UNIVERSITY OF OSLO
Department of Informatics

Observable Behavior of Distributed Systems: Component Reasoning for Concurrent Objects

Research Report 401

Crystal Chang Din

Johan Dovland

Einar Broch Johnsen

Olaf Owe

ISBN 82-7368-362-1
ISSN 0806-3036

November 2010

Contents

1 Introduction **2**

2 Syntax for the *ABS* Language **3**

3 Observable Behavior **5**
3.1 Invariant Reasoning 8

4 Analysis of *ABS* Programs **9**
4.1 Semantic Definition by a Syntactic Encoding 9
4.2 Weakest Liberal Preconditions 11
4.3 Hoare Logic 12
4.4 Object Composition 14

5 Reader/Writer Example **15**
5.1 Implementation 15
5.2 Specification and Verification 16

6 Related and Future Work **18**

7 Conclusion **18**

A Complete Code of Fairness Reader/Writer **20**

B Definition of Writers **21**

C Definition of Writing **21**

D Verification Details for RWController **21**
D.1 openR 22
D.2 openW 22
D.3 closeR 22
D.4 closeW 22
D.5 read 22
D.6 write 23

Observable Behavior of Distributed Systems: Component Reasoning for Concurrent Objects

Crystal Chang Din, Johan Dovland, Einar Broch Johnsen, 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

We present a partial correctness proof system for ABS, an imperative, concurrent and object-oriented language which provides asynchronous communication model that is suitable for loosely coupled objects in the distributed setting. The proof system is derived from a standard sequential language by means of a syntactic encoding and applies Hoare rules. The execution of a distributed system is represented by its communication history, which can be predicated by history invariant. Modularity is achieved by establishing history invariants independently for each object and composed at need. This results in behavioral specification of distributed system in an open environment. As a case study we model and analyze the reader-writer example in the framework we developed.

1 Introduction

Distributed systems play an essential role in the modern world, and the quality of such systems is often crucial. Quality assurance is however non-trivial since distributed systems depend on unpredictable factors such as different processing speeds of independent components and network transmission speeds. Such systems are therefore hard to test under the different conditions, which calls for precise modeling and analysis frameworks with suitable tool support. In particular, there is a need for compositional verification systems such that each component can be analyzed independently from its surrounding components.

Object orientation is the leading framework for concurrent and distributed systems, recommended by the RM-ODP [12]. Many distributed systems are today programmed in object-oriented, imperative languages like Java and C++. Programs written in these languages are in general difficult to analyze due to composition and alias problems, and the complexity of their concurrency, communication, and synchronization mechanisms. Rather than performing analysis at the level of the Java and C++ code, it may be easier to consider a model of the program at a suitable level. In this paper, we consider *ABS*, a high-level object-oriented language, which is inspired by the *Creol* language [13]. *ABS* supports concurrent objects with an asynchronous communication model that is suitable for loosely coupled objects in a distributed setting. The language is imperative, and avoids some of the mentioned difficulties of analyzing distributed systems.

In ABS, there is no access to the internal state variables of other objects, and a concurrent object has its own execution thread. Object communication is only by method calls, allowing asynchronous communication in order to avoid undesirable waiting in the distributed setting, where one object need not depend on the responsiveness of other objects. Internally in an object, there is at most one process executing at a time, and

intra-object synchronization is programmed by *processor release points*. These mechanisms provide high-level constructs for process control, and in particular allow an object to change dynamically between active and reactive behavior. Concurrency problems inside the object are avoided since each region from a release point to another release point is performed as a critical region. The operational semantics of ABS has been worked out in [10].

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 [5,11]. At any point in time the communication history abstractly captures the system state [6,7]. Therefore a system may be specified by the finite initial segments of its communication histories. The *local* history of an object reflects 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 [2].

In this paper, we develop a partial correctness proof system for the *ABS* language. A class is specified by a *class invariant* over the class attributes and the local communication history. The proof system is derived from a standard sequential language by means of a syntactic encoding, using nondeterministic assignments to the history for reflecting activity of other processes . The reasoning inside a class is comparable to reasoning about a sequential while-language, and it amounts to proving that the class invariant is maintained from one release point to another. Since remote access is restricted to method calls, the classical Hoare rule for assignment is sound.

For each object, a history invariant can be derived from the class invariant by hiding the local state of the object, allowing objects to be specified independently of their internal implementation details. Such specifications describe the observable communication between each object and the environment. 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.

Report 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. Section 5 provides an 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 cp of class parameters, defines 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 *init*. There is read-only access to the class parameters 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 only. 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 ignored in this paper (except in the *ABS* examples).

P	::=	$Dd^*\ F^*\ In^*\ Cl^*\ [s]^?$	program
In	::=	**interface** I [**extends** I^+]$^?${S^*}	interface declaration
Cl	::=	**class** C([$T\ x$]*) [**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	::=	$v \mid$ **null** $\mid$ **this** $\mid p \mid t \mid f(e^*)$	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

Dd	::=	**data** D {[$Co(T^*)$]*}	data type declaration
F	::=	**def** $T\ fn$([$T\ x$]*) $= rhs$	function declaration
t	::=	v	local variables or attributes
	$\mid$	$Co(e^*)$	constructor expressions
	$\mid$	(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

Figure 1: The BNF syntax of the ABS language with the imperative sublanguage and the functional sublanguage, where *rhs* denotes a side-effect free expression in the underlying functional language for defining data types Dd and functions F. 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.

A method definition has the form $m(\overline{x})${**var** $\overline{y}$; s; **return** e}, ignoring type information, where $\overline{x}$ is the list of parameters, $\overline{y}$ an optional list of *local variables*, and s; **return** e the body. As for class parameters, there is read-only access to the parameter list $\overline{x}$. 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*.

Processor release points influence the internal control flow in an object. A processor release point is either declared by a guarded command **await** g or unconditionally by **suspend**. After a processor release, an *enabled* and suspended process is selected for execution. The **suspend** statement suspends the executing process and releases the processor, and the suspended process is enabled whenever the processor is free. If the guard of an **await** statement evaluates to false during process execution, the *continuation* of the process is *suspended*, and the processor is released. By execution of an external *asynchronous method call* **await** $x.m(\overline{e})$ or **await** $v := x.m(\overline{e})$, the method m is invoked on x with input values $\overline{e}$. The continuation of the calling process is then 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. Execution of statement **await** b, where b is a

Boolean side-effect free expression over the state of the object, leads to no suspension if b evaluates to true. The process is otherwise suspended, and the continuation is enabled whenever b evaluates to true. The statement $e!m(\overline{e})$ invokes $e.m$ asynchronously, and the calling process continues without waiting for the reply. The language additionally contains statements for object creation, synchronous method calls, **skip**, conditionals, loops, and sequential composition. 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 provieds no methods and invokes no methods on the generated objects.

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 [5, 11]. At any point in time the communication history abstractly captures the system state [6, 7]. 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 [2]. 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 $\_ \vdash \_ : Seq[T] \times T \to Seq[T]$ (where "_" indicates an argument position). Let $a, b : Seq[T]$, $x, y, z : T$, and $s : Set[T]$. Projection $\_/\_ : Seq[T] \times Set[T] \to Seq[T]$ is defined inductively by $\varepsilon/s \triangleq \varepsilon$ and $(a \vdash x)/s \triangleq \textbf{if } x \in s \textbf{ then } (a/s) \vdash x \textbf{ else } a/s \textbf{ fi}$. The "ends with" and "begins with" predicates $\_\textbf{ew}\_ : Seq[T] \times T \to Bool$ and $\_\textbf{bw}\_ : Seq[T] \times T \to Bool$ are defined inductively by $\varepsilon \textbf{ ew } x \triangleq false$, $(a \vdash x) \textbf{ ew } y \triangleq x = y$, $\varepsilon \textbf{ bw } x \triangleq false$, $(\varepsilon \vdash x) \textbf{ bw } y \triangleq x = y$, and $(a \vdash z \vdash x) \textbf{ bw } y \triangleq (a \vdash z) \textbf{ bw } y$. Furthermore, let $a \leqslant b$ denote that a is a prefix of 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 type of values that may occur as actual parameters to method calls, including *Obj*, *Nat*, and *Bool*. Communication events are defined next.

Definition 1 ***(Communication events)*** *Let $o, o' : Obj, m : Mtd, c : Cls, \overline{e} : List[Data]$, and $v : Data$. 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, $Ev = IEv \cup IREv \cup CEv \cup CREv \cup NEv \cup NREv$.*

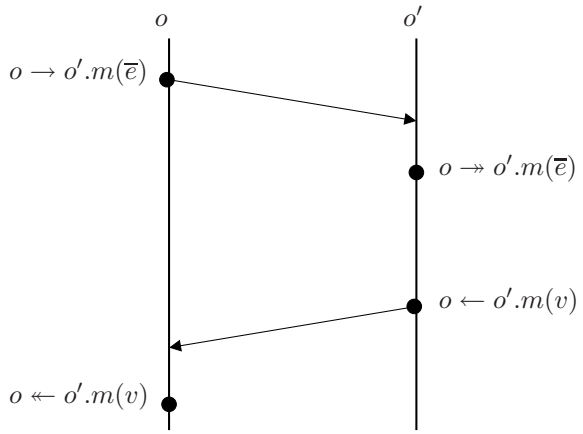

Figure 2: A method call cycle, where object o calls a method m on object o'. The arrows indicate message passing, and the bullets indicates events. The events on the left hand side are visible to o, whereas the events on the right hand side are visible to o'. Remark that there is an arbitrary delay between message receiving and reaction.

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

A method call is in our model reflected by four communication events, as illustrated in Fig. 2 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 $o \rightarrow o'.m(\overline{e})$ 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 $o \twoheadleftarrow o'.m(v)$. Next we define *communication histories* as a sequence of events. The creation of an object o' by an object o is reflected by the events $o \rightarrow o'.\mathbf{new}\ C(\overline{e})$ 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 $o \rightarrow o'.\mathbf{new}\ C(\overline{e})$ reflects that o initiates the creation, whereas $o \twoheadrightarrow o'.\mathbf{new}\ C(\overline{e})$ reflects that o' is created. 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

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

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 $o \rightarrow o'.\textbf{new}\ C(\overline{e})$ is local to o whereas $o \twoheadrightarrow o'.\textbf{new}\ C(\overline{e})$ is local to o'.

Functions may extract information from the history. In particular, we define $oid : Seq[Ev] \rightarrow Set[Obj]$ as follows:

$$oid(\varepsilon) \triangleq \{\mathsf{null}\}$$
$$oid(o \rightarrow o'.m(\overline{e})) \triangleq \{o, o'\} \cup oid(\overline{e})$$
$$oid(o \leftarrow o'.m(v)) \triangleq \{o, o'\} \cup oid(v)$$
$$oid(o \rightarrow o'.\textbf{new}\ C(\overline{e})) \triangleq \{o, o'\} \cup oid(\overline{e})$$
$$oid(o \twoheadrightarrow o'.\textbf{new}\ C(\overline{e})) \triangleq \{o, o'\} \cup oid(\overline{e})$$
$$oid(h \vdash \gamma) \triangleq oid(h) \cup oid(\gamma)$$
$$oid(o \twoheadrightarrow o'.m(\overline{e})) \triangleq \{o, o'\} \cup oid(\overline{e})$$
$$oid(o \twoheadleftarrow o'.m(v)) \triangleq \{o, o'\} \cup oid(v)$$

where $\gamma : Ev$, and $oid(\overline{e})$ returns the set of object identifiers occurring in the list $\overline{e}$. The function $ob : Seq[Ev] \rightarrow Set[Obj \times Cls \times List[Data]]$ returns the set of created objects in a history: $ob(\varepsilon) \triangleq \emptyset$, $ob(h \vdash o \rightarrow o'.\textbf{new}\ C(\overline{e})) \triangleq ob(h) \cup \{o' : C(\overline{e})\}$, and $ob(h \vdash \textbf{\textit{others}}) \triangleq ob(h)$ for all other events; the function $obId : Seq[Ev] \rightarrow Set[Obj]$ returns identities of the created objects: $obId(\varepsilon) \triangleq \emptyset$, $obId(h \vdash o \rightarrow o'.\textbf{new}\ C(\overline{e})) \triangleq obId(h) \cup \{o'\}$, and $obId(h \vdash \textbf{\textit{others}}) \triangleq obId(h)$ for all other events. For a local history h/o [1], the projection $ob(h/o)$ returns all objects *created* by 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 well-formedness predicate over communication histories:

Definition 4 ***(Well-formed histories)*** *Let O be a set of object identities and $h : Seq[Ev_O]$, the well-formedness predicate $wf : Seq[Ev_{Set[Obj]}] \times Set[Obj] \rightarrow Bool$ is defined*

[1] Let h/o be the shorthand for h/Ev_o

by:

$$
\begin{aligned}
&\mathit{wf}(\varepsilon, O) \triangleq \mathit{true}\\
&\mathit{wf}(h \vdash o \rightarrow o'.m(\overline{e}), O) \triangleq \mathit{wf}(h, O) \wedge o \neq \mathsf{null} \wedge o' \neq \mathsf{null}\\
&\mathit{wf}(h \vdash o \twoheadrightarrow o'.m(\overline{e}), O) \triangleq \mathit{wf}(h, O) \wedge o \neq \mathit{null}\\
&\qquad\qquad \wedge\ (o \in O \Rightarrow \mathit{valid}(h, o \rightarrow o'.m(\overline{e}), o \twoheadrightarrow o'.m(\overline{e})))\\
&\mathit{wf}(h \vdash o \leftarrow o'.m(v), O) \triangleq \mathit{wf}(h, O) \wedge \mathit{valid}(h, o \twoheadrightarrow o'.m(\_), o \leftarrow o'.m(\_))\\
&\mathit{wf}(h \vdash o \twoheadleftarrow o'.m(v), O) \triangleq \mathit{wf}(h, O) \wedge o' \neq \mathit{null}\\
&\qquad\qquad \wedge\ \mathit{valid}(h, o \rightarrow o'.m(\_), o \twoheadleftarrow o'.m(\_)))\\
&\qquad\qquad \wedge\ (o' \in O \Rightarrow \mathit{valid}(h, o \leftarrow o'.m(v), o \twoheadleftarrow o'.m(v)))\\
&\mathit{wf}(h \vdash o \rightarrow o'.\mathsf{new}\ C(\overline{e}), O) \triangleq \mathit{wf}(h, O) \wedge \mathit{parent}(o') = o \wedge o' \notin \mathit{oid}(h)\\
&\mathit{wf}(h \vdash o \twoheadrightarrow o'.\mathsf{new}\ C(\overline{e}), O) \triangleq \mathit{wf}(h, O) \wedge \mathit{parent}(o') = o\\
&\qquad\qquad \wedge\ (o \in O) \Rightarrow h\ \mathsf{ew}\ o \rightarrow o'.\mathsf{new}\ C(\overline{e})
\end{aligned}
$$

where the validity function $\mathit{valid} : \mathit{Seq}[Ev] \times Ev \times Ev \rightarrow \mathit{Bool}$ *is defined by:*

$$\mathit{valid}(h, e_1, e_2) \triangleq \#(h/e_1) > \#(h/e_2)$$

The validity check of well-formedness is defined as following. 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 where the callee is in O, there must be more completion events than completion reaction events. Remark that for object creation, the parent object and the created object synchronize, i.e., the creation event $o \twoheadrightarrow o'.\mathsf{new}\ C(\overline{e})$ is immediately preceded by $o \rightarrow o'.\mathsf{new}\ C(\overline{e})$ on a composed history.

For an object o', message sending is not visible on the local history of o' if the sender o is different from o'. For the message receiving $o \twoheadrightarrow o'.m(E)$ in Def.4, the validity check is trivially satisfied for a well-formed local history of o' when $o \neq o'$. Consequently, for the local history of o', the validity check only applies to local calls (i.e., where $o = o'$). For a global system, i.e., where O contains all objects in the system, the validity check is applied to all events since both the caller and the callee must be in O.

3.1 Invariant Reasoning

In interactive and nonterminating 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 instance: A method implements its part of the contract by ensuring that the invariant holds upon termination and when the method is suspended, assuming that the invariant holds initially and after suspensions. To facilitate compositional and component-based reasoning about programs, the class invariant is used to establish a *relationship between the internal state and the observable behavior of class instances*. The internal state reflects the values of class attributes, whereas the observable behavior is expressed as a set of potential communication histories. By hiding the internal state, class invariants form a suitable basis for compositional reasoning about object systems.

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

$$\begin{aligned} I_C(\overline{w}, h_{\mathsf{this}}) \triangleq\ & I(\overline{w}, h_{\mathsf{this}}) \wedge \mathit{wf}(h_{\mathsf{this}}, \{\mathsf{this}\}) \\ & \wedge\ h_{\mathsf{this}}\ \mathbf{bw}\ \mathit{parent}(\mathsf{this}) \twoheadrightarrow \mathsf{this}.\mathbf{new}\ C(\mathsf{cp}). \end{aligned}$$

4 Analysis of *ABS* Programs

The semantics is expressed as an encoding into a sequential sublanguage without shared variables, but with a nondeterministic assignment operator [8]. Nondeterministic 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 4.4.

A call to a method of an object o' by an object o is modeled as passing an invocation message from o to o', and the reply as passing a completion message from o' to o. Similarly, object creation is captured by a message from the parent object to the generated object. This communication is captured by four *events* on the communication history, as illustrated in Fig. 2. For a local call (i.e., $o = o'$), all four events are visible on the local history of o.

4.1 Semantic Definition by a Syntactic Encoding

We consider a simple *sequential* language with the syntax

$$\mathbf{skip} \mid \mathbf{abort} \mid \overline{v} := \overline{e} \mid s_1; s_2 \mid \mathbf{if}\ b\ \mathbf{then}\ s_1\ [\mathbf{else}\ s_2]^?\ \mathbf{fi}.$$

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

$$\overline{y} := \mathbf{some}\ \overline{x}\ .\ P(\overline{x})$$

For partial correctness, we assume that the statement does not terminate normally if no such $\overline{x}$ can be found. 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 this state, and has otherwise no effect. Similarly *assume statements* are used to encode known facts. Semantically the statement

$$\mathbf{assume}\ b$$

could be understood as $\mathbf{if}\ b\ \mathbf{then\ skip\ else\ abort\ fi}$.

A process with release points and asynchronous method calls is interpreted as a nondeterministic *SEQ* program *without* shared variables and release points, by the mapping $\langle\!\langle\ \rangle\!\rangle$, as defined in Fig. 3. Let $P^{\overline{x}}_{\overline{e}}$, where $\overline{x}$ and $\overline{e}$ are of the same length, denote the substitution of every free occurrence of $\overline{x}$ in P by $\overline{e}$. Expressions and types are mapped by the identity function. At the class level, the list of class attributes is augmented with

$$\langle\!\langle\, s_1; s_2 \,\rangle\!\rangle \triangleq \langle\!\langle\, s_1 \,\rangle\!\rangle; \langle\!\langle\, s_2 \,\rangle\!\rangle$$
$$\langle\!\langle\, \mathbf{skip} \,\rangle\!\rangle \triangleq \mathbf{skip}$$
$$\langle\!\langle\, \mathbf{abort} \,\rangle\!\rangle \triangleq \mathbf{abort}$$
$$\langle\!\langle\, v := e \,\rangle\!\rangle \triangleq v := e$$
$$\langle\!\langle\, \mathbf{suspend} \,\rangle\!\rangle \triangleq \mathbf{assert}\ I_C(\overline{w}, \mathcal{H});\ \overline{w}, \mathcal{H} := \mathbf{some}\ \overline{w}', \mathcal{H}'\,.\, \mathcal{H} \leqslant \mathcal{H}' \wedge I_C(\overline{w}', \mathcal{H}') \wedge Pending(\mathcal{H}', \mathsf{caller} \twoheadrightarrow \mathsf{this}.n(\overline{x}))$$
$$\langle\!\langle\, \mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{fi} \,\rangle\!\rangle \triangleq \mathbf{if}\ \ b\ \mathbf{then}\ \langle\!\langle\, s_1 \,\rangle\!\rangle\ \ \mathbf{else}\ \ \langle\!\langle\, s_2 \,\rangle\!\rangle\ \mathbf{fi}$$
$$\langle\!\langle\, o.m(\overline{e}) \,\rangle\!\rangle \triangleq \mathcal{H} := \mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e}); v' := \mathbf{some}\ v'\,.\, true;\ \mathcal{H} := \mathcal{H} \vdash \mathsf{this} \twoheadleftarrow o.m(v') \qquad (o \neq \mathsf{this})$$
$$\langle\!\langle\, v := o.m(\overline{e}) \,\rangle\!\rangle \triangleq \langle\!\langle\, o.m(\overline{e}) \,\rangle\!\rangle; v := v';$$
$$\langle\!\langle\, o!m(\overline{e}) \,\rangle\!\rangle \triangleq \mathcal{H} := \mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e})$$
$$\langle\!\langle\, \mathbf{await}\ o.m(\overline{e}) \,\rangle\!\rangle \triangleq \mathcal{H} := \mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e}); o' := o; \langle\!\langle\, \mathbf{suspend} \,\rangle\!\rangle; v' := \mathbf{some}\ v'\,.\, true;\ \mathcal{H} := \mathcal{H} \vdash \mathsf{this} \twoheadleftarrow o'.m(v'); \mathbf{assume}\ \mathit{wf}(\mathcal{H}) \qquad (o \neq \mathsf{this})$$
$$\langle\!\langle\, \mathbf{await}\ v := o.m(\overline{e}) \,\rangle\!\rangle \triangleq \langle\!\langle\, \mathbf{await}\ o.m(\overline{e}) \,\rangle\!\rangle; v := v';$$
$$\langle\!\langle\, \mathsf{this}.m(\overline{e}) \,\rangle\!\rangle \triangleq \mathcal{H} := \mathcal{H} \vdash \mathsf{this} \rightarrow \mathsf{this}.m(\overline{e}); \mathbf{assert}\ I_C(\overline{w}, \mathcal{H});\ v', \overline{w}, \mathcal{H} := \mathbf{some}\ v', \overline{w}', \mathcal{H}'\,.\, \mathcal{H} \leqslant \mathcal{H}' \wedge I_C(\overline{w}', \mathcal{H}') \wedge Pending(\mathcal{H}', \mathsf{caller} \twoheadrightarrow \mathsf{this}.n(\overline{x})) \wedge (\forall \overline{z}\,.\, S^{\overline{x}, \mathsf{caller}}_{\overline{e}, \mathsf{this}} \Rightarrow R^{\mathsf{return}, \overline{w}, \mathcal{H}}_{v', \overline{w}', \mathcal{H}'});\ \mathcal{H} := \mathcal{H} \vdash \mathsf{this} \twoheadleftarrow \mathsf{this}.m(v'); \mathbf{assume}\ \mathit{wf}(\mathcal{H}) \qquad (\overline{z} = FV(S, R) \setminus \overline{w}, \mathcal{H}, \mathsf{cp})$$
$$\langle\!\langle\, v := \mathsf{this}.m(\overline{e}) \,\rangle\!\rangle \triangleq \langle\!\langle\, \mathsf{this}.m(\overline{e}) \,\rangle\!\rangle; v := v';$$
$$\langle\!\langle\, \mathbf{await}\ \mathsf{this}.m(\overline{e}) \,\rangle\!\rangle \triangleq \mathcal{H} := \mathcal{H} \vdash \mathsf{this} \rightarrow \mathsf{this}.m(\overline{e}); \mathbf{assert}\ I_C(\overline{w}, \mathcal{H});\ v', \overline{w}, \mathcal{H} := \mathbf{some}\ v', \overline{w}', \mathcal{H}'\,.\, \mathcal{H} \leqslant \mathcal{H}' \wedge I_C(\overline{w}', \mathcal{H}') \wedge Pending(\mathcal{H}', \mathsf{caller} \twoheadrightarrow \mathsf{this}.n(\overline{x}))\ \mathcal{H} := \mathcal{H} \vdash \mathsf{this} \twoheadleftarrow \mathsf{this}.m(v'); \mathbf{assume}\ \mathit{wf}(\mathcal{H})$$
$$\langle\!\langle\, \mathbf{await}\ v := \mathsf{this}.m(\overline{e}) \,\rangle\!\rangle \triangleq \langle\!\langle\, \mathbf{await}\ \mathsf{this}.m(\overline{e}) \,\rangle\!\rangle; v := v';$$
$$\langle\!\langle\, \mathbf{await}\ b \,\rangle\!\rangle \triangleq \mathbf{if}\ b\ \mathbf{then\ skip\ else\ assert}\ I_C(\overline{w}, \mathcal{H});\ \overline{w}, \mathcal{H} := \mathbf{some}\ \overline{w}', \mathcal{H}'\,.\, \mathcal{H} \leqslant \mathcal{H}' \wedge I_C(\overline{w}', \mathcal{H}') \wedge Pending(\mathcal{H}', \mathsf{caller} \twoheadrightarrow \mathsf{this}.n(\overline{x})) \wedge b^{\overline{w}}_{\overline{w}'}\ \mathbf{fi}$$
$$\langle\!\langle\, x := \mathbf{new}\ C(\overline{e}) \,\rangle\!\rangle \triangleq x' := \mathbf{some}\ x'.\mathit{parent}(x') = \mathsf{this} \wedge x' \notin \mathit{oid}(\mathcal{H});\ \mathcal{H} := \mathcal{H} \vdash \mathsf{this} \rightarrow x'.\mathbf{new}_C(\overline{e}); x := x'$$
$$\langle\!\langle\, m(\overline{x})\ B \,\rangle\!\rangle \triangleq \mathcal{H} := \mathcal{H} \vdash \mathsf{caller} \twoheadrightarrow \mathsf{this}.m(\overline{x}); \langle\!\langle\, B \,\rangle\!\rangle;\ \mathcal{H} := \mathcal{H} \vdash \mathsf{caller} \leftarrow \mathsf{this}.m(\mathsf{return}); \mathbf{assume}\ \mathit{wf}(\mathcal{H})$$
$$\langle\!\langle\, \mathbf{return}\ e \,\rangle\!\rangle \triangleq \mathsf{return} := e$$

Figure 3: ABS syntactic equations. The assumptions reflect that the history in an execution is welldefined. For non-deterministic extension of $\mathcal{H}$, we let $\mathrm{n}(\overline{x})$ be the enclosing method of the statements.

$\mathsf{this} : \mathit{Obj}$ and $\mathcal{H} : \mathit{Seq}[Ev_{\mathsf{this}}]$, representing self reference and the history, respectively. We let $\mathit{wf}(\mathcal{H})$ abbreviate $\mathit{wf}(\mathcal{H}, \{\mathsf{this}\})$.

The semantics of a method is defined from the local perspective of processes. A *SEQ* processes executes on a state $\overline{w} \cup \mathcal{H}$ extended with local variables. The local effect of executing an invocation or 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 nondeterministic assignments to $\overline{w}$ and $\mathcal{H}$. When a method $m(\overline{x})$ starts execution, the local communication history is extended by $\mathcal{H} := \mathcal{H} \vdash \mathsf{caller} \twoheadrightarrow \mathsf{this}.m(\overline{e})$. The termination of a local process, representing a method invocation, extends $\mathcal{H}$ with a completion message: $\mathcal{H} := \mathcal{H} \vdash \mathsf{caller} \twoheadleftarrow \mathsf{this}.m(v)$, where v is the return value of m.

When the process executes an invocation statement $x.m(\overline{e})$, the history is extended by an output message: $\mathcal{H} := \mathcal{H} \vdash \mathsf{this} \rightarrow x.m(\overline{e})$, and fetching the reply is encoded by $\mathcal{H} := \mathcal{H} \vdash \mathsf{this} \twoheadleftarrow x.m(v)$. When a process is suspended waiting for a reply, it preserves the invariants and a nondeterministic update of $\overline{w}$ and $\mathcal{H}$ captures execution by the en-

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

Figure 4: Verification conditions for *ABS* methods. Condition 1 is for class initialization to establish the invariant. *init* refers to the initialization block of the class. Condition 2 ensures that the invariant is maintained by each encoding of method $m(\overline{x})\ B$. Condition 3 is used to verify local synchronous calls, where S is the precondition and R is the postcondition for the encoding of method $m(\overline{x})\ B$.

vironment and by other processes in the same object. For partial correctness reasoning, we may assume that processes are not suspended infinitely long. Consequently, nondeterministic assignment captures the possible interleaving of processes in an abstract manner.

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

The following Pending function describes that method calls are not completed. $Pending : Seq[Ev] \times Ev \rightarrow Bool$:

$$Pending(h, o \twoheadrightarrow o'.m(\overline{e})) \triangleq \#(h/o \twoheadrightarrow o'.m(\overline{e})) > \#(h/o \leftarrow o'.m(\_))$$

In Fig. 3, the usage of *Pending* captures that the current process is not terminated during suspension.

Lemma 1 *The local history of an object is well-formed for any legal execution.*

Proof. Preservation of well-formedness is trivial for statements that do not extend the local history $\mathcal{H}$, and we need to ensure well-formedness after extensions of $\mathcal{H}$. Well-formedness is maintained by processor release points since the class invariant implies well-formedness. Extending the history with invocation or invocation reaction events maintains well-formedness 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, well-formedness is guaranteed by the **assume** statements following the different extensions.

4.2 Weakest Liberal Preconditions

Based on the encoding from *ABS* to *SEQ*, we may define *weakest liberal preconditions* for the different *ABS* statements. The verification conditions of a class C with invariant $I_C(\mathcal{H}, \overline{w})$ are summarized in Fig. 4. Condition (1) applies to the initialization block *init* of C, requiring that the invariant is established when *init* terminates. We may reason about possible processor release points in *init* by the weakest liberal preconditions given below. Condition (2) of Fig. 4 applies to each method $m(\overline{x})\ B$ 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. For method $m(\overline{x})\ B$ which starts execution in a state where precondition S holds, the postcondition R must hold when the method terminates. Remark that S and R may violate the class invariant.

The weakest liberal precondition for nondeterministic assignment is given by:

$$wlp(\overline{y} := \mathbf{some}\ \overline{x}\,.\,P(\overline{x}), Q) = \forall \overline{x}\,.\,(P(\overline{x}) \Rightarrow Q^{\overline{y}}_{\overline{x}})$$

$$
\begin{array}{l}
wlp(s_1; s_2, Q) \triangleq wlp(s_1, wlp(s_2, Q)) \\
wlp(\textbf{skip}, Q) \triangleq Q \\
wlp(\textbf{abort}, Q) \triangleq \mathit{false} \\
wlp(v := e, Q) \triangleq Q^v_e \\
wlp(\textbf{suspend}, Q) \triangleq I_C(\overline{w}, \mathcal{H}) \wedge \forall \overline{w}', \mathcal{H}' \,.\, (\mathcal{H} \leqslant \mathcal{H}' \wedge I_C(\overline{w}', \mathcal{H}') \wedge \\
\quad Pending(\mathcal{H}', \mathsf{caller} \twoheadrightarrow \mathsf{this}.n(\overline{x}))) \Rightarrow Q^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'} \\
wlp(\textbf{if}\ b\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi}, Q) \triangleq \textbf{if}\ b\ \textbf{then}\ wlp(s_1, Q)\ \textbf{else}\ wlp(s_2, Q) \\
wlp(o.m(\overline{e}), Q) \triangleq \forall v' \,.\, Q^{\mathcal{H}}_{\mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e}) \vdash \mathsf{this} \twoheadleftarrow o.m(v')} \qquad (o \neq \mathsf{this}) \\
wlp(v := o.m(\overline{e}), Q) \triangleq \forall v' \,.\, Q^{v,\mathcal{H}}_{v',\mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e}) \vdash \mathsf{this} \twoheadleftarrow o.m(v')} \qquad (o \neq \mathsf{this}) \\
wlp(o!m(\overline{e}), Q) \triangleq Q^{\mathcal{H}}_{\mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e})} \\
wlp(\textbf{await}\ \ o.m(\overline{e}), Q) \triangleq I_C(\overline{w}, \mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e})) \wedge \\
\quad \forall v', \overline{w}', \mathcal{H}' \,.\, ((\mathcal{H} \vdash \mathsf{this} \rightarrow o.m(\overline{e}) \leqslant \mathcal{H}') \wedge I_C(\overline{w}', \mathcal{H}') \wedge wf(\mathcal{H}' \vdash \mathsf{this} \twoheadleftarrow o.m(v'))) \\
\qquad \Rightarrow Q^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}' \vdash \mathsf{this} \twoheadleftarrow o.m(v')} \qquad (o \neq \mathsf{this}) \\
wlp(\textbf{await}\ \ v := o.m(\overline{e}), Q) \triangleq wlp(\textbf{await}\ \ o.m(\overline{e}), Q^v_{v'}) \\
wlp(\mathsf{this}.m(\overline{e}), Q) \triangleq I_C(\overline{w}, \mathcal{H} \vdash \mathsf{this} \rightarrow \mathsf{this}.m(\overline{e})) \wedge \forall v', \overline{w}', \mathcal{H}' \,. \\
\quad (\mathcal{H} \vdash \mathsf{this} \rightarrow \mathsf{this}.m(\overline{e}) \leqslant \mathcal{H}' \wedge (\forall \overline{z} \,.\, S^{\overline{x},\mathsf{caller},\mathcal{H}}_{\overline{e},\mathsf{this},\mathcal{H} \vdash \mathsf{this} \rightarrow \mathsf{this}.m(\overline{e})} \Rightarrow R^{\mathsf{return},\overline{w},\mathcal{H}}_{v',\overline{w}',\mathcal{H}'}) \\
\quad \wedge I_C(\overline{w}', \mathcal{H}') \wedge Pending(\mathcal{H}', \mathsf{caller} \twoheadrightarrow \mathsf{this}.n(\overline{x})) \wedge wf(\mathcal{H}' \vdash \mathsf{this} \twoheadleftarrow \mathsf{this}.m(v'))) \\
\quad \Rightarrow Q^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}' \vdash \mathsf{this} \twoheadleftarrow \mathsf{this}.m(v')} \qquad (\overline{z} = FV(S, R) \setminus \overline{w}, \mathcal{H}, \mathsf{cp}) \\
wlp(v := \mathsf{this}.m(\overline{e}), Q) \triangleq wlp(\mathsf{this}.m(\overline{e}), Q^v_{v'}) \\
wlp(\textbf{await}\ \ \mathsf{this}.m(\overline{e}), Q) \triangleq I_C(\overline{w}, \mathcal{H} \vdash \mathsf{this} \rightarrow \mathsf{this}.m(\overline{e})) \wedge \forall v', \overline{w}', \mathcal{H}' \,. \\
\quad (\mathcal{H} \vdash \mathsf{this} \rightarrow \mathsf{this}.m(\overline{e}) \leqslant \mathcal{H}' \wedge I_C(\overline{w}', \mathcal{H}') \wedge Pending(\mathcal{H}', \mathsf{caller} \twoheadrightarrow \mathsf{this}.n(\overline{x})) \wedge \\
\quad wf(\mathcal{H}' \vdash \mathsf{this} \twoheadleftarrow \mathsf{this}.m(v'))) \Rightarrow Q^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}' \vdash \mathsf{this} \twoheadleftarrow \mathsf{this}.m(v')} \\
wlp(\textbf{await}\ \ v := \mathsf{this}.m(\overline{e}), Q) \triangleq wlp(\textbf{await}\ \ \mathsf{this}.m(\overline{e}), Q^v_{v'}) \\
wlp(\textbf{await}\ \ b, Q) \triangleq \textbf{if}\ b\ \textbf{then}\ Q\ \textbf{else}\ I_C(\overline{w}, \mathcal{H}) \wedge \\
\quad \forall \overline{w}', \mathcal{H}' \,.\, (\mathcal{H} \leqslant \mathcal{H}' \wedge\ I_C(\overline{w}', \mathcal{H}') \wedge Pending(\mathcal{H}', \mathsf{caller} \twoheadrightarrow \mathsf{this}.n(\overline{x})) \wedge b^{\overline{w}}_{\overline{w}'}) \Rightarrow Q^{\overline{w},\mathcal{H}}_{\overline{w}',\mathcal{H}'} \\
wlp(x := \textbf{new}\ \ C(\overline{e}), Q) \triangleq \forall x' \,.\, (parent(x') = \mathsf{this}\ \wedge\ x' \notin oid(\mathcal{H})) \Rightarrow Q^{x,\mathcal{H}}_{x',\mathcal{H} \vdash \mathsf{this} \leftrightarrow x' \,.\, \mathbf{new}_C(\overline{e})} \\
wlp(m(\overline{x})\ B, Q) \triangleq wlp(\mathcal{H} := \mathcal{H} \vdash \mathsf{caller} \twoheadrightarrow \mathsf{this}.m(\overline{x}); B; \\
\quad \mathcal{H} := \mathcal{H} \vdash \mathsf{caller} \leftarrow \mathsf{this}.m(\mathsf{return}), wf(\mathcal{H}) \Rightarrow Q) \\
wlp(\textbf{return}\ e, Q) \triangleq Q^{\mathsf{return}}_e
\end{array}
$$

Figure 5: Weakest liberal preconditions for *ABS* statements.

assuming that $\overline{x}$ is disjoint from FV[Q] - $\{\overline{y}\}$. The side condition may easily be satisfied, since variable names in **some** expressions may be renamed to avoid name captures. The weakest precondition for assert statements is given by:

$$wlp(\textbf{assert}\ \ b, Q) = b \wedge Q$$

and similarly for assumptions:

$$wlp(\textbf{assume}\ \ b, Q) = b \Rightarrow Q$$

Weakest liberal preconditions for the different *ABS* statements are summarized in Fig. 5, which are straight forwardly derived from the encoding in Fig. 3.

4.3 Hoare Logic

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

$$\{wf(\mathcal{H})\}\ \mathsf{s}\ \{wf(\mathcal{H})\}$$

$$\{\mathcal{H}_0\ =\ \mathcal{H}\}\ \mathsf{s}\ \{\mathcal{H}_0\ \leqslant\ \mathcal{H}\}$$

$$\{Pending(\mathcal{H}, \mathsf{caller} \twoheadrightarrow \mathsf{this}.m(\overline{e}))\}\ \mathsf{s}\ \{Pending(\mathcal{H}, \mathsf{caller} \twoheadrightarrow \mathsf{this}.m(\overline{e}))\}$$

$$\{P\}\ \textbf{skip}\ \{P\}$$

$$\{true\}\ \textbf{abort}\ \{false\}$$

$$\{P^x_e\}\ \mathsf{x} := \mathsf{e}\ \{P\}$$

$$\{I_C\}\ \textbf{suspend}\ \{I_C\}$$

$$\{I_C\}\ \textbf{await}\ \mathsf{b}\ \{I_C \wedge b\}$$

$$\{Q \wedge b\}\ \textbf{await}\ \mathsf{b}\ \{Q \wedge b\}$$

$$\{I_C{}^{\mathcal{H}}_{\mathcal{H}\ \vdash\ \mathsf{this} \rightarrow o.m(\overline{e})} \wedge\ o' = o\}\ \textbf{await}\ \ \mathsf{v}\ :=\ \mathsf{o.m}(\overline{e})\ \{\mathcal{H}\ \textbf{ew}\ \mathsf{this} \twoheadleftarrow o'.m(v) \wedge \exists v\ .\ I_C{}^{\mathcal{H}}_{pop(\mathcal{H})}\}$$

$$\frac{\{P\}S_1\{Q\}\quad \{Q\}S_2\{R\}}{\{P\}S_1;\ S_2\{R\}}$$

$$\frac{\{P \wedge B\}\mathsf{S}\{Q\}\quad (P \wedge \neg B) \Rightarrow Q}{\{P\}\textbf{if}\ \mathsf{B}\ \textbf{then}\ \mathsf{S}\ \textbf{fi}\{Q\}}$$

$$\frac{\{P \wedge B\}S_1\{Q\}\quad \{P \wedge \neg B\}S_2\{Q\}}{\{P\}\textbf{if}\ \mathsf{B}\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\{Q\}}$$

$$\frac{(P' \Rightarrow P)\quad \{P\}\mathsf{S}\{Q\}\quad (Q\ \Rightarrow\ Q')}{\{P'\}\mathsf{S}\{Q'\}}$$

Figure 6: Derived Hoare Rules. I_C denotes the class invariant. Primed variables are logical variables, and *pop* denotes the (left) rest operation, and S ranges over ABS statements, which may not use $\mathcal{H}$ as a program variable.

P holds and the execution terminates, then Q holds after s has terminated. Both weakest liberal preconditions and Hoare reasoning may be used in the same proof, since proving $\{P\}\mathsf{s}\{Q\}$ is the same as proving $P \Rightarrow wlp(\mathsf{s}, Q)$. If $wlp(\mathsf{s}, Q)$ also implies P, we say that $\{P\}\mathsf{s}\{Q\}$ is complete with respect to the weakest liberal precondition. The Hoare rules in Fig. 6 follow directly from the weakest liberal preconditions in Fig. 5 and Lemma 1. Application of Hoare rules instead of *wlp* may simplify proofs since quantifiers are not used for nondeterministic assignment. For instance, for a boolean guard b, the triple $\{I_C\}\textbf{await}\ \mathsf{b}\{I_C \wedge b\}$ follows directly from $wlp(\textbf{await}\ b, I_C \wedge b)$. 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, i.e., writing $\textbf{if}\ y \neq 0\ \textbf{then}\ x := 1/y\ \textbf{else}$ abort $\textbf{fi}$ instead of $x := 1/y$.

Processor release points are encoded as nondeterministic assignments to $\overline{w}$ and $\mathcal{H}$. Thus, the values of variables declared local to the method are not changed during method suspension. For an assertion L over local variables and any guard g, the Hoare triples $\{L\}\textbf{suspend}\{L\}$ and $\{L\}\textbf{await}\ \mathsf{g}\{L\}$ follow directly from the weakest liberal precon-

$$\{I_{C_{pop(\mathcal{H})}}^{\mathcal{H}} \wedge_{\mathcal{H}} \mathbf{ew}\ \mathsf{caller} \twoheadrightarrow \mathsf{this}.m(\overline{x})\} B \{wf(\mathcal{H} \vdash \mathsf{caller} \leftarrow \mathsf{this}.m(\mathsf{return})) \Rightarrow I_{C\,\mathcal{H}\vdash \mathsf{caller}\leftarrow \mathsf{this}.m(\mathsf{return})}^{\mathcal{H}}\}$$

Figure 7: Hoare triple formulation of verification condition (2) in Fig. 4 for the method $m(\overline{x})\ B$.

ditions.

The syntactic encoding of a method m in Fig. 3 reveals the reaction event ($\mathcal{H} \vdash_{\mathcal{H}} \mathsf{caller} \twoheadrightarrow \mathsf{this}.m(\overline{x})$) and completion event($\mathcal{H} \vdash_{\mathcal{H}} \mathsf{caller} \leftarrow \mathsf{this}.m(\mathsf{return})$). Verification condition (2) in Fig. 4 may then be formulated as the Hoare triple given in Fig. 7, where the pre- and postconditions to the method body are derived by standard reasoning.

4.4 Object Composition

By organizing the state space in terms of only locally accessible variables, including a local history variable recording local communication messages, we obtain a compositional reasoning system. By hiding the internal state variables of an object o of class C, an *external invariant* $I_{o:C(\overline{e})}$ defining its observable behavior on its local history h_o may be obtained:

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

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

For object composition, it suffices to compare the local histories of the composed objects. For this purpose, we adapt a composition method introduced by Soundarajan [14,15]. 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 on common messages when composed, expressed by projections from the common history. Thus, for a set O of objects with history H, we require that the projection of H on each object, e.g. o, is the same as the *local history* of object o:

$$H/o = h_o$$

When reasoning about a *global system*, we assume the existence of system generated object main, such that all object are created by main or generated objects. Thus, main is an ancestor of all objects. The global invariant of such a system of dynamically created objects may be constructed from the local invariants of the composed objects, requiring well-formedness of global history. The global invariants $I(H)$ of a global system with history H is

$$I(H) \triangleq (\bigwedge_{(o:C(\overline{e}))\in ob(H)} I_{o:C(\overline{e})}(H/o)) \wedge wf(H, obId(H) \cup \{\mathsf{main}\})$$

The quantification ranges over all generated objects in the composition, which is a finite number at any execution point. Note that the global invariant is obtained directly from the external invariants of the composed objects, without any restrictions on the local reasoning. This ensures compositional reasoning. Notice also that we consider dynamic systems where the number and identities of the composed objects are nondeterministic. Since object identities are only created by **new** statements, it follows that for a global system with history H, that $obId(H) \cup \{\mathsf{main}\} = oid(H) \setminus \{\mathsf{null}\}$. Since main is the initial root object, the creation of main is not reflected on the global history H, i.e., $\mathsf{main} \notin obId(H)$. The following lemma ensures that parent chains are cycle free for global systems.

Lemma 2 *Given a global system with history H and invariant $I(H)$, then*

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

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

$$\forall o \in obId(H \vdash \gamma)\,.\, o \notin anc(o) \wedge \mathsf{main} \in anc(o) \wedge (anc(o) \setminus \{\mathsf{main}\}) \subseteq obId(H \vdash \gamma)$$

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

For the case $H \vdash o \rightarrow o'.\mathbf{new}$ (ignoring the class of o'), well-formedness of H gives $parent(o') = o \wedge o' \notin oid(H)$. We distinguish two cases, $o = \mathsf{main}$ and $o \neq \mathsf{main}$.
Case $o = \mathsf{main}$: The conclusion follows directly by $anc(o') = \mathsf{main}$.
Case $o \neq \mathsf{main}$: The conclusion follows from IH and the proof obligation:

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

By the definition of anc, the proof obligation becomes

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

By $o \in oid(H \vdash o \rightarrow o'.\mathbf{new})$, we have $o \in obId(H \vdash o \rightarrow o'.\mathbf{new})$ since H is global. Thus, $o \in obId(H)$, and since $o' \notin oid(H)$, we have $o \neq o'$. The proof obligation then reduces to

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

Since $o \in obId(H)$, we have $\mathsf{main} \in anc(o)$ and $(anc(o) \setminus \{\mathsf{main}\}) \subseteq obId(H)$ by IH. The proof obligation then follows since $o' \notin oid(H)$.

5 Reader/Writer Example

This section gives an example of the readers and writers problem implemented in the ABS language. Also, we define safety invariants and illustrate the reasoning system through verification of these invariants.

5.1 Implementation

We assume given a shared database db, which provides two basic operations read and write. Through interface specifications, these are assumed to be accessible for RWController objects. Clients will communicate with a RWController object to obtain read and write access to the database. An implementation of RWController can be found in Fig.8. 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. A client is assumed not to terminate unless it has invoked closeR and closeW at least as many times as openR and openW, respectively. To ensure *fair* competition between readers and writers, invocations of openR and openW compete on equal terms for a guard writer = null. The attribute readers contains the clients with read access, writer contains the client with write access, and pr counts the number of pending calls to method db.read. The readers set is extended by execution of openR, where the guard

```
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 := Cons(caller, readers);}

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

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

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

   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 && readers = Empty && pr = 0;
      db.write(key, value);
   }
}
```

Figure 8: Implementation of the fair reader/writer controller

ensures that there is no writer. By openW, a client will gain write access if there currently is no writer. A client may thereby become the writer even if readers is nonempty. 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 activity is completed. The client contained in writer will eventually be allowed to perform write operations since all active readers are assumed to call closeR at some point. Thus, even though readers may be nonempty while writer contains a client, the controller ensures that reading and writing *activity* cannot happen simultaneously on the database. Remark that a client contained in writer is not allowed to sign up for reading (the example may be modified in order to give read access to the writer). The complete implementation of the example can be found in Appendix A.

5.2 Specification and Verification

Next we define a safety invariant for the RWController class, expressing a relation between observable communication and internal state of class instances. The internal state is given by the values of the class attributes, and functions over the local communication history are used to extract relevant information from the history. Define the function $Readers : Seq[Ev] \to Set[Obj]$ by:

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

in which ***others*** matches all ground terms not giving any match in the above equations. Upon termination of openR, the caller is added to the set of readers and the caller is removed from the set upon termination of closeR. We furthermore assume a function

Writers, defined over completions of openW and closeW in a corresponding manner, see Appendix B. Also define $\mathit{Reading} : \mathit{Seq}[\mathit{Ev}] \to \mathit{Nat}$ by:

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

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

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

$$I \triangleq I_1 \wedge I_2 \wedge I_3 \wedge I_4 \wedge I_5 \wedge I_6 \wedge I_7$$

where

$$\begin{aligned}
I_1 &\triangleq \mathit{Readers}(\mathcal{H}) = \mathsf{readers} \\
I_2 &\triangleq \mathit{Writers}(\mathcal{H}) = \{\mathsf{writer}\} \\
I_3 &\triangleq \#\{\mathsf{writer}\} \leqslant 1 \\
I_4 &\triangleq \mathit{Reading}(\mathcal{H}) = \mathsf{pr} \\
I_5 &\triangleq \mathit{Writing}(\mathcal{H}) > 0 \Rightarrow \mathsf{pr} = 0 \\
I_6 &\triangleq \mathit{Reading}(\mathcal{H}) \geqslant 0 \wedge \mathit{Writing}(\mathcal{H}) \geqslant 0 \\
I_7 &\triangleq \mathit{OK}(\mathcal{H})
\end{aligned}$$

The invariant illustrates how the values of class attributes may be expressed in terms of observable communication, e.g., $\mathit{Readers}(\mathcal{H})$ has the same value as readers. In addition, the invariant I implies $\mathit{Reading}(\mathcal{H}) = 0 \vee \mathit{Writing}(\mathcal{H}) = 0$, i.e. no reading and writing *activity* happens simultaneously. The predicate $\mathit{OK} : \mathit{Seq}[\mathit{Ev}] \to \mathit{Bool}$ is defined inductively over the history by:

$$\begin{aligned}
&\mathit{OK}(\varepsilon) \triangleq \mathit{true} \\
&\mathit{OK}(h \vdash _ \leftarrow \mathsf{this.openR}) \triangleq \mathit{OK}(h) \wedge \mathit{Writers}(h) = \emptyset && (1) \\
&\mathit{OK}(h \vdash _ \leftarrow \mathsf{this.openW}) \triangleq \mathit{OK}(h) \wedge \mathit{Writers}(h) = \emptyset && (2) \\
&\mathit{OK}(h \vdash \mathsf{this} \to \mathsf{db.write}) \triangleq \mathit{OK}(h) \wedge \mathit{Readers}(h) = \emptyset \ \wedge \\
&\qquad\qquad \mathit{Reading}(h) = 0 \wedge \mathit{Pending}(h, \mathit{Writers}(h) \twoheadrightarrow \mathsf{this.write}) && (3) \\
&\mathit{OK}(h \vdash \mathsf{this} \to \mathsf{db.read}) \triangleq \mathit{OK}(h) \ \wedge \\
&\qquad\qquad (\exists r \,.\, \mathit{Pending}(h, r \twoheadrightarrow \mathsf{this.read}) \wedge r \in \mathit{Readers}(h)) && (4) \\
&\mathit{OK}(h \vdash \boldsymbol{others}) \triangleq \mathit{OK}(h) && (5)
\end{aligned}$$

Here, conditions (1) and (2) reflects the *fairness condition*: invocations of openR and openW compete on equal terms for the guard writer = null, which equals $\mathit{Writers}(\mathcal{H}) = \emptyset$ by I_2. While writer is not null, conditions (1) and (2) additionally ensure that no clients can be included in the readers set or be assigned to writer. Condition (3) is an abstraction of the guard in write: when invoking db.write, there cannot be any readers or any pending calls to db.read. Furthermore, there must be an uncompleted invocation of this.write, with the registered writer as the caller. Correspondingly, Condition (4) expresses that when invoking db.read, there must be a pending call to this.read, and the caller of this.read must be in the set of registered readers.

As a verification example, the verification of read with respect to I_1 is presented in Fig. 9. The verification succeeds by application of the Hoare rules in Fig. 6 and the derivation in Fig. 7. Especially, the **`await`** statement is analyzed by the derived rule. The complete verification of this case study can be found in Appendix D.

$\{Readers(\mathcal{H}) = \mathsf{readers}\}$
await $\mathsf{writer} = \mathsf{null}$;
$\{Readers(\mathcal{H}) = \mathsf{readers} \wedge \mathsf{writer} = \mathsf{null}\}$
$\{Readers(\mathcal{H}) \cup \{\mathsf{caller}\} = \mathsf{Cons}(\mathsf{caller}, \mathsf{readers})\}$
$\mathsf{readers} := \mathsf{Cons}(\mathsf{caller}, \mathsf{readers})$
$\{Readers(\mathcal{H}) \cup \{\mathsf{caller}\} = \mathsf{readers}\}$

Figure 9: Verification details for the body of method read with respect to the invariant I_1. Here, two consecutive predicates $\{P\}\{Q\}$ resolves to the verification condition $P \Rightarrow Q$.

6 Related and Future Work

Related Work. The work is based on [1, 9], but the syntactic encoding is based on a different notion of locality. In [9] message sending is visible on the local history of both the caller and the callee. Thus, message sending leads to restrictions on the local history of the receiver. Here we solved the problem by redefining message sending, which is considered as a local action of the sender, whereas reacting upon the message is considered as a local action of the receiver. When composed, each reaction by the receiver must match a sent message. In contrast to the earlier work, the current approach allows unrestricted use of assumptions on the environment. This is valuable when reasoning about objects in an open environment.

Future Work. A sound and complete reasoning system for ABS language have been developed in this work. The next milestone of our research is to implement this reasoning system in the theorem prover framework of KeY, and make (semi-)automatic verification possible. We will then make a larger case study in ABS with the reasoning support of KeY.

7 Conclusion

The state space of distributed system will often be infinite. Therefore reasoning about a distributed system with model checking approaches is problematic. In order to solve this problem, we develop a compositional reasoning systems, choosing ABS as our language for modeling distributed systems, due to its high-level concurrency and communication mechanisms. In our reasoning system, the communication between objects in the distributed setting can be analyzed locally for each object and composed at need without having any knowledge of other objects' state. A complete communication of a method call between two objects is recorded by four events, with a partial ordering captured by the notion of well-formedness. The verification of a class can be done locally by means of verifying a class invariant, letting the invariant refer to the attributes and the local communication history. The history reflects the sequence of communication events and allows a compositional system for reasoning about distributed concurrent objects. Our system is sound and complete by construction, and 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 methods calls. In this paper, we have shown the verification of a reader-writer example in the *ABS* language using the proof system we develop.

References

[1] W. Ahrendt and M. Dylla. A system for compositional verification of asynchronous objects. *Science of Computer Programming*, In Press, 2010.

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

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

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

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

[6] O.-J. Dahl. Object-oriented specifications. In *Research directions in object-oriented programming*, pages 561–576. MIT Press, Cambridge, MA, USA, 1987.

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

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

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

[10] R. Hähnle, E. B. Johnsen, B. M. Østvold, J. Schäfer, M. Steffen, and 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.

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

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

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

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

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

A 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 Cons(Data, DataSet)}


def Bool isElement(Data data, DataSet list) =
  case list{
    Empty => False;
    Cons(d, l) =>
        data == d || isElement(data, l)
  }


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


def ListData delete(Data data, DataSet list) =
  case list{
    Empty => Empty;
    Cons(d, l) =>
      if (data == d) {delete(data, l)}
      else {Cons(d, delete(data, l))}
  }


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


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


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


class DataBase implements DB{
  Map map;

  {map == Empty;}
```

```
    Data read(Int key){
      return lookup(key, map);
    }

    Void write(Int key, Data data){
      map := modify(key, data, 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 := Cons(caller, readers);}

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

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

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

    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 && readers = Empty && pr = 0;
      db.write(key, value);
    }
  }
```

B Definition of Writers

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

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

C Definition of Writing

$Writing : Seq[Ev] \to Nat$

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

D Verification Details for RWController

We here present the verification details for each method in RWController, with respect to the user defined invariant I (Section 5.2). For each method we only show the relevant parts of I, the verification for the remaining parts of I is trivial.

D.1 openR

$I_{2\wedge 7}$:

$\{OK(\mathcal{H}) \wedge \mathit{Writers}(\mathcal{H}) = \{\mathsf{writer}\}\}$
await writer = null;
$\{OK(\mathcal{H}) \wedge \mathit{Writers}(\mathcal{H}) = \{\mathsf{writer}\} \wedge \mathsf{writer} = \mathsf{null}\}$
$\{OK(\mathcal{H}) \wedge \mathit{Writers}(\mathcal{H}) = \emptyset \wedge \mathit{Writers}(\mathcal{H}) = \{\mathsf{writer}\}\}$
readers := Cons(caller, readers)
$\{OK(\mathcal{H}) \wedge \mathit{Writers}(\mathcal{H}) = \emptyset \wedge \mathit{Writers}(\mathcal{H}) = \{\mathsf{writer}\}\}$

D.2 openW

$I_{2\wedge 3\wedge 7}$:

$\{(OK(\mathcal{H}) \wedge \mathit{Writers}(\mathcal{H}) = \{\mathsf{writer}\} \wedge \#(\mathit{Writers}(\mathcal{H})) \leqslant 1)\}$
await writer = null;
$\{(OK(\mathcal{H}) \wedge \mathit{Writers}(\mathcal{H}) = \{\mathsf{writer}\} \wedge \#(\mathit{Writers}(\mathcal{H})) \leqslant 1) \wedge \mathsf{writer} = \mathsf{null}\}$
$\{OK(\mathcal{H}) \wedge \mathit{Writers}(\mathcal{H}) = \emptyset \wedge \mathit{Writers}(\mathcal{H}) \cup \{\mathsf{caller}\} = \{\mathsf{caller}\} \wedge$
$\#(\mathit{Writers}(\mathcal{H}) \cup \{\mathsf{caller}\}) \leqslant 1\}$
writer := caller
$\{OK(\mathcal{H}) \wedge \mathit{Writers}(\mathcal{H}) = \emptyset \wedge \mathit{Writers}(\mathcal{H}) \cup \{\mathsf{caller}\} = \{\mathsf{writer}\} \wedge$
$\#(\mathit{Writers}(\mathcal{H}) \cup \{\mathsf{caller}\}) \leqslant 1\}$

D.3 closeR

I_1 :

$\{\mathit{Readers}(\mathcal{H}) = \mathsf{readers}\}$
$\{\mathit{Readers}(\mathcal{H}) \setminus \{\mathsf{caller}\} = \mathit{delete}(\mathsf{caller}, \mathsf{readers})\}$
readers := delete(caller, readers);
$\{\mathit{Readers}(\mathcal{H}) \setminus \{\mathsf{caller}\} = \mathsf{readers}\}$

D.4 closeW

I_2 :

$\{\mathit{Writers}(\mathcal{H}) = \{\mathsf{writer}\}\}$
await writer = caller;
$\{\mathit{Writers}(\mathcal{H}) = \{\mathsf{writer}\} \wedge \mathsf{writer} = \mathsf{caller}\}$
$\{\mathit{Writers}(\mathcal{H}) \setminus \{\mathsf{caller}\} = \{\mathit{null}\}\}$
writer := null;
$\{\mathit{Writers}(\mathcal{H}) \setminus \{\mathsf{caller}\} = \{\mathsf{writer}\}\}$

D.5 read

$I_{1\wedge 4\wedge 7}$:

$\{OK(\mathcal{H}) \wedge \mathit{Readers}(\mathcal{H}) = \mathsf{readers} \wedge \mathit{Reading}(\mathcal{H}) = \mathsf{pr} \wedge \mathit{Pending}(\mathcal{H}, \mathsf{caller} \twoheadrightarrow \mathsf{this.write}))\}$
await isElement(caller, readers);
$\{OK(\mathcal{H}) \wedge \mathit{Readers}(\mathcal{H}) = \mathsf{readers} \wedge \mathit{Reading}(\mathcal{H}) = \mathsf{pr} \wedge \mathit{Pending}(\mathcal{H}, \mathsf{caller} \twoheadrightarrow \mathsf{this.write}))) \wedge$

$\quad$ isElement(caller, readers)$\}$
$\{OK(\mathcal{H}) \ \wedge (\exists r \,.\, Pending(\mathcal{H}, r \twoheadrightarrow \mathsf{this.read}) \wedge r \in Readers(\mathcal{H}))$
$\quad \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Reading(\mathcal{H}) + 1 = \mathsf{pr} + 1\}$
$\mathsf{pr} := \mathsf{pr} + 1;$
$\{OK(\mathcal{H}) \ \wedge (\exists r \,.\, Pending(\mathcal{H}, r \twoheadrightarrow \mathsf{this.read}) \wedge r \in Readers(\mathcal{H}))$
$\quad \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Reading(\mathcal{H}) + 1 = \mathsf{pr}\}$
await result := db.read(key);
$\{(\exists \mathsf{result} \ . \ I_{1\wedge 4\wedge 7}^{\mathcal{H}}{}_{pop(\mathcal{H})}) \wedge \mathcal{H}$ **ew** this $\twoheadleftarrow db.\mathsf{read(result)}\}$
$\{OK(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Reading(\mathcal{H}) = \mathsf{pr} - 1\}$
$\mathsf{pr} := \mathsf{pr} - 1;$
$\{OK(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Reading(\mathcal{H}) = \mathsf{pr}\}$
return result;
$\{OK(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Reading(\mathcal{H}) = \mathsf{pr}\}$

D.6 write

$I_{1\wedge 2\wedge 4\wedge 5\wedge 7}$:

$\{OK(\mathcal{H}) \wedge Writers(\mathcal{H}) = \{writer\} \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge$
$\quad (Writing(\mathcal{H}) > 0 \Rightarrow \mathsf{pr} = 0) \wedge Pending(\mathcal{H}, \mathsf{caller} \twoheadrightarrow \mathsf{this.write}))\}$
await caller = writer && readers = Empty && pr = 0;
$\{OK(\mathcal{H}) \wedge Writers(\mathcal{H}) = \{writer\} \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge$
$\quad (Writing(\mathcal{H}) > 0 \Rightarrow \mathsf{pr} = 0 \wedge Pending(\mathcal{H}, \mathsf{caller} \twoheadrightarrow \mathsf{this.write}))) \wedge$
$\quad (\mathsf{caller} = \mathsf{writer} \wedge \mathsf{readers} = \mathsf{Empty} \wedge \mathsf{pr} = 0)\}$
$\{OK(\mathcal{H}) \wedge Readers(\mathcal{H}) = \emptyset \ \wedge Reading(\mathcal{H}) = 0 \wedge Pending(\mathcal{H}, Writers(\mathcal{H}) \twoheadrightarrow \mathsf{this.write}) \wedge$
$\quad Writers(\mathcal{H}) = \{writer\} \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge$
$\quad (Writing(\mathcal{H}) > 0 \Rightarrow \mathsf{pr} = 0)\}$
db.write(key, value);
$\{OK(\mathcal{H}) \wedge Writers(\mathcal{H}) = \{writer\} \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge$
$\quad (Writing(\mathcal{H}) > 0 \Rightarrow \mathsf{pr} = 0)\}$