An Approach to Compositional Reasoning about Concurrent Objects and Futures

Research Report 415

Crystal Chang Din
Johan Dovland
Olaf Owe

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

Feb 2012
# Contents

1 Introduction .................................................. 2

2 Syntax for the ABS Language ................................ 4

3 Observable Behavior ........................................... 6

4 Analysis of ABS Programs ...................................... 10
   4.1 Semantic Definition by a Syntactic Encoding .......... 10
   4.2 Weakest Liberal Preconditions ............................ 11
   4.3 Dynamic Logic ............................................. 13
   4.4 Object Composition ....................................... 15

5 Reader Writer Example ......................................... 16
   5.0.1 Implementation ......................................... 17
   5.0.2 Specification and Verification ........................ 18

6 Related and Future Work ....................................... 19

7 Conclusion ...................................................... 20

A Syntax of the ABS functional sublanguage ................... 22

B Auxiliary Functions ............................................ 22

C Reader Writer Example in ABS ................................. 23
   C.1 Complete implementation .................................. 23
   C.2 Definition of \texttt{irev} .................................. 24
   C.3 Definition of Writers ...................................... 24
   C.4 Definition of Writing ...................................... 24
   C.5 Proof outline .............................................. 24
      C.5.1 \texttt{openR} ........................................... 24
      C.5.2 \texttt{openW} .......................................... 25
      C.5.3 \texttt{closeR} ......................................... 25
      C.5.4 \texttt{closeW} ......................................... 25
      C.5.5 \texttt{read} ............................................ 25
      C.5.6 \texttt{write} .......................................... 25
   C.6 Proof details ................................................ 25
      C.6.1 \texttt{openR} ........................................... 25
      C.6.2 \texttt{openW} .......................................... 26
      C.6.3 \texttt{closeR} ......................................... 26
      C.6.4 \texttt{closeW} ......................................... 27
      C.6.5 \texttt{read} ............................................ 27
      C.6.6 \texttt{write} .......................................... 29
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

\footnote{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.} 

2
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

\[ P ::= Dd^* \quad E^* \quad In^* \quad Cl^* \quad [s]^? \quad \text{program} \]

\[ In ::= \text{interface } I \{ \text{extends } I^* \} \{ S^* \} \quad \text{interface declaration} \]

\[ Cl ::= \text{class } C \{ [T \; cp^*] \} \{ [\text{implements } I^*] \} \{ [T \; w^*] \{ s \} \quad M^* \} \quad \text{class definition} \]

\[ M ::= S \quad B \quad \quad \text{method definition} \]

\[ S ::= T \quad m(T \; x^*) \quad \text{method signature} \]

\[ B ::= \{ [\text{var } T \; x[{-} \; e]^*]; [s^*]; t^* \} \quad \text{return } e \quad \text{method blocks} \]

\[ v ::= x \mid w \mid fr \quad \text{local variables or attributes} \]

\[ e ::= \text{null } \mid \text{this } \mid \text{caller } \mid \text{destiny } \mid v \mid cp \mid t \quad \text{pure expressions} \]

\[ s ::= v := e \mid fr := e \mid \text{m}(e^*) \mid \text{await } e \mid \text{suspend} \]

\[ | \text{skip } | \text{abort } | \text{if } e \text{ then } s \text{ [else } s^* \text{] [fi]} | s ; s \]

\[ v := \text{new } C(e^*) \mid v := e \text{.m}(e^*) \mid \text{await } v ::= fr? \]

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 \( cC \), defines class attributes \( cA \), 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 \( \text{init} \). There is read-only access to the formal class parameters \( cA \). 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 \( \text{run} \) 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 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 \texttt{await} statement
causes a release point, which suspends the executing process, releasing the processor and
allowing an \textit{enabled} and suspended process to be selected for execution. The continuation
of a process suspended by \texttt{await \ e} is enabled when the guard \( e \) evaluates to true. The
\texttt{suspend} statement is considered equivalent to \texttt{await true}. Note that an alternative
semantic definition of \texttt{await true} as \texttt{skip} is given in [15], and also used in several other
ABS papers.
A \textit{future}, \texttt{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 \textit{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 \( m(\tau)|\texttt{var} \ \eta; \ s; \ \texttt{return} \ e \),
ignoring type information, where \( \tau \) is the list of parameters, \( \eta \) 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 \texttt{Void} are assumed to
end with a \texttt{return void} statement, where \texttt{void} is the only value of type \texttt{Void}. Compound
return types can be defined by means of data types.
Object communication in ABS is \textit{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 \( \tau \), allowing the caller to wait for the reply in various manners:
- \texttt{fr} := \texttt{x!m(\tau)}: Here the calling process generates a fresh \textit{future identity} and continues
  without waiting for the \textit{future} to become resolved.
- \texttt{await v} := \texttt{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.
- \texttt{v} := \texttt{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 \( v := \texttt{fr.get} \).)
- \texttt{v} := \texttt{x.m(\tau)}: If \( x \) is different from this, the statement can be interpreted as \texttt{fr} := \texttt{x!m(\tau)}; \texttt{v} := \texttt{fr}?
  for a fresh future variable \( fr \). The method is invoked without releasing the processor of the calling object; the calling process \textit{blocks} the processor
  while waiting for the \textit{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, \texttt{skip},
\texttt{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 _·_ : Seq[T] × T → 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 : Seq[T], x, y, z : T, and s : Seq[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 _/_: Seq[T] × Seq[T] → Seq[T] is defined inductively by ε/s = ε and (a · x)/s = if x ∈ s then (a/s) · x else a/s fi.

The “ends with” and “begins with” predicates _ew_: Seq[T] × T → Bool and _bw_: Seq[T] × T → Bool are defined inductively by ε ew x = false, (a · y) ew x = x = y, ε bw x = false, (ε · y) bw x = x = y, and (a · z · y) bw x = (a · z) bw x. Furthermore, let a ≤ b denote that a is a prefix of b, a ⊢ b denote the concatenation of a and b, agree(a) denote that all elements (if any) are equal, #a denote the length of a, and [x1, x2, ..., xi] denote the sequence of x1, x2, ..., xi 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 caller, callee, receiver : Obj, future : Fid, method : Mtd, class : Cls, args : List[Data], and result : Data. The set Ev of all communication events is defined by Ev = IEv ∪ IREv ∪ CEv ∪ CREv ∪ NEv ∪ NREv, including

- the set IEv of invocation events ⟨caller → callee, future, method, args⟩
- the set IREv of invocation reaction events ⟨caller → callee, future, method, args⟩
- the set CEv of completion events ⟨callee, future, method, result⟩
- the set CREv of completion reaction events ⟨receiver ← , future, result⟩
- the set NEv of object creation events ⟨callee → callee, class, args⟩
- the set NREv of object creation reaction events ⟨callee → callee, class, args⟩

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 $\_\_\_\text{caller}$, $\_\_\_\text{callee}$, $\_\_\_\text{receiver}$ : $\text{Ev} \rightarrow \text{Obj}$, $\_\_\_\text{future}$ : $\text{Ev} \rightarrow \text{Fid}$, $\_\_\_\text{method}$ : $\text{Ev} \rightarrow \text{Mtd}$, $\_\_\_\text{class}$ : $\text{Ev} \rightarrow \text{Cls}$, $\_\_\_\text{args}$ : $\text{Ev} \rightarrow \text{List[Data]}$, and $\_\_\_\text{result}$ : $\text{Ev} \rightarrow \text{Data}$. For example, $(o \rightarrow d', u, m, \pi)$. caller returns $o$.

Equality is the only executable operation on object identities. Given the parent function, we may define an ancestor function $\text{anc}$ : $\text{Obj} \rightarrow \text{Set[Obj]}$ by $\text{anc}(\text{main}) = \text{main}$ and $\text{anc}(o) \supseteq \text{main}$ (where $o$ $\neq$ $\text{main}$). We say that parent chains are cycle free if $o$ $\notin$ $\text{anc}(o)$ for all generated objects $o$, i.e., for $o$ $\neq$ $\text{main}$.

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 $o'$, and the method is reflected by the invocation event $(o \rightarrow o', u, m, \pi)$ where $\pi$ is the list of actual parameters. The event $(o \rightarrow o', u, m, \pi)$ reflects that $o'$ starts execution of the method, and the event $(o \rightarrow o', u, m, \pi)$ reflects the future is resolved upon method termination. The event $(o \rightarrow o', u, m, \pi)$ reflects the future is resolved upon method termination. The event $(o \rightarrow o', u, m, \pi)$ 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 $(o \rightarrow o', C, \pi)$ and $(o \rightarrow o', C, \pi)$, where $o'$ is an object of class $C$ and $\pi$ 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 $\text{Seq[Ev}_O\text{]}$. The set of all communication events in $O$ is defined by $\text{Ev}_O \equiv \text{IEv}_O \cup \text{REv}_O \cup \text{CEv}_O \cup \text{REv}_O \cup \text{CREv}_O \cup$.
\[ \text{Definition 3 (Local communication histories) The local communication history of an object } o \text{ is a finite sequence of type } \text{Seq}[\text{Ev}(o)]. \text{ For a given history } h, \text{ the local history of } o \text{ is the subsequence of } h \text{ visible to the object } o, \text{ defined by } h/\text{Ev}(o) \text{ and abbreviated } h/o. \]

In this manner, the local communication history reflects the local activity of each object. For a method call \( m \) of object \( d \) made by \( o \), the events \( \langle o \to d, u, m, t \rangle \) and \( \langle o \to d, u, e \rangle \) are local to \( o \) and \( d \) respectively. For object creation, the event \( \langle o \to d, C, \tau \rangle \) is local to \( o \) whereas \( \langle o \to d', C, \tau \rangle \) is local to \( d' \). Let \( h_o \) denote that \( h \) is a local history of object \( o \), i.e., \( h_o : \text{Seq}[\text{Ev}(o)]. \)

It follows by the definitions above that \( \text{Ev}(o) \cap \text{Ev}(d) = \emptyset \) for \( o \neq d \), i.e., the two local histories \( h_o \) and \( h_d \) have no common events.

We define functions over the history to extract information, such as \( \text{fid} : \text{Seq}[\text{Ev}] \to \text{Sel}[\text{Fid}] \) extracting all identities of future occurring in a history, as follows:

\[
\begin{align*}
\text{ fid}(\varepsilon) &\triangleq \{\text{null}\} & \text{ fid}(h \cdot \gamma) &\triangleq \text{ fid}(h) \cup \text{ fid}(\gamma) \\
\text{ fid}(\langle o \to d', u, m, t \rangle) &\triangleq \{u \cup \text{ fid}(t)\} & \text{ fid}(\langle d \to o, a, u, m, t \rangle) &\triangleq \{u \cup \text{ fid}(t)\} \\
\text{ fid}(\langle o \leftarrow u, m, e \rangle) &\triangleq \{u \cup \text{ fid}(e)\} & \text{ fid}(\langle o \leftarrow u, u, e \rangle) &\triangleq \{u \cup \text{ fid}(e)\} \\
\text{ fid}(\langle o \to d', C, \tau \rangle) &\triangleq \text{ fid}(t) & \text{ fid}(\langle d \to o, C, \tau \rangle) &\triangleq \text{ fid}(t)
\end{align*}
\]

where \( \gamma : \text{Ev}, \text{ and } \text{ fid}(\tau) \) returns the set of future identities occurring in the expression list \( \tau \). Similarly the function \( \text{oid}(h) \) returns all object identities occurring in a history. The definition can be found in Appendix B.

The function \( \text{new}_{\text{obj}} : \text{Seq}[\text{Ev}] \to \text{Sel}[\text{Obj} \times \text{Cls} \times \text{List}[\text{Data}]] \) returns the set of created objects (each given by its object identity, associated class and class parameters) in a history:

\[
\begin{align*}
\text{ new}_{\text{obj}}(\varepsilon) &\triangleq \emptyset & \text{ new}_{\text{obj}}(h \cdot \langle o \to d', C, \tau \rangle) &\triangleq \text{ new}_{\text{obj}}(h) \cup \{d : \text{C}(\tau)\} \\
\text{ new}_{\text{obj}}(h \cdot \text{ others}) &\triangleq \text{ new}_{\text{obj}}(h)
\end{align*}
\]

(where \( \text{ others } \) matches all other events). The function \( \text{new}_{\text{obj}} : \text{Sel}[\text{Obj} \times \text{Cls} \times \text{List}[\text{Data}]] \to \text{Sel}[\text{Obj}] \) extracts object identities from the output of function \( \text{new}_{\text{obj}} \). For a local history \( h_o \), all objects \( \text{created by } o \) are returned by \( \text{new}_{\text{obj}}(h_o) \).

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:

\[ \text{Definition 4 (Wellformed histories) Let } u : \text{Fid}, h : \text{Seq}[\text{Ev}], \text{ the wellformedness predicate } \text{wf} : \text{Seq}[\text{Ev}] \times \text{Sel}[\text{Obj}] \to \text{Bool} \text{ is defined by:} \]

\[
\begin{align*}
\text{ wf}(\varepsilon, O) &\triangleq \text{ true } \\
\text{ wf}(h \cdot \langle o \to d', u, m, t \rangle, O) &\triangleq \text{ wf}(h, O) \land o \neq \text{ null} \land d \neq \text{ null} \land u \notin \text{ fid}(h) \\
\text{ wf}(h \cdot \langle o \leftarrow u, m, e \rangle, O) &\triangleq \text{ wf}(h, O) \land d \neq \text{ null} \land o \neq \text{ null} \land h/u \langle \leftarrow \rangle \to o, u, m, e \rangle \\
\text{ wf}(h \cdot \langle o \leftarrow u, u, e \rangle, O) &\triangleq \text{ wf}(h, O) \land u \in \text{ fid}(h/o) \land \text{agree}((h/u).\text{result}) \cdot e \\
\text{ wf}(h \cdot \langle o \to d', C, \tau \rangle, O) &\triangleq \text{ wf}(h, O) \land \text{ oid}(h/o) \land \text{parent}(d) \neq o \land d \neq \text{ oid}(h) \\
\text{ wf}(h \cdot \langle d \to o, C, \tau \rangle, O) &\triangleq \text{ wf}(h, O) \land d \neq \text{ null} \land \text{parent}(d) \neq o \land o \notin \text{ oid}(h/(O\backslash d')) \lor \text{ oid}(\tau) \land o - \varepsilon \land (d \in O) \Rightarrow h/d \cdot \text{ew} \langle d \to o, C, \tau \rangle
\end{align*}
\]
The wellformedness of appending an invocation event to the history captures the freshness of the generated future identity, i.e., \( u \not\in \text{fid}(h) \). For invocation reaction events, the corresponding invocation message must be visible in the past history if the caller is in \( O \), i.e., \( h/u \leftarrow \langle d' \rightarrow o, u, m, v_r \rangle \)/\( O \), 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., \( u \in \text{fid}(h/o) \). Moreover, all completion events and completion reaction events connected with the same future, must have the same return result, i.e., \( \text{agree}((h/u).\text{result}) \cdot c \). Remark that for object creation, the parent object and the created object synchronize, i.e., if \( d' \) is the parent of \( o \) and the history of \( O \) ends with the creation reaction event \( \langle d' \rightarrow o, C, v_r \rangle \), then the last event generated by \( d' \) is \( \langle d \rightarrow o, C, v_r \rangle \). 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 \( (d \in O \Rightarrow e_v \in h) \).

**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* \( I_C(\overline{w}, h_{\text{his}}) \) for a class \( C \) is a predicate over the attributes \( \overline{w} \) and the local history \( h_{\text{his}} \), as well as the formal class parameters \( \overline{cp} \) and \( \text{this} \), which are constant (read-only) variables.
We consider a simple semantic definition by a syntactic encoding. For a local call (i.e., Method communication is captured by four statements in order to state required conditions. The language, where statements have the syntax

\[ s := \text{skip} | \text{abort} | \tau := \pi | s | s | \text{if } b \text{ then } s | \text{else } s | \text{fi} | v := m(\pi) \]

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:

\[ \overline{y} := \text{some} \]

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

\[ \text{assert } b \]
means that one is obliged to verify the condition $b$ for the current state, and has otherwise no effect. Similarly, \textit{assume} statements are used to encode known facts. Semantically the statement
\begin{verbatim}
assume b
\end{verbatim}
is understood as $\textbf{if } b \textbf{ then skip else abort fi}$. To summarize, we have the following syntax for \texttt{SEQ} statements:
\begin{verbatim}
s ::= \texttt{skip} | \texttt{abort} | \tau ::= \tau | s | \texttt{if } b \texttt{ then } s \texttt{ else } s \texttt{ fi} | v ::= m(\tau)
| \texttt{y} ::= \texttt{some} | \texttt{assert} \ b | \texttt{assume} \ b
\end{verbatim}

Method definitions are of the form $m'(\tau, \texttt{caller, destiny}) \texttt{ body}$, where \texttt{body} is of the form $\{\texttt{var } y; s\}$. Thus a body contains declaration of method-local variables followed by a sequence of statements. For simplicity we use the same \texttt{body} notation as in \textit{ABS}. However, in \textit{ABS} the body must end with a final \texttt{return} statement, whereas \texttt{SEQ} uses a \texttt{return} variable. Since \texttt{caller} and \texttt{destiny} are not part of the \textit{ABS} language, they appear as explicit parameters in the encoding. At the class level, the list of class attributes is augmented with $\texttt{this} : \texttt{Obj}$ and $H : \texttt{Seq}[E_{\text{this}}]$, representing self reference and the local communication history, respectively. The semantics of a method is defined from the local perspective of processes. An \textit{ABS} process with release points and asynchronous method calls is interpreted as a non-deterministic \texttt{SEQ} process \textit{without} shared variables and release points, by the mapping $\langle \rangle$, as defined in Fig. 3. Expressions and types are mapped by the identity function. A \texttt{SEQ} process executes on a state $\pi \cup H$ extended with local variables and auxiliary variables introduced by the encoding. As in \textit{ABS}, there is read-only access to the formal class parameters. We let $\omega(H)$ abbreviate $\omega(H, \{\text{this}\})$.

When an instance of $m(\tau)$ starts execution, the history $H$ is extended by an invocation reaction event: $H := H \cdot (\texttt{caller} \rightarrow \texttt{this, destiny, m, } \tau)$. Process termination is reflected by appending a completion event: $H := H \cdot (\texttt{this, destiny, m, return})$, where \texttt{return} is the return value of $m$. When invoking some method $o(m(\tau))$, the history is extended with an invocation event: $H := H \cdot (\texttt{this \rightarrow o, fr, m, } \tau)$, and fetching the result value $v$ is encoded by $H := H \cdot (\texttt{this \leftarrow fr, v})$. The local effect of executing a release statement is that $\pi$ and $H$ may be updated due to the execution of other processes. In the encoding, these updates are captured by non-deterministic assignments to $\pi$ and $H$, as reflected by the encoding of the \texttt{suspend} statement. Here, the \texttt{assume} and \texttt{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 \textit{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.

\textbf{Lemma 1} \textit{The local history of an object is wellformed for any legal execution}
\begin{proof}
Preservation of wellformedness is trivial for statements that do not extend the local history $H$, and we need to ensure wellformedness after extensions of $H$. 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 $\omega(H)$ is preserved by the encoding of statement $v ::= o.m(\tau)$. For the remaining extensions, i.e., completion and completion reaction events, wellformedness is guaranteed by the \texttt{assume} statements following the different extensions.
\end{proof}

4.2 \textit{Weakest Liberal Preconditions}

We may define \textit{weakest liberal preconditions} for the different \textit{ABS} statements, reflecting that we consider partial correctness. The definitions are based on the encoding from \textit{ABS} to \texttt{SEQ}. The verification conditions of a class $C$ with invariant $I_C(\pi, H)$ are summarized in Fig. 4. Condition (1) applies to the initialization block $\texttt{init}$ of $C$, ensuring that the invariant is established upon
Figure 4: Verification conditions for ABS methods. Condition (1) ensures that the class invariant is established by the class initialization block \textit{init}. Condition (2) ensures that each method \(m(T)\) body maintains the class invariant. Condition (3) is used to verify additional properties for a method \(m(T)\) body, verifying the pre/post specification \(P/Q\) for the implementation. Notice that this \(\neq\) null follows from each premise.

\[
\begin{align*}
\text{termination. We may reason about possible processor release points in } init \text{ by the weakest liberal preconditions given below. Condition (2) applies to each method } m(T) \text{ body defined in } C; \text{ ensuring that each method maintains the class invariant. Condition (3) is used in order to prove additional knowledge for local synchronous calls, where } P \text{ is the precondition and } Q \text{ is the postcondition (given by a user specification). The weakest liberal precondition for non-deterministic assignment is given by:}
\end{align*}
\]

Figure 5: Weakest liberal preconditions for ABS statements. We let \(f', f'', h', \text{ and } v'\), denote fresh variables, and let \(wp_{SEQ}\) denote weakest liberal precondition for \(SEQ\).

termination. We may reason about possible processor release points in \textit{init} by the weakest liberal preconditions given below. Condition (2) applies to each method \(m(T)\) 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:

\[
wp(\text{some }, Q) = \forall \gamma . Q
\]

where the universal quantifier reflects that the chosen value of \(y\) is not known in the prestate.

The weakest liberal preconditions for \textit{assert} and \textit{assume} statements are given by:

\[
\begin{align*}
wp(\text{assert } b, Q) &\equiv b \land Q \\
wp(\text{assume } b, Q) &\equiv b \Rightarrow Q
\end{align*}
\]
In addition, let \( R^\pi, \) where \( \pi \) and \( \tau \) are of the same length, denote \( R \) where every free occurrence of each \( x_i \in \pi \) is replaced by \( e_i \). Weakest liberal preconditions for the different \( \text{ABS} \) 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 \( \psi \Rightarrow [s] \phi \) expresses that a program \( s \), starting in a state where \( \psi \) is true, will terminate in a state in which \( \phi \) holds. The formula \( \psi \Rightarrow [s] \phi \) does not demand termination and describes that if \( s \) is executed in a state where \( \psi \) holds and the execution terminates, then \( \phi \) holds in the final state. We are only concerned with partial correctness reasoning, where termination is not required, thus the diamond modality \( \langle \rangle \) 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 formula \( \psi \Rightarrow [s] \phi \), where assignment is the first statement can be rewritten to \( \{ v := \tau(e) \} [s] \phi \), in which \( \tau \) 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 \( \{ v_1 := \tau(e_1) \} \ldots \{ v_n := \tau(e_n) \} \) 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 \( \psi_1, \ldots, \psi_n \vdash \phi_1, \ldots, \phi_n \) represents a set of assumption formulae \( \psi_1, \ldots, \psi_n \) and a set of formulae \( \phi_1, \ldots, \phi_n \) to be proved. We say a sequent is valid if at least one formula \( \phi_i \) follows from the assumptions. Thus, a sequent can be interpreted as \( \psi_1 \land \ldots \land \psi_n \Rightarrow \phi_1 \lor \ldots \lor \phi_n \).

Sequent calculus rules:

\[
\text{ruleName} \quad \frac{\Gamma_1 \vdash \Delta_1 \ldots \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}
\]

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.

\[
\begin{align*}
\text{andRight} & \quad \frac{\Gamma \vdash \phi, \Delta}{\Gamma \vdash \phi \land \psi, \Delta} \\
\text{transitivity} & \quad \frac{\Gamma \vdash \psi, \Delta \quad \Gamma \vdash \psi \Rightarrow \phi, \Delta}{\Gamma \vdash \phi, \Delta} \\
\text{impRight} & \quad \frac{\Gamma, \phi \vdash \psi, \Delta}{\Gamma \vdash \phi \Rightarrow \psi, \Delta} \\
\text{allRight} & \quad \frac{\Gamma \vdash \phi^c, \Delta}{\Gamma \vdash \forall v. \phi, \Delta} \quad \text{(with a fresh constant \( c \) which is not used in \( \Gamma \))}
\end{align*}
\]

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 \( \psi \Rightarrow \text{wp}(s, \phi) \), we derive dynamic logic formulae for each \( \text{ABS} \)
(1) $\Gamma_C \vdash (H = \langle \text{paren}(\text{this}) \rightarrow \text{this}, C, \overline{x} \rangle) \Rightarrow [\text{init}_C] I_C(\overline{n}, H)$

(2) $\Gamma_C \vdash (w(\overline{h}) \land I_C(\overline{n}, H)) \Rightarrow \left( H = \overline{H} \cdot \langle \text{caller} \rightarrow \text{this}, \text{destiny}, m, \overline{x} \rangle; \text{body}_m; \right.$
$\left. H = \overline{H} \cdot \langle \leftarrow \text{this}, \text{destiny}, m, \text{return} \rangle \right) (w(\overline{h}) \Rightarrow I_C(\overline{n}, H))$

(3) $\Gamma_C \vdash (w(\overline{h}) \land P(\overline{n}, H)) \Rightarrow \left( H = \overline{H} \cdot \langle \text{caller} \rightarrow \text{this}, \text{destiny}, m, \overline{x} \rangle; \text{body}_m; \right.$
$\left. H = \overline{H} \cdot \langle \leftarrow \text{this}, \text{destiny}, m, \text{return} \rangle \right) (w(\overline{h}) \Rightarrow Q(\overline{n}, H))$

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 $\Gamma_C$ denotes the set of method specifications in $C$ (including invariance $I_C$), and which are proved by verification conditions (2) and (3).

\begin{align*}
&\vdash I_C(\overline{n}, H) \land w(\overline{h}) \\
&\vdash \forall \overline{n}, \overline{h}, \{ H := H \vdash h'(I_C(\overline{n}, H) \land \tau(e) \Rightarrow [s] \phi) \\
&\vdash [\text{await } e; \ s] \phi \\
&\vdash \forall \overline{f}, \{ H := H \vdash \langle \text{this} \rightarrow a.f, m, \overline{x} \rangle | fr := f \} (w(\overline{h}) \Rightarrow [s] \phi) \\
&\vdash [fr = a!m(\overline{x}); \ s] \phi \\
&\vdash \forall \overline{v}, \{ H := H \vdash \langle \text{this} \rightarrow \overline{f}, v \rangle | v := v' \} (w(\overline{h}) \Rightarrow [s] \phi) \\
&\vdash [v := \overline{f}'; \ s] \phi \\
&\vdash \forall \overline{v}, \{ H := H \vdash \langle \text{this} \rightarrow \overline{f}, \overline{v} \rangle | v := v' \} (w(\overline{h}) \Rightarrow [s] \phi) \\
&\vdash [v := \text{new } C(\overline{v}); \ s] \phi \\
&\Gamma \wedge \{ m(\overline{x}) \} \text{ sat } (P_1, Q_1) \vdash (\exists \overline{f}, \overline{h}, \overline{m}, \overline{v}, H \text{ ew}(\text{this} \leftarrow \overline{h}, \overline{v}) \land R^p_{\overline{n}, \overline{m}, \overline{v}} \wedge (P_1)_{\overline{n}, \overline{m}, \overline{v}, \overline{h}} \Rightarrow (Q_1)_{\text{this}, \overline{f}, \overline{v}, \text{return}, H} )) \Rightarrow [s] \phi \\
&\Gamma \vdash R \Rightarrow [v := m(\overline{v}); \ s] \phi
\end{align*}

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

statement from Section 4.2. In Fig. 7 we present the sequents for verifying methods and the initialization of classes, where $\Gamma_C$ 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
Therefore the history invariant is in general weaker than the class invariant, i.e.,

\[ I_C(\pi, H) \Rightarrow I_{\text{abs}}(\pi)(H) \]

Figure 9: Equivalent substitutions for other ABS statements. Here \( \tau \) is a function evaluating an expression \( e \), primes denote fresh variables, \( \phi \) is the postcondition, \( s \) is the remaining program yet to be executed, \( s_{\phi}^e \) is \( s \) with all (free) occurrences of \( e \) replaced by \( e^\prime \), and \( \text{default}_T \) 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, \([\text{suspend}]; s \phi \) can be rewritten to \([\text{await} \ true; s] \phi \), and we can then apply the sequent rule \text{awaitCond}. The substitution rules are introduced in Fig. 9 with the following syntax

\[
\text{ruleName} \ \phi = \phi' \]

which expresses the formula \( \phi \) can be rewritten to \( \phi' \).

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 \( o : C(\pi) \) can be captured by a prefix-closed history invariant, \( I_{\text{abs}}(\pi)(h_o) \). 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.,
By hiding the internal state variables of an object \( o \) of class \( C \), an external, prefix-closed history invariant \( I_{o:C}[\pi](h_o) \) defining its observable behavior on its local history \( h_o \) may be obtained from the class invariant of \( C \):

\[
I_{o:C}[\pi](h_o) \triangleq \exists h', \exists \pi. h_o \preceq h' \land (I_{C}[\pi, h'_o])_{\exists o:\pi}
\]

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 \( \exists \pi. (I_{C}[\pi, h'_o])_{\exists o:\pi} \). Also observe that a prefix-closed property \( P(h_o) \) is the same as the property \( \forall h \leq h_o. P(h) \).

Alternatively, a history invariant can be verified by showing that it is maintained by each statement \( s \) affecting the local history, i.e., one must prove

\[
(I_{\text{thick},C}[\pi](H) \land P) \vdash \exists s (Q \Rightarrow I_{\text{thick},C}[\pi](H))
\]

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 \( o : C[\pi] \) together with dynamically generated objects. The invariant \( I_O(H) \) of such a subsystem is given by

\[
I_O(H) \triangleq \bigwedge_{o : C[\pi] \in O, new_ob(H)} I_{o:C}[\pi](H[o])
\]

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

\[
(o\text{id}(H) \setminus \{\text{null}\}) \subseteq new_ob(O \cup new_ob(H))
\]

Reasoning about a global system can be done as above assuming the existence of an initial object \( \text{main} \) of some class \( \text{Main} \), such that all objects are created by \( \text{main} \) or generated objects. Thus \( \text{main} \) 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 \( I_{\text{main},\text{Main}}(H) \) of a global system with history \( H \) is

\[
wf(H, new_id(new_ob(H)) \cup \{\text{main}\}) \land \\
(o\text{id}(H) \setminus \{\text{null}, \text{main}\}) \subseteq new_id(new_ob(H)) \bigwedge_{o : C[\pi] \in new_ob(H)} I_{o:C}[\pi](H[o])
\]

assuming \( \text{true} \) as the class invariant for \( \text{main} \). Since \( \text{main} \) is the initial root object, the creation of \( \text{main} \) is not reflected on the global history \( H \), i.e., \( \text{main} \notin new_id(new_ob(H)) \).

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\( Readers : Seq[Ent] \rightarrow Seq[Obj] :\)
\[
\begin{align*}
Readers(x) & \triangleq \emptyset \\
Readers(h.\langle \leftarrow this, fr', openR, \_\rangle) & \triangleq Readers(h) \cup \{irev(h, fr').caller\} \\
Readers(h.\langle \leftarrow this, fr', openW, \_\rangle) & \triangleq Readers(h) \cup \{irev(h, fr').caller\} \\
Readers(h.\langle \leftarrow this, fr', closeR, \_\rangle) & \triangleq Readers(h) \setminus \{irev(h, fr').caller\} \\
Readers(h.\langle \leftarrow this, fr', closeW, \_\rangle) & \triangleq Readers(h) \setminus \{irev(h, fr').caller\} \\
Readers(h.\langle \leftarrow this, fr', others\rangle) & \triangleq Readers(h)
\end{align*}
\]
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[Ent] \rightarrow Nat` by:
\[
Reading(h) = \#(h/(this \rightarrow db, \_, read, \_)) - \#(h/(this \leftarrow, Futures(h, db, read), \_))
\]
where `Futures(h, db, read) \triangleq ToSet((h/(this \rightarrow db, \_, read, \_), 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:
\[
I = I_1 \land I_2 \land I_3 \land I_4
\]
where
\[
\begin{align*}
I_1 & \triangleq Readers(\mathcal{H}) = readers \\
I_2 & \triangleq Writers(\mathcal{H}) = \{writer\} \\
I_3 & \triangleq Reading(\mathcal{H}) = pr \\
I_4 & \triangleq OK(\mathcal{H})
\end{align*}
\]
where `\{writer\} = \emptyset` 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
\[ \vdash (\text{wf}(H) \land I_C) \Rightarrow \{ H := H \cdot H_{\text{begin}} \} (I_C \land \text{wf}(H)) \]

\[ \vdash (\text{wf}(H) \land I_C) \Rightarrow \forall \mathcal{H}. \{ H := H \cdot H_{\text{begin}} \vdash H \}'(I_C \land \text{wf}(H) \land (\text{writer} = \text{None}) \Rightarrow \{ \text{readers} := \text{insertElement}(\text{readers}, \text{caller}) \} [H := H \cdot H_{\text{end}}] (\text{wf}(H) \Rightarrow I_C) \]

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

The same value as \( \text{readers} \). The predicate \( \text{OK} : \text{Seq}[E_\text{d}] \rightarrow \text{Bool} \) is defined inductively over the history by:

\[
\begin{align*}
\text{OK}(e) & \triangleq \text{true} \\
\text{OK}(h \cdot \langle \rightarrow \text{this}, f', \text{openR}, \_ \rangle) & \triangleq \text{OK}(h) \land \# \text{Writers}(h) = 0 \quad (1) \\
\text{OK}(h \cdot \langle \rightarrow \text{this}, f', \text{openW}, \_ \rangle) & \triangleq \text{OK}(h) \land \# \text{Writers}(h) = 0 \quad (2) \\
\text{OK}(h \cdot \langle \rightarrow \text{this} \rightarrow \text{db}, f', \text{write}, \_ \rangle) & \triangleq \text{OK}(h) \land \text{Reading}(h) = 0 \land \# \text{Writers}(h) = 1 \quad (3) \\
\text{OK}(h \cdot \langle \rightarrow \text{this} \rightarrow \text{db}, f', \text{read}, \_ \rangle) & \triangleq \text{OK}(h) \land \text{Writing}(h) = 0 \quad (4) \\
\text{OK}(h \cdot \text{others}) & \triangleq \text{OK}(h)
\end{align*}
\]

Here, conditions (1) and (2) reflect the fairness condition: invocations of \text{openR} and \text{openW} compete on equal terms for the guard \text{writer} = \text{null}, which equals \text{Writers}(h) = \emptyset by I2. If \text{writer} is different from \text{null}, conditions (1) and (2) additionally ensure that no clients can be included in the \text{readers} set or be assigned to \text{writer}. Condition (3) captures the guard in \text{write}: when invoking \text{db.write}, there cannot be any pending calls to \text{db.read}. Correspondingly, Condition (4) expresses that when invoking \text{db.read}, there is no incomplete writing operation. The invariant \( I \) implies that no reading and writing activity happens simultaneously:

\[
\text{Reading}(H) = 0 \lor \text{Writing}(H) = 0
\]

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 prefix-closed verification of method openR with respect to the invariant \( I_1 : \text{Readers}(H) = \text{readers} \) 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 \( \text{OK}(H) \) is prefix-closed and may be used as a composable history invariant. Remark that the property \( \text{Writing}(H) = 0 \) can be verified as a part of the class invariant since \text{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

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

References


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:

$$Dd ::= \text{data } D = [Co(T^*])^*$$

$$F ::= \text{def } T \ fn([T \rightarrow D^*]) ::= \text{rhs}$$

$$t ::= Co(c^e) \mid fn([c]^e)$$

$$p ::= v \mid Co(p^e) \mid (p, p)$$

$$rhs ::= e$$

$$b ::= p \Rightarrow rhs$$

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 $\text{and}$ and $\text{or}$ for booleans, and $=$ for equality.

B Auxiliary Functions

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

$$\text{oid}(c) \triangleq \{\text{null}\}$$

$$\text{oid}(o \rightarrow d', u, m, \gamma) \triangleq \{o, d'\} \cup \text{oid}(\gamma)$$

$$\text{oid}(\langle o \leftarrow, u, m, c\rangle) \triangleq \{o\} \cup \text{oid}(c)$$

$$\text{oid}(o \rightarrow d', C, \gamma) \triangleq \{o, d'\} \cup \text{oid}(\gamma)$$

$$\text{oid}(h \leftarrow \gamma) \triangleq \text{oid}(h) \cup \text{oid}(\gamma)$$

$$\text{oid}(\langle o \rightarrow, u, m, \gamma\rangle) \triangleq \{o\} \cup \text{oid}(\gamma)$$

$$\text{oid}(\langle o \leftarrow, u, \gamma\rangle) \triangleq \{o\} \cup \text{oid}(\gamma)$$

$$\text{oid}(\langle o \rightarrow, u, C, \gamma\rangle) \triangleq \{d, o\} \cup \text{oid}(\gamma)$$
where $\gamma : E_v$, and $oid(\tau)$ returns the set of object identifiers occurring in the expression list $\tau$.

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] × Fid → IREv

irev(ε, fr') ≜ ⊥
irev(h · event, fr') ≜ if fr' ∈ event then event else irev(h, fr') fi

C.3 Definition of Writers

Writers : Seq[Ev] → Set[Obj]

Writers(ε) ≜ ∅
Writers(h · (this, fr', openW, _)) ≜ Writers(h) ∪ {irev(h, fr').caller}
Writers(h · (this, fr', closeW, _)) ≜ Writers(h) \ {irev(h, fr').caller}
Writers(h · others) ≜ Writers(h)

C.4 Definition of Writing

Writing : Seq[Ev] → Nat

Writing(h) ≜ #(h⟨this → db, _, write, _⟩) − #(h⟨this ← Futures(h, db, write), _⟩)
Futures(h, db, write) ≜ ToSet((h⟨this → db, _, write, _⟩).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 Hbegin
and Hend respectively. Hbegin : ⟨(call → this, destiny, m, J)⟩ and Hend : ⟨(← this, destiny, m, return)⟩.

C.5.1 openR

⊢ (wf(H) ∧ IC) ⇒ {H := H · Hbegin} (IC ∧ wf(H))
⊢ (wf(H) ∧ IC) ⇒ ∀h′, h. {(H := H · Hbegin) ⇒ h′}(IC ∧ wf(H) ∧ (writer = None) ⇒
                             {readers := insertElement(readers, caller)}[H := H · Hend] (wf(H) ⇒ IC))

⊢ (wf(H) ∧ IC) ⇒ [H := H · Hbegin; await writer = None;
                             readers := insertElement(readers, caller); H := H · Hend] (wf(H) ⇒ IC)
C.5.2 openW

\[(\text{wf}(H) \land I_C) \Rightarrow \{ H := H \cdot \text{begin} \}(I_C \land \text{wf}(H)) \]

\[(\text{wf}(H) \land I_C) \Rightarrow \forall \pi, \pi'. \{ H := H \cdot \text{begin} \vdash \pi' \}(I_C \land \text{wf}(H)) \land (\text{write} = \text{None}) \Rightarrow
\begin{align*}
& \text{writer := caller} \land \text{readers := insertElement(readers, caller)}; H := H \cdot \text{end}(\text{wf}(H) \Rightarrow I_C) \\
& \text{write := None; writer := caller;}
\end{align*}
\]

C.5.3 closeR

\[(\text{wf}(H) \land I_C) \Rightarrow \{ H := H \cdot \text{begin} \}(I_C \land \text{wf}(H)) \]

\[(\text{wf}(H) \land I_C) \Rightarrow \forall \pi, \pi'. \{ H := H \cdot \text{begin} \vdash \pi' \}(I_C \land \text{wf}(H)) \land (\text{writer} = \text{caller}) \Rightarrow
\begin{align*}
& \text{writer := None} \land \text{readers := remove(readers, caller)}; H := H \cdot \text{end}(\text{wf}(H) \Rightarrow I_C) \\
& \text{readers := remove(readers, caller)}; H := H \cdot \text{end}(\text{wf}(H) \Rightarrow I_C)
\end{align*}
\]

C.5.4 closeW

\[(\text{wf}(H) \land I_C) \Rightarrow \{ H := H \cdot \text{begin} \}(I_C \land \text{wf}(H)) \]

\[(\text{wf}(H) \land I_C) \Rightarrow \forall \pi, \pi'. \{ H := H \cdot \text{begin} \vdash \pi' \}(I_C \land \text{wf}(H)) \land (\text{write} = \text{None}) \Rightarrow
\begin{align*}
& \text{writer := None} \land \text{readers := remove(readers, caller)}; H := H \cdot \text{end}(\text{wf}(H) \Rightarrow I_C) \\
& \text{readers := remove(readers, caller)}; H := H \cdot \text{end}(\text{wf}(H) \Rightarrow I_C)
\end{align*}
\]

C.5.5 read

\[(\text{wf}(H) \land I_C) \Rightarrow \{ H := H \cdot \text{begin} \}(\text{fr := null} \land \text{data := } \epsilon \}(I_C \land \text{wf}(H)) \]

\[(\text{wf}(H) \land I_C) \Rightarrow \forall \pi, \pi'. \{ H := H \cdot \text{begin} \vdash \pi' \}(\text{fr := null} \land \text{data := } \epsilon \}(I_C \land \text{wf}(H)) \land \text{contains(readers, caller)} \Rightarrow
\begin{align*}
& \forall \pi, \pi'. \{ \text{pr := pr + 1} \}(H := H \cdot \langle \text{this } \Rightarrow \text{db}, \text{fr'}, \text{read}, \text{key} \rangle; \text{fr := fr'} \}(I_C \land \text{wf}(H)) \\
& \text{fr := fr'} \vdash \{ H := H \cdot \langle \text{this } \Rightarrow \text{db}, \text{fr'}, \text{read}, \text{key} \rangle; \text{fr := fr'} \}(I_C \land \text{wf}(H)) \Rightarrow
\end{align*}
\]

C.5.6 write

\[B := \text{caller} \land \text{write} \land \text{pr} = 0 \land \text{size(readers) = 1}) \]

\[(\text{readers := EmptySet} \land \text{contains(readers, writer)} \land \text{size(readers) = 1}) \]

\[(\text{wf}(H) \land I_C) \Rightarrow \{ H := H \cdot \text{begin} \}(I_C \land \text{wf}(H)) \land \text{write} \land \text{pr = 0} \land \text{size(readers) = 1}) \Rightarrow
\begin{align*}
& \text{readers := EmptySet} \land \text{contains(readers, writer)} \land \text{size(readers) = 1}) \land \text{write} \land \text{pr = 0} \land \text{size(readers) = 1}) \Rightarrow
\end{align*}
\]

C.6 Proof details

We write \{\text{caller}\} as the abbreviation of \{\text{irex(h,destiny,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

\[I_C = I_1 : \text{Readers(H)} = \text{readers} \]

\[H_{\text{begin}} : \langle \text{caller } \Rightarrow \text{this, destiny, openR, x} \rangle \]

\[H_{\text{end}} : \langle \text{this, destiny, openR, return} \rangle \]

\[(1)
\]

\[(\text{wf}(H) \land \text{Readers(H) = readers}) \Rightarrow
\begin{align*}
& \{ H := H \cdot \text{begin} \}(\text{Readers(H) = readers} \land \text{wf}(H)) \Rightarrow
\end{align*}
\]
\((wfh(\mathcal{H}) \land \text{Readers}(\mathcal{H}) = \text{readers}) \Rightarrow \)
\((\text{Readers}(\mathcal{H} \cdot \mathcal{H}_{\text{begin}}) = \text{readers} \land wfh(\mathcal{H} \cdot \mathcal{H}_{\text{begin}}))\)
\((wfh(\mathcal{H}) \land \text{Readers}(\mathcal{H}) = \text{readers}) \Rightarrow \)
\((\text{Readers}(\mathcal{H} \cdot \mathcal{H}_{\text{begin}}) = \text{readers} \land wfh(\mathcal{H} \cdot \mathcal{H}_{\text{begin}}))\)

\((wfh(\mathcal{H}) \land \text{Readers}(\mathcal{H}) = \text{readers}) \Rightarrow \)
\(\forall \mathcal{H}, \mathcal{H}', (\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{\text{begin}} \mapsto \mathcal{H}') (\text{Readers}(\mathcal{H}) = \text{readers} \land wfh(\mathcal{H}) \land (\text{writer} = \text{None}) \Rightarrow \)
\{\text{readers} := \text{insertElement}(\text{readers}, \text{caller}) \| \mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{\text{end}}\}(wfh(\mathcal{H}) \Rightarrow \text{Readers}(\mathcal{H}) = \text{readers})\)
\((wfh(\mathcal{H}) \land \text{Readers}(\mathcal{H}) = \text{readers}) \Rightarrow \)
\(\forall \mathcal{H}, \mathcal{H}', (\text{Readers}(\mathcal{H} \cdot \mathcal{H}_{\text{begin}} \mapsto \mathcal{H}') = \text{readers} \land wfh(\mathcal{H} \cdot \mathcal{H}_{\text{begin}} \mapsto \mathcal{H}') \land (\text{writer} = \text{None}) \Rightarrow \)
\{\text{readers} := \text{insertElement}(\text{readers}, \text{caller})\}(wfh(\mathcal{H}) \Rightarrow \text{Readers}(\mathcal{H} \cdot \mathcal{H}_{\text{begin}} \mapsto \mathcal{H}') = \text{insertElement}(\text{readers}, \text{caller}))\)

\((wfh(\mathcal{H}) \land \text{Readers}(\mathcal{H}) = \text{readers}) \Rightarrow \)
\(\forall \mathcal{H}, \mathcal{H}', (\text{Readers}(\mathcal{H} \cdot \mathcal{H}_{\text{begin}} \mapsto \mathcal{H}') = \text{readers} \land wfh(\mathcal{H} \cdot \mathcal{H}_{\text{begin}} \mapsto \mathcal{H}') \land (\text{writer} = \text{None}) \Rightarrow \)
\{\text{readers} := \text{insertElement}(\text{readers}, \text{caller})\}(wfh(\mathcal{H}) \Rightarrow \text{Readers}(\mathcal{H} \cdot \mathcal{H}_{\text{begin}} \mapsto \mathcal{H}') = \text{insertElement}(\text{readers}, \text{caller}))\)

**C.6.2 openW**

\(I_C = I_1 \land I_2 : \text{Readers}(\mathcal{H}) = \text{readers} \land \text{Writers}(\mathcal{H}) = \{\text{writer}\}\)
\(\mathcal{H}_{\text{begin}} : \langle \text{caller} \mapsto \text{this, destiny, openW}, \_ \rangle\)
\(\mathcal{H}_{\text{end}} : \langle \_ \mapsto \text{this, destiny, openW, return} \rangle\)

\((wfh(\mathcal{H}) \land \text{Readers}(\mathcal{H}) = \text{readers} \land \text{Writers}(\mathcal{H}) = \{\text{writer}\}) \Rightarrow \)
\(\{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{\text{begin}} \mapsto \mathcal{H}'\}(\text{Readers}(\mathcal{H}) = \text{readers} \land \text{Writers}(\mathcal{H}) = \{\text{writer}\} \land wfh(\mathcal{H}))\)
\((wfh(\mathcal{H}) \land \text{Readers}(\mathcal{H}) = \text{readers} \land \text{Writers}(\mathcal{H}) = \{\text{writer}\}) \Rightarrow \)
\(\{\text{Readers}(\mathcal{H} \cdot \mathcal{H}_{\text{begin}}) = \text{readers} \land \text{Writers}(\mathcal{H} \cdot \mathcal{H}_{\text{begin}}) = \{\text{writer}\} \land wfh(\mathcal{H} \cdot \mathcal{H}_{\text{begin}})\}(wfh(\mathcal{H}) \Rightarrow \text{Readers}(\mathcal{H} \cdot \mathcal{H}_{\text{begin}}) = \{\text{writer}\} \land wfh(\mathcal{H} \cdot \mathcal{H}_{\text{begin}}))\)

\((wfh(\mathcal{H}) \land \text{Readers}(\mathcal{H}) = \text{readers} \land \text{Writers}(\mathcal{H}) = \{\text{writer}\}) \Rightarrow \)
\(\forall \mathcal{H}, \mathcal{H}', (\text{Readers}(\mathcal{H} \cdot \mathcal{H}_{\text{begin}} \mapsto \mathcal{H}') = \text{readers} \land wfh(\mathcal{H} \cdot \mathcal{H}_{\text{begin}} \mapsto \mathcal{H}') \land (\text{writer} = \text{None}) \Rightarrow \)
\{\text{writer} := \text{caller} \| \text{readers} := \text{insertElement}(\text{readers}, \text{caller})\}(\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{\text{end}}\)(wfh(\mathcal{H}) \Rightarrow \text{Readers}(\mathcal{H}) = \text{readers} \land \text{Writers}(\mathcal{H}) = \{\text{writer}\})))\)

\((wfh(\mathcal{H}) \land \text{Readers}(\mathcal{H}) = \text{readers} \land \text{Writers}(\mathcal{H}) = \{\text{writer}\}) \Rightarrow \)
\(\forall \mathcal{H}, \mathcal{H}', (\text{Readers}(\mathcal{H} \cdot \mathcal{H}_{\text{begin}} \mapsto \mathcal{H}') = \text{readers} \land wfh(\mathcal{H} \cdot \mathcal{H}_{\text{begin}} \mapsto \mathcal{H}') \land (\text{writer} = \text{None}) \Rightarrow \)
\{\text{writer} := \text{caller} \| \text{readers} := \text{insertElement}(\text{readers}, \text{caller})\}(\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{\text{end}}\)(wfh(\mathcal{H}) \Rightarrow \text{Readers}(\mathcal{H}) = \text{readers} \land \text{Writers}(\mathcal{H}) = \{\text{writer}\})))\)

\((wfh(\mathcal{H}) \land \text{Readers}(\mathcal{H}) = \text{readers} \land \text{Writers}(\mathcal{H}) = \{\text{writer}\}) \Rightarrow \)
\(\forall \mathcal{H}, \mathcal{H}', (\text{Readers}(\mathcal{H} \cdot \mathcal{H}_{\text{begin}} \mapsto \mathcal{H}') = \text{readers} \land wfh(\mathcal{H} \cdot \mathcal{H}_{\text{begin}} \mapsto \mathcal{H}') \land (\text{writer} = \text{None}) \Rightarrow \)
\{\text{writer} := \text{caller} \| \text{readers} := \text{insertElement}(\text{readers}, \text{caller})\}(\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{\text{end}}\)(wfh(\mathcal{H}) \Rightarrow \text{Readers}(\mathcal{H}) = \text{readers} \land \text{Writers}(\mathcal{H}) = \{\text{writer}\})))\)

**C.6.3 closeR**

\(I_C = I_1 : \text{Readers}(\mathcal{H}) = \text{readers}\)
\(\mathcal{H}_{\text{begin}} : \langle \text{caller} \mapsto \text{this, destiny, closeR}, \_ \rangle\)
\(\mathcal{H}_{\text{end}} : \langle \_ \mapsto \text{this, destiny, closeR, return} \rangle\)

\((wfh(\mathcal{H}) \land \text{Readers}(\mathcal{H}) = \text{readers}) \Rightarrow \)
\{\mathcal{H} := \mathcal{H} \cdot \mathcal{H}_{\text{begin}} \cdot \mathcal{H}_{\text{end}}\}(\text{readers} := \text{remove}(\text{readers}, \text{caller}))\)(wfh(\mathcal{H}) \Rightarrow \text{Readers}(\mathcal{H}) = \text{readers})\)
\((wfh(\mathcal{H}) \land \text{Readers}(\mathcal{H}) = \text{readers}) \Rightarrow \)
\{wfh(\mathcal{H} \cdot \mathcal{H}_{\text{begin}} \cdot \mathcal{H}_{\text{end}}) \Rightarrow \text{Readers}(\mathcal{H} \cdot \mathcal{H}_{\text{begin}} \cdot \mathcal{H}_{\text{end}}) = \text{remove}(\text{readers}, \text{caller})\)
I C = I 1 \land I 2 \land I 3 \land I 4 : Readers(H) = readers \land Writers(H) = \{writer\} \land Writing(H) = pr \land OK(H)

The property Writing(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.

H begin : \langle caller \rightarrow this, destiny, read, \pi \rangle
H end : \langle \leftarrow this, destiny, readW, return \rangle

C.6.5 \textbf{read}

IC = I 1 \land I 2 \land I 3 \land I 4 : Readers(H) = readers \land Writers(H) = \{writer\} \land Reading(H) = pr \land OK(H)

\textbf{C.6.5.1 closeW}

H begin : \langle caller \rightarrow this, destiny, closeW, \pi \rangle
H end : \langle \leftarrow this, destiny, closeW, return \rangle

(1)
\begin{align*}
wf(H) \land Readers(H) & = readers \land Writers(H) = \{writer\}\rangle \Rightarrow \\
\{H := H \cdot Hbegin\} (Readers(H) & = readers \land Writers(H) = \{writer\} \land wf(H)) \\
wf(H) \land Readers(H) & = readers \land Writers(H) = \{writer\}\rangle \Rightarrow \\
\{Readers(H \cdot Hbegin) \land Writers(H \cdot Hbegin) = \{writer\} \land wf(H \cdot Hbegin)\}
\end{align*}

(2)
\begin{align*}
wf(H) \land Readers(H) & = readers \land Writers(H) = \{writer\}\rangle \Rightarrow \\
\forall \forall. h'. \{H := H \cdot Hbegin | h'\} (Readers(H) & = readers \land \\
Writers(H) = \{writer\} \land writer = caller\} \Rightarrow \\
\{writer := \text{None } \mid \text{ readers := remove(readers, caller)}\} \{H := H \cdot Hend\} \\
wf(H) & \rightarrow \{Readers(H) = readers \land Writers(H) = \{writer\}\}
\end{align*}

\begin{align*}
wf(H) \land Readers(H) & = readers \land Writers(H) = \{writer\}\rangle \Rightarrow \\
\forall \forall. h'. \{H := H \cdot Hbegin | h'\} (Readers(H) & = readers \land \\
Writers(H) = \{writer\} \land writer = caller\} \Rightarrow \\
\{wf(H \cdot Hbegin | h' \cdot Hend) \Rightarrow \\
\{Readers(H \cdot Hbegin | h' \cdot Hend) = \text{remove(readers, caller)} \land \\
Writers(H \cdot Hbegin | h' \cdot Hend) = \{\text{None}\})\}
\end{align*}

(2)
\begin{align*}
wf(H) \land Readers(H) & = readers \land Writers(H) = \{writer\}\rangle \Rightarrow \\
\forall \forall. h'. \{H := H \cdot Hbegin | h'\} (Readers(H) & = readers \land \\
Writers(H) = \{writer\} \land writer = caller\} \Rightarrow \\
\{wf(H \cdot Hbegin | h' \cdot Hend) \Rightarrow \\
\{Readers(H \cdot Hbegin | h' \cdot Hend) = \text{remove(readers, caller)} \land \\
Writers(H \cdot Hbegin | h' \cdot Hend) = \{\text{None}\})\}
\end{align*}

C.6.5 \textbf{read}

IC = I 1 \land I 2 \land I 3 \land I 4 : Readers(H) = readers \land Writers(H) = \{writer\} \land Reading(H) = pr \land OK(H)

The property Writing(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.

H begin : \langle caller \rightarrow this, destiny, read, \pi \rangle
H end : \langle \leftarrow this, destiny, read, return \rangle

(1)
\begin{align*}
wf(H) \land Readers(H) & = readers \land Writers(H) = \{writer\} \land Reading(H) = pr \land OK(H)\rangle \Rightarrow \\
\{H := H \cdot Hbegin\} fr := \text{null} \mid data := \epsilon \langle \{Readers(H) = readers \land Writers(H) = \{writer\} \land Reading(H) = pr \land OK(H) \land wf(H)\}
\end{align*}

(2)
\begin{align*}
wf(H) \land Readers(H) & = readers \land Writers(H) = \{writer\} \land Reading(H) = pr \land OK(H)\rangle \Rightarrow \\
\{Readers(H \cdot Hbegin) = \text{readers } \land Writers(H \cdot Hbegin) = \{writer\} \land Reading(H \cdot Hbegin) = pr \land OK(H) \land wf(H \cdot Hbegin)\}
\end{align*}

\begin{align*}
wf(H) \land Readers(H) & = readers \land Writers(H) = \{writer\} \land Reading(H) = pr \land OK(H)\rangle \Rightarrow \\
\{Readers(H) = readers \land Writers(H) = \{writer\} \land Reading(H) = pr \land OK(H) \land wf(H \cdot Hbegin)\}
\end{align*}

(2)
\begin{align*}
wf(H) \land Writing(H) & = 0 \land Readers(H) = readers \land Writers(H) = \{writer\} \land \\
Reading(H) & = pr \land OK(H)\rangle \Rightarrow \\
\forall \forall. h'. \{H := H \cdot Hbegin | h'\} fr := \text{null} \mid data := \epsilon
\end{align*}
(Writing(H) = 0 ∧ Readers(H) = readers ∧ Writers(H) = {writer} ∧ Reading(H) = pr ∧ OK(H) ∧ wfr(H) ∧ contains(readers, caller) ⇒
∀fr f . (pr := pr + 1)[H := H · (this → db, fr, read, key)][fr := fr]
(Writing(H) = 0 ∧ Readers(H) = readers ∧ Writers(H) = {writer} ∧ Reading(H) = pr ∧ OK(H) ∧ wfr(H)))

(wfr(H) ∧ Writing(H) = 0 ∧ Readers(H) = readers ∧ Writers(H) = {writer} ∧ Reading(H) = pr ∧ OK(H) ⇒
∀pr, hr . (Writing(H · Hbegin = hr) = 0 ∧ Readers(H · Hbegin = hr) = readers ∧ Writers(H · Hbegin = hr) = {writer} ∧ Reading(H · Hbegin = hr) = pr ∧ OK(H · Hbegin = hr)) ⇒
∀fr f . (Writing(H · Hbegin = hr) = fr) ∧ contains(readers, caller) ⇒
∀fr f . (Writing(H · Hbegin = hr) = fr) ∧ contains(readers, caller) ⇒
∀fr f . (Writing(H · Hbegin = hr) = fr) ∧ contains(readers, caller) ⇒
∀fr f . (Writing(H · Hbegin = hr) = fr) ∧ contains(readers, caller) ⇒
∀fr f . (Writing(H · Hbegin = hr) = fr) ∧ contains(readers, caller) ⇒
∀fr f . (Writing(H · Hbegin = hr) = fr) ∧ contains(readers, caller) ⇒
∀fr f . (Writing(H · Hbegin = hr) = fr) ∧ contains(readers, caller) ⇒
∀fr f . (Writing(H · Hbegin = hr) = fr) ∧ contains(readers, caller) ⇒
C.6.6 write

$I_C = I_2 \land I_3 \land I_4 : \text{Writers}(H) = \{\text{writer}\} \land \text{Reading}(H) = \text{pr} \land OK(H)$

$H_{begin} : \langle \text{caller} \rightarrow \text{this, destiny, write, \pi} \rangle$

$H_{end} : \langle \langle \text{this, destiny, write, return} \rangle \rangle$

$B : \text{caller} = \text{writer} \land \& \land \text{pr} = 0 \land \& \&$

$(\text{readers} = \text{EmptySet} \cup \{\text{contains(readers, writer} \& \& \text{size(readers) = 1})\}$

(1)

$(\text{wf}(H) \land \text{Writers}(H) = \{\text{writer}\} \land \text{Reading}(H) = \text{pr} \land OK(H)) \Rightarrow$

$(\{H : H \cdot \text{Hbegin} \vdash h' : (\text{this} \rightarrow \text{db, fr'), read, keyy} \rightarrow h'') \}$

$(\text{Writers}(H \cdot H_{begin} \vdash h' : (\text{this} \rightarrow \text{db, fr'), read, keyy} \rightarrow h'') \Rightarrow$

$(\text{OK}(H \cdot H_{begin} \vdash h' : (\text{this} \rightarrow \text{db, fr'), read, keyy} \rightarrow h'') = \text{pr} \land$

# Writers(H \cdot H_{begin} \vdash h'') = 1\}$

(2)

$(\text{wf}(H) \land \text{Writers}(H) = \{\text{writer}\} \land \text{Reading}(H) = \text{pr} \land OK(H)) \Rightarrow$

$\forall h', fr', v'. \{H : H \cdot \text{Hbegin} \vdash h' \}$

$(\text{Writers}(H) = \{\text{writer}\} \land \text{Reading}(H) = \text{pr} \land OK(H) \land \text{wf}(H) \land B \Rightarrow$

$(\{H : H \cdot \text{this} \rightarrow \text{db, fr'}, \text{write} \cdot \text{(key, value)} \cdot \langle \text{this} \rightarrow \rightarrow, fr', v'\rangle \cdot \text{Hend}) \}$

$(\text{wf}(H) \Rightarrow \text{Writers}(H) = \{\text{writer}\} \land \text{Reading}(H) = \text{pr} \land OK(H)) \Rightarrow$

$\forall h', fr', v'. \{\text{Writers}(H \cdot H_{begin} \vdash h') = \{\text{writer}\} \land \text{Reading}(H \cdot H_{begin} \vdash h') = \text{pr} \land$

OK(H \cdot H_{begin} \vdash h') \land \text{wf}(H \cdot H_{begin} \vdash h') \land B \Rightarrow$

$(\text{Writers}(H \cdot H_{begin} \vdash h' : \langle \text{this} \rightarrow \text{db, fr'), read, write, (key, value)} \cdot \langle \text{this} \rightarrow \rightarrow, fr', v'\rangle \cdot \text{Hend}) \Rightarrow$

$(\text{Writers}(H \cdot H_{begin} \vdash h' : \langle \text{this} \rightarrow \text{db, fr'), read, write, (key, value)} \cdot \langle \text{this} \rightarrow \rightarrow, fr', v'\rangle \cdot \text{Hend}) = \{\text{writer}\} \land$

OK(H \cdot H_{begin} \vdash h') \Rightarrow \text{Reading}(H \cdot H_{begin} \vdash h') = \text{pr} \land$

# Writers(H \cdot H_{begin} \vdash h') = 1\}$

29