UNIVERSITY OF OSLO
Department of Informatics

An Approach to Compositional Reasoning about Concurrent Objects and Futures [1]

Research Report 415

Crystal Chang Din

Johan Dovland

Olaf Owe

ISBN 82-7368-378-8
ISSN 0806-3036

Feb 2012

Contents

An Approach to Compositional Reasoning about Concurrent Objects and Futures [†]

Crystal Chang Din, Johan Dovland, Olaf Owe
Dept. of Informatics, University of Oslo,
P.O. Box 1080 Blindern, N-0316 Oslo, Norway.
E-mails: {crystald,johand,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 code level of mainstream object-oriented languages such as Java and C++, we consider an imperative, object-oriented language with a simpler concurrency model. This language, based on concurrent objects communicating by asynchronous method calls and futures, avoids some difficulties of mainstream object-oriented programming languages related to compositionality and aliasing. In particular, reasoning about futures is handled by means of histories. *Compositional verification systems* facilitate system analysis, allowing components to be analyzed independently of their environment. In this paper, a compositional proof system in dynamic logic for partial correctness is established based on communication histories and class invariants. The soundness and relative completeness of this proof system follow by construction using a transformational approach from a sequential language with a non-deterministic assignment operator.

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 [20]. Many distributed systems are programmed in 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, emerging from the complexity of their concurrency, communication, and synchronization mechanisms. Therefore, one may benefit from analyzing 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

[†]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`) and supported by the Short Term Scientific Mission, COST Action IC0701.

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 communication model of *Creol* [23] with futures, but the language ignores other aspects of *Creol* such as inheritance. *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* [12]: 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* and futures only.

A concurrent object has its own execution thread. Asynchronous method calls do not transfer control between the caller and the callee. In 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 [18]. Recently, this notion of cooperative scheduling and asynchronous method calls has been integrated in Java by means of concurrent object groups [26]. A Java code generator for *ABS* model is available. Thus programmers can model and analyze 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 [7, 19]. At any point in time the communication history abstractly captures the system state [9, 10]. In fact communication traces are used in semantics for full abstraction results (e.g., [1, 21]). 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 [3].

In this paper, we develop a partial correctness proof system for the *ABS* language (ignoring object groups, interfaces, data types). 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 encoding, extending a transformational technique originally proposed by Olderog and Apt [24] 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 history invariants of the different objects. Modularity is achieved since history invariants can be

established independently for each object and composed at need. Compared to previous work [14], we here consider a reasoning system for *futures* in dynamic logic: A *future* is a placeholder for the result value of the method call. An *ABS* reasoning system in dynamic logic is established in this paper.
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 5. Section 6 discusses related and future work, and Section 7 concludes the paper.

2 Syntax for the ABS Language

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[= e]]*;]$^?$ [s;]$^?$ **return** e}	method blocks
T	::=	$I \mid D \mid Void \mid Fid$	types(interface or data type)
v	::=	$x \mid w \mid fr$	local variables or attributes
e	::=	**null** | this | caller | destiny | $v \mid cp \mid t$	pure expressions
s	::=	$v := e \mid fr := e!m(e^*) \mid$ **await** $e \mid$ **suspend**	statements
		| **skip** | **abort** | **if** e **then** s [**else** s]$^?$ **fi** | $s; s$	
		| $v :=$ **new** $C(e^*) \mid v := e.m(e^*) \mid$ [**await**] $v := fr?$	

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, m method name, w class attribute, fr future variable, 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. Expressions e are side-effect free. Object groups are omitted. The functional sublanguage for terms t can be found in Appendix. A.

The syntax of the *ABS* language (slightly simplified) with futures 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 assume that all future variables are initialized before use, and that this is ensured by static checking. 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** e is enabled when the guard e evaluates to true. The **suspend** statement is considered equivalent to **await** *true*. Note that an alternative semantic definition of **await** *true* as **skip** is given in [15], and also used in several other *ABS* papers.

A *future*, *fr*, is a placeholder for the result of a method call: a future is generated with a unique identity upon invoking the method call. Upon termination of the called method, the return value is placed in the future. We then say that the future is *resolved*. The caller, or any other object that obtains the future identity, can fetch the result value when the future is resolved. 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 placed in the future of the call upon method termination. The predefined formal parameter caller gives access to the calling object, and the variable destiny is the future identity generated by caller. 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.

Object communication in *ABS* is *asynchronous*, as there is no explicit transfer of execution control between the caller and the callee. 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:

- $fr := x!m(\overline{e})$: Here the calling process generates a fresh *future identity* and continues without waiting for the *future* to become resolved.
- **await** $v := fr?$: The continuation of the process is here suspended until the *future* is resolved and it is selected for execution. The return value contained in fr is then assigned to v. Other processes of the caller may thereby execute during the suspension.
- $v := fr?$: The process is blocked until the *future* is resolved, and then assigns the value, contained in fr, to v. (In the original *ABS* language, the syntax of this statement is written $v := fr.\mathsf{get}$).
- $v := x.m(\overline{e})$: If x is different from this, the statement can be interpreted as $fr := x!m(\overline{e}); v := fr?$ for a fresh future variable fr. The method is invoked without releasing the processor of the calling object; the calling process *blocks* the processor while waiting for the *future* to become resolved. For the caller, the statement thereby appears to be synchronous which may potentially lead to deadlock, and should be used with care; however, the call statement are typically 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**, and conditionals. 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 root object main. Object main is allowed to generate objects, but cannot otherwise participate in the execution. Especially, main provides no methods and invokes no methods on generated objects.

3 Observable Behavior

The observable behavior of an object or a subsystem is described by communication histories over observable events [7, 19]. Each event is observable to only one object, namely the one generating it. We consider separate events for invoking and reacting upon a call, for generating and fetching the result of a future, as well as for initiation and fulfillment of object creation. The connection between the different events is formalized through the notion of *wellformedness*.

Notation. Sequences are constructed by the empty sequence ε and the right append function $\_ \cdot \_ : \mathit{Seq}[T] \times T \to \mathit{Seq}[T]$ (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 [10]. Let $a, b : \mathit{Seq}[T]$, $x, y, z : T$, and $s : \mathit{Set}[T]$. For a set s, the projection of a sequence a with respect to s, denoted a/s, is the subsequence of a consisting of all events belonging to s. Projection $\_/\_ : \mathit{Seq}[T] \times \mathit{Set}[T] \to \mathit{Seq}[T]$ is defined inductively by $\varepsilon/s \triangleq \varepsilon$ and $(a \cdot x)/s \triangleq \mathbf{if}\ x \in s\ \mathbf{then}\ (a/s) \cdot x\ \mathbf{else}\ a/s\ \mathbf{fi}$. 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 $\varepsilon\ \mathbf{ew}\ x \triangleq \mathit{false}$, $(a \cdot y)\ \mathbf{ew}\ x \triangleq x = y$, $\varepsilon\ \mathbf{bw}\ x \triangleq \mathit{false}$, $(\varepsilon \cdot y)\ \mathbf{bw}\ x \triangleq x = y$, and $(a \cdot z \cdot y)\ \mathbf{bw}\ x \triangleq (a \cdot z)\ \mathbf{bw}\ x$. Furthermore, let $a \leqslant b$ denote that a is a prefix of b, $a \mapsto b$ denote the concatenation of a and b, $\mathit{agree}(a)$ denote that all elements (if any) are equal, $\#a$ denote the length of a, and $[x_1, x_2, \ldots, x_i]$ denote the sequence of $x_1, x_2, \ldots, x_i$ for $i > 0$. The second argument of the "ends with" and "begins with" predicates can be lifted to sets of events. Let *Data* be the supertype of all kinds of data, including *Fid*. Communication events are defined next.

Definition 1 ***(Communication events)*** *Let* $\mathit{caller}, \mathit{callee}, \mathit{receiver} : \mathit{Obj}$, $\mathit{future} : \mathit{Fid}$, $\mathit{method} : \mathit{Mtd}$, $\mathit{class} : \mathit{Cls}$, $\mathit{args} : \mathit{List}[\mathit{Data}]$, *and* $\mathit{result} : \mathit{Data}$. *The set Ev of all communication events is defined by* $\mathit{Ev} \triangleq \mathit{IEv} \cup \mathit{IREv} \cup \mathit{CEv} \cup \mathit{CREv} \cup \mathit{NEv} \cup \mathit{NREv}$, *including*

- *the set IEv of invocation events* $\langle \mathit{caller} \to \mathit{callee}, \mathit{future}, \mathit{method}, \mathit{args} \rangle$
- *the set IREv of invocation reaction events* $\langle \mathit{caller} \twoheadrightarrow \mathit{callee}, \mathit{future}, \mathit{method}, \mathit{args} \rangle$
- *the set CEv of completion events* $\langle \leftarrow \mathit{callee}, \mathit{future}, \mathit{method}, \mathit{result} \rangle$
- *the set CREv of completion reaction events* $\langle \mathit{receiver} \twoheadleftarrow, \mathit{future}, \mathit{result} \rangle$
- *the set NEv of object creation events* $\langle \mathit{caller} \to \mathit{callee}, \mathit{class}, \mathit{args} \rangle$
- *the set NREv of object creation reaction events* $\langle \mathit{caller} \twoheadrightarrow \mathit{callee}, \mathit{class}, \mathit{args} \rangle$

The arrow notation is used to improve readability, and to identify the object generating the event. Invocation events are generated by the caller, invocation reaction events by the callee, completion events by the callee, completion reaction events by the receiver, object creation events by the caller, and object creation reaction events by the callee.

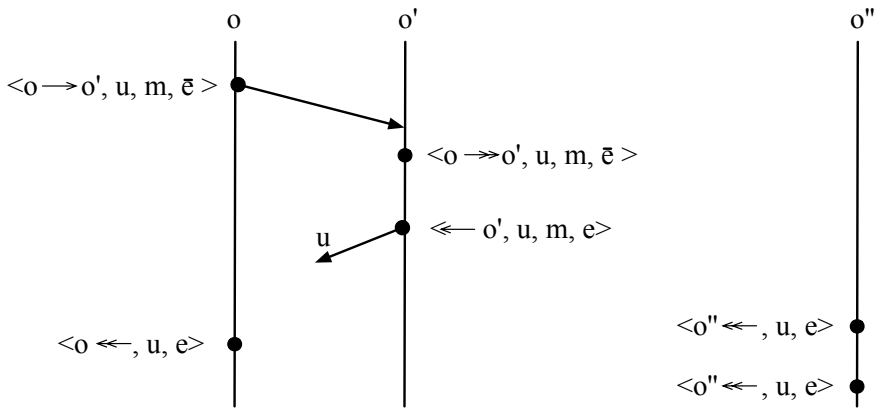

Figure 2: A method call cycle: object o calls a method m on object o' with *future* u. The arrows indicate message passing, and the bullets indicate events. The events on the left-hand side are visible to o, those in the middle are visible to o', and the ones on the right-hand side are visible to o''. Remark that there is an arbitrary delay between message receiving and reaction.

Events may be decomposed by the functions $\_.caller, \_.callee, \_.receiver : Ev \to Obj$, $\_.future : Ev \to Fid$, $\_.method : Ev \to Mtd$, $\_.class : Ev \to Cls$, $\_.args : Ev \to List[Data]$, and $\_.result : Ev \to Data$. For example, $\langle o \to o', u, m, \overline{e}\rangle.caller$ returns o. The decomposition functions are lifted to sequences in the standard way, for instance, $\_.result : Seq[Ev] \to Seq[Data]$, ignoring elements for which the decomposition is not defined. As in [15], we assume a total function $parent : Obj \to Obj$ where $parent(o)$ denotes the creator of o, such that $parent(\mathsf{main}) = \mathsf{main}$ and $parent(o) = \mathsf{null} \Leftrightarrow o = \mathsf{null}$. Equality is the only executable operation on object identities. Given the *parent* function, we may define an *ancestor* function $anc : Obj \to Set[Obj]$ by $anc(\mathsf{main}) \triangleq \{\mathsf{main}\}$ and $anc(o) \triangleq parent(o) \cup anc(parent(o))$ (where $o \neq \mathsf{main}$). We say that parent chains are *cycle free* if $o \notin anc(o)$ for all generated objects o, i.e., for $o \neq \mathsf{main}$.
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 $\langle o \to o', u, m, \overline{e}\rangle$ where $\overline{e}$ is the list of actual parameters. The event $\langle o \twoheadrightarrow o', u, m, \overline{e}\rangle$ reflects that o' starts execution of the method, and the event $\langle \leftarrow o', u, m, e\rangle$ reflects that the future is resolved upon method termination. The event $\langle o \twoheadleftarrow, u, e\rangle$ captures that object o fetches the result value from the resolved future. The creation of an object o' by an object o is reflected by the events $\langle o \to o', C, \overline{e}\rangle$ and $\langle o \twoheadrightarrow o', C, \overline{e}\rangle$, where o' is an instance of class C and $\overline{e}$ are the actual values for the class parameters. The first event reflects that o initiates the creation, and the latter 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 (a set O of objects) up to a given time is a finite sequence of type $Seq[Ev_O]$. The set of all communication events in O is defined by $Ev_O \triangleq IEv_O \cup IREv_O \cup CEv_O \cup CREv_O \cup$*

$NEv_O \cup NREv_O$, *including*

$$\begin{array}{ll} IEv_O \triangleq \{e : IEv \mid e.caller \in O\} & IREv_O \triangleq \{e : IREv \mid e.callee \in O\} \\ CEv_O \triangleq \{e : CEv \mid e.callee \in O\} & CREv_O \triangleq \{e : CREv \mid e.receiver \in O\} \\ NEv_O \triangleq \{e : NEv \mid e.caller \in O\} & NREv_O \triangleq \{e : NREv \mid e.callee \in O\} \end{array}$$

Definition 3 ***(Local communication histories)*** *The* local communication history *of an object o is a finite sequence of type $Seq[Ev_{\{o\}}]$. For a given history h, the local history of o is the subsequence of h visible to the object o, defined by $h/Ev_{\{o\}}$ and abbreviated h/o.*

In this manner, the local communication history reflects the local activity of each object. For a method call of m on object o' made by o, the events $\langle o \rightarrow o', u, m, \overline{e}\rangle$ and $\langle o \twoheadleftarrow, u, e\rangle$ are local to o. Correspondingly, the events $\langle o \twoheadrightarrow o', u, m, \overline{e}\rangle$ and $\langle \leftarrow o', u, m, e\rangle$ are local to o'. For object creation, the event $\langle o \rightarrow o', C, \overline{e}\rangle$ is local to o whereas $\langle o \twoheadrightarrow o', C, \overline{e}\rangle$ 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 $Ev_{\{o\}} \cap Ev_{\{o'\}} = \varnothing$ for $o \neq o'$, i.e., the two local histories h_o and $h_{o'}$ have *no* common events.
We define functions over the history to extract information, such as $fid : Seq[Ev] \rightarrow Set[Fid]$ extracting all identities of future occurring in a history, as follows:

$$\begin{array}{ll} fid(\varepsilon) \triangleq \{\mathsf{null}\} & fid(h \cdot \gamma) \triangleq fid(h) \cup fid(\gamma) \\ fid(\langle o \rightarrow o', u, m, \overline{e}\rangle) \triangleq \{u\} \cup fid(\overline{e}) & fid(\langle o' \twoheadrightarrow o, u, m, \overline{e}\rangle) \triangleq \{u\} \cup fid(\overline{e}) \\ fid(\langle \leftarrow o, u, m, e\rangle) \triangleq \{u\} \cup fid(e) & fid(\langle o \twoheadleftarrow, u, e\rangle) \triangleq \{u\} \cup fid(e) \\ fid(\langle o \rightarrow o', C, \overline{e}\rangle) \triangleq fid(\overline{e}) & fid(\langle o' \twoheadrightarrow o, C, \overline{e}\rangle) \triangleq fid(\overline{e}) \end{array}$$

where $\gamma : Ev$, and $fid(\overline{e})$ returns the set of future identities occurring in the expression list $\overline{e}$. Similarly the function $oid(h)$ returns all object identities occurring in a history. The definition can be found in Appendix. B.
The function $new_{ob} : Seq[Ev] \rightarrow Set[Obj \times Cls \times List[Data]]$ returns the set of created objects (each given by its object identity, associated class and class parameters) in a history:

$$\begin{array}{l} new_{ob}(\varepsilon) \triangleq \varnothing \\ new_{ob}(h \cdot \langle o \rightarrow o', C, \overline{e}\rangle) \triangleq new_{ob}(h) \cup \{o' : C(\overline{e})\} \\ new_{ob}(h \cdot \textbf{\textit{others}}) \triangleq new_{ob}(h) \end{array}$$

(where ***others*** matches all other events). The function $new_{id} : Set[Obj \times Cls \times List[Data]] \rightarrow Set[Obj]$ 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 visible methods are invoked for objects of given interfaces. 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. 2:

Definition 4 ***(Wellformed histories)*** *Let u: Fid, $h : Seq[Ev_O]$, the wellformedness predicate wf: $Seq[Ev_O] \times Set[Obj] \rightarrow Bool$ is defined by:*

$$\begin{array}{l} wf(\varepsilon, O) \triangleq true \\ wf(h \cdot \langle o \rightarrow o', u, m, \overline{e}\rangle, O) \triangleq wf(h, O) \wedge o \neq \textbf{\textit{null}} \wedge o' \neq \textbf{\textit{null}} \wedge u \notin fid(h) \\ wf(h \cdot \langle o' \twoheadrightarrow o, u, m, \overline{e}\rangle, O) \triangleq wf(h, O) \wedge o' \neq \textbf{\textit{null}} \wedge o \neq \textbf{\textit{null}} \wedge h/u = [\langle o' \rightarrow o, u, m, \overline{e}\rangle]/O \\ wf(h \cdot \langle \leftarrow o, u, m, e\rangle, O) \triangleq wf(h, O) \wedge h/u \ \textbf{\textit{ew}} \ \langle \_ \twoheadrightarrow o, u, m, \_\rangle \\ wf(h \cdot \langle o \twoheadleftarrow, u, e\rangle, O) \triangleq wf(h, O) \wedge u \in fid(h/o) \wedge agree(((h/u).result) \cdot e) \\ wf(h \cdot \langle o \rightarrow o', C, \overline{e}\rangle, O) \triangleq wf(h, O) \wedge o \neq \textbf{\textit{null}} \wedge parent(o') = o \wedge o' \notin oid(h) \\ wf(h \cdot \langle o' \twoheadrightarrow o, C, \overline{e}\rangle, O) \triangleq wf(h, O) \wedge o' \neq \textbf{\textit{null}} \wedge parent(o) = o' \wedge o \notin oid(h/(O \backslash \{o'\})) \cup oid(\overline{e}) \\ \qquad\qquad h/o = \varepsilon \wedge (o' \in O) \Rightarrow h/o' \ \textbf{\textit{ew}} \ \langle o' \rightarrow o, C, \overline{e}\rangle \end{array}$$

The wellformedness of appending an invocation event to the history captures the freshness of the generated future identity, i.e., $u \notin \mathit{fid}(h)$. For invocation reaction events, the corresponding invocation message must be visible in the past history if the caller is in O, i.e., $h/u = [\langle o' \rightarrow o, u, m, \overline{e}\rangle]/O$, where the projection h/u denotes the subhistory of all events with future u. For a completion event, the corresponding invocation reaction event must have appeared in the past history. And each method only returns the result to the appointed future once. For a completion reaction event, the future identity must be known from the past history, i.e., $u \in \mathit{fid}(h/o)$. Moreover, all completion events and completion reaction events connected with the same future, must have the same return result, i.e., $agree(((h/u).result) \cdot e)$. Remark that for object creation, the parent object and the created object synchronize, i.e., if o' is the parent of o and the history of O ends with the creation reaction event $\langle o' \twoheadrightarrow o, C, \overline{e}\rangle$, then the last event generated by o' is $\langle o' \rightarrow o, C, \overline{e}\rangle$. We have chosen to include information of the called method in completion events, in order to make history specification more easily readable. And in order to make the definition of wellformedness constructive, we have chosen to include information of the caller in invocation and invocation reaction events. We then avoid quantifiers to formalize the presence of required prior events e_v by formulating conditions of the form $(o' \in O \Rightarrow e_v \in h)$.

Invariant Reasoning. The communication history abstractly captures the system state at any point in time [9,10]. Therefore partial correctness properties of a system may be specified by 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-closed) set of possible histories, expressing safety properties [3]. In a distributed and concurrent system, it is desirable to reason about one component at a time. According to this, we develop a compositional reasoning system so that programmers can reason about one class at a time and compose history invariants at need.
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.

$$
\begin{array}{l}
\langle\!\langle\, m(\overline{x})\ \{\mathbf{var}\ \overline{y}; s\}\,\rangle\!\rangle \triangleq\ m'(\overline{x}, \mathsf{caller}, \mathsf{destiny})\ \{\mathbf{var}\ \overline{y}, \mathsf{return}; \mathcal{H} := \mathcal{H} \cdot \langle \mathsf{caller} \twoheadrightarrow \mathsf{this}, \mathsf{destiny}, m, \overline{x}\rangle;\\
\quad \langle\!\langle\, s\,\rangle\!\rangle; \mathcal{H} := \mathcal{H} \cdot \langle \leftarrow \mathsf{this}, \mathsf{destiny}, m, \mathsf{return}\rangle; \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\!\dashv 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\, \mathit{fr} := o!m(\overline{e})\,\rangle\!\rangle \triangleq\ \mathit{fr}' := \mathbf{some}; \mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \rightarrow o, \mathit{fr}', m, \overline{e}\rangle; \mathit{fr} := \mathit{fr}'; \mathbf{assume}\ \ \mathit{wf}(\mathcal{H})\\
\langle\!\langle\, v := \mathit{fr}?\,\rangle\!\rangle \triangleq\ v' := \mathbf{some}; \mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \twoheadleftarrow, \mathit{fr}, v'\rangle; v := v'; \mathbf{assume}\ \ \mathit{wf}(\mathcal{H})\\
\langle\!\langle\, \mathbf{await}\ \ v := \mathit{fr}?\,\rangle\!\rangle \triangleq\ \langle\!\langle\, \mathbf{suspend}; v := \mathit{fr}?\,\rangle\!\rangle\\
\langle\!\langle\, \mathbf{await}\ \ v := o.m(\overline{e})\,\rangle\!\rangle \triangleq\ \langle\!\langle\, \mathit{fr}' := o!m(\overline{e}); \mathbf{await}\ \ v := \mathit{fr}'?\,\rangle\!\rangle\\
\langle\!\langle\, v := o.m(\overline{e})\,\rangle\!\rangle \triangleq\ \langle\!\langle\, \mathit{fr}' := o!m(\overline{e})\,\rangle\!\rangle; \mathbf{if}\ \ o = \mathsf{this}\ \ \mathbf{then}\ \ v' := m'(\overline{e}, \mathsf{this}, \mathit{fr}')\\
\qquad \mathbf{else}\ \ v' := \mathbf{some}\ \ \mathbf{fi}; \mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \twoheadleftarrow, \mathit{fr}', v'\rangle; v := v'; \mathbf{assume}\ \ \mathit{wf}(\mathcal{H})\\
\langle\!\langle\, v := \mathbf{new}\ \ C(\overline{e})\,\rangle\!\rangle \triangleq\ v' := \mathbf{some};\ \ \mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \rightarrow v', C, \overline{e}\rangle; v := v'; \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 3: The *ABS* semantic encoding. Here, C is the class enclosing the encoded statements, I_C is the class invariant, and fr', h', v' denote fresh variables. The assumptions reflect that the history of an execution is wellformed, that suspension maintains the local invariant, and that a waiting condition holds when control returns.

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 [16]. 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 dynamic logic 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. Method 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. Similarly, object creation is captured by a message from the parent object to the generated object, and captured by two events.

4.1 Semantic Definition by a Syntactic Encoding

We consider a simple *sequential* language, *SEQ*, 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 [4, 5, 25]. 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

$$\textbf{assume}\ b$$

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

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

Method definitions are of the form $m'(\overline{x}, \mathsf{caller}, \mathsf{destiny})\ body$, where $body$ is of the form $\{\textbf{var}\ \overline{y}; s\}$. 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. Since caller and destiny are not part of the *ABS* language, they appear as explicit parameters in the encoding.
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. 3. 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 $wf(\mathcal{H})$ abbreviate $wf(\mathcal{H}, \{\mathsf{this}\})$.
When an instance of $m(\overline{x})$ starts execution, the history $\mathcal{H}$ is extended by an invocation reaction event: $\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{caller} \twoheadrightarrow \mathsf{this}, \mathsf{destiny}, m, \overline{x} \rangle$. Process termination is reflected by appending a completion event: $\mathcal{H} := \mathcal{H} \cdot \langle \leftarrow \mathsf{this}, \mathsf{destiny}, m, \mathsf{return} \rangle$, where return is the return value of m. When invoking some method $o!m(\overline{e})$, the history is extended with an invocation event: $\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \rightarrow o, fr, m, \overline{e} \rangle$, and fetching the result value e is encoded by $\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \twoheadleftarrow, fr, e \rangle$.
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 formalizes 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.
In the encoding of *object creation*, non-deterministic assignment is used to construct object identifiers, and the history is extended with the creation event. The final wellformedness assumption ensures the parent relationships and uniqueness of the generated identifiers. 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 $v := o.m(\overline{e})$. 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. 4. Condition (1) applies to the initialization block *init* of C, ensuring that the invariant is established upon

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

Figure 4: 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 P/Q for the implementation. Notice that $\mathsf{this} \neq \mathsf{null}$ follows from each premise.

$wlp(m(\overline{x})\ \{\mathbf{var}\ \overline{y}; s\}, Q) \triangleq wlp_{SEQ}(m'(\overline{x}, \mathsf{caller}, \mathsf{destiny})\ \{\mathbf{var}\ \overline{y}, \mathsf{return};$
$\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{caller} \twoheadrightarrow \mathsf{this}, \mathsf{destiny}, m, \overline{x}\rangle; s; \mathcal{H} := \mathcal{H} \cdot \langle \leftarrow \mathsf{this}, \mathsf{destiny}, m, \mathsf{return}\rangle\}, wf(\mathcal{H}) \Rightarrow Q)$
for $\overline{y} \notin FV[Q]$
$wlp(\mathbf{suspend}, Q) \triangleq I_C(\overline{w}, \mathcal{H}) \wedge wf(\mathcal{H}) \wedge \forall \overline{w}, h'\,.\,(I_C(\overline{w}, \mathcal{H}) \wedge wf(\mathcal{H}) \Rightarrow Q)^{\mathcal{H}}_{\mathcal{H}\vdash h'}$
$wlp(\mathbf{await}\ \ b, Q) \triangleq I_C(\overline{w}, \mathcal{H}) \wedge wf(\mathcal{H}) \wedge \forall \overline{w}, h'\,.\,(I_C(\overline{w}, \mathcal{H}) \wedge wf(\mathcal{H}) \wedge b \Rightarrow Q)^{\mathcal{H}}_{\mathcal{H}\vdash h'}$
$wlp(fr := o!m(\overline{e}), Q) \triangleq \forall fr'\,.\,(wf(\mathcal{H}) \Rightarrow Q)^{fr,\mathcal{H}}_{fr',\mathcal{H}\cdot\langle \mathsf{this}\to o, fr', m, \overline{e}\rangle}$
$wlp(v := fr?, Q) \triangleq \forall v'\,.\,(wf(\mathcal{H}) \Rightarrow Q)^{v,\mathcal{H}}_{v',\mathcal{H}\cdot\langle \mathsf{this}\leftarrow, fr, v'\rangle}$
$wlp(\mathbf{await}\ \ v := fr?, Q) \triangleq wlp(\mathbf{suspend}; v := fr?, Q)$
$wlp(\mathbf{await}\ \ v := o.m(\overline{e}), Q) \triangleq wlp(fr' := o!m(\overline{e}); \mathbf{await}\ \ v := fr'?, Q)$
$wlp(v := o.m(\overline{e}), Q) \triangleq \forall fr''\,.\,\mathbf{if}\ \ o = \mathsf{this}\ \mathbf{then}\ \ (wlp_{SEQ}(v' := m'(\overline{e}, \mathsf{this}, fr'), Q'))^{fr',\mathcal{H}}_{fr'',\mathcal{H}\cdot\langle \mathsf{this}\to o, fr'', m, \overline{e}\rangle}$
$\mathbf{else}\ \ (\forall v'\,.\,Q')^{fr',\mathcal{H}}_{fr'',\mathcal{H}\cdot\langle \mathsf{this}\to o, fr'', m, \overline{e}\rangle}$
where $Q' = (wf(\mathcal{H}) \Rightarrow Q)^{v,\mathcal{H}}_{v',\mathcal{H}\cdot\langle \mathsf{this}\leftarrow, fr', v'\rangle}$
$wlp_{SEQ}(v' := m'(\overline{e}), Q) \triangleq (wlp_{SEQ}(m'(\overline{x})\ body, Q^{\overline{y}, v'}_{\overline{y}', \mathsf{return}}))^{\overline{x}, \overline{y}'}_{\overline{e}, \overline{y}}$
where $\overline{y}$ are the local variables of the caller (including caller), and $\overline{y}'$ are fresh logical variables
$wlp_{SEQ}(m'(\overline{x})\ body, Q) \triangleq wlp(body, Q)$
$wlp(v := \mathbf{new}\ \ C(\overline{e}), Q) \triangleq \forall v'\,.\,(wf(\mathcal{H}) \Rightarrow Q)^{v,\mathcal{H}}_{v',\mathcal{H}\cdot\langle \mathsf{this}\to v', C, \overline{e}\rangle}$
$wlp(\mathbf{var}\ \overline{y}, Q) \triangleq \forall \overline{y}\,.\,Q$
$wlp(\mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{fi}, Q) \triangleq \mathbf{if}\ b\ \mathbf{then}\ wlp(s_1, Q)\ \mathbf{else}\ wlp(s_2, Q)$
$wlp(s_1; s_2, Q) \triangleq wlp(s_1, wlp(s_2, Q))$
$wlp(\mathbf{skip}, Q) \triangleq Q$
$wlp(\mathbf{abort}, Q) \triangleq true$
$wlp(v := e, Q) \triangleq Q^v_e$
$wlp(\mathbf{return}\ e, Q) \triangleq Q^{\mathsf{return}}_e$

Figure 5: Weakest liberal preconditions for *ABS* statements. We let fr', fr'', h', and v', denote fresh variables, and let wlp_{SEQ} denote weakest liberal precondition for *SEQ*.

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, where P is the precondition and Q is the postcondition (given by a user specification).
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:

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

In addition, let $R^{\overline{x}}_{\overline{e}}$, where $\overline{x}$ and $\overline{e}$ are of the same length, denote R where every free occurrence of each $x_i \in \overline{x}$ is replaced by e_i. Weakest liberal preconditions for the different ABS statements are summarized in Fig. 5. These are straightforwardly derived from the encoding in Fig. 3, where the quantifiers reflect 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 Dynamic Logic

The principle of dynamic logic is the formulation of statements about program behavior by integrating programs and formulae within a single language. The formula $\psi \Rightarrow \langle s \rangle \phi$ expresses that a program s, starting in a state where ψ is true, will terminate in a state in which ϕ holds. The formula $\psi \Rightarrow [s]\phi$ does not demand termination and describes that if s is executed in a state where ψ holds and the execution terminates, then ϕ holds in the final state. We are only concerned with partial correctness reasoning, where termination is not required, thus the diamond modality $\langle \cdot \rangle$ will not be used in our context.
The state of a program evolves by the execution of assignments. Accordingly, substitution in the formulae is required. In this paper, we develop a forward reasoning system where the effect of substitutions in the precondition is delayed. This is achieved by the *update* mechanism [6]. An *update* $\{v := t\}$ on an expression e, i.e. $\{v := t\}\ e$, can be evaluated as e^v_t. A dynamic formula $[v := e; s]\phi$ where assignment is the first statement can be rewritten to $\{v := \tau(e)\}[s]\phi$, in which τ is a function evaluating the (side-effect free) terminal expression e and s is the rest of the program. Since an *update* can only be applied to terms and formulae that do not contain programs, the effect of *update* on the formula $[s]\phi$ is accumulated and delayed until s has been completely and symbolically executed. The parallel update $\{v_1 := \tau(e_1)||...||v_n := \tau(e_n)\}$ represents the accumulated *updates*, which will be applied simultaneously on the formula/term. Upon conflict when $v_i = v_j$ but $\tau(e_i) \neq \tau(e_j)$ where $i < j$, the later update $\{v_j := \tau(e_j)\}$ wins.
A sequent $\psi_1, ..., \psi_n \vdash \phi_1, ..., \phi_n$ represents a set of assumption formulae $\psi_1, ..., \psi_n$ and a set of formulae $\phi_1, ..., \phi_n$ to be proved. We say a sequent is *valid* if at least one formula ϕ_i follows from the assumptions. Thus, a sequent can be interpreted as $\psi_1 \wedge ... \wedge \psi_n \Rightarrow \phi_1 \vee ... \vee \phi_n$.
Sequent calculus rules :

$$\mathsf{ruleName}\ \frac{\Gamma_1 \vdash \Delta_1 \quad ... \quad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}$$

always have zero, one, or more sequents as premises on the top and one sequent as conclusion at the bottom. For simplicity, we use capital Greek letters to denote (possibly empty) sets of formulae. Semantically, a rule states the validity of all n premises implies the validity of the conclusion. Operationally, rules are typically applied bottom-up, reducing the provability of the conclusion to that of the premises. In Fig. 6 we present a selection of the sequent rules.

$$\mathsf{andRight}\ \frac{\Gamma \vdash \phi, \Delta \quad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta} \qquad \mathsf{transitivity}\ \frac{\Gamma \vdash \psi, \Delta \quad \Gamma \vdash \psi \Rightarrow \phi, \Delta}{\Gamma \vdash \phi, \Delta}$$

$$\mathsf{impRight}\ \frac{\Gamma, \phi \vdash \psi, \Delta}{\Gamma \vdash \phi \Rightarrow \psi, \Delta} \qquad \mathsf{allRight}\ \frac{\Gamma \vdash \phi^v_c, \Delta}{\Gamma \vdash \forall v\,.\,\phi, \Delta}\ \text{(with a fresh constant c which is not used in } \Gamma)$$

Figure 6: Selected sequent rules.

Since dynamic logic formulae and weakest liberal preconditions are closely related, namely $\psi \Rightarrow [s]\phi$ is the same as $\psi \Rightarrow wlp(s, \phi)$, we derive dynamic logic formulae for each ABS

$$
\begin{array}{lll}
(1) & \Gamma_C \vdash & (\mathcal{H} = \langle \mathit{parent}(\mathsf{this}) \twoheadrightarrow \mathsf{this}, C, \overline{e}\rangle) \Rightarrow [\mathit{init}_C] I_C(\overline{w}, \mathcal{H}) \\
(2) & \Gamma_C \vdash & (\mathit{wf}(\mathcal{H}) \wedge I_C(\overline{w}, \mathcal{H})) \Rightarrow [\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{caller} \twoheadrightarrow \mathsf{this}, \mathsf{destiny}, m, \overline{x}\rangle; \mathit{body}_m; \\
 & & \mathcal{H} := \mathcal{H} \cdot \langle \twoheadleftarrow \mathsf{this}, \mathsf{destiny}, m, \mathsf{return}\rangle](\mathit{wf}(\mathcal{H}) \Rightarrow I_C(\overline{w}, \mathcal{H})) \\
(3) & \Gamma_C \vdash & (\mathit{wf}(\mathcal{H}) \wedge P(\overline{w}, \mathcal{H})) \Rightarrow [\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{caller} \twoheadrightarrow \mathsf{this}, \mathsf{destiny}, m, \overline{x}\rangle; \mathit{body}_m; \\
 & & \mathcal{H} := \mathcal{H} \cdot \langle \twoheadleftarrow \mathsf{this}, \mathsf{destiny}, m, \mathsf{return}\rangle](\mathit{wf}(\mathcal{H}) \Rightarrow Q(\overline{w}, \mathcal{H}))
\end{array}
$$

Figure 7: *ABS* verification conditions for a class C in dynamic logic, derived from Fig. 4. Conditions of form (1) and (2) verify the class invariant I_C and conditions of form (3) verify user defined pre/post specification pairs P/Q. Here Γ_C denotes the set of method specifications in C (including invariance I_C), and which are proved by verification conditions (2) and (3).

$$
\mathsf{awaitCond}\ \frac{\begin{array}{l}\vdash I_C(\overline{w}, \mathcal{H}) \wedge \mathit{wf}(\mathcal{H}) \\ \vdash \forall \overline{w}, h' \,.\, \{\mathcal{H} := \mathcal{H} \mapsto h'\}\ (I_C(\overline{w}, \mathcal{H}) \wedge \mathit{wf}(\mathcal{H}) \wedge \tau(e) \Rightarrow [s]\phi)\end{array}}{\vdash [\mathbf{await}\ e;\ s]\phi}
$$

$$
\mathsf{async}\ \frac{\vdash \forall \mathit{fr}' \,.\, \{\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \to o, \mathit{fr}', m, \overline{e}\rangle || \ \mathit{fr} := \mathit{fr}'\}\ (\mathit{wf}(\mathcal{H}) \Rightarrow [s]\phi)}{\vdash [\mathit{fr} = o\,!\,\mathtt{m}(\overline{e});\ s]\phi}
$$

$$
\mathsf{get}\ \frac{\vdash \forall v' \,.\, \{\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \twoheadleftarrow, \mathit{fr}, v'\rangle || \ v := v'\}\ (\mathit{wf}(\mathcal{H}) \Rightarrow [s]\phi)}{\vdash [v := \mathit{fr}?;\ s]\phi}
$$

$$
\mathsf{new}\ \frac{\vdash \forall v' \,.\, \{\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \to v', C, \overline{e}\rangle || \ v := v'\}\ (\mathit{wf}(\mathcal{H}) \Rightarrow [s]\phi)}{\vdash [v\ \ = \mathbf{new}\ C(\overline{e});\ s]\phi}
$$

$$
\mathsf{syncIntra}\ \frac{\begin{array}{l}\Gamma \bigwedge_i m(\overline{x})\ \mathbf{sat}\ (P_i, Q_i) \vdash (\exists \mathit{fr}', h', \overline{w}', v' \,.\, \mathcal{H}\ \mathbf{ew} \langle \mathsf{this} \twoheadleftarrow, \mathit{fr}', v\rangle \wedge \\ R^{\mathcal{H}, \overline{w}, v}_{h', \overline{w}', v'} \bigwedge_i ((P_i)^{\overline{x}, \mathsf{caller}, \mathsf{destiny}, \overline{w}, \mathcal{H}}_{\overline{e}, \mathsf{this}, \mathit{fr}', \overline{w}', h''} \Rightarrow (Q_i)^{\mathsf{caller}, \mathsf{destiny}, \mathsf{return}, \mathcal{H}}_{\mathsf{this}, \mathit{fr}', v, \mathit{pop}(\mathcal{H})})) \Rightarrow [s]\phi\end{array}}{\Gamma \vdash R \Rightarrow [v := \mathtt{m}(\overline{e});\ s]\phi}
$$

Figure 8: Dynamic logic rules for basic *ABS* communication statements. fr', h', v' are fresh variables. The last rule assumes neither P, R, nor e, refer to return, and $h'' = h' \cdot \langle \mathsf{this} \to \mathsf{this}, \mathit{fr}', m, \overline{e}^{w, v}_{w', v'}\rangle$.

statement from Section 4.2. In Fig. 7 we present the sequents for verifying methods and the initialization of classes, where Γ_C denotes the set of method specifications and invariance of the class C. These three sequents are derived from the ones in Fig. 4, respectively.
The first statement s_1 in the formula $[s_1; s]\phi$ determines which sequent rule to apply, typically reducing the formula to $[s]\phi$ and the reduction continues repeatedly until the remaining statements s are reduced. In Fig. 8, we present the sequent rules for the await condition (awaitCond), asynchronous message call (async), get statement for synchronously fetching the result from the future (get), object creation (new) and synchronous intra-call (syncIntra). The rule for **await** statement gives two premises, one dealing with the pre-state and one dealing with the post-state, both reflecting that the invariant must hold. The sequent rule async says that the validity of the reduced formula $[s]\phi$ should be proved under the condition of the updated history, the generated future and the assumption of history wellformedness. The rules get and new follow the same pattern as async. For synchronous intra-calls we allow local proofs of the called method to be reused: In the sequent rule syncIntra, the assumption uses a selection of pre- and post condition pairs of the called method (typically those already verified). At the right-hand side of the sequent, we need to prove the reduced formula $[s]\phi$ under the assumption of the

update	$\{v := t\}\ e$	$=$	e_t^v
abort	$[\mathbf{abort};\ s]\phi$	$=$	$true$
skip	$[\mathbf{skip};\ s]\phi$	$=$	$[s]\phi$
return	$[\mathbf{return}\ e;]\phi$	$=$	$\{\mathsf{return} := \tau(e)\}\ \phi$
assign	$[v := e;\ s]\phi$	$=$	$\{v := \tau(e)\}\ [s]\phi$
declInit	$[T\ v = e;\ s]\phi$	$=$	$[v' := e;\ s_{v'}^v]\phi$
declNoInit	$[T\ v;\ s]\phi$	$=$	$[v' := \mathsf{default}_T;\ s_{v'}^v]\phi$
ifElse	$[\mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{fi};\ s]\phi$	$=$	$\mathbf{if}\ b\ \mathbf{then}\ [s_1; s]\phi\ \mathbf{else}\ [s_2; s]\phi\ \mathbf{fi}$
suspend	$[\mathbf{suspend};\ s]\phi$	$=$	$[\mathbf{await}\ true;\ s]\phi$
awaitFuture	$[\mathbf{await}\ v := fr?;\ s]\phi$	$=$	$[\mathbf{suspend}; v := fr?;\ s]\phi$
awaitCall	$[\mathbf{await}\ v := o.\mathtt{m}(\overline{e});\ s]\phi$	$=$	$[fr' := o!\mathtt{m}(\overline{e});\ \mathbf{await}\ v := fr'?;\ s]\phi$
syncCall	$[v := o.\mathtt{m}(\overline{e});\ s]\phi$	$=$	$\mathbf{if}\ o = \mathsf{this}\ \mathbf{then}\ [v := \mathtt{m}(\overline{e});\ s]\phi$
			$\mathbf{else}\ [fr' := o!\mathtt{m}(\overline{e});\ v := fr'?;\ s]\phi\ \mathbf{fi}$

Figure 9: Equivalent substitutions for other *ABS* statements. Here τ is a function evaluating an expression e, primes denote fresh variables, ϕ is the postcondition, s is the remaining program yet to be executed, $s_{v'}^v$ is s with all (free) occurrences of v replaced by v', and $\mathsf{default}_T$ is the default value defined for type T.

starting condition R (can be none) and the chosen method contract(s) with the substitution of actual method parameters.
The rules for the rest of the *ABS* statements can be defined as substitution rules. For instance, $[\mathbf{suspend}; s]\phi$ can be rewritten to $[\mathbf{await}\ true; s]\phi$, and we can then apply the sequent rule awaitCond. The substitution rules are introduced in Fig. 9 with the following syntax

$$\mathsf{ruleName}\ \phi = \phi'$$

which expresses the formula ϕ can be rewritten to ϕ'.

4.4 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 [27, 28]. 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 $\forall h \leqslant h_o\,.\,P(h)$.
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

$$(I_{\mathsf{this}:C(\overline{cp})}(\mathcal{H}) \wedge P) \vdash [s](Q \Rightarrow I_{\mathsf{this}:C(\overline{cp})}(\mathcal{H}))$$

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

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

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

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

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., $\mathsf{main} \notin new_{id}(new_{ob}(H))$.

5 Reader Writer Example

In this section we study a fair implementation of the reader/writer problem in *ABS*. We define safety invariants and illustrate the reasoning system by verification of these invariants.

```
interface DB{
   String read(Int key);
   Unit write(Int key, String element);}

interface RWinterface{
   Unit openR();
   Unit closeR();
   Unit openW();
   Unit closeW();
   String read(Int key);
   Unit write(Int key, String element);}

class RWController implements RWinterface{
   DB db; Set<Caller> readers := EmptySet;
   Caller writer := None; Int pr := 0;

   {db := new DataBase();}

   Unit openR(){
      await writer = None;
      readers := insertElement(readers, caller);}

   Unit closeR(){
      readers := remove(readers, caller); }

   Unit openW(){
      await writer = None; writer := caller;
      readers := insertElement(readers, caller);}

   Unit closeW(){
      await writer = caller; writer := None;
      readers := remove(readers, caller);}

   String read(Int key){
      Fut<String> fr; String data;
      await contains(readers, caller);
      pr := pr + 1; fr := db!read(key); await data := fr?; pr := pr −1;
      return data;}

   Unit write(Int key, String value){
      await caller = writer && pr = 0 && (readers = EmptySet ||
      (contains(readers, writer) && size(readers) = 1));
      db.write(key, value);}
}
```

Figure 10: Implementation of the fair reader/writer controller. See Appendix. C.1 for full implementation including data type definitions and implementation of the DataBase class.

5.0.1 Implementation

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. 10, 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 perform 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 Appendix. C.1. For simplicity we have omitted **`return`** *void* statements.

5.0.2 Specification and Verification

For the RWController class in Fig. 10, 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 $Readers : Seq[Ev] \rightarrow Set[Obj]$:

$$
\begin{array}{l}
Readers(\varepsilon) \triangleq \varnothing \\
Readers(h \cdot \langle \leftarrow \mathsf{this}, fr', \mathsf{openR}, \_ \rangle) \triangleq Readers(h) \cup \{irev(h, fr').caller\} \\
Readers(h \cdot \langle \leftarrow \mathsf{this}, fr', \mathsf{openW}, \_ \rangle) \triangleq Readers(h) \cup \{irev(h, fr').caller\} \\
Readers(h \cdot \langle \leftarrow \mathsf{this}, fr', \mathsf{closeR}, \_ \rangle) \triangleq Readers(h) \backslash \{irev(h, fr').caller\} \\
Readers(h \cdot \langle \leftarrow \mathsf{this}, fr', \mathsf{closeW}, \_ \rangle) \triangleq Readers(h) \backslash \{irev(h, fr').caller\} \\
Readers(h \cdot \boldsymbol{others}) \triangleq Readers(h)
\end{array}
$$

where ***others*** matches all events not matching any of the above cases. The function $irev(h, fr')$ extracts the invocation reaction event, containing the future fr', from the history h. Definition of *irev* can be found in Appendix. C.2. 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 Appendix. C.3. Next we define $Reading : Seq[Ev] \rightarrow Nat$ by:

$$Reading(h) \triangleq \#(h/\langle \mathsf{this} \rightarrow \mathsf{db}, \_, \mathsf{read}, \_ \rangle) - \#(h/\langle \mathsf{this} \twoheadleftarrow, Futures(h, \mathsf{db}, \mathsf{read}), \_ \rangle)$$

where $Futures(h, \mathsf{db}, \mathsf{read}) \triangleq ToSet((h/\langle \mathsf{this} \rightarrow \mathsf{db}, \_, \mathsf{read}, \_ \rangle).future)$. Thus the function $Reading(h)$ computes the difference between the number of initiated calls to db.read and the corresponding reaction events. The function *Writing(h)* follows the same pattern over calls to db.write, the definition can be found in Appendix. C.4.
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

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

where $\{\mathsf{writer}\} = \varnothing$ if writer = 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

$$\frac{\begin{array}{l}\vdash (wf(\mathcal{H}) \wedge I_C) \Rightarrow \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin}\}(I_C \wedge wf(\mathcal{H})) \\ \vdash (wf(\mathcal{H}) \wedge I_C) \Rightarrow \forall \overline{w}, h' \,.\, \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin} \mapsto h'\}(I_C \wedge wf(\mathcal{H}) \wedge (\text{writer} = \text{None}) \Rightarrow \\ \quad \{\text{readers} := \text{insertElement}(\text{readers}, \text{caller})||\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{end}\}(wf(\mathcal{H}) \Rightarrow I_C))\end{array}}{\begin{array}{l}\vdash (wf(\mathcal{H}) \wedge I_C) \Rightarrow [\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin};\ \text{await writer} = \text{None}; \\ \quad \text{readers} := \text{insertElement}(\text{readers}, \text{caller}); \mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{end}](wf(\mathcal{H}) \Rightarrow I_C)\end{array}}$$

Figure 11: Verification details for the body of method openR with respect to the invariant I_1 : $Readers(\mathcal{H})$ = readers. $\mathcal{H}_{begin}$: $\langle \text{caller} \twoheadrightarrow \text{this}, \text{destiny}, \text{openR}, \overline{x} \rangle$, $\mathcal{H}_{end}$: $\langle \leftarrow \text{this}, \text{destiny}, \text{openR}, \text{return} \rangle$. Remark that the ordering between readers is not concerned. The verification condition follows from insertElement(x,s)= s∪{x}.

the same value as readers. The predicate $OK : Seq[Ev] \rightarrow Bool$ is defined inductively over the history by:

$$\begin{array}{ll} OK(\varepsilon) \triangleq true & \\ OK(h \cdot \langle \leftarrow \text{this}, fr', \text{openR}, \_ \rangle) \triangleq OK(h) \wedge \#Writers(h) = 0 & (1) \\ OK(h \cdot \langle \leftarrow \text{this}, fr', \text{openW}, \_ \rangle) \triangleq OK(h) \wedge \#Writers(h) = 0 & (2) \\ OK(h \cdot \langle \text{this} \rightarrow \text{db}, fr', \text{write}, \_ \rangle) \triangleq OK(h) \wedge Reading(h) = 0 \wedge \#Writers(h) = 1 & (3) \\ OK(h \cdot \langle \text{this} \rightarrow \text{db}, fr', \text{read}, \_ \rangle) \triangleq OK(h) \wedge Writing(h) = 0 & (4) \\ OK(h \cdot \textbf{\textit{others}}) \triangleq OK(h) & \end{array}$$

Here, conditions (1) and (2) reflect the *fairness condition*: invocations of openR and openW compete on equal terms for the guard writer = null, which equals $Writers(\mathcal{H}) = \varnothing$ 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. Condition (3) captures the guard in write: when invoking db.write, there cannot be any pending calls to db.read. Correspondingly, Condition (4) expresses that when invoking db.read, there is no incomplete writing operation. The invariant I implies that no reading and writing *activity* happens simultaneously:

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

Notice that I is (by construction) prefix-closed, thus this property holds at all times, and expresses the desired mutual exclusion of reading and writing at all times (not only at release points).
As a verification example, the successful verification of method openR with respect to the invariant I_1 : $Readers(\mathcal{H})$ = 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. 7. The complete verification of this case study can be found in Appendix. C.5.
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.

6 Related and Future Work

Related 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 [2]. The present approach follows the line of work based on communication histories to model object communication events in a distributed setting [7, 8, 19]. Objects are concurrent and interact solely by method calls and futures, 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 [17]. History invariants can be naturally included in interface definitions, 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.

Olderog and Apt consider transformation of program statements preserving semantical equivalence [24]. This approach is extended in [13] to a general methodology for transformation of language constructions resulting in sound and relative complete proof systems. The approach resembles our encoding, but is non-compositional in contrast to our work. In particular, extending the approach of [13] to multi-threaded systems will in general require interference freedom tests.

The four-event semantics applied in the current paper is based on [15] which leads to *disjoint alphabets* for different objects. 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 a history invariant describing the external behavior of an object of the class, and as composition rule, similar to previous approaches [27, 28], gives global history invariants for a system or subsystem. Dylla and Ahrendt [2] present a compositional verification system in dynamic logic for *Creol* but without futures. The denotational Creol semantics features the similar four communication events, there called 'invoc', 'begin', 'end', and 'comp'. However, the reasoning system [2] is based on the two-event semantics of [16], which requires more complex rules than the present one. In the current work, we revise the four-event semantics [15] to deal with futures [18], allowing several readings of the same future, possibly by different objects.

A reasoning system for futures has been presented in [12], using a combination of global and local invariants. Futures are treated as visible objects rather than reflected by events in histories. In contrast to our work, global reasoning is obtained by means of global invariants, and not by compositional rules. Thus the environment of a class must be known at verification time.

Future work. We believe that our verification system is suitable for implementation within the KeY framework. Having support for (semi-)automatic verification, such an implementation will be valuable when developing larger case studies for the *ABS* programs. 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., [11, 22] 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 with futures. Compositional reasoning is facilitated by expressing object properties in terms of observable interaction between the object and its environment, recorded on communication histories. Method invocation, writing/reading of futures, and object creation, are reflected by two kinds of events each, in such a way that different objects have different alphabets. This is essential for obtaining a simple reasoning system. 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 dynamic logic system is derived from *wlp*. 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 the reader-writer example.

Acknowledgements. The authors would like to thank Wolfgang Ahrendt and Richard Bubel for fruitful discussions on the subject.

References

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

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

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

[4] 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.

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

[6] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer, 2007.

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

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

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

[10] O.-J. Dahl. *Verifiable Programming*. Intl. Series in Computer Science. Prentice Hall, 1992.

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

[12] F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 316–330. Springer, Mar. 2007.

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

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

[15] C. C. Din, J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. *Journal of Logic and Algebraic Programming*, in press, 2012. issn = "1567-8326", doi = "10.1016/j.jlap.2012.01.003".

[16] 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.

[17] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. *Journal of Logic and Algebraic Programming*, 79(7):578–607, 2010.

[18] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 142–164. Springer, 2011.

[19] C. A. R. Hoare. *Communicating Sequential Processes*. Intl. Series in C.S., Prentice Hall, 1985.

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

[21] A. S. A. Jeffrey and J. Rathke. Java Jr.: Fully abstract trace semantics for a core Java language. In *Proc. European Symposium on Programming*, *LNCS* 3444, pages 423–438. Springer, 2005.

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

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

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

[25] C. Pierik and F. S. d. Boer. A Syntax-Directed Hoare Logic for Object-Oriented Programming Concepts. Formal Methods for Open Object-Based Distributed Systems (FMOODS) VI. Volume 2884 of LNCS, pages 64–78. Springer, 2003.

[26] J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In T. D'Hondt, editor, *European Conference on Object-Oriented Programming (ECOOP 2010)*, volume 6183 of *LNCS*, pages 275–299. Springer, June 2010.

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

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

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	::=	$\mathbf{data}\ D\ = [Co(T^*)]^*$	data type declaration
F	::=	$\mathbf{def}\ T\ fn([T\ x]^*) == rhs$	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
		$\mid \mathbf{case}\ e\{b^*\}$	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 construct and infix operator are not included in the syntax above. We use + and − for the usual numbers, **and** and **or** for booleans, and = for equality.

B Auxiliary Functions

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:

$$oid(\varepsilon) \triangleq \{\mathsf{null}\} \qquad oid(h \vdash \gamma) \triangleq oid(h) \cup oid(\gamma)$$

$$oid(\langle o \to o', u, m, \overline{e}\rangle) \triangleq \{o, o'\} \cup oid(\overline{e}) \qquad oid(\langle o' \twoheadrightarrow o, u, m, \overline{e}\rangle) \triangleq \{o', o\} \cup oid(\overline{e})$$

$$oid(\langle \leftarrow o, u, m, e\rangle) \triangleq \{o\} \cup oid(e) \qquad oid(\langle o \twoheadleftarrow, u, e\rangle) \triangleq \{o\} \cup oid(e)$$

$$oid(\langle o \to o', C, \overline{e}\rangle) \triangleq \{o, o'\} \cup oid(\overline{e}) \qquad oid(\langle o' \twoheadrightarrow o, C, \overline{e}\rangle) \triangleq \{o', o\} \cup oid(\overline{e})$$

where $\gamma : Ev$, and $oid(\overline{e})$ returns the set of object identifiers occurring in the expression list $\overline{e}$.

C Reader Writer Example in ABS

C.1 Complete implementation

```
module RW;


data Caller = Readers | Writers | None;


interface RWinterface{
   Unit openR();
   Unit closeR();
   Unit openW();
   Unit closeW();
   String read(Int key);
   Unit write(Int key, String element);
}

interface DB{
   String read(Int key);
   Unit write(Int key, String element);
}

class DataBase implements DB{
   Map<Int, String> map := EmptyMap;

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

   Unit write(Int key, String element){
      map := put(map, key, element);
   }
}

class RWController implements RWinterface{
   DB db; Set<Caller> readers := EmptySet;
   Caller writer := None; Int pr := 0;

   {db := new DataBase();}

   Unit openR(){
      await writer = None;
      readers := insertElement(readers, caller);
   }

   Unit closeR(){
      readers := remove(readers, caller);
   }

   Unit openW(){
      await writer = None; writer := caller;
      readers := insertElement(readers, caller);
   }

   Unit closeW(){
      await writer = caller; writer := None;
      readers := remove(readers, caller);
```

```
    }

    String read(Int key){
      Fut<String> fr; String data;
      await contains(readers, caller);
      pr := pr + 1; fr := db!read(key); await data := fr?; pr := pr −1;
      return data;
    }

    Unit write(Int key, String value){
      await caller = writer && pr = 0 && (readers = EmptySet ||
      (contains(readers, writer) && size(readers) = 1));
      db.write(key, value);
    }
  }
```

C.2 Definition of *irev*

$irev : Seq[Ev] \times Fid \to IREv$

$irev(\varepsilon, fr') \triangleq \bot$
$irev(h \cdot \text{event}, fr') \triangleq \textbf{if } fr' \in \text{event } \textbf{then } \text{event } \textbf{else } irev(h, fr') \textbf{ fi}$

C.3 Definition of Writers

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

$Writers(\varepsilon) \triangleq \varnothing$
$Writers(h \cdot \langle \leftarrow \text{this}, fr', \text{openW}, \_\rangle) \triangleq Writers(h) \cup \{irev(h, fr').caller\}$
$Writers(h \cdot \langle \leftarrow \text{this}, fr', \text{closeW}, \_\rangle) \triangleq Writers(h) \backslash \{irev(h, fr').caller\}$
$Writers(h \cdot \textbf{\textit{others}}) \triangleq Writers(h)$

C.4 Definition of Writing

$Writing : Seq[Ev] \to Nat$

$Writing(h) \triangleq \#(h/\langle \text{this} \to \text{db}, \_, \text{write}, \_\rangle) - \#(h/\langle \text{this} \twoheadleftarrow, Futures(h, \text{db}, \text{write}), \_\rangle)$
$Futures(h, \text{db}, \text{write}) \triangleq ToSet((h/\langle \text{this} \to \text{db}, \_, \text{write}, \_\rangle).future)$

C.5 Proof outline

In the proof outlines we present the first and the final derived sequents for each method of the RWController class. The corresponding proof details can be found in Appendix. C.6.

For simplicity, the invocation reaction event and the completion event are abbreviated to $\mathcal{H}_{begin}$ and $\mathcal{H}_{end}$ respectively. $\mathcal{H}_{begin} : \langle \text{caller} \twoheadrightarrow \text{this}, \text{destiny}, m, \overline{x} \rangle$ and $\mathcal{H}_{end} : \langle \leftarrow \text{this}, \text{destiny}, m, \text{return} \rangle$.

C.5.1 openR

$$\frac{\begin{array}{l}\vdash (wf(\mathcal{H}) \wedge I_C) \Rightarrow \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin}\}(I_C \wedge wf(\mathcal{H})) \\ \vdash (wf(\mathcal{H}) \wedge I_C) \Rightarrow \forall \overline{w}, h' . \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h'\}(I_C \wedge wf(\mathcal{H}) \wedge (\text{writer} = \text{None}) \Rightarrow \\ \quad \{\text{readers} := \text{insertElement}(\text{readers}, \text{caller}) || \mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{end}\}(wf(\mathcal{H}) \Rightarrow I_C))\end{array}}{\begin{array}{l}\vdash (wf(\mathcal{H}) \wedge I_C) \Rightarrow [\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin};\ \text{await writer} = \text{None}; \\ \quad \text{readers} := \text{insertElement}(\text{readers}, \text{caller}); \mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{end}](wf(\mathcal{H}) \Rightarrow I_C)\end{array}}$$

C.5.2 openW

$$\frac{\begin{array}{l}\vdash (wf(\mathcal{H}) \wedge I_C) \Rightarrow \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin}\}(I_C \wedge wf(\mathcal{H}))\\ \vdash (wf(\mathcal{H}) \wedge I_C) \Rightarrow \forall \overline{w}, h' \,.\, \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h'\}(I_C \wedge wf(\mathcal{H}) \wedge (\text{writer} = \text{None}) \Rightarrow\\ \quad \{\text{writer} := \text{caller} \;||\; \text{readers} := \text{insertElement(readers, caller)}||\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{end}\}(wf(\mathcal{H}) \Rightarrow I_C))\end{array}}{\begin{array}{l}\vdash (wf(\mathcal{H}) \wedge I_C) \Rightarrow [\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin};\ \text{await writer} = \text{None; writer} := \text{caller;}\\ \quad \text{readers} := \text{insertElement(readers, caller)};\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{end}](wf(\mathcal{H}) \Rightarrow I_C)\end{array}}$$

C.5.3 closeR

$$\frac{\vdash (wf(\mathcal{H}) \wedge I_C) \Rightarrow \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin} \cdot \mathcal{H}_{end}||\ \text{readers} := \text{remove(readers, caller)}\}(wf(\mathcal{H}) \Rightarrow I_C)}{\vdash (wf(\mathcal{H}) \wedge I_C) \Rightarrow [\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin}; \text{readers} := \text{remove(readers, caller)}; \mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{end}](wf(\mathcal{H}) \Rightarrow I_C)}$$

C.5.4 closeW

$$\frac{\begin{array}{l}\vdash (wf(\mathcal{H}) \wedge I_C) \Rightarrow \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin}\}(I_C \wedge wf(\mathcal{H}))\\ \vdash (wf(\mathcal{H}) \wedge I_C) \Rightarrow \forall \overline{w}, h' \,.\, \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h'\}(I_C \wedge wf(\mathcal{H}) \wedge (\text{writer} = \text{caller}) \Rightarrow\\ \quad \{\text{writer} := \text{None} \;||\; \text{readers} := \text{remove(readers, caller)}||\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{end}\}(wf(\mathcal{H}) \Rightarrow I_C))\end{array}}{\begin{array}{l}\vdash (wf(\mathcal{H}) \wedge I_C) \Rightarrow [\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin};\ \text{await writer} = \text{caller; writer} := \text{None;}\\ \quad \text{readers} := \text{remove(readers, caller)};\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{end}](wf(\mathcal{H}) \Rightarrow I_C)\end{array}}$$

C.5.5 read

$$\frac{\begin{array}{l}\vdash (wf(\mathcal{H}) \wedge I_C) \Rightarrow \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin}||fr := \mathsf{null}||data := \epsilon\}(I_C \wedge wf(\mathcal{H}))\\ \vdash (wf(\mathcal{H}) \wedge I_C) \Rightarrow \forall \overline{w}, h' \,.\, \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h'||fr := \mathsf{null}||data := \epsilon\}\\ \quad (I_C \wedge wf(\mathcal{H}) \wedge \text{contains(readers, caller)} \Rightarrow\\ \quad \forall fr' \,.\, \{pr := pr + 1||\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key\rangle||fr := fr'\}(I_C \wedge wf(\mathcal{H})))\\ \vdash (wf(\mathcal{H}) \wedge I_C) \Rightarrow \forall \overline{w}, h' \,.\, \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h'||fr := \mathsf{null}||data := \epsilon\}\\ \quad (I_C \wedge wf(\mathcal{H}) \wedge \text{contains(readers, caller)} \Rightarrow\\ \quad \forall \overline{w}', h'', fr' \,.\, \{pr := pr + 1||\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key\rangle \vdash\!\dashv h''||fr := fr'\}(I_C \wedge wf(\mathcal{H})) \Rightarrow\\ \quad \forall v' \,.\, \{\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \twoheadleftarrow, fr, v'\rangle \cdot \mathcal{H}_{end}||\text{data:= } v'||\text{pr := pr -1}||\mathsf{return} := \text{data}\}(wf(\mathcal{H}) \Rightarrow I_C))\end{array}}{\begin{array}{l}\vdash (wf(\mathcal{H}) \wedge I_C) \Rightarrow [\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin};\ \text{Fut<String> fr; String data; await contains(readers, caller);}\\ \quad \text{pr := pr + 1; fr := db!read(key); await data := fr?; pr := pr -1; return data;}\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{end}](wf(\mathcal{H}) \Rightarrow I_C)\end{array}}$$

C.5.6 write

B : caller = writer && pr = 0 &&
(readers = EmptySet || (contains(readers, writer) && size(readers) = 1))

$$\frac{\begin{array}{l}\vdash (wf(\mathcal{H}) \wedge I_C) \Rightarrow \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin}\}(I_C \wedge wf(\mathcal{H}))\\ \vdash (wf(\mathcal{H}) \wedge I_C) \Rightarrow \forall \overline{w}, h', fr', v' \,.\, \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h'\}(I_C \wedge wf(\mathcal{H}) \wedge B \Rightarrow\\ \quad \{\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \rightarrow db, fr', \mathsf{write}, (\text{key,value})\rangle \cdot \langle \mathsf{this} \twoheadleftarrow, fr', v'\rangle \cdot \mathcal{H}_{end}\}(wf(\mathcal{H}) \Rightarrow I_C))\end{array}}{\vdash (wf(\mathcal{H}) \wedge I_C) \Rightarrow [\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin};\ \text{await } B; \text{ db.write(key, value)};\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{end}](wf(\mathcal{H}) \Rightarrow I_C)}$$

C.6 Proof details

We write {caller} as the abbreviation of $\{irev(h, \mathsf{destiny}).\mathsf{caller}\}$. For each method, we show the detail proofs for each final derived sequent. If there are more than one sequent to be proved for one method, we distinguish the cases by listing them.

C.6.1 openR

$I_C = I_1 : Readers(\mathcal{H}) = \mathsf{readers}$
$\mathcal{H}_{begin} : \langle \mathsf{caller} \twoheadrightarrow \mathsf{this}, \mathsf{destiny}, \mathsf{openR}, \overline{x}\rangle$
$\mathcal{H}_{end} : \langle \leftarrow \mathsf{this}, \mathsf{destiny}, \mathsf{openR}, \mathsf{return}\rangle$

(1)
$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers}) \Rightarrow$
$\{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin}\}(Readers(\mathcal{H}) = \mathsf{readers} \wedge wf(\mathcal{H}))$

$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers}) \Rightarrow$
$\quad (Readers(\mathcal{H} \cdot \mathcal{H}_{begin}) = \mathsf{readers} \ \wedge \ wf(\mathcal{H} \cdot \mathcal{H}_{begin}))$
$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers}) \Rightarrow$
$\quad (Readers(\mathcal{H}) = \mathsf{readers} \ \wedge \ wf(\mathcal{H} \cdot \mathcal{H}_{begin}))$
(2)
$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers}) \Rightarrow$
$\quad \forall \overline{w}, h' \,.\, \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h'\}(Readers(\mathcal{H}) = \mathsf{readers} \wedge wf(\mathcal{H}) \wedge (\text{writer} = \text{None}) \Rightarrow$
$\quad\quad \{\text{readers} := \text{insertElement}(\mathsf{readers}, \mathsf{caller})||\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{end}\}(wf(\mathcal{H}) \Rightarrow Readers(\mathcal{H}) = \mathsf{readers}))$
$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers}) \Rightarrow$
$\quad \forall \overline{w}, h' \,.\, (Readers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \mathsf{readers} \ \wedge \ wf(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') \wedge (\text{writer} = \text{None}) \Rightarrow$
$\quad\quad (wf(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \mathcal{H}_{end}) \Rightarrow Readers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \mathcal{H}_{end}) = \text{insertElement}(\mathsf{readers}, \mathsf{caller})))$
$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers}) \Rightarrow$
$\quad \forall \overline{w}, h' \,.\, (Readers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \mathsf{readers} \ \wedge \ wf(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') \wedge (\text{writer} = \text{None}) \Rightarrow$
$\quad\quad (wf(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \mathcal{H}_{end}) \Rightarrow Readers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') \cup \{\mathsf{caller}\} = \text{insertElement}(\mathsf{readers}, \mathsf{caller})))$

C.6.2 openW

$I_C = I_1 \wedge I_2 : Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\}$
$\mathcal{H}_{begin} : \langle \mathsf{caller} \twoheadrightarrow \mathsf{this}, \mathsf{destiny}, \mathsf{openW}, \overline{x} \rangle$
$\mathcal{H}_{end} : \langle \leftarrow \mathsf{this}, \mathsf{destiny}, \mathsf{openW}, \mathsf{return} \rangle$

(1)
$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\}) \Rightarrow$
$\quad \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin}\}(Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge wf(\mathcal{H}))$
$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\}) \Rightarrow$
$\quad (Readers(\mathcal{H} \cdot \mathcal{H}_{begin}) = \mathsf{readers} \wedge Writers(\mathcal{H} \cdot \mathcal{H}_{begin}) = \{\mathsf{writer}\} \ \wedge \ wf(\mathcal{H} \cdot \mathcal{H}_{begin}))$
$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\}) \Rightarrow$
$\quad (Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \ \wedge \ wf(\mathcal{H} \cdot \mathcal{H}_{begin}))$
(2)
$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\}) \Rightarrow$
$\quad \forall \overline{w}, h' \,.\, \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h'\}(Readers(\mathcal{H}) = \mathsf{readers} \wedge$
$\quad Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge wf(\mathcal{H}) \wedge (\text{writer} = \text{None}) \Rightarrow$
$\quad\quad \{\text{writer} := \text{caller} \ || \ \text{readers} := \text{insertElement}(\mathsf{readers}, \mathsf{caller})||\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{end}\}$
$\quad\quad (wf(\mathcal{H}) \Rightarrow (Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\})))$
$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\}) \Rightarrow$
$\quad \forall \overline{w}, h' \,.\, (Readers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \mathsf{readers} \wedge$
$\quad Writers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \{\mathsf{writer}\} \wedge wf(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') \wedge (\text{writer} = \text{None}) \Rightarrow$
$\quad\quad (wf(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \mathcal{H}_{end}) \Rightarrow$
$\quad\quad (Readers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \mathcal{H}_{end}) = \text{insertElement}(\mathsf{readers}, \mathsf{caller}) \wedge$
$\quad\quad Writers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \mathcal{H}_{end}) = \{\mathsf{caller}\})))$
$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\}) \Rightarrow$
$\quad \forall \overline{w}, h' \,.\, (Readers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \mathsf{readers} \wedge$
$\quad Writers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \{\mathsf{writer}\} \wedge wf(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') \wedge (\text{writer} = \text{None}) \Rightarrow$
$\quad\quad (wf(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \mathcal{H}_{end}) \Rightarrow$
$\quad\quad (Readers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') \cup \{\mathsf{caller}\} = \text{insertElement}(\mathsf{readers}, \mathsf{caller}) \wedge$
$\quad\quad Writers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') \cup \{\mathsf{caller}\} = \{\mathsf{caller}\})))$

C.6.3 closeR

$I_C = I_1 : Readers(\mathcal{H}) = \mathsf{readers}$
$\mathcal{H}_{begin} : \langle \mathsf{caller} \twoheadrightarrow \mathsf{this}, \mathsf{destiny}, \mathsf{closeR}, \overline{x} \rangle$
$\mathcal{H}_{end} : \langle \leftarrow \mathsf{this}, \mathsf{destiny}, \mathsf{closeR}, \mathsf{return} \rangle$

$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers}) \Rightarrow$
$\quad \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin} \cdot \mathcal{H}_{end}|| \ \text{readers} := \text{remove}(\mathsf{readers}, \mathsf{caller})\}(wf(\mathcal{H}) \Rightarrow Readers(\mathcal{H}) = \mathsf{readers})$
$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers}) \Rightarrow$
$\quad (wf(\mathcal{H} \cdot \mathcal{H}_{begin} \cdot \mathcal{H}_{end}) \Rightarrow Readers(\mathcal{H} \cdot \mathcal{H}_{begin} \cdot \mathcal{H}_{end}) = \text{remove}(\mathsf{readers}, \mathsf{caller}))$

$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers}) \Rightarrow$
$\quad (wf(\mathcal{H} \cdot \mathcal{H}_{begin} \cdot \mathcal{H}_{end}) \Rightarrow Readers(\mathcal{H})\backslash\{\mathsf{caller}\} = \text{remove}(\mathsf{readers}, \mathsf{caller}))$

C.6.4 closeW

$I_C = I_1 \wedge I_2 : Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\}$
$\mathcal{H}_{begin} : \langle \mathsf{caller} \twoheadrightarrow \mathsf{this}, \mathsf{destiny}, \mathsf{closeW}, \overline{x} \rangle$
$\mathcal{H}_{end} : \langle \leftarrow \mathsf{this}, \mathsf{destiny}, \mathsf{closeW}, \mathsf{return} \rangle$

(1)
$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\}) \Rightarrow$
$\quad \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin}\}(Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge wf(\mathcal{H}))$
$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\}) \Rightarrow$
$\quad (Readers(\mathcal{H} \cdot \mathcal{H}_{begin}) = \mathsf{readers} \wedge Writers(\mathcal{H} \cdot \mathcal{H}_{begin}) = \{\mathsf{writer}\} \ \wedge \ wf(\mathcal{H} \cdot \mathcal{H}_{begin}))$
$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\}) \Rightarrow$
$\quad (Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \ \wedge \ wf(\mathcal{H} \cdot \mathcal{H}_{begin}))$
(2)
$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\}) \Rightarrow$
$\quad \forall \overline{w}, h' \,.\, \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin} \vdash h'\}(Readers(\mathcal{H}) = \mathsf{readers} \wedge$
$\quad Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge wf(\mathcal{H}) \wedge (\text{writer} = \text{caller}) \Rightarrow$
$\quad\quad \{\text{writer} := \text{None} \,||\, \text{readers} := \text{remove}(\text{readers}, \text{caller})||\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{end}\}$
$\quad\quad (wf(\mathcal{H}) \Rightarrow (Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\})))$
$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\}) \Rightarrow$
$\quad \forall \overline{w}, h' \,.\, (Readers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash h') = \mathsf{readers} \wedge$
$\quad Writers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash h') = \{\mathsf{writer}\} \wedge wf(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash h') \wedge (\text{writer} = \text{caller}) \Rightarrow$
$\quad\quad (wf(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash h' \cdot \mathcal{H}_{end}) \Rightarrow$
$\quad\quad (Readers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash h' \cdot \mathcal{H}_{end}) = \text{remove}(\text{readers}, \text{caller}) \wedge$
$\quad\quad Writers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash h' \cdot \mathcal{H}_{end}) = \{\text{None}\})))$
$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\}) \Rightarrow$
$\quad \forall \overline{w}, h' \,.\, (Readers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash h') = \mathsf{readers} \wedge$
$\quad Writers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash h') = \{\mathsf{writer}\} \wedge wf(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash h') \wedge (\text{writer} = \text{caller}) \Rightarrow$
$\quad\quad (wf(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash h' \cdot \mathcal{H}_{end}) \Rightarrow$
$\quad\quad (Readers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash h')\backslash\{\mathsf{caller}\} = \text{remove}(\text{readers}, \text{caller}) \wedge$
$\quad\quad Writers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash h')\backslash\{\mathsf{caller}\} = \{\text{None}\})))$

C.6.5 read

$I_C = I_1 \wedge I_2 \wedge I_3 \wedge I_4 : Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})$
The property $Writing(\mathcal{H}) = 0$ can be verified as a part of the class invariant since $db.write$ is only called synchronously. We need to include it in order to complete the second branch of the proof tree.
$\mathcal{H}_{begin} : \langle \mathsf{caller} \twoheadrightarrow \mathsf{this}, \mathsf{destiny}, \mathsf{read}, \overline{x} \rangle$
$\mathcal{H}_{end} : \langle \leftarrow \mathsf{this}, \mathsf{destiny}, \mathsf{read}, \mathsf{return} \rangle$

(1)
$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})) \Rightarrow$
$\quad \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin}||fr := \mathsf{null}||data := \epsilon\}(Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge$
$\quad Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge wf(\mathcal{H}))$
$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})) \Rightarrow$
$\quad (Readers(\mathcal{H} \cdot \mathcal{H}_{begin}) = \mathsf{readers} \wedge Writers(\mathcal{H} \cdot \mathcal{H}_{begin}) = \{\mathsf{writer}\} \wedge$
$\quad Reading(\mathcal{H} \cdot \mathcal{H}_{begin}) = \mathsf{pr} \wedge OK(\mathcal{H} \cdot \mathcal{H}_{begin}) \wedge wf(\mathcal{H} \cdot \mathcal{H}_{begin}))$
$(wf(\mathcal{H}) \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})) \Rightarrow$
$\quad (Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge wf(\mathcal{H} \cdot \mathcal{H}_{begin}))$
(2)
$(wf(\mathcal{H}) \wedge Writing(\mathcal{H}) = 0 \wedge Readers(\mathcal{H}) = \mathsf{readers} \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge$
$Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})) \Rightarrow$
$\quad \forall \overline{w}, h' \,.\, \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin} \vdash h'||fr := \mathsf{null}||data := \epsilon\}$

$(\mathit{Writing}(\mathcal{H}) = 0 \wedge \mathit{Readers}(\mathcal{H}) = \mathsf{readers} \wedge \mathit{Writers}(\mathcal{H}) = \{\mathsf{writer}\} \wedge$
$\mathit{Reading}(\mathcal{H}) = \mathsf{pr} \wedge \mathit{OK}(\mathcal{H}) \wedge \mathit{wf}(\mathcal{H}) \wedge \mathrm{contains}(\mathrm{readers},\ \mathrm{caller}) \Rightarrow$
$\forall fr' \,.\, \{pr := pr + 1 || \mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key \rangle || fr := fr'\}$
$(\mathit{Writing}(\mathcal{H}) = 0 \wedge \mathit{Readers}(\mathcal{H}) = \mathsf{readers} \wedge \mathit{Writers}(\mathcal{H}) = \{\mathsf{writer}\} \wedge$
$\mathit{Reading}(\mathcal{H}) = \mathsf{pr} \wedge \mathit{OK}(\mathcal{H}) \wedge \mathit{wf}(\mathcal{H})))$
$(\mathit{wf}(\mathcal{H}) \wedge \mathit{Writing}(\mathcal{H}) = 0 \wedge \mathit{Readers}(\mathcal{H}) = \mathsf{readers} \wedge \mathit{Writers}(\mathcal{H}) = \{\mathsf{writer}\} \wedge$
$\mathit{Reading}(\mathcal{H}) = \mathsf{pr} \wedge \mathit{OK}(\mathcal{H})) \Rightarrow$
$\forall \overline{w}, h' \,.\, (\mathit{Writing}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = 0 \wedge \mathit{Readers}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \mathsf{readers} \wedge$
$\mathit{Writers}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \{\mathsf{writer}\} \wedge \mathit{Reading}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \mathsf{pr} \wedge \mathit{OK}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') \wedge$
$\mathit{wf}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') \wedge \mathrm{contains}(\mathrm{readers},\ \mathrm{caller}) \Rightarrow$
$\forall fr' \,.\, (\mathit{Writing}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key \rangle) = 0 \wedge$
$\mathit{Readers}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key \rangle) = \mathsf{readers} \wedge$
$\mathit{Writers}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key \rangle) = \{\mathsf{writer}\} \wedge$
$\mathit{Reading}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key \rangle) = \mathsf{pr} + 1 \wedge$
$\mathit{OK}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key \rangle) \wedge \mathit{wf}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key \rangle)))$
$(\mathit{wf}(\mathcal{H}) \wedge \mathit{Writing}(\mathcal{H}) = 0 \wedge \mathit{Readers}(\mathcal{H}) = \mathsf{readers} \wedge \mathit{Writers}(\mathcal{H}) = \{\mathsf{writer}\} \wedge$
$\mathit{Reading}(\mathcal{H}) = \mathsf{pr} \wedge \mathit{OK}(\mathcal{H})) \Rightarrow$
$\forall \overline{w}, h' \,.\, (\mathit{Writing}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = 0 \wedge \mathit{Readers}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \mathsf{readers} \wedge$
$\mathit{Writers}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \{\mathsf{writer}\} \wedge \mathit{Reading}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \mathsf{pr} \wedge \mathit{OK}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') \wedge$
$\mathit{wf}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') \wedge \mathrm{contains}(\mathrm{readers},\ \mathrm{caller}) \Rightarrow$
$\forall fr' \,.\, (\mathit{Writing}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = 0 \wedge$
$\mathit{Readers}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \mathsf{readers} \wedge$
$\mathit{Writers}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \{\mathsf{writer}\} \wedge$
$\mathit{Reading}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') + 1 = \mathsf{pr} + 1 \wedge$
$\mathit{OK}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') \wedge \mathit{Writing}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = 0 \wedge$
$\mathit{wf}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key \rangle)))$
(3)
$(\mathit{wf}(\mathcal{H}) \wedge \mathit{Readers}(\mathcal{H}) = \mathsf{readers} \wedge \mathit{Writers}(\mathcal{H}) = \{\mathsf{writer}\} \wedge \mathit{Reading}(\mathcal{H}) = \mathsf{pr} \wedge \mathit{OK}(\mathcal{H})) \Rightarrow$
$\forall \overline{w}, h' \,.\, \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' || fr := \mathsf{null} || data := \epsilon\} (\mathit{Readers}(\mathcal{H}) = \mathsf{readers} \wedge$
$\mathit{Writers}(\mathcal{H}) = \{\mathsf{writer}\} \wedge \mathit{Reading}(\mathcal{H}) = \mathsf{pr} \wedge \mathit{OK}(\mathcal{H}) \wedge \mathit{wf}(\mathcal{H}) \wedge \mathrm{contains}(\mathrm{readers},\ \mathrm{caller}) \Rightarrow$
$\forall \overline{w}', h'', fr' \,.\, \{pr := pr + 1 || \mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key \rangle \vdash\!\dashv h'' || fr := fr'\}$
$(\mathit{Readers}(\mathcal{H}) = \mathsf{readers} \wedge \mathit{Writers}(\mathcal{H}) = \{\mathsf{writer}\} \wedge \mathit{Reading}(\mathcal{H}) = \mathsf{pr} \wedge \mathit{OK}(\mathcal{H}) \wedge \mathit{wf}(\mathcal{H})) \Rightarrow$
$\forall v' \,.\, \{\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \twoheadleftarrow, fr, v' \rangle \cdot \mathcal{H}_{end} || \mathrm{data} := v' || \mathrm{pr} := \mathrm{pr}\ \text{-}1 || \mathsf{return} := \mathrm{data}\}$
$(\mathit{wf}(\mathcal{H}) \Rightarrow \mathit{Readers}(\mathcal{H}) = \mathsf{readers} \wedge \mathit{Writers}(\mathcal{H}) = \{\mathsf{writer}\} \wedge \mathit{Reading}(\mathcal{H}) = \mathsf{pr} \wedge \mathit{OK}(\mathcal{H})))$
$(\mathit{wf}(\mathcal{H}) \wedge \mathit{Readers}(\mathcal{H}) = \mathsf{readers} \wedge \mathit{Writers}(\mathcal{H}) = \{\mathsf{writer}\} \wedge \mathit{Reading}(\mathcal{H}) = \mathsf{pr} \wedge \mathit{OK}(\mathcal{H})) \Rightarrow$
$\forall \overline{w}, h' \,.\, (\mathit{Readers}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \mathsf{readers} \wedge \mathit{Writers}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \{\mathsf{writer}\} \wedge$
$\mathit{Reading}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \mathsf{pr} \wedge \mathit{OK}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') \wedge$
$\mathit{wf}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') \wedge \mathrm{contains}(\mathrm{readers},\ \mathrm{caller}) \Rightarrow$
$(\mathit{Readers}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key \rangle \vdash\!\dashv h'') = \mathsf{readers} \wedge$
$\mathit{Writers}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key \rangle \vdash\!\dashv h'') = \{\mathsf{writer}\} \wedge$
$\mathit{Reading}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key \rangle \vdash\!\dashv h'') = \mathsf{pr} + 1 \wedge$
$\mathit{OK}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key \rangle \vdash\!\dashv h'') \wedge$
$\mathit{wf}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key \rangle \vdash\!\dashv h'') \Rightarrow$
$(\mathit{wf}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key \rangle \vdash\!\dashv h'' \cdot \langle \mathsf{this} \twoheadleftarrow, fr, v' \rangle \cdot \mathcal{H}_{end}) \Rightarrow$
$\mathit{Readers}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key \rangle \vdash\!\dashv h'' \cdot \langle \mathsf{this} \twoheadleftarrow, fr, v' \rangle \cdot \mathcal{H}_{end}) = \mathsf{readers} \wedge$
$\mathit{Writers}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key \rangle \vdash\!\dashv h'' \cdot \langle \mathsf{this} \twoheadleftarrow, fr, v' \rangle \cdot \mathcal{H}_{end}) = \{\mathsf{writer}\} \wedge$
$\mathit{Reading}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key \rangle \vdash\!\dashv h'' \cdot \langle \mathsf{this} \twoheadleftarrow, fr, v' \rangle \cdot \mathcal{H}_{end}) =$
$\mathsf{pr} - 1 + 1 \wedge$
$\mathit{OK}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key \rangle \vdash\!\dashv h'' \cdot \langle \mathsf{this} \twoheadleftarrow, fr, v' \rangle \cdot \mathcal{H}_{end})))$
$(\mathit{wf}(\mathcal{H}) \wedge \mathit{Readers}(\mathcal{H}) = \mathsf{readers} \wedge \mathit{Writers}(\mathcal{H}) = \{\mathsf{writer}\} \wedge \mathit{Reading}(\mathcal{H}) = \mathsf{pr} \wedge \mathit{OK}(\mathcal{H})) \Rightarrow$
$\forall \overline{w}, h' \,.\, (\mathit{Readers}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \mathsf{readers} \wedge \mathit{Writers}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \{\mathsf{writer}\} \wedge$
$\mathit{Reading}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \mathsf{pr} \wedge \mathit{OK}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') \wedge$
$\mathit{wf}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') \wedge \mathrm{contains}(\mathrm{readers},\ \mathrm{caller}) \Rightarrow$
$(\mathit{Readers}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key \rangle \vdash\!\dashv h'') = \mathsf{readers} \wedge$
$\mathit{Writers}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key \rangle \vdash\!\dashv h'') = \{\mathsf{writer}\} \wedge$
$\mathit{Reading}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key \rangle \vdash\!\dashv h'') = \mathsf{pr} + 1 \wedge$
$\mathit{OK}(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key \rangle \vdash\!\dashv h'') \wedge$

$wf(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key\rangle \vdash\!\dashv h'')) \Rightarrow$
$(wf(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key\rangle \vdash\!\dashv h'' \cdot \langle \mathsf{this} \twoheadleftarrow, fr, v'\rangle \cdot \mathcal{H}_{end}) \Rightarrow$
$Readers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key\rangle \vdash\!\dashv h'') = \mathsf{readers} \wedge$
$Writers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key\rangle \vdash\!\dashv h'') = \{\mathsf{writer}\} \wedge$
$Reading(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key\rangle \vdash\!\dashv h'') - 1 = \mathsf{pr} \wedge$
$OK(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow \mathsf{db}, fr', \mathsf{read}, key\rangle \vdash\!\dashv h'')))$

C.6.6 write

$I_C = I_2 \wedge I_3 \wedge I_4 : Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})$
$\mathcal{H}_{begin} : \langle \mathsf{caller} \twoheadrightarrow \mathsf{this}, \mathsf{destiny}, \mathsf{write}, \overline{x}\rangle$
$\mathcal{H}_{end} : \langle \leftarrow \mathsf{this}, \mathsf{destiny}, \mathsf{write}, \mathsf{return}\rangle$
B : caller = writer && pr = 0 &&
(readers = EmptySet || (contains(readers, writer) && size(readers) = 1))

(1)
$(wf(\mathcal{H}) \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})) \Rightarrow$
$\{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin}\}(Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge wf(\mathcal{H}))$
$(wf(\mathcal{H}) \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})) \Rightarrow$
$(Writers(\mathcal{H} \cdot \mathcal{H}_{begin}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H} \cdot \mathcal{H}_{begin}) = \mathsf{pr} \wedge$
$OK(\mathcal{H} \cdot \mathcal{H}_{begin}) \wedge wf(\mathcal{H} \cdot \mathcal{H}_{begin}))$
$(wf(\mathcal{H}) \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})) \Rightarrow$
$(Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge wf(\mathcal{H} \cdot \mathcal{H}_{begin}))$
(2)
$(wf(\mathcal{H}) \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})) \Rightarrow$
$\forall \overline{w}, h', fr', v' \,.\, \{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h'\}$
$(Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H}) \wedge wf(\mathcal{H}) \wedge B \Rightarrow$
$\{\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \rightarrow db, fr', \mathsf{write}, (\mathrm{key,value})\rangle \cdot \langle \mathsf{this} \twoheadleftarrow, fr', v'\rangle \cdot \mathcal{H}_{end}\}$
$(wf(\mathcal{H}) \Rightarrow Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})))$
$(wf(\mathcal{H}) \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})) \Rightarrow$
$\forall \overline{w}, h', fr', v' \,.\, (Writers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \{\mathsf{writer}\} \wedge Reading(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \mathsf{pr} \wedge$
$OK(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') \wedge wf(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') \wedge B \Rightarrow$
$(wf(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow db, fr', \mathsf{write}, (\mathrm{key,value})\rangle \cdot \langle \mathsf{this} \twoheadleftarrow, fr', v'\rangle \cdot \mathcal{H}_{end}) \Rightarrow$
$Writers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow db, fr', \mathsf{write}, (\mathrm{key,value})\rangle \cdot \langle \mathsf{this} \twoheadleftarrow, fr', v'\rangle \cdot \mathcal{H}_{end}) = \{\mathsf{writer}\} \wedge$
$Reading(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow db, fr', \mathsf{write}, (\mathrm{key,value})\rangle \cdot \langle \mathsf{this} \twoheadleftarrow, fr', v'\rangle \cdot \mathcal{H}_{end}) = \mathsf{pr} \wedge$
$OK(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow db, fr', \mathsf{write}, (\mathrm{key,value})\rangle \cdot \langle \mathsf{this} \twoheadleftarrow, fr', v'\rangle \cdot \mathcal{H}_{end})))$
$(wf(\mathcal{H}) \wedge Writers(\mathcal{H}) = \{\mathsf{writer}\} \wedge Reading(\mathcal{H}) = \mathsf{pr} \wedge OK(\mathcal{H})) \Rightarrow$
$\forall \overline{w}, h', fr', v' \,.\, (Writers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \{\mathsf{writer}\} \wedge Reading(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \mathsf{pr} \wedge$
$OK(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') \wedge wf(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') \wedge B \Rightarrow$
$(wf(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h' \cdot \langle \mathsf{this} \rightarrow db, fr', \mathsf{write}, (\mathrm{key,value})\rangle \cdot \langle \mathsf{this} \twoheadleftarrow, fr', v'\rangle \cdot \mathcal{H}_{end}) \Rightarrow$
$Writers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \{\mathsf{writer}\} \wedge Reading(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = \mathsf{pr} \wedge$
$OK(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') \wedge Reading(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = 0 \wedge$
$\#Writers(\mathcal{H} \cdot \mathcal{H}_{begin} \vdash\!\dashv h') = 1))$