{out}(\mathsf{this}, \mathcal{H}) \ \vdash (\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H}))) \wedge \mathsf{cnt} \neq 0$
$\Rightarrow$
$(\mathit{out}(\mathsf{next}, \mathcal{H}) \leq \mathit{in}(\mathsf{next}, \mathcal{H}) \vdash x) \Rightarrow \mathit{in}(\mathsf{this}, \mathcal{H}) \vdash x = \mathit{out}(\mathsf{this}, \mathcal{H}) \ \vdash (\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H}) \vdash x)$

F.2. The get method

Proof outline:

$\{I\}$
$\{I^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{caller}\twoheadrightarrow\mathsf{this}.get}\}$
$\mathcal{H} = \mathcal{H} \vdash \mathsf{caller} \twoheadrightarrow \mathsf{this}.get;$
$\{I\}\ \textbf{var}\ \mathsf{Obj}\ r; \{I\}\ \textbf{await}(\mathsf{cnt} > 0); \{I \wedge \mathsf{cnt} > 0\}$
$\{((\mathsf{cell} = \mathsf{null} \Rightarrow \forall r'. Q^{r,\mathcal{H}}_{r',\mathcal{H}\vdash \mathsf{this}\rightarrow\mathsf{next}.get\vdash \mathsf{this}\twoheadleftarrow\mathsf{next}.get(r')}) \wedge (\mathsf{cell} \neq \mathsf{null} \Rightarrow (Q^{\mathsf{cell}}_{\mathsf{null}})^{r}_{\mathsf{cell}}))^{\mathsf{cnt}}_{\mathsf{cnt}-1}\}$
$\mathsf{cnt} := \mathsf{cnt} - 1;$
$\{(\mathsf{cell} = \mathsf{null} \Rightarrow \forall r'. Q^{r,\mathcal{H}}_{r',\mathcal{H}\vdash \mathsf{this}\rightarrow\mathsf{next}.get\vdash \mathsf{this}\twoheadleftarrow\mathsf{next}.get(r')}) \wedge (\mathsf{cell} \neq \mathsf{null} \Rightarrow (Q^{\mathsf{cell}}_{\mathsf{null}})^{r}_{\mathsf{cell}})\}$
$\textbf{if}(\mathsf{cell} = \mathsf{null})\ \textbf{then}\ \{\forall r'. Q^{r,\mathcal{H}}_{r',\mathcal{H}\vdash \mathsf{this}\rightarrow\mathsf{next}.get\vdash \mathsf{this}\twoheadleftarrow\mathsf{next}.get(r')}\}\ r := \mathsf{next}.get()$
$\textbf{else}\ \{(Q^{\mathsf{cell}}_{\mathsf{null}})^{r}_{\mathsf{cell}}\}\ r := \mathsf{cell}; \mathsf{cell} := \mathsf{null}\ \textbf{fi};$
$\{Q\}$
$\textbf{return}\ r;$
$\{I^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{caller}\twoheadleftarrow\mathsf{this}.get(r)}\}$
$\mathcal{H} = \mathcal{H} \vdash \mathsf{caller} \twoheadleftarrow \mathsf{this}.get(r);$
$\{I\}$

where $Q \triangleq I^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{caller}\leftarrow \mathsf{this}.get(r)}$

The proof outline leads to three verification conditions:

(1) $I \Rightarrow I^{\mathcal{H}}_{\mathcal{H}\vdash \mathsf{caller}\rightarrow \mathsf{this}.get}$
(2) $I \wedge \mathsf{cnt} > 0 \wedge \mathsf{cell} = \mathsf{null} \Rightarrow (\forall r'.Q^{r,\mathcal{H}}_{r',\mathcal{H}\vdash \mathsf{this}\rightarrow \mathsf{next}.get\vdash \mathsf{this}\leftarrow \mathsf{next}.get(r')})^{\mathsf{cnt}}_{\mathsf{cnt}-1}$
(3) $I \wedge \mathsf{cnt} > 0 \wedge \mathsf{cell} \neq \mathsf{null} \Rightarrow ((Q^{\mathsf{cell}}_{\mathsf{null}})^{r}_{\mathsf{cell}})^{\mathsf{cnt}}_{\mathsf{cnt}-1}$

F.2.1. Invariant Analysis

The two class invariants are proved by the following verification conditions:

$I_1 : \mathsf{cnt} = \#(\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H}))$
(1) :
$\mathsf{cnt} = \#(\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H}))$
$\Rightarrow$
$\mathsf{cnt} = \#(\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H}))$
(2) :
$\mathsf{cnt} = \#(\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H})) \wedge \mathsf{cnt} > 0 \wedge \mathsf{cell} = \mathsf{null}$
$\Rightarrow$
$\mathsf{cnt} - 1 = \#(\mathsf{cell} + (in(\mathsf{next}, h)\ \mathbf{after}\ \#(out(\mathsf{next}, h) \vdash x)))$
(3) :
$\mathsf{cnt} = \#(\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H})) \wedge \mathsf{cnt} > 0 \wedge \mathsf{cell} \neq \mathsf{null}$
$\Rightarrow$
$\mathsf{cnt} - 1 = \#(\mathsf{null} + \mathit{buf}(\mathsf{next}, \mathcal{H}))$

$I_2 : \mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow in(\mathsf{this}, \mathcal{H}) = out(\mathsf{this}, \mathcal{H})\ \vdash (\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H}))$
(1) :
$\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow in(\mathsf{this}, \mathcal{H}) = out(\mathsf{this}, \mathcal{H})\ \vdash (\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H}))$
$\Rightarrow$
$\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow in(\mathsf{this}, \mathcal{H}) = out(\mathsf{this}, \mathcal{H})\ \vdash (\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H}))$
(2) :
$I_1 \wedge (\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow in(\mathsf{this}, \mathcal{H}) = out(\mathsf{this}, \mathcal{H})\ \vdash (\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H}))) \wedge \mathsf{cnt} > 0 \wedge \mathsf{cell} = \mathsf{null}$
$\Rightarrow$
$(out(\mathsf{next}, \mathcal{H}) \vdash r \leq in(\mathsf{next}, \mathcal{H})) \Rightarrow in(\mathsf{this}, \mathcal{H}) = (out(\mathsf{this}, \mathcal{H}) \vdash r)\ \vdash (\mathsf{cell} + rest(\mathit{buf}(\mathsf{next}, \mathcal{H})))$
(3) :
$\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow in(\mathsf{this}, \mathcal{H}) = out(\mathsf{this}, \mathcal{H})\ \vdash (\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H})) \wedge \mathsf{cnt} > 0 \wedge \mathsf{cell} \neq \mathsf{null}$
$\Rightarrow$
$\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow in(\mathsf{this}, \mathcal{H}) = (out(\mathsf{this}, \mathcal{H}) \vdash \mathsf{cell})\ \vdash (\mathsf{null} + \mathit{buf}(\mathsf{next}, \mathcal{H}))$

F.3. Deriving the conditional FIFO property, $\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \mathit{fifo}(\mathsf{this}, \mathcal{H})$, from the assumption of class invariant (7)

$\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow in(\mathsf{this}, \mathcal{H}) = out(\mathsf{this}, \mathcal{H})\ \vdash (\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H}))$
$\Rightarrow$
$\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow\ \mathit{fifo}(\mathsf{this}, \mathcal{H})$

$in(\mathsf{this}, \mathcal{H}) = out(\mathsf{this}, \mathcal{H}) \ \vdash (\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H}))$
$\Rightarrow$
$out(\mathsf{this}, \mathcal{H}) \leq in(\mathsf{this}, \mathcal{H})$

F.4. Verification of the History Invariant

Here we consider the details for verifying the conditional FIFO property, named $Cond_{fifo}$, as a history invariant. The invariant is formulated as: $\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \mathit{fifo}(\mathsf{this}, \mathcal{H})$ A history invariant, $I_{\mathsf{this}:C(\overline{cp})}$, can be verified by showing that it is maintained by each local statement s affecting the history, i.e., one must prove $\{I_{\mathsf{this}:C(\overline{cp})}(\mathcal{H}) \wedge P\}\ s\ \{Q \Rightarrow I_{\mathsf{this}:C(\overline{cp})}(\mathcal{H})\}$ where P and Q are the pre- and postconditions of s used in the proof outline of the method.

F.4.1. The put method

In the above sections, we have proved that $Cond_{fifo}$ follows by implication from the class invariant, and that the class invariant holds at method termination. Thus, it remains to prove that $Cond_{fifo}$ holds after each history extension inside the method body, i.e., we must prove that the property holds after next.put(x):

$$\{Cond_{fifo} \wedge P\}\ \mathsf{next.put(x)}\ \{Q \Rightarrow Cond_{fifo}\}$$

where
$Q \triangleq (\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow in(\mathsf{this}, \mathcal{H}) = out(\mathsf{this}, \mathcal{H}) \ \vdash (\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H})))^{\mathcal{H},\mathsf{cnt}}_{\mathcal{H}\vdash\mathsf{caller}\leftarrow\mathsf{this}.put,\mathsf{cnt}+1}$
the actual assertion P is not needed here, but it is given in the proof outline above. The postcondition: $Q \Rightarrow Cond_{fifo}$ is proved as follows:

$(\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow in(\mathsf{this}, \mathcal{H}) = out(\mathsf{this}, \mathcal{H}) \ \vdash (\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H})))$
$\Rightarrow$
$(\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \ \mathit{fifo}(\mathsf{this}, \mathcal{H}))$

$in(\mathsf{this}, \mathcal{H}) = out(\mathsf{this}, \mathcal{H}) \ \vdash (\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H}))$
$\Rightarrow$
$out(\mathsf{this}, \mathcal{H}) \leq in(\mathsf{this}, \mathcal{H})$

F.4.2. The get method

In the above sections, we have proved that $Cond_{fifo}$ follows by implication from the class invariant, and that the class invariant holds at method termination. Thus, it remains to prove that $Cond_{fifo}$ holds after each history extension inside the method body, i.e., we must prove that the property holds after r:=next.get():

$$\{Cond_{fifo} \wedge P\}\ \mathsf{r{:=}next.get()}\ \{Q \Rightarrow Cond_{fifo}\}$$

where
$Q \triangleq (\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow in(\mathsf{this}, \mathcal{H}) = out(\mathsf{this}, \mathcal{H}) \ \vdash (\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H})))^{\mathcal{H}}_{\mathcal{H}\vdash\mathsf{caller}\leftarrow\mathsf{this}.get(r)}$
the actual assertion P is not needed here, but it is given in the proof outline above. The postcondition: $Q \Rightarrow Cond_{fifo}$ is proved as follows:

$(\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow in(\mathsf{this}, \mathcal{H}) = out(\mathsf{this}, \mathcal{H}) \vdash r \ \vdash (\mathsf{cell} + \mathit{buf}(\mathsf{next}, \mathcal{H})))$
$\Rightarrow$
$(\mathit{fifo}(\mathsf{next}, \mathcal{H}) \Rightarrow \ \mathit{fifo}(\mathsf{this}, \mathcal{H}))$

$in(\mathsf{this}, \mathcal{H}) = out(\mathsf{this}, \mathcal{H}) \vdash r \ \vdash\!\dashv (\mathsf{cell} + buf(\mathsf{next}, \mathcal{H}))$
$\Rightarrow$
$out(\mathsf{this}, \mathcal{H}) \leq in(\mathsf{this}, \mathcal{H})$

G. Proof of Lemma 2

For an assertion P, we let P' abbreviate $P^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'}$, and I abbreviates I_C.

G.1. Soundness

For the rules (NOTNULL), (RETURN), (SUSPEND), (AWAIT), (CALLASYNC), (CALLSYNC1), (CALLSYNC1-1), (AWAITCALL1), (AWAITCALL2), and (NEW)which are of the form $\{P\}\ s\ \{Q\}$, soundness follow directly from the *wlp*, i.e., for each rule the formula $P \Rightarrow wlp(s, Q)$ holds.

Rule (METHOD). Let $s' \triangleq \mathcal{H} := \mathcal{H} \vdash \mathsf{caller} \twoheadrightarrow \mathsf{this}.m(\overline{x}); s; \mathcal{H} := \mathcal{H} \vdash \mathsf{caller} \twoheadleftarrow \mathsf{this}.m(\mathsf{return})$
The rule may then be formulated as:

$$(\text{METHOD}) \quad \frac{\{S\}\ s'\ \{wf(\mathcal{H}) \Rightarrow R\}}{\{\forall \overline{y}\,.\,S\}\ m(\overline{x})\{\mathbf{var}\ \overline{y}; s\}\ \{\exists \overline{y}\,.\,R\}}$$

and we have the following *wlp*:

$$wlp(m(\overline{x})\{\mathbf{var}\ \overline{y}; s\}, Q) \triangleq wlp(\mathbf{var}\ \overline{y}; s', wf(\mathcal{H}) \Rightarrow Q)$$

where $\overline{y} \notin FV[Q]$.

For soundness, we need to ensure

$$\forall \overline{y}\,.\,S \Rightarrow wlp(\mathbf{var}\ \overline{y}; s', wf(\mathcal{H}) \Rightarrow \exists \overline{y}\,.\,R)$$

under the assumption $S \Rightarrow wlp(s', wf(\mathcal{H}) \Rightarrow R)$ (rule premise). Since $R \Rightarrow \exists \overline{y}\,.\,R$, we have $wlp(s', wf(\mathcal{H}) \Rightarrow R) \Rightarrow wlp(s', wf(\mathcal{H}) \Rightarrow \exists \overline{y}\,.\,R)$. Thus it suffices to prove $\forall \overline{y}\,.\,S \Rightarrow wlp(\mathbf{var}\ \overline{y}, S)$ which is trivial.

Rule (CALLSYNC2). Let i denote the event $\mathsf{this} \rightarrow o.m(\overline{e})$, and $c_{(o,v)}$ denote the event $\mathsf{this} \twoheadleftarrow o.m(v)$. We have the following proof obligation:

$$S^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i} \wedge o = \mathsf{this} \Rightarrow wlp(v := o.m(\overline{e}), \exists z\,.\,R^{v,\mathsf{return},\mathsf{caller},\mathcal{H}}_{z,v,\mathsf{this},pop(\mathcal{H})} \wedge \mathcal{H}\ \mathbf{ew}\ c_{(\mathsf{this},v)})$$

under the assumption $S \wedge \mathsf{caller} = \mathsf{this} \Rightarrow wlp(body, R \wedge \mathsf{caller} = \mathsf{this})$ (rule premise). Here, the *wlp* is defined by:

$$wlp(v := o.m(\overline{e}), Q) \triangleq o = \mathsf{this} \Rightarrow (wlp(body, Q^{v,\mathcal{H},\overline{y},v'}_{v',\mathcal{H}\vdash c_{(\mathbf{this},v')},\overline{y}',\mathsf{return}}))^{\overline{x},\mathsf{caller},\overline{y}',\mathcal{H}}_{\overline{e},\mathsf{this},\overline{y},\mathcal{H}\vdash i}$$

which means that the above proof obligation can be written as:

$$S^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i} \Rightarrow (wlp(body, (\exists z\,.\,R^{v,\mathsf{return},\mathsf{caller},\mathcal{H}}_{z,v,\mathsf{this},pop(\mathcal{H})} \wedge \mathcal{H}\ \mathbf{ew}\ c_{(\mathsf{this},v)})^{v,\mathcal{H},\overline{y},v'}_{v',\mathcal{H}\vdash c_{(\mathbf{this},v')},\overline{y}',\mathsf{return}}))^{\overline{x},\mathsf{caller},\overline{y}',\mathcal{H}}_{\overline{e},\mathsf{this},\overline{y},\mathcal{H}\vdash i}$$

Remark that S and R are assertions over the state of the called method, i.e., $\overline{y}$ and $\overline{y}'$ does not occur in these assertions. Since $\overline{y} \notin FV[R]$, we have the following implication:

$$R \wedge \mathsf{caller} = \mathsf{this} \Rightarrow (\exists z\,.\,R^{v,\mathsf{return},\mathsf{caller},\mathcal{H}}_{z,v,\mathsf{this},pop(\mathcal{H})} \wedge \mathcal{H}\ \mathbf{ew}\ c_{(\mathsf{this},v)})^{v,\mathcal{H},\overline{y},v'}_{v',\mathcal{H}\vdash c_{(\mathbf{this},v')},\overline{y}',\mathsf{return}}$$

Since *wlp* is monotonic, it therefore suffices to prove:

$$S^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i} \Rightarrow (wlp(body, R \wedge \mathsf{caller} = \mathsf{this}))^{\overline{x},\mathsf{caller},\overline{y}',\mathcal{H}}_{\overline{e},\mathsf{this},\overline{y},\mathcal{H}\vdash i}$$

which follows by rule premise and the trivial implication:

$$S^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i} \Rightarrow (S \wedge \mathsf{caller} = \mathsf{this})^{\overline{x},\mathsf{caller},\overline{y}',\mathcal{H}}_{\overline{e},\mathsf{this},\overline{y},\mathcal{H}\vdash i}$$

Rule (CALLSYNC2-1). This proof follows the same pattern as for (CALLSYNC2). As above we have $\overline{y}$ and $\overline{y}'$ not in $FV[S]$ and $FV[R]$, and we therefore ignore these variables below. Here we have the proof obligation:

$$S^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i} \wedge o = \mathsf{this} \Rightarrow wlp(o.m(\overline{e}), \exists v' \,.\, R^{\mathsf{caller},\mathsf{return},\mathcal{H}}_{\mathsf{this},v',pop(\mathcal{H})} \wedge \mathcal{H}\ \mathbf{ew}\ c_{(\mathsf{this},v')})$$

under the assumption $S \wedge \mathsf{caller} = \mathsf{this} \Rightarrow wlp(body, R \wedge \mathsf{caller} = \mathsf{this})$ (rule premise). Here, the *wlp* is defined by:

$$wlp(o.m(\overline{e}), Q) \triangleq o = \mathsf{this} \Rightarrow (wlp(body, Q^{\mathcal{H},v'}_{\mathcal{H}\vdash c_{(\mathbf{this},v')},\mathsf{return}}))^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i}$$

Since

$$R \wedge \mathsf{caller} = \mathsf{this} \Rightarrow (\exists v' \,.\, R^{\mathsf{caller},\mathsf{return},\mathcal{H}}_{\mathsf{this},v',pop(\mathcal{H})} \wedge \mathcal{H}\ \mathbf{ew}\ c_{(\mathsf{this},v')})^{\mathcal{H},v'}_{\mathcal{H}\vdash c_{(\mathbf{this},v')},\mathsf{return}}$$

The proof obligation reduces to

$$S^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i} \Rightarrow (wlp(body, R \wedge \mathsf{caller} = \mathsf{this}))^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i}$$

which is satisfied by the same argument as above.

G.2. Completeness

Statement **suspend**.

1.	$\{I\}\ \mathbf{suspend}\ \{I\}$	(SUSPEND)
2.	$\{\mathcal{H} = h_0\}\ \mathbf{suspend}\ \{h_0 \leq \mathcal{H}\}$	(HIS)
3.	$\{wf(\mathcal{H})\}\ s\ \{wf(\mathcal{H})\}$	(WF)
4.	$\{I \wedge \mathcal{H} = h_0 \wedge wf(\mathcal{H})\}\ \mathbf{suspend}\ \{I \wedge h_0 \leq \mathcal{H} \wedge wf(\mathcal{H})\}$	$(1, 2, 3, (\text{CONJ}))$
5.	$\{\forall \overline{w}', \mathcal{H}' \,.\, (\forall h_0 \,.\, I \wedge \mathcal{H} = h_0 \wedge wf(\mathcal{H}) \Rightarrow I' \wedge h_0 \leq \mathcal{H}' \wedge wf(\mathcal{H}')) \Rightarrow Q'\}$ $\mathbf{suspend}\ \{Q\}$	$(4, (\text{ADAP}))$
6.	$\{\forall \overline{w}', \mathcal{H}' \,.\, (I \wedge wf(\mathcal{H}) \Rightarrow I' \wedge \mathcal{H} \leq \mathcal{H}' \wedge wf(\mathcal{H}')) \Rightarrow Q'\}\ \mathbf{suspend}\ \{Q\}$	$(5, math)$
7.	$\{I \wedge wf(\mathcal{H}) \wedge \forall \overline{w}', \mathcal{H}' \,.\, (I' \wedge \mathcal{H} \leq \mathcal{H}' \wedge wf(\mathcal{H}')) \Rightarrow Q'\}\ \mathbf{suspend}\ \{Q\}$	$(6, (\text{CONS}))$
8.	$\{wlp(\mathbf{suspend}, Q)\}\ \mathbf{suspend}\ \{Q\}$	$(7, def)$

Statement **await** b.

1.	$\{I\}\ \mathbf{await}\ b\ \{I \wedge b\}$	(AWAIT)
2.	$\{\mathcal{H} = h_0\}\ \mathbf{await}\ b\ \{h_0 \leq \mathcal{H}\}$	(HIS)
3.	$\{wf(\mathcal{H})\}\ s\ \{wf(\mathcal{H})\}$	(WF)
4.	$\{I \wedge \mathcal{H} = h_0 \wedge wf(\mathcal{H})\}\ \mathbf{await}\ b\ \{I \wedge b \wedge h_0 \leq \mathcal{H} \wedge wf(\mathcal{H})\}$	$(1, 2, 3, (\text{CONJ}))$
5.	$\{\forall \overline{w}', \mathcal{H}' \,.\, (\forall h_0 \,.\, I \wedge \mathcal{H} = h_0 \wedge wf(\mathcal{H}) \Rightarrow I' \wedge b' \wedge h_0 \leq \mathcal{H}' \wedge wf(\mathcal{H}')) \Rightarrow Q'\}$ $\mathbf{await}\ b\ \{Q\}$	$(4, (\text{ADAP}))$
6.	$\{\forall \overline{w}', \mathcal{H}' \,.\, (I \wedge wf(\mathcal{H}) \Rightarrow I' \wedge b' \wedge \mathcal{H} \leq \mathcal{H}' \wedge wf(\mathcal{H}')) \Rightarrow Q'\}$ $\mathbf{await}\ b\ \{Q\}$	$(5, math)$
7.	$\{I \wedge wf(\mathcal{H}) \wedge \forall \overline{w}', \mathcal{H}' \,.\, (I' \wedge b' \wedge \mathcal{H} \leq \mathcal{H}' \wedge wf(\mathcal{H}')) \Rightarrow Q'\}$ $\mathbf{await}\ b\ \{Q\}$	$(6, (\text{CONS}))$
8.	$\{wlp(\mathbf{await}\ b, Q)\}\ \mathbf{await}\ b\ \{Q\}$	$(7, def)$

Statement **await** $o.m(\overline{e})$.

1. $\{h_0 = \mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e}) \wedge I^{\mathcal{H}}_{h_0} \wedge o = o_0\}$
   await $o.m(\overline{e})$
   $\{h_0 \leq \mathcal{H} \wedge I^{\mathcal{H}}_{pop(\mathcal{H})} \wedge \exists v\,.\,\mathcal{H}$ **ew** $\mathsf{this} \twoheadleftarrow o_0.m(v)\}$ (awaitCall2)
2. $\{wf(\mathcal{H})\}\ s\ \{wf(\mathcal{H})\}$ (wf)
3. $\{h_0 = \mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e}) \wedge I^{\mathcal{H}}_{h_0} \wedge o = o_0 \wedge wf(\mathcal{H})\}$
   await $o.m(\overline{e})$
   $\{h_0 \leq \mathcal{H} \wedge I^{\mathcal{H}}_{pop(\mathcal{H})} \wedge \exists v\,.\,\mathcal{H}$ **ew** $\mathsf{this} \twoheadleftarrow o_0.m(v) \wedge wf(\mathcal{H})\}$ $(1, 2, (\text{conj}))$
4. $\{\forall \overline{w}', \mathcal{H}'\,.\,(\forall h_0, o_0\,.\,(h_0 = \mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e}) \wedge I^{\mathcal{H}}_{h_0} \wedge o = o_0 \wedge wf(\mathcal{H}))$
   $\Rightarrow h_0 \leq \mathcal{H}' \wedge (I^{\mathcal{H}}_{pop(\mathcal{H}')})^{\overline{w}}_{\overline{w}'} \wedge \exists v\,.\,\mathcal{H}'$ **ew** $\mathsf{this} \twoheadleftarrow o_0.m(v) \wedge wf(\mathcal{H}'))$
   $\Rightarrow Q^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'}\}$**await** $o.m(\overline{e})\{Q\}$ $(3, (\text{adap}))$
5. $\{\forall \overline{w}', \mathcal{H}'\,.\,(I^{\mathcal{H}}_{\mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e})} \wedge wf(\mathcal{H}) \Rightarrow$
   $\mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e}) \leq \mathcal{H}' \wedge (I^{\mathcal{H}}_{pop(\mathcal{H}')})^{\overline{w}}_{\overline{w}'} \wedge \exists v\,.\,\mathcal{H}'$ **ew** $\mathsf{this} \twoheadleftarrow o.m(v)$
   $\wedge wf(\mathcal{H}')) \Rightarrow Q^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'}\}$**await** $o.m(\overline{e})\{Q\}$ $(4, math)$
6. $\{I^{\mathcal{H}}_{\mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e})} \wedge wf(\mathcal{H}) \wedge$
   $\forall \overline{w}', \mathcal{H}'\,.\,(\mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e}) \leq \mathcal{H}' \wedge$
   $(I^{\mathcal{H}}_{pop(\mathcal{H}')})^{\overline{w}}_{\overline{w}'} \wedge \exists v\,.\,\mathcal{H}'$ **ew** $\mathsf{this} \twoheadleftarrow o.m(v) \wedge wf(\mathcal{H}')) \Rightarrow Q^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'}\}$
   await $o.m(\overline{e})\{Q\}$ $(5, (\text{cons}))$
7. $\{I^{\mathcal{H}}_{\mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e})} \wedge wf(\mathcal{H}) \wedge$
   $\forall v', \overline{w}', \mathcal{H}'\,.\,(\mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e}) \leq \mathcal{H}' \wedge$
   $I^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'} \wedge wf(\mathcal{H}' \vdash \mathsf{this} \twoheadleftarrow o.m(v'))) \Rightarrow Q^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}' \vdash \mathsf{this} \twoheadleftarrow o.m(v')}\}$
   await $o.m(\overline{e})\{Q\}$ $(6, (\text{cons}))$
8. $\{o = \mathsf{null}\}$ **await** $o.m(\overline{e})\ \{false\}$ (notNull)
9. $\{o \neq \mathsf{null} \Rightarrow (I_C{}^{\mathcal{H}}_{\mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e})} \wedge wf(\mathcal{H}) \wedge$
   $\forall v', \overline{w}', \mathcal{H}'\,.\,(\mathcal{H} \vdash \mathsf{this} \to o.m(\overline{e}) \leq \mathcal{H}' \wedge$
   $I^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'} \wedge wf(\mathcal{H}' \vdash \mathsf{this} \twoheadleftarrow o.m(v'))) \Rightarrow Q^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}' \vdash \mathsf{this} \twoheadleftarrow o.m(v')})\}$
   await $o.m(\overline{e})\{Q\}$ $(7, 8, (\text{disj}))$
10. $\{wlp(\textbf{await}\ o.m(\overline{e}), Q)\}$ **await** $o.m(\overline{e})\ \{Q\}$ $(9, def)$

Statement $\mathbf{await}\ v := o.m(\overline{e})$.

1. $\{h_0 = \mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e}) \wedge I^{\mathcal{H}}_{h_0} \wedge o = o_0\}$
   $\mathbf{await}\ v := o.m(\overline{e})$
   $\{h_0 \leq \mathcal{H} \wedge \mathcal{H}\ \mathbf{ew}\ \mathsf{this} \twoheadleftarrow o_0.m(v) \wedge \exists v\,.\, I^{\mathcal{H}}_{pop(\mathcal{H})}\}$ (awaitCall1)
2. $\{wf(\mathcal{H})\}\ s\ \{wf(\mathcal{H})\}$ (wf)
3. $\{h_0 = \mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e}) \wedge I^{\mathcal{H}}_{h_0} \wedge o = o_0 \wedge wf(\mathcal{H})\}$
   $\mathbf{await}\ v := o.m(\overline{e})$
   $\{h_0 \leq \mathcal{H} \wedge \mathcal{H}\ \mathbf{ew}\ \mathsf{this} \twoheadleftarrow o_0.m(v) \wedge \exists v\,.\, I^{\mathcal{H}}_{pop(\mathcal{H})} \wedge wf(\mathcal{H})\}$ $(1, 2, (\textsc{conj}))$
4. $\{\forall v', \overline{w}', \mathcal{H}'\,.\,(\forall h_0, o_0\,.\,(h_0 = \mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e}) \wedge I^{\mathcal{H}}_{h_0} \wedge o = o_0 \wedge wf(\mathcal{H}))$
   $\Rightarrow h_0 \leq \mathcal{H}' \wedge \mathcal{H}'\ \mathbf{ew}\ \mathsf{this} \twoheadleftarrow o_0.m(v') \wedge (\exists v\,.\, I^{\mathcal{H}}_{pop(\mathcal{H}')})^{\overline{w}}_{\overline{w}'} \wedge wf(\mathcal{H}'))$
   $\Rightarrow Q^{v,\overline{w},\mathcal{H}}_{v',\overline{w}',\mathcal{H}'}\}\mathbf{await}\ v := o.m(\overline{e})\{Q\}$ $(3, (\textsc{adap}))$
5. $\{\forall v', \overline{w}', \mathcal{H}'\,.\,(I^{\mathcal{H}}_{\mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e})} \wedge wf(\mathcal{H})$
   $\Rightarrow \mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e}) \leq \mathcal{H}' \wedge \mathcal{H}'\ \mathbf{ew}\ \mathsf{this} \twoheadleftarrow o.m(v') \wedge$
   $(\exists v\,.\, I^{\mathcal{H}}_{pop(\mathcal{H}')})^{\overline{w}}_{\overline{w}'} \wedge wf(\mathcal{H}')) \Rightarrow Q^{v,\overline{w},\mathcal{H}}_{v',\overline{w}',\mathcal{H}'}\}\mathbf{await}\ v := o.m(\overline{e})\{Q\}$ $(4, \mathit{math})$
6. $\{I^{\mathcal{H}}_{\mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e})} \wedge wf(\mathcal{H}) \wedge$
   $\forall v', \overline{w}', \mathcal{H}'\,.\,(\mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e}) \leq \mathcal{H}' \wedge \mathcal{H}'\ \mathbf{ew}\ \mathsf{this} \twoheadleftarrow o.m(v') \wedge$
   $(\exists v\,.\, I^{\mathcal{H}}_{pop(\mathcal{H}')})^{\overline{w}}_{\overline{w}'} \wedge wf(\mathcal{H}')) \Rightarrow Q^{v,\overline{w},\mathcal{H}}_{v',\overline{w}',\mathcal{H}'}\}\mathbf{await}\ v := o.m(\overline{e})\{Q\}$ $(5, (\textsc{cons}))$
7. $\{I^{\mathcal{H}}_{\mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e})} \wedge wf(\mathcal{H}) \wedge$
   $\forall v', \overline{w}', \mathcal{H}'\,.\,(\mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e}) \leq \mathcal{H}' \wedge$
   $I^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'} \wedge wf(\mathcal{H}' \vdash \mathsf{this} \twoheadleftarrow o.m(v'))) \Rightarrow (Q^{v}_{v'})^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}' \vdash \mathsf{this} \twoheadleftarrow o.m(v')}\}$
   $\mathbf{await}\ v := o.m(\overline{e})\{Q\}$ $(6, (\textsc{cons}))$
8. $\{o = \mathsf{null}\}\ \mathbf{await}\ v := o.m(\overline{e})\ \{\mathit{false}\}$ (notNull)
9. $\{o \neq \mathsf{null} \Rightarrow (I^{\mathcal{H}}_{\mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e})} \wedge wf(\mathcal{H}) \wedge$
   $\forall v', \overline{w}', \mathcal{H}'\,.\,(\mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e}) \leq \mathcal{H}' \wedge$
   $I^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'} \wedge wf(\mathcal{H}' \vdash \mathsf{this} \twoheadleftarrow o.m(v'))) \Rightarrow (Q^{v}_{v'})^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}' \vdash \mathsf{this} \twoheadleftarrow o.m(v')})\}$
   $\mathbf{await}\ v := o.m(\overline{e})\{Q\}$ $(7, 8, (\textsc{disj}))$
10. $\{wlp(\mathbf{await}\ v := o.m(\overline{e}), Q)\}\ \mathbf{await}\ v := o.m(\overline{e})\ \{Q\}$ $(9, \mathit{def})$

Statement $v := o.m(\overline{e})$.
Let i abbreviate the event $\mathsf{this} \rightarrow o.m(\overline{e})$ (which equals $\mathsf{this} \rightarrow \mathsf{this}.m(\overline{e})$ under the assumption $\mathsf{this} = o$), and let $c_{(o,v)}$ abbreviate the event $\mathsf{this} \twoheadleftarrow o.m(v)$. Given an arbitrary postcondition Q to the statement $v := o.m(\overline{e})$, i.e., $FV[Q] \subseteq \{\overline{w}, \mathcal{H}, \overline{cp}, \overline{y}, \overline{l}\}$ where $\overline{y}$ are the method-local variables of the caller (including the formal parameters and caller) and $\overline{l}$ is a list of logical variables. Observe that $\mathsf{return} \notin FV[Q]$ since Q appears inside the body of the calling method. By definition, the assertion $wlp(v := o.m(\overline{e}), Q)$ may then be written as:

$$o \neq \mathsf{null} \Rightarrow \forall v'\,.\,\mathbf{if}\ o = \mathsf{this}\ \mathbf{then}\ (wlp(v' := m'(\overline{e}, \mathsf{this}), Q^{v,\mathcal{H}}_{v',\mathcal{H} \vdash c_{(\mathbf{this},v')}}))^{\mathcal{H}}_{\mathcal{H} \vdash i}$$
$$\mathbf{else}\ Q^{v,\mathcal{H}}_{v',\mathcal{H} \vdash i \vdash c_{(o,v')}}$$

which by definition of $wlp(v' := m'(\overline{e}, \mathsf{this}), Q)$ can be rewritten as:

$$o \neq \mathsf{null} \Rightarrow \forall v'\,.\,\mathbf{if}\ o = \mathsf{this}\ \mathbf{then}$$
$$((wlp(m'(\overline{x}, \mathsf{caller})\ body, (Q^{v,\mathcal{H}}_{v',\mathcal{H} \vdash c_{(\mathbf{this},v')}})^{\overline{y},v'}_{\overline{y}',\mathsf{return}}))^{\overline{x},\mathsf{caller},\overline{y}'}_{\overline{e},\mathsf{this},\overline{y}})^{\mathcal{H}}_{\mathcal{H} \vdash i}$$
$$\mathbf{else}\ Q^{v,\mathcal{H}}_{v',\mathcal{H} \vdash i \vdash c_{(o,v')}}$$

By simplifying the substitutions, this formula may be written as:

$$o \neq \mathsf{null} \Rightarrow \forall v' \,.\, \mathtt{if}\ o = \mathsf{this}\ \mathtt{then} \\ (wlp(m(\overline{x})body, Q^{v,\mathcal{H},\overline{y}}_{\mathsf{return},\mathcal{H}\vdash c_{(\mathbf{this},\mathbf{return})},\overline{y}'}))^{\overline{x},\mathsf{caller},\mathcal{H},\overline{y}'}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i,\overline{y}} \\ \mathtt{else}\ Q^{v,\mathcal{H}}_{v',\mathcal{H}\vdash i\vdash c_{(o,v')}}$$

In the proof below, we let P denote the following assertion:

$$(wlp(m(\overline{x})\ body, Q^{v,\mathcal{H},\overline{y}}_{\mathsf{return},\mathcal{H}\vdash c_{(\mathbf{this},\mathbf{return})},\overline{y}'}))^{\overline{x},\mathsf{caller},\mathcal{H},\overline{y}'}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i,\overline{y}}$$

which means that the wlp can be written as:

$$wlp(v := o.m(\overline{e}), Q)\ \triangleq\ o \neq \mathsf{null} \Rightarrow \forall v' \,.\, \mathtt{if}\ o = \mathsf{this}\ \mathtt{then}\ P\ \mathtt{else}\ Q^{v,\mathcal{H}}_{v',\mathcal{H}\vdash i\vdash c_{(o,v')}}$$

In the proof below, we let S denote the formula $wlp(m(\overline{x})\ body, Q^{v,\mathcal{H},\overline{y}}_{\mathsf{return},\mathcal{H}\vdash c_{(\mathbf{this},\mathbf{return})},\overline{y}'})$, and R denote the formula $Q^{v,\mathcal{H},\overline{y}}_{\mathsf{return},\mathcal{H}\vdash c_{(\mathbf{this},\mathbf{return})}\overline{y}'}$.

1.	$\{S\}\ m(\overline{x})body\ \{R\}$	(*premise*)
2.	$\{S^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i} \wedge o = \mathsf{this}\}$ $v := o.m(\overline{e})\{\exists z \,.\, R^{v,\mathsf{return},\mathsf{caller},\mathcal{H}}_{z,v,\mathsf{this},pop(\mathcal{H})} \wedge \mathcal{H}\ \mathtt{ew}\ c_{(\mathsf{this},v)}\}$	(1, (CALLSYNC2))
3.	$\{\forall \overline{w}', \mathcal{H}', v' \,.\, (\forall \overline{y}', \overline{l} \,.\, S^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i} \wedge o = \mathsf{this} \Rightarrow$ $(\exists z \,.\, R^{v,\mathsf{return},\mathsf{caller},\mathcal{H}}_{z,v,\mathsf{this},pop(\mathcal{H})} \wedge \mathcal{H}\ \mathtt{ew}\ c_{(\mathsf{this},v)})^{\overline{w},\mathcal{H},v}_{\overline{w}',\mathcal{H}',v'}) \Rightarrow Q^{\overline{w},\mathcal{H},v}_{\overline{w}',\mathcal{H}',v'}\}$ $v := o.m(\overline{e})\{Q\}$	(2, (ADAP))
4.	$\{S^{\overline{x},\mathsf{caller},\mathcal{H},\overline{y}'}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i,\overline{y}} \wedge o = \mathsf{this}\}\ v := o.m(\overline{e})\ \{Q\}$	(3, (CONS))
5.	$\{P \wedge o = \mathsf{this}\}\ v := o.m(\overline{e})\ \{Q\}$	(4, *def*)
6.	$\{\forall v'.Q^{v,\mathcal{H}}_{v',\mathcal{H}\ \vdash i\vdash c_{(o,v')}} \wedge o \neq \mathsf{this}\}\ v := o.m(\overline{e})\ \{Q\}$	(CALLSYNC1)
7.	$\{(P \wedge o = \mathsf{this}) \vee (\forall v'.Q^{v,\mathcal{H}}_{v',\mathcal{H}\ \vdash i\vdash c_{(o,v')}} \wedge o \neq \mathsf{this})\}\ v := o.m(\overline{e})\ \{Q\}$	(5, 6, (DISJ))
8.	$\{\forall v' \,.\, \mathtt{if}\ o = \mathsf{this}\ \mathtt{then}\ P\ \mathtt{else}\ Q^{v,\mathcal{H}}_{v',\mathcal{H}\ \vdash i\vdash c_{(o,v')}}\}\ v := o.m(\overline{e})\ \{Q\}$	(7, *math*)
9.	$\{o = \mathsf{null}\}\ v := o.m(\overline{e})\ \{false\}$	(NOTNULL)
10.	$\{o \neq \mathsf{null} \Rightarrow \forall v' \,.\, \mathtt{if}\ o = \mathsf{this}\ \mathtt{then}\ P\ \mathtt{else}\ Q^{v,\mathcal{H}}_{v',\mathcal{H}\ \vdash i\vdash c_{(o,v')}}\}\ v := o.m(\overline{e})\ \{Q\}$	(8, 9, (DISJ))
11.	$\{wlp(v := o.m(\overline{e}), Q)\}\ v := o.m(\overline{e})\ \{Q\}$	(10, *def*)

Statement $o.m(\overline{e})$.
Following the same outline as for statement $v := o.m(\overline{e})$ above, $wlp(o.m(\overline{e}), Q)$ can be written as:

$$o \neq \mathsf{null} \Rightarrow \forall v' \,.\, \mathtt{if}\ o = \mathsf{this}\ \mathtt{then} \\ (wlp(m(\overline{x})body, Q^{\mathcal{H},\overline{y}}_{\mathcal{H}\vdash c_{(\mathbf{this},\mathbf{return})},\overline{y}'}))^{\overline{x},\mathsf{caller},\mathcal{H},\overline{y}'}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i,\overline{y}} \\ \mathtt{else}\ Q^{\mathcal{H}}_{\mathcal{H}\vdash i\vdash c_{(o,v')}}$$

Below we let P denote the assertion:

$$(wlp(m(\overline{x})body, Q^{\mathcal{H},\overline{y}}_{\mathcal{H}\vdash c_{(\mathbf{this},\mathbf{return})},\overline{y}'}))^{\overline{x},\mathsf{caller},\mathcal{H},\overline{y}'}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i,\overline{y}}$$

Assertion S denotes $wlp(m(\overline{x})body, Q^{\mathcal{H},\overline{y}}_{\mathcal{H}\vdash c_{(\mathbf{this},\mathbf{return})},\overline{y}'})$, and R denotes $Q^{\mathcal{H},\overline{y}}_{\mathcal{H}\vdash c_{(\mathbf{this},\mathbf{return})},\overline{y}'}$. The proof then corresponds to the one above:

1. $\{S\}\ m(\overline{x})body\ \{R\}$ (*premise*)
2. $\{S^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i} \wedge o = \mathsf{this}\}$
   $o.m(\overline{e})\{\exists v' \,.\, R^{\mathsf{return},\mathsf{caller},\mathcal{H}}_{v',\mathsf{this},pop(\mathcal{H})} \wedge \mathcal{H}\ \mathbf{ew}\ c_{(\mathsf{this},v')}\}$ (1, (CALLSYNC2-1))
3. $\{\forall \overline{w}', \mathcal{H}' \,.\, (\forall \overline{y}', \overline{l} \,.\, S^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i} \wedge o = \mathsf{this} \Rightarrow$
   $(\exists v' \,.\, R^{\mathsf{return},\mathsf{caller},\mathcal{H}}_{v',\mathsf{this},pop(\mathcal{H})} \wedge \mathcal{H}\ \mathbf{ew}\ c_{(\mathsf{this},v')})^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'}) \Rightarrow Q^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'}\}$
   $o.m(\overline{e})\{Q\}$ (2, (ADAP))
4. $\{S^{\overline{x},\mathsf{caller},\mathcal{H},\overline{y}'}_{\overline{e},\mathsf{this},\mathcal{H}\vdash i,\overline{y}} \wedge o = \mathsf{this}\}\ o.m(\overline{e})\ \{Q\}$ (3, (CONS))
5. $\{P \wedge o = \mathsf{this}\}\ o.m(\overline{e})\ \{Q\}$ (4, *def*)
6. $\{\forall v'.Q^{\mathcal{H}}_{\mathcal{H}\ \vdash i\vdash c_{(o,v')}} \wedge o \neq \mathsf{this}\}\ o.m(\overline{e})\ \{Q\}$ (CALLSYNC1-1)
7. $\{(P \wedge o = \mathsf{this}) \vee (\forall v'.Q^{\mathcal{H}}_{\mathcal{H}\ \vdash i\vdash c_{(o,v')}} \wedge o \neq \mathsf{this})\}\ o.m(\overline{e})\ \{Q\}$ (5, 6, (DISJ))
8. $\{\forall v' \,.\, \mathbf{if}\ o = \mathsf{this}\ \mathbf{then}\ P\ \mathbf{else}\ Q^{\mathcal{H}}_{\mathcal{H}\ \vdash i\vdash c_{(o,v')}}\}\ o.m(\overline{e})\ \{Q\}$ (7, *math*)
9. $\{o = \mathsf{null}\}\ o.m(\overline{e})\ \{false\}$ (NOTNULL)
10. $\{o \neq \mathsf{null} \Rightarrow \forall v' \,.\, \mathbf{if}\ o = \mathsf{this}\ \mathbf{then}\ P\ \mathbf{else}\ Q^{\mathcal{H}}_{\mathcal{H}\ \vdash i\vdash c_{(o,v')}}\}\ o.m(\overline{e})\ \{Q\}$ (8, 9, (DISJ))
11. $\{wlp(o.m(\overline{e}), Q)\}\ o.m(\overline{e})\ \{Q\}$ (10, *def*)

Rule (METHOD).

The side condition $\overline{y} \notin FV[Q]$ means that $\exists \overline{y} \,.\, Q = Q$. By the rule premise, we have $S = wlp(s', wf(\mathcal{H}) \Rightarrow Q)$, where s' is as for the soundness proof of (METHOD) above. The remaining verification condition $wlp(\mathbf{var}\ \overline{y}, S) \Rightarrow \forall \overline{y} \,.\, S$ is trivial.