UNIVERSITY OF OSLO
Department of Informatics

Compositional and Sound Reasoning about Active Objects with Shared Futures

Research Report 437

Crystal Chang Din

Olaf Owe

ISBN 978-82-7368-402-8
ISSN 0806-3036

February 2014

Contents

Compositional and Sound Reasoning about Active Objects with Shared Futures [*]

Crystal Chang Din, Olaf Owe
Dept. of Informatics, University of Oslo,
P.O. Box 1080 Blindern, N-0316 Oslo, Norway.
E-mails: {crystald,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. The *future mechanism* extends the traditional method call communication model by facilitating sharing of references to futures. By assigning method call result values to futures, third party objects may pick up these values. This may reduce the time spent waiting for replies in a distributed environment. However, futures add a level of complexity to program analysis, as the program semantics becomes more involved.

This paper presents a model for asynchronously communicating objects, where return values from method calls are handled by futures. The model facilitates invariant specifications over the locally visible communication history of each object. Compositional reasoning is supported and proved sound, as each object may be specified and verified independently of its environment. A kernel object-oriented language with futures inspired by the ABS modeling language is considered. A compositional proof system for this language is presented, formulated within dynamic logic.

1 Introduction

Distributed systems play an essential role in society today. However, quality assurance of distributed systems is non-trivial since they depend on unpredictable factors, such as different processing speeds of independent components. Therefore, it is highly challenging to test such distributed systems after deployment under different relevant conditions. These challenges motivates frameworks combining precise modeling and analysis with suitable tool support. In particular, *compositional verification systems* allow the different components to be analyzed independently from their surrounding components.
Object orientation is the leading framework for concurrent and distributed systems, recommended by the RM-ODP [26]. However, method-based communication between concurrent units may cause busy-waiting, as in the case of remote and synchronous method invocation, e.g., Java RMI [3]. Concurrent objects communicating by *asynchronous method calls* have been proposed as a promising framework to combine object-orientation and distribution in a natural manner. Each concurrent object encapsulates its own state and processor, and internal interference is avoided as at most one process

[*] 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.

is executing on an object at a time. Asynchronous method calls allow the caller to continue with its own activity without blocking while waiting for the reply, and a method call leads to a new process on the called object. The notion of *futures* [6, 23, 30, 35] improves this setting by providing a decoupling of the process invoking a method and the process reading the returned value. By sharing *future identities*, the caller enables other objects to wait for method results. However, futures complicate program analysis since programs become more involved compared to semantics with traditional method calls, and in particular local reasoning is a challenge.

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 [8, 25]. At any point in time the communication history abstractly captures the system state [11, 12]. In fact, traces are used in semantics for full abstraction results (e.g., [1, 27]). The *local history* of an object reflects the communication visible to that object, i.e., between the object and its surroundings. A system may be specified by the finite initial segments of its communication histories, and a *history invariant* is a predicate which holds for all finite sequences in the set of possible histories, expressing safety properties [5].

In this work we consider a kernel object-oriented language, where futures are used to manage return values of method calls. Objects are concurrent and communicate asynchronously. We formalize object communication by a four event operational semantics, capturing shared futures, where each event is visible to only one object. Consequently, the local histories of two different objects share no common events, and history invariants can be established independently for each object. We present a *dynamic logic* proof system for class verification, facilitating independent reasoning about each class. A verified *class invariant* can be instantiated to each object of that class, resulting in an invariant over the local history of the object. Modularity is achieved as the independently derived history invariants can be composed to form *global* system specifications. Global history consistency is captured by a notion of history *wellformedness.* The formalization of object communication extends previous work [18] which considered concurrent objects and asynchronous communication, but without futures.

Paper overview. Sect. 2 presents a core language with shared futures. The communication model is presented in Sect. 3, and Sect. 4 defines the operational semantics. Sect. 5 presents the compositional reasoning system, and Sect. 6 contains related work and concludes the paper.

2 A Core Language with Shared Futures

A *future* is a placeholder for the return value of a method call. Each future has a unique identity which is *generated* when a method is invoked. The future is *resolved* upon method termination, by placing the return value of the method in the future. Thus, unlike the traditional method call mechanism, the callee does not send the return value directly back to the caller. However, the caller may keep a *reference* to the future, allowing the caller to *fetch* the future value once resolved. References to futures may be shared between objects, e.g., by passing them as parameters. After achieving a future reference, this means that third party objects may fetch the future value. Thus, the future value may be fetched several times, possibly by different objects. In this manner, shared futures provide an efficient way to distribute method call results to a number of objects.

For the purposes of this paper, we consider a core object-oriented language with futures, presented in Fig 1. It includes basic statements for first order futures, inspired by *ABS* [24]. Class instances are concurrent, encapsulating their own state and processor. Each method invoked on the object leads to a new process, and at most one process is executing

$$
\begin{array}{lcll}
Cl & ::= & \mathbf{class}\ C([T\ cp]^*)\ \{[T\ w\ [:= e]^?]^*\ s\ M^*\} & \text{class definition}\\
M & ::= & T\ m([T\ x]^*)\ \{[\mathbf{var}\ [T\ x]^*]^?\ s\ ;\ \mathbf{return}\ e\} & \text{method definition}\\
T & ::= & C \mid \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{String} \mid \mathsf{Void} \mid \mathsf{Fut}{<}T{>} & \text{types}\\
v & ::= & x \mid w & \text{variables (local or field)}\\
e & ::= & \mathbf{null} \mid \mathsf{this} \mid v \mid cp \mid f(\overline{e}) & \text{pure expressions}\\
s & ::= & v := e \mid fr := v!m(\overline{e}) \mid v := e? & \text{statements}\\
 & & \mid \mathbf{skip} \mid \mathbf{if}\ e\ \mathbf{then}\ s\ [\mathbf{else}\ s]^?\ \mathbf{fi} \mid s; s & \\
 & & \mid \mathbf{while}\ e\ \mathbf{do}\ s\ \mathbf{od} \mid v := \mathbf{new}\ C(\overline{e}) &
\end{array}
$$

Figure 1: Core language syntax, with C class name, cp formal class parameter, m method name, w fields, x method parameter or local variable, and where fr is a future variable. We let $[\ ]^*$ and $[\ ]^?$ denote repeated and optional parts, respectively, and $\overline{e}$ is a (possibly empty) expression list. Expressions e and functions f are side-effect free.

on an object at a time. Object communication is *asynchronous*, as there is no explicit transfer of control between the caller and the callee. Methods are organized in classes in a standard manner. A class C takes a list of formal parameters $\overline{cp}$, defines fields $\overline{w}$, initialization block s and methods $\overline{M}$. There is read-only access to the parameters $\overline{cp}$. A method definition has the form $m(\overline{x})\{\mathbf{var}\ \overline{y};\ s;\ \mathbf{return}\ e\}$, ignoring type information, where $\overline{x}$ is the list of parameters, $\overline{y}$ an optional list of *method-local variables*, s is a sequence of statements, and the value of e is returned upon termination.

A future variable fr is declared by $\mathsf{Fut}{<}T{>}\ fr$, indicating that fr may refer to futures which will eventually contain values of type T. The call statement $fr := x!m(\overline{e})$ invokes the method m on object x with input values $\overline{e}$. The identity of the generated future is assigned to fr, and the calling process continues execution without waiting for fr to become resolved. The query statement $v := fr?$ is used to fetch the value of a future. The statement blocks until fr is resolved, and then assigns the value contained in fr to v. The language contains additional statements for assignment, **skip**, conditionals, and sequential composition.

We assume that call and query statements are well-typed. If x refers to an object where m is defined with no input values and return type Int, the following is a well-typed blocking method call: $\mathsf{Fut}{<}\mathsf{Int}{>}\ fr;\ \mathsf{Int}\ v;\ fr := x!m();\ v := fr?$.

To avoid blocking, *ABS* provides statements for process control, including a statement **await** $fr?$, which releases the current process as long as fr is not yet resolved. This gives rise to more efficient computing with futures. It is possible to add a treatment of process release statements as a straight forward extension of the present work, following the approach of [18]. We here focus on a core language for futures, with a simple semantics, avoiding specialized features such as process control. The core language ignores *ABS* features that are orthogonal to shared futures, including interface encapsulation, inheritance, local synchronous calls, and internal scheduling of processes by means of cooperative multitasking. We refer to the report version of this paper for a treatment of these issues [19].

2.1 An Example

In order to illustrate the usage of futures, we consider the problem of counting the number of occurrences of each word in a large collection of documents. We consider the computing model MapReduce in Fig. 2. MapReduce is invented and used heavily by Google for efficient distributed computing over large data sets [17]. It has three major steps: Map, Shuffle and Reduce. The Map phase runs over input data, which might be

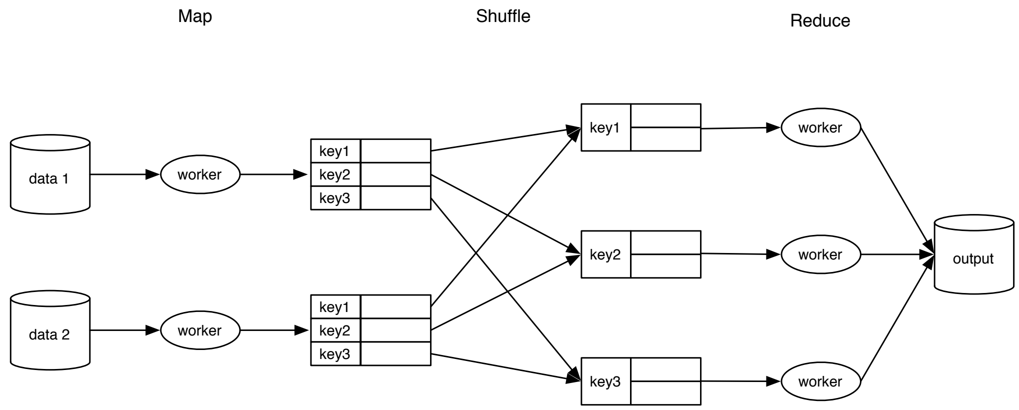

Figure 2: MapReduce model.

a database or some files, and output key-value pairs. The input data is split in parts so they can be processed by workers in parallel. The second step is the Shuffle phase, which collates values with the same key together. At last, the Reduce function is called by workers in parallel on the shuffled data distinguished by keys.

We assume two interfaces, *WorkerI* and *MapReduceI*. The interface *WorkerI* is implemented by a class *Worker* shown in Listing. 1, in which the method *invokeMap* takes a file and emits a list of pairs. Such that each word in the file is associated with a counting number: '1' in this example. For instance, if the content of the file is 'I am fine', the output of *invokeMap* is '(I,1),(am,1),(fine,1)'. The method *invokeReduce* in class *Worker* sums together all counts emitted for a particular word. For instance, *invokeReduce* takes '(am, (1,1))' and outputs 2.

The interface *MapReduceI* is implemented by class *MapReduce*, shown in Listing. 2. We here assume generic data types for sets, lists, and pairs, the latter with *fst* and *snd* to extract the first and second element, respectively.

The input to the method *mapReduce* is a list of files each starts with a filename and contains a list of words, i.e. the content of the file. Each file are handled by a worker in parallel. To achieve concurrency, for each file the object of *MapReduce* calls asynchronously the method *invokeMap* on the assigned worker *w*. This is realized by the statement *fMap := w!invokeMap(filename, content)*. The function *insertElement* collects all the futures into a set *fMapResults*. Next is the Shuffle phase. The function *take* randomly

```
class Worker () implements WorkerI {

  List<Pair<String, Int>>
    invokeMap(String filename, List<String> content) {...}

  Int invokeReduce(String key, List<Int> value) {...}
  ...
}

class WorkerPool() implements WorkerPoolI {
  WorkerI getWorker() {// provides idle workers,
                        // or generates new workers if needed.}
}
```

Listing 1: Sketch of the classes Worker and WorkerPool.

```
class MapReduce(WorkerPoolI wp) implements MapReduceI {

 List<Pair<String, Int>> mapReduce(
  List<Pair<String, List<String>>> files) {

   Set<Fut<List<Pair<String, Int>>>> fMapResults := EmptySet;
   Set<Pair<String, Fut<Int>>> fReduceResults := EmptySet;
   List<Pair<String, Int>> result := Nil;

   // Map phase //
   while (~isEmpty(files)) do
     ...
     WorkerI w := wp.getWorker();
     ...
     Fut<List<Pair<String, Int>>>
        fMap := w!invokeMap(filename, content);
     fMapResults := insertElement(fMapResults, fMap)
   od;

   // Shuffle phase //
   while(~emptySet(fMapResults)) do
     Fut<List<Pair<String, Int>>>
       fMapResult := take(fMapResults);
     ...
     List<Pair<String, Int>> mapResult := fMapResult?;
     ... // collates values with the same key together
   od;

   // Reduce phase //
   while(~emptySet(keys)) do
     ...
     WorkerI w := wp.getWorker();
     Fut<Int> fReduce := w!invokeReduce(key, values);
     fReduceResults := insertElement(
       fReduceResults, Pair(key, fReduce)) od;
   while (~emptySet(fReduceResults)) do
     Pair<String, Fut<Int>> reduceResult := take(fReduceResults);
     ...
     String key := fst(reduceResult);
     Fut<Int> fValue := snd(reduceResult);
     Int value := fValue?;
     result := Cons(Pair(key, value), result) od;
   return result;
  }
}
```

Listing 2: **The MapReduce class.** Here the notation $x := o.m(\overline{e})$ abbreviates $u := o!m(\overline{e}); x := u?$ (for some fresh future u) to de-emphasize trivial usage of futures.

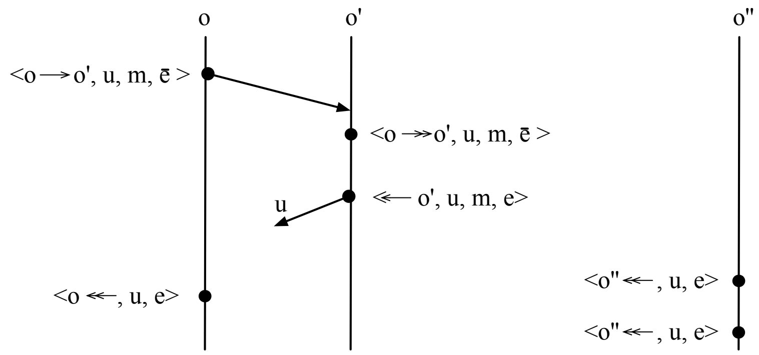

Figure 3: A method call cycle: object o calls a method m on object o' with *future* u. The events on the left-hand side are visible to o, those in the middle are visible to o', and the ones on the right-hand side are visible to o''. There is an arbitrary delay between message receiving and reaction.

extracts an element from a set. The method *mapReduce* waits upon each future, gets the results from each future: *mapResult := fMapResult?*, and collates all the values with the same key, i.e. word, together. For instance, '(I,1),(am,1),(who,1),(I,1),(am,1)' is shuffled to '(I,(1,1)),(am,(1,1)),(who,(1))'. In the Reduce phase, each 'key' and the corresponding values are handled by a worker in parallel. In the same way as the Map phase for achieving concurrency, the first part of the reduce phase calls asynchronously the method *invokeReduce* on the assigned worker *w*. This is realized by the statement *fReduce := w!invokeReduce(key, values)*. The function *insertElement* collects all the futures into a set *fReduceResults*. At the very last, the method *mapReduce* waits upon each future, gets the results from each future: *value := fValue?*, and return the number of occurrences of each word in a large collection of files.

Here the future mechanism is exploited to make an efficient implementation, avoiding blocking calls on the workers: The Map phase is not waiting for the workers to do *invokeMap*, storing future identities only, thereby allowing many workers to start and work concurrently. Likewise in the loop calling *invokeReduce*, only futures identities are recorded. Blocking is delayed to phases where the future value information is actually needed.

3 Observable Behaviour

In this section we describe a communication model for concurrent objects communicating by means of asynchronous message passing and futures. The model is defined in terms of the *observable* communication between objects in the system. We consider how the execution of an object may be described by different *communication events* which reflect the observable interaction between the object and its environment. The observable behavior of a system is described by communication histories over observable events [8, 25].

3.1 Communication Events

Since message passing is asynchronous, we consider separate events for method invocation, reacting upon a method call, resolving a future, and for fetching the value of a future. Each event is observable to only one object, which is the one that *generates* the event. The events generated by a method call cycle is depicted in Fig. 3. The object o calls a method m on object o' with input values $\overline{e}$ and where u denotes the future identity. An invocation message is sent from o to o' when the method is invoked. This is reflected by the *invocation event* $\langle o \rightarrow o', u, m, \overline{e}\rangle$ generated by o. An *invocation reaction event* $\langle o \twoheadrightarrow o', u, m, \overline{e}\rangle$ is generated by o' once the method starts execution. When the method terminates, the object o' generates the *future event* $\langle \leftarrow o', u, m, e\rangle$. This event reflects that u is resolved with return value e. The *fetching event* $\langle o \twoheadleftarrow, u, e\rangle$ is generated by o when o fetches the value of the resolved future. Since future identities may be passed to other objects, e.g, o'', this object may also fetch the future value, reflected by the event $\langle o'' \twoheadleftarrow, u, e\rangle$, generated by o''. The creation of an object o' by an object o is reflected by the event $\langle o \xrightarrow{new} o', C, \overline{e}\rangle$, where o' is the instance of class C and $\overline{e}$ are the actual values for the class parameters. Let type Mid include all method names, and let Data be the supertype of all values occurring as actual parameters, including future identities Fid and object identities Oid.

Definition 1 (Events) *Let $caller, callee, receiver$: Oid, $future$: Fid, $method$: Mid, $class$: Cls, $args$: $\mathsf{List}[\mathsf{Data}]$, and $result$: Data. Communication events Ev include:*

- Invocation events $\langle caller \rightarrow callee, future, method, args\rangle$*, generated by caller.*
- Invocation reaction events $\langle caller \twoheadrightarrow callee, future, method, args\rangle$*, generated by callee.*
- Future events $\langle \leftarrow callee, future, method, result\rangle$*, generated by callee.*
- Fetching events $\langle receiver \twoheadleftarrow, future, result\rangle$*, generated by receiver*
- Creation events $\langle caller \xrightarrow{new} callee, class, args\rangle$*, generated by caller*

Events may be decomposed by functions. For instance, $\_.result$: $\mathsf{Ev} \rightarrow \mathsf{Data}$ is well-defined for future and fetching events, e.g., $\langle \leftarrow o', u, m, e\rangle.result = e$.
For a method invocation with future u, the ordering of events depicted in Fig. 3 is described by the following regular expression (using $\cdot$ for sequential composition of events)

$$\langle o \rightarrow o', u, m, \overline{e}\rangle \cdot \langle o \twoheadrightarrow o', u, m, \overline{e}\rangle \cdot \langle \leftarrow o', u, m, e\rangle [\cdot \langle \_ \twoheadleftarrow, u, e\rangle]^*$$

for some fixed o, o', m, $\overline{e}$, e, and where $\_$ denotes an arbitrary value. This implies that the result value may be read several times, each time with the same value, namely that given in the preceding future event.

3.2 Communication Histories

The execution of a system up to present time may be described by its history of observable events, defined as a sequence. A sequence over some type T is constructed by the empty sequence ε and the right append function $\_ \cdot \_$: $\mathsf{Seq}[T] \times T \rightarrow \mathsf{Seq}[T]$ (where "$\_$" indicates an argument position). The choice of constructors gives rise to generate inductive function definitions, in the style of [12]. Projection, $\_/\_$: $\mathsf{Seq}[T] \times \mathsf{Set}[T] \rightarrow \mathsf{Seq}[T]$ is defined inductively by $\varepsilon/s \triangleq \varepsilon$ and $(a \cdot x)/s \triangleq$ **if** $x \in s$ **then** $(a/s) \cdot x$ **else** a/s **fi**, for a : $\mathsf{Seq}[T]$, $x : T$, and s : $\mathsf{Set}[T]$, restricting a to the elements in s. We use dot notation to extract components from record-like structures, for instance $\langle o \rightarrow o', f, m, \overline{e}\rangle.\mathsf{callee}$ is o', and also lift the dot notation to sequences. For a sequence h of events, $h/ \leftarrow$ is the subsequence of invocation events, and $(h/ \leftarrow).callee$ is the sequence of callee elements from these invocation events.
A *communication history* for a set S of objects is defined as a sequence of events generated by the objects in S. We say that a history is *global* if S includes all objects in the system.

Definition 2 (Communication histories) *The communication history h of a system of objects S is a sequence of type $\mathsf{Seq}[\mathsf{Ev}]$, such that each event in h is generated by an object in S.*

We observe that the *local history* of a single object o is achieved by restricting S to the single object, i.e., the history contains only elements generated by o. For a history h, we let h/o abbreviate the projection of h to the events generated by o. Since each event is generated by only one object, it follows that the local histories of two different objects are disjoint.

Definition 3 (Local histories) *For a global history h and an object o, the projection h/o is the local history of o.*

4 Operational Semantics

Rewriting logic [31] is a logical and semantic framework in which concurrent and distributed systems can be specified in an object-oriented style. Unbounded data structures and user-defined data types are defined in this framework by means of equational specifications. Rewriting logic extends membership equational logic with *rewrite rules*, so that in a *rewrite theory*, the *dynamic* behavior of a system is specified as a set of rules on top of its *static* part, defined by a set of equations. Informally, a *labeled conditional rewrite rule* is a transition $l:\ t \longrightarrow t'$ **if** $cond$, where l is a *label*, t and t' are terms over typed variables and function symbols of given arities, and $cond$ is a condition that must hold for the transition to take place. Rewrite rules are used to specify local transitions in a system, from a state fragment that matches the pattern t, to another state fragment that is an instance of the pattern t'. Rules are selected nondeterministic if there are at least two rule instantiations with left-hand sides matching overlapping fragments of a term. Concurrent rewriting is possible if the fragments are non-overlapping. Furthermore, matching is made modulo the properties of the function symbols that appear in the rewrite rule, like associativity, commutativity, identity (ACI), which introduces further nondeterminism. The Maude tools [9] allow simulation, state exploration, reachability analysis, and LTL model checking of rewriting logic specifications. The state of a concurrent object system is captured by a *configuration*, which is an ACI multiset of *units* such as objects and messages, and other relevant system parts, which in our case includes futures. Concurrency is then supported in the framework by allowing concurrent application of rules when there are non-overlapping matches of left-hand sides. The following context rule, which is implicit in rewriting logic, describes interleaving semantics (letting G, G_1, G_2 denote subconfigurations):

$$\textit{context rule}\quad \frac{G_1 \to G_2}{G\ G_1 \longrightarrow G\ G_2}$$

4.1 Operational Rules

For our purpose, a *configuration* is a multiset of (concurrent) objects, classes, messages, futures, as well as a representation of the global history. We use blank-space as the multiset constructor, allowing ACI pattern matching. Objects have the form $\mathbf{object}(\mathit{Id} : o, A)$ where o is the unique identity of the object and A is a set of semantic attributes, including

$Cl : c$ the class c of the object,
$Pr : s$ the remaining code s of the active process,
$Lvar : l$ the local state l of the active process, including method parameters and the implicit future identity destiny,
$Flds : a$ the state a of the fields, including class parameters,
$Cnt : n$ a counter n used to generate future identities,
$Mtd : m$ the name m of the current method.

Similarly, classes have the form

$$\mathbf{class}(Id : c, Par : z, Flds : a, Init : s, Mtds : q, Cnt : n)$$

where c is the class name, z the class parameters, a the fields, and s the initialization code. The variable q is a multiset of method definitions of the form

$$(m, \overline{p}, l, s)$$

where m is the method name, $\overline{p}$ is the list of parameters, l contains the local variables (including default values), and s is the code. The counter n in the class is used to generate object identities.

Messages have the form of invocation events as described above. And, a future unit is of the form $\mathbf{fut}(Id : u, Val : v)$ where u is the future identity and v is its value. The global history is represented by a unit $\mathbf{hist}(h)$ where h is finite sequence of events (initially empty). Remark that a system configuration contains exactly one history. The history is included to define the interleaving semantics upon which we derive our history-based reasoning formalism.

The initial state of an object o of class C with actual class parameter values $\overline{v}$ is denoted $init_{o:C(\overline{v})}$ and is defined by

$$init_{o:C(\overline{v})} \triangleq \mathbf{object}(Id : o, Cl : C, Pr : init_C, Lvar : \varnothing, Flds : a, Cnt : 0, Mtd : init)$$

where a is the initial state of the object fields given by $[\mathsf{this} \mapsto o, Par_C \mapsto \overline{v}, Flds_C \mapsto \overline{d}]$. Here Par_C, $Flds_C$, and $init_C$, represent the class parameters, the fields, and the initialization code of C, respectively. The class parameters Par_C are initialized by the actual parameters $\overline{v}$, the fields $Flds_C$ are initialized by default values $\overline{d}$ (of the appropriate types), and the initial code is ready to be executed with an empty local state.

A system is given by a set of self-contained classes $\overline{Cl}$, including a class $Main$, without class parameters, used to generate the initial object $init_{main:Main(\varepsilon)}$. The initial configuration of a system is defined by

$$init_{\overline{Cl}} \triangleq \overline{Cl} \;\; init_{main:Main(\varepsilon)} \;\; \mathbf{hist}(\varepsilon)$$

The operational rules are summarized in Fig. 4. The rules for skip, assignment, initialized variable declarations, if- and while-statements are standard. Note that $(a; l)$ represents the total object state, composed by a, the state of the fields/class parameters, and l, the state of the local variables/parameters of the method. Lookup of a variable if left to right, i.e., l is tried before a. Expressions e without side-effects are evaluated by a semantic function depending on the total state, i.e., $eval(e, (a; l))$.

Method invocation is captured by the rule call. The generated future identity $ft(o, n)$ is globally unique (assuming the *next* function is producing locally unique values). The future unit itself is not generated yet; it will be generated by return from the called method.

If there is no active process in an object, denoted $Pr : empty$, a method call is selected for execution by rule method. The invocation message is consumed by this rule, and the future identity of the call is assigned to the implicit parameter destiny. Method

$$
\begin{array}{ll}
\textsf{skip:} & \mathbf{object}(\mathit{Id} : o, \mathit{Pr} : \ (\mathtt{skip}; s)) \longrightarrow \mathbf{object}(\mathit{Id} : o, \mathit{Pr} : s) \\[1ex]
\textsf{assign :} & \mathbf{object}(\mathit{Id} : o, \mathit{Pr} : (v := e; s), \mathit{Lvar} : l, \mathit{Flds} : a) \\
 & \longrightarrow \\
 & \mathtt{if}\ \ v\ \mathit{in}\ \mathit{Lvar}\ \mathtt{then}\ \mathbf{object}(\mathit{Id} : o, \mathit{Pr} : s, \mathit{Lvar} : l[v \mapsto \mathit{eval}(e, (a; l))], \mathit{Flds} : a) \\
 & \qquad\qquad\qquad \mathtt{else}\ \mathbf{object}(\mathit{Id} : o, \mathit{Pr} : s, \mathit{Lvar} : l, \mathit{Flds} : a[v \mapsto \mathit{eval}(e, (a; l))]) \\[1ex]
\textsf{init :} & \mathbf{object}(\mathit{Id} : o, \mathit{Pr} : (\overline{Tv := e}; s), \mathit{Lvar} : l, \mathit{Flds} : a) \\
 & \longrightarrow \\
 & \mathbf{object}(\mathit{Id} : o, \mathit{Pr} : (\overline{Tv}; \overline{v := e}; s), \mathit{Lvar} : l, \mathit{Flds} : a) \\[1ex]
\textsf{if-else :} & \mathbf{object}(\mathit{Id} : o, \mathit{Pr} : (\mathit{if}\ \ e\ \mathit{then}\ s_1\ \mathit{else}\ s_2\ \mathit{fi}; s), \mathit{Lvar} : l, \mathit{Flds} : a) \\
 & \longrightarrow \\
 & \mathtt{if}\ \ \mathit{eval}(e, (a; l))\ \mathtt{then}\ \mathbf{object}(\mathit{Id} : o, \mathit{Pr} : (s_1; s), \mathit{Lvar} : l, \mathit{Flds} : a) \\
 & \qquad\qquad\qquad\quad \mathtt{else}\ \mathbf{object}(\mathit{Id} : o, \mathit{Pr} : (s_2; s), \mathit{Lvar} : l, \mathit{Flds} : a) \\[1ex]
\textsf{while :} & \mathbf{object}(\mathit{Id} : o, \mathit{Pr} : (\mathit{while}\ e\ \mathit{do}\ s_1\ \mathit{od}; s), \mathit{Lvar} : l, \mathit{Flds} : a) \\
 & \longrightarrow \\
 & \mathbf{object}(\mathit{Id} : o, \mathit{Pr} : (\mathit{if}\ \ e\ \mathit{then}\ s_1; \mathit{while}\ e\ \mathit{do}\ s_1\ \mathit{od}\ \mathit{fi}; s), \mathit{Lvar} : l, \mathit{Flds} : a) \\[1ex]
\textsf{new :} & \mathbf{hist}(h)\ \mathbf{class}(\mathit{Id} : C, \mathit{Cnt} : n) \\
 & \mathbf{object}(\mathit{Id} : o, \mathit{Pr} : (v := \mathit{new}\ C(\overline{e}); s), \mathit{Lvar} : l, \mathit{Flds} : a) \\
 & \longrightarrow \\
 & \mathbf{hist}(h \cdot \langle o \overset{new}{\longrightarrow} \mathit{ob}(C, n), C, \mathrm{eval}(\overline{e}, (a; l))\rangle)\ \mathbf{class}(\mathit{Id} : C, \mathit{Cnt} : \mathit{next}(n)) \\
 & \mathbf{object}(\mathit{Id} : o, \mathit{Pr} : (v := \mathit{ob}(C, n); s), \mathit{Lvar} : l, \mathit{Flds} : a) \\
 & \mathit{init}_{\mathit{ob}(C,n):C(\mathrm{eval}(\overline{e},(a;l)))} \\[1ex]
\textsf{call :} & \mathbf{hist}(h)\ \mathbf{object}(\mathit{Id} : o, \mathit{Pr} : (\mathit{fr} := v!m(\overline{e}); s), \mathit{Lvar} : l, \mathit{Flds} : a, \mathit{Cnt} : n) \\
 & \longrightarrow \\
 & \textsc{msg}\ \mathbf{hist}(h \cdot \textsc{msg}) \\
 & \mathbf{object}(\mathit{Id} : o, \mathit{Pr} : (\mathit{fr} := \mathit{ft}(o, n); s), \mathit{Lvar} : l, \mathit{Flds} : a, \mathit{Cnt} : \mathit{next}(n)) \\[1ex]
\textsf{method :} & \langle o' \rightarrow o, u, m, \overline{v}\rangle\ \mathbf{hist}(h)\ \mathbf{class}(\mathit{Id} : c, \mathit{Mtds} : (q\ \ (m, \overline{p}, l, s))) \\
 & \mathbf{object}(\mathit{Id} : o, \mathit{Cl} : c, \mathit{Pr} : \mathit{empty}, \mathit{Flds} : a) \\
 & \longrightarrow \\
 & \mathbf{hist}(h \cdot \langle o' \twoheadrightarrow o, u, m, \overline{v}\rangle)\ \mathbf{class}(\mathit{Id} : c, \mathit{Mtds} : (q\ \ (m, \overline{p}, l, s))) \\
 & \mathbf{object}(\mathit{Id} : o, \mathit{Cl} : c, \mathit{Pr} : s, \mathit{Lvar} : l[\overline{p} \mapsto \overline{v}][\mathsf{destiny} \mapsto u], \mathit{Flds} : a, \mathit{Mtd} : m) \\[1ex]
\textsf{return :} & \mathbf{hist}(h)\ \mathbf{object}(\mathit{Id} : o, \mathit{Pr} : \mathtt{return}\ e, \mathit{Lvar} : l, \mathit{Flds} : a, \mathit{Mtd} : m) \\
 & \longrightarrow \\
 & \mathbf{hist}(h \cdot \langle\leftarrow o, \mathit{eval}(\mathsf{destiny}, l), m, \mathit{eval}(e, (a; l))\rangle) \\
 & \mathbf{fut}(\mathit{Id} : \mathit{eval(\mathsf{destiny},l)}, \mathit{Val} : \mathit{eval(e,(a;l))}) \\
 & \mathbf{object}(\mathit{Id} : o, \mathit{Pr} : \mathit{empty}, \mathit{Flds} : a) \\[1ex]
\textsf{query :} & \mathbf{hist}(h)\ \mathbf{fut}(\mathit{Id} : u, \mathit{Val} : d)\ \mathbf{object}(\mathit{Id} : o, \mathit{Pr} : (v := e?; s), \mathit{Lvar} : l, \mathit{Flds} : a) \\
 & \longrightarrow \\
 & \mathbf{hist}(h \cdot \langle o \twoheadleftarrow, u, d\rangle)\ \mathbf{fut}(\mathit{Id} : u, \mathit{Val} : d) \\
 & \mathbf{object}(\mathit{Id} : o, \mathit{Pr} : (v := d; s), \mathit{Lvar} : l, \mathit{Flds} : a) \\
 & \mathtt{if}\ \mathit{eval}(e, (a; l)) = u
\end{array}
$$

Figure 4: Operational rules, using the standard rewriting logic convention that irrelevant attributes may be omitted in a rule. Variables are denoted by single characters (the uniform naming convention is left implicit), $(a; l)$ represents the total object state, and $a[v \mapsto d]$ is the state a updated by binding the variable v to the data value d. The *eval* function evaluates an expression in a given state, and *in* is used for testing domain membership. In rule call, MSG denotes $\langle o \rightarrow \mathit{eval}(v, (a; l)), \mathit{ft}(o, n), m, \mathit{eval}(\overline{e}, (a; l))\rangle$ where $\mathit{ft}(o, n)$ is the generated future identity. In rule new, $\mathit{ob}(C, n)$ is the generated object identity.

execution is completed by rule return, and a future value is fetched by rule query. A query can only succeed if the appropriate future unit is generated. A future unit appears in the configuration when resolved by rule return, which means that a query statement blocks until the future is resolved. Remark that rule query does not remove the future unit from the configuration, which allows several processes to fetch the value of the same future.

In the rule new, the new object gets a unique identity $\mathit{ob}(C, n)$, given by that of the generating object and a counter, the actual class parameters are evaluated, and the initialization is performed. The given language fragment may be extended with constructs for inter object process control and suspension, e.g., by using the *ABS* approach of [18].

4.2 Semantic Properties

Semantic properties are stated by means of notions of validity. We define *global validity* (denoted $\models$) and *local validity* with respect to a class C (denoted $\models_C$). A global object system initiated by a configuration $init_{\overline{Cl}}$ is said to satisfy a global invariant property $I(h)$, if the global history h of any reachable configuration G satisfies $I(h)$:

$$\overline{Cl} \models I(h) \ \triangleq\ \forall G \,.\, init_{\overline{Cl}} \longrightarrow^{*} G \wedge G.hist = h \Rightarrow I(h)$$

where $\longrightarrow^{*}$ denotes the transitive and reflexive extension of the transition relation, lifted to configurations, and where $G.hist$ extracts the history of the configuration G.
Similarly, an object system initiated by a configuration $init_{\overline{Cl}}$ is said to satisfy a C-local invariant property $I(h)$ if every object o of class C in any reachable configuration G satisfies $I(h/o)$, i.e., the projection from global history to the object o:

$$\overline{Cl} \models_C I(h) \triangleq \forall G, o \,.\, init_{\overline{Cl}} \longrightarrow^{*} G \wedge G.hist = h \wedge o \in G.obj \wedge G[o].class = C \Rightarrow I(h/o)$$

where $G.obj$ extracts the object identities from the objects in the configuration G.
We next provide notions of global and local wellformedness for global histories. We first introduce some notation and functions used in defining wellformed histories. For sequences a and b, let a **ew** x denote that x is the last element of a, $agree(a)$ denote that all elements (if any) are equal, and $a \leqslant b$ denote that a is a prefix of b. Let $[x_1, x_2, \ldots, x_i]$ denote the sequence of $x_1, x_2, \ldots, x_i$ for $i > 0$ (allowing repeated parts [...]*). Functions for event decomposition are lifted to sequences in the standard way, ignoring events for which the decomposition is not defined, e.g., $\_.result : \mathsf{Seq[Ev]} \rightarrow \mathsf{Seq[Data]}$.
Functions may extract information from the history. In particular, we define oid : $\mathsf{Seq[Ev]} \rightarrow \mathsf{Set[Obj]}$ extracting all object identities occurring in a history, as follows:

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

where γ : Ev, and $oid(\overline{e})$ returns the set of object identifiers occurring in the expression list $\overline{e}$. The function $\mathit{fid} : \mathsf{Seq[Ev]} \rightarrow \mathsf{Set[Fid]}$ extracts future identities from a history:

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

where γ : Ev, and $\mathit{fid}(\overline{e})$ returns the set of future identities occurring in the expression list $\overline{e}$. For a global history h, the function $\mathit{fid}(h)$ returns all future identities on h, and for a local history h/o, the function $\mathit{fid}(h/o)$ returns the futures generated by o or received as parameters. At last, h/u abbreviates the projection of history h to the set $\{\gamma \mid \gamma.\mathit{future} = u\}$, i.e., all events with future u.

Definition 4 (Wellformed histories) *Let h : $\mathsf{Seq[Ev]}$ be a history of a global object* *system S. The wellformedness predicate $wf : \mathsf{Seq}[\mathsf{Ev}] \to \mathsf{Bool}$ is defined by:*

$$\begin{array}{lcl}
wf(\varepsilon) & \triangleq & true \\
wf(h \cdot \langle o \to o', u, m, \overline{e} \rangle) & \triangleq & wf(h) \wedge o \neq \mathsf{null} \wedge u \notin fid(h) \cup fid(\overline{e}) \\
wf(h \cdot \langle o' \twoheadrightarrow o, u, m, \overline{e} \rangle) & \triangleq & wf(h) \wedge o \neq \mathsf{null} \wedge h/u = [\langle o' \to o, u, m, \overline{e} \rangle] \\
wf(h \cdot \langle \leftarrow o, u, m, e \rangle) & \triangleq & wf(h) \wedge h/u \ \boldsymbol{\mathsf{ew}} \ \langle \_ \twoheadrightarrow o, u, m, \_ \rangle \\
wf(h \cdot \langle o \twoheadleftarrow, u, e \rangle) & \triangleq & wf(h) \wedge u \in fid(h/o) \wedge agree(((h/u).result) \cdot e) \\
wf(h \cdot \langle o \xrightarrow{new} o', C, \overline{e} \rangle) & \triangleq & wf(h) \wedge o \neq \mathsf{null} \wedge o' \neq \mathsf{null} \wedge o' \notin oid(h) \cup oid(\overline{e})
\end{array}$$

It follows directly that a wellformed global history satisfies the communication order pictured in Fig. 3, i.e.,

$$\begin{array}{l}
\forall u \,.\, \exists o, o', m, \overline{e}, e \,. \\
\qquad h/u \leqslant [\langle o' \to o, u, m, \overline{e} \rangle, \langle o' \twoheadrightarrow o, u, m, \overline{e} \rangle, \langle \leftarrow o, u, m, e \rangle, [\langle \_ \twoheadleftarrow, u, e \rangle]^*]
\end{array}$$

Also, it ensures the uniqueness of object identifiers and future identities. We can prove that the operational semantics guarantees wellformedness:

Lemma 1 *The global history h of a global object system S obtained by the given operational semantics, is wellformed, i.e., $\models wf(h)$ where $wf(h)$ is strengthened by the two conditions $fid(h) \subseteq (h/\to).future$ and $oid(h) - \mathsf{null} \subseteq (h/ \xrightarrow{new}).callee$.*

The two conditions ensure that a history may not refer to object and future identities before generated by creation and invocation events, respectively. This lemma follows by induction over the number of rule applications.
Wellformedness of a local history for an object o, denoted $wf_o(h)$, is defined as in Def. 4, except that the last conjunct of the case $\langle o' \twoheadrightarrow o, u, m, \overline{e} \rangle$ only holds for self calls, i.e., where o and o' are equal. For local wellformedness, the conjunct is therefore weakened to $o = o' \Rightarrow h/u = [\langle o' \to o, u, m, \overline{e} \rangle]$. If h is a wellformed global history, it follows immediately that each projection h/o is locally wellformed, i.e.,

$$wf(h) \Rightarrow wf_o(h/o)$$

5 Program Verification

The communication history abstractly captures the system state at any point in time [11, 12]. Partial correctness properties of a system may thereby 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 [5]. In this section we present a framework for compositional reasoning about object systems, establishing an invariant over the global history from invariants over the local histories of each object. Since the local object histories are disjoint with our four event semantics, it is possible to reason locally about each object. In particular, the history updates of the operational semantics affect the local history of the active object only, and can be treated simply as an assignment to the local history. The local history is not effected by the environment, and interference-free reasoning is then possible. Correspondingly, the reasoning framework consists of two parts: A proof system for local (class-based) reasoning, and a rule for composition of object specifications.

$$\textsf{invoc}\ \frac{\vdash \forall u\,.\,\{\mathcal{H} := \mathcal{H}\cdot\langle \textsf{this} \to v, u, m, \overline{e}\rangle ||\ \mathit{fr} := u\}\ [s]\phi}{\vdash [\mathit{fr} := v!m(\overline{e});\ \ s]\phi}$$

$$\textsf{fetch}\ \frac{\vdash \forall v'\,.\,\{\mathcal{H} := \mathcal{H}\cdot\langle \textsf{this} \twoheadleftarrow, e, v'\rangle ||\ v := v'\}\ ([s]\phi \wedge \exists \overline{w}\,.\,I(\overline{w},\mathcal{H}))}{\vdash [v := e?;\ \ s]\phi}$$

$$\textsf{new}\ \frac{\vdash \forall v'\,.\,\{\mathcal{H} := \mathcal{H}\cdot\langle \textsf{this} \xrightarrow{new} v', C, \overline{e}\rangle ||\ v := v'\}\ [s]\phi}{\vdash [v\ \ := \mathtt{\textbf{new}}\ C(\overline{e});\ \ s]\phi}$$

Figure 5: Dynamic logic rules for method invocation, future query and object creation. $I(\overline{w},\mathcal{H})$ is the class invariant.

5.1 Local Reasoning

Pre- and postconditions to method definitions are in our setting used to establish a *class invariant*. The class invariant must hold after initialization of all class instances and must be maintained by all methods, serving as a contract for the different methods: A method implements its part of the contract by ensuring that the invariant holds upon termination, assuming that it holds when the method starts execution. A class invariant establishes a *relationship between the internal state and the observable behavior of class instances*. The internal state reflects the values of the fields, and the observable behavior is expressed as potential communication histories. A *user-provided invariant* $I(\overline{w},\mathcal{H})$ for a class C is a predicate over the fields $\overline{w}$, the read-only parameters $\overline{cp}$ and this, in addition to the local history $\mathcal{H}$ which is a sequence of events generated by this. The proof system for class-based verification is formulated within *dynamic logic* as used by the KeY framework [7], facilitating class invariant verification by considering each method independently. The dynamic logic formulation suggests that the proof system is suitable for an implementation in the KeY framework.

Dynamic logic provides a structured way to describe program behavior by an integration of programs and assertions within a single language. The formula $\psi \Rightarrow [s]\phi$ express partial correctness properties: if statement s is executed in a state where ψ holds and the execution terminates, then ϕ holds in the final state. The formula is verified by a symbolic execution of s, where state modifications are handled by the *update* mechanism [7]. A dynamic formula $[s_1; s]\phi$ is equal to $[s_1][s]\phi$, where $[s]\phi$ is the precondition of s. A dynamic formula $[v := e; s]\phi$, i.e., where an assignment is the first statement, reduces to $\{v := e\}[s]\phi$, where $\{v := e\}$ is an update. We assume that expressions e can be evaluated within the assertion language. Updates can only be *applied* on formulas without programs, which means that updates on a formula $[s]\phi$ are accumulated and *delayed* until the symbolic execution of s is complete. Update application $\{v := t\}e$, on an expression e, evaluates to the substitution e^v_t, replacing all free occurrences of v in e by t. The parallel update $\{v_1 := e_1||...||v_n := e_n\}$, for disjoint variables $v_1, ..., v_n$, represents an accumulated update, and the application of a parallel update leads to a simultaneous substitution. For an update U, we have $U(\phi_1 \wedge \phi_2) = U\phi_1 \wedge U\phi_2$. A *sequent* $\psi_1, ..., \psi_n \vdash \phi_1, ..., \phi_m$ contains assumptions $\psi_1, ..., \psi_n$, and formulas $\phi_1, ..., \phi_m$ to be proved. The sequent is *valid* if at least one formula ϕ_i follows from the assumptions, and it can be interpreted as $\psi_1 \wedge ... \wedge \psi_n \Rightarrow \phi_1 \vee ... \vee \phi_m$.

In order to verify a class invariant $I(\overline{w},\mathcal{H})$, we must prove that the invariant is established by the initialization code and maintained by all method definitions in C, assuming wellformedness of the local history. For a method definition $m(\overline{x})\{s;\ \mathtt{\textbf{return}}\ e\}$ in C,

skip	$[\texttt{\textbf{skip}};\ s]\phi$	$=$	$[s]\phi$
assign	$[v := e;\ s]\phi$	$=$	$\{v := e\}\ [s]\phi$
declInit	$[T\ v = e;\ s]\phi$	$=$	$[v' := e;\ s^{v}_{v'}]\phi$
declNoInit	$[T\ v;\ s]\phi$	$=$	$[v' := \mathsf{default}_T;\ s^{v}_{v'}]\phi$
ifElse	$[\texttt{\textbf{if}}\ b\ \texttt{\textbf{then}}\ s_1\ \texttt{\textbf{else}}\ s_2\ \texttt{\textbf{fi}};\ s]\phi$	$=$	$\texttt{\textbf{if}}\ b\ \texttt{\textbf{then}}\ [s_1; s]\phi\ \texttt{\textbf{else}}\ [s_2; s]\phi$
while	$[\texttt{\textbf{while}}\ b\ \texttt{\textbf{do}}\ s'\ \texttt{\textbf{od}};\ s]\phi$	$=$	$\texttt{\textbf{if}}\ b\ \texttt{\textbf{then}}\ (\exists w\,.\,I(\overline{w}, \mathcal{H})) \wedge$ $([s';\ \texttt{\textbf{while}}\ b\ \texttt{\textbf{do}}\ s'\ \texttt{\textbf{od}};\ s]\phi)\ \texttt{\textbf{else}}\ [s]\phi$

Figure 6: Semantical definitions for standard *ABS* statements. Here ϕ is the postcondition, s is the remaining program yet to be executed, primes denote fresh variables, $s^{v}_{v'}$ is s with all (free) occurrences of v replaced by v', and $\mathsf{default}_T$ is the default value defined for type T.

this amounts to a proof of the sequent:

$$\begin{aligned} \vdash (wf_{\mathit{this}}(\mathcal{H}) \wedge I(\overline{w}, \mathcal{H}) \Rightarrow [\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{caller} \twoheadrightarrow \mathsf{this}, \mathsf{destiny}, m, \overline{x}\rangle;\\ s;\ \mathcal{H} := \mathcal{H} \cdot \langle \leftarrow \mathsf{this}, \mathsf{destiny}, m, e\rangle](wf_{\mathit{this}}(\mathcal{H}) \Rightarrow I(\overline{w}, \mathcal{H})) \end{aligned}$$

Here, the method body is extended with a statement for extending the history with the invocation reaction event, and the **return** statement is treated as a history extension. Dynamic logic rules for method invocation, future query, and object creation, can be found in Fig. 5. When invoking a method, the update in the premise of rule invoc captures the history extension and the generation of a fresh future identity u. Similarly, the update in rule fetch captures the history extension and the assignment of a fresh value to v, where the wellformedness assumptions ensure that all values received from the same future are equal. The update in the premise of rule new captures the history extension and the generation of a fresh object identity v', and the universal quantifier reflects non-determinism. The prime is needed here since v may occur in $\overline{e}$. The query rule insists that the class invariant holds for local history, ignoring the field values of the current state, as discussed in the soundness proof. Assignments are analyzed as explained above, and rules for **skip** and conditionals are standard. We refer to Din et al. for further details [19].
The rules for the rest of the *ABS* statements can be defined as substitution rules introduced in Fig. 6. For instance, $[\texttt{\textbf{skip}};\ s]\phi$ can be rewritten to $[s]\phi$. In rule declInit and declNoInit v' is needed since the postcondition may talk about a field with the same name v. If-statements without an else-branch are as usual.

5.2 Soundness

The reasoning system for statements in dynamic logic is sound if any provable property is valid, i.e.,

$$\vdash \psi \Rightarrow [s]\phi \Rightarrow\ \models \psi \Rightarrow [s]\phi$$

Validity of a dynamic logic formula, denoted $\models \psi \Rightarrow [s]\phi$, is defined by means of the operational semantics. We base the semantics on the operational semantics above, as given by unlabeled transitions of the form $G_1 \rightarrow G_2$.
Note that each rule is local to one object, and we write $G_1 \xrightarrow{o:s} G_2$ to indicate an execution involving only object o such that exactly the statement (list) s has been executed by o. And we write $G_1 \overset{o:s}{\rightrightarrows} G_2$ if o executes s while other objects may execute.

Definition 5 (Explicit execution step)

$$G_1 \overset{o:s}{\rightrightarrows} G_2 \triangleq G_1 \longrightarrow^* G_2 \wedge G_1[o].Pr = s; G_2[o].Pr$$

$$G_1 \xrightarrow{o:s} G_2 \triangleq G_1 \overset{o:s}{\rightrightarrows} G_2 \wedge \forall o' \,.\, o' \neq o \Rightarrow G_1[o'] = G_2[o']$$

expressing one or more transitions from the configuration G_1 to G_2 such that o executes s, with or without, respectively, interleaved execution by other objects. The notation $G[o]$ denotes the object o of the configuration G.

We consider pre- and postconditions over local states and the local history. Such an assertion can be evaluated in a state defining values for attributes (of the appropriate class), parameters and local variables (of the method) and the local history. We let $[\![\psi \xrightarrow{o:s} \phi]\!]_{G,o}$ express that if the condition ψ holds for object o before execution of s by the object in configuration G, then ϕ holds for o after the execution. As above, we let $\xrightarrow{o:s}$ express local execution by o, and $\overset{o:s}{\rightrightarrows}$ execution by o interleaved with other objects:

Definition 6 (Validity of pre/post-conditions over execution steps)

$$[\![\psi \xrightarrow{o:s} \phi]\!]_{G,o} \triangleq \forall G', \overline{z} \,.\, \mathit{wf}(G'.\mathit{hist}) \wedge G \xrightarrow{o:s} G' \wedge \mathit{loc}(G,o)[\psi] \Rightarrow \mathit{loc}(G',o)[\phi]$$

$$[\![\psi \overset{o:s}{\rightrightarrows} \phi]\!]_{G,o} \triangleq \forall G', \overline{z} \,.\, \mathit{wf}(G'.\mathit{hist}) \wedge G \overset{o:s}{\rightrightarrows} G' \wedge \mathit{loc}(G,o)[\psi] \Rightarrow \mathit{loc}(G',o)[\phi]$$

where $\overline{z}$ is the list of auxiliary variables in ψ and/or ϕ, not bound by G nor G'. Here $\mathit{loc}(G,o)$ denotes the local state of object o, as derived from the global state G. The function $\mathit{loc} : \mathsf{Config} \times \mathsf{Oid} \rightarrow \mathsf{State}$ is defined by

$$\mathit{loc}(G,o) \triangleq (G[o].\mathit{Flds}; G[o].\mathit{Lvar}) + [\mathcal{H} \mapsto (G.\mathit{hist})/o]$$

where the resulting $\mathcal{H}$ ranges over local histories (i.e., in the alphabet of o), and where this is bound to o in G as explained earlier. Thus the extraction is made by taking the state of object o and adding the history localized to o. We let $\mathit{loc}(G,o)[\psi]$ denote the value of ψ in state $\mathit{loc}(G,o)$.

It follows that local reasoning suffices for local pre/post-conditions, in the sense that when reasoning about one object in our system, one may ignore the activity of other objects.

Lemma 2 *$[\![\psi \xrightarrow{o:s} \phi]\!]_{G,o}$ is the same as $[\![\psi \overset{o:s}{\rightrightarrows} \phi]\!]_{G,o}$*

The lemma follows by induction on the length of executions, and the fact that $\mathit{loc}(G,o)$ for any G is not affected by execution steps by other objects than o, since remote access to fields is not allowed in our language and since h/o only contains events generated by o.

In our setting, we may understand a sequent by means of the $\xrightarrow{o:s}$ relation, letting a dynamic logic subformula depend on a given pre-configuration G and object o.

Definition 7 (Validity of dynamic logic sequents)

$$\begin{array}{rcl}
\models \psi_1, ..., \psi_n \vdash \phi_1, ..., \phi_m & \triangleq & \forall G, o \,.\, \mathit{wf}(G.\mathit{hist}) \Rightarrow [\![\psi_1 \wedge ... \wedge \psi_n \Rightarrow \phi_1 \vee ... \vee \phi_m]\!]_{G,o} \\
[\![[s]\phi]\!]_{G,o} & \triangleq & [\![\mathit{true} \xrightarrow{o:s} \phi]\!]_{G,o} \\
[\![e]\!]_{G,o} & \triangleq & \mathit{loc}(G,o)[e] \\
[\![U[s]\phi]\!]_{G,o} & \triangleq & [\![[U'; s]\phi]\!]_{G,o} \\
[\![\psi \wedge \phi]\!]_{G,o} & \triangleq & [\![\psi]\!]_{G,o} \wedge [\![\phi]\!]_{G,o}, \\
[\![\psi \vee \phi]\!]_{G,o} & \triangleq & [\![\psi]\!]_{G,o} \vee [\![\phi]\!]_{G,o} \\
[\![\psi \Rightarrow \phi]\!]_{G,o} & \triangleq & [\![\psi]\!]_{G,o} \Rightarrow [\![\phi]\!]_{G,o}
\end{array}$$

where e is a formula without the dynamic logic operators, and the equations for updates are as given earlier. U' is the sequentialized version of the parallel update U introducing temporary variables if needed. For instance the sequentialized version of $x := y || y := x$ is $x_0 := x; x := y; y := x_0$. It follows from the definition that

$$[\![\psi \Rightarrow [s]\phi]\!]_{G,o} = [\![\psi \xrightarrow{o:s} \phi]\!]_{G,o}$$

Here o is the executing object and the object on which ψ and ϕ are interpreted. Thus the formula is valid if for any object o executing s, the postcondition holds in the poststate, provided the precondition holds in the prestate. In dynamic logic the prestate given by G and o is fixed for the whole sequent, and therefore the meaning of the individual operators is given in the context of G and o.

We verify an invariant $I(\overline{w}, \mathcal{H})$ for a class C by showing that $I(\overline{w}, \mathcal{H})$ is established by the initialization of C, i.e. $init_C$, and is maintained by all methods in C, assuming local wellfomedness. The rule is:

$$\textsf{class}\ \frac{\begin{array}{l}\vdash \exists \overline{w} \,.\, I(\overline{w}, \mathcal{H} \cdot \gamma) \Rightarrow \exists \overline{w} \,.\, I(\overline{w}, \mathcal{H}) \\ \vdash\ \mathcal{H} = \varepsilon \Rightarrow [init_C](wf_{\textsf{this}}(\mathcal{H}) \Rightarrow I(\overline{w}, \mathcal{H})) \\ \vdash (wf_{\textsf{this}}(\mathcal{H}) \wedge I(\overline{w}, \mathcal{H})) \Rightarrow [body_{C,m}](wf_{\textsf{this}}(\mathcal{H}) \Rightarrow I(\overline{w}, \mathcal{H})), \text{for all methods } m \text{ in } C\end{array}}{\vdash_C \exists \overline{w} \,.\, I(\overline{w}, \mathcal{H})}$$

where $body_{C,m}$ denotes the body s of method m of C augmented with effects on the local history reflecting the start and end of the method, namely

$$\mathcal{H} := \mathcal{H} \cdot \langle \textsf{caller} \twoheadrightarrow \textsf{this}, \textsf{destiny}, m, \overline{x} \rangle; s;\ \mathcal{H} := \mathcal{H} \cdot \langle \leftarrow \textsf{this}, \textsf{destiny}, m, e \rangle$$

Lemma 3 *Reasoning about statements is sound:*

$$\vdash \psi \Rightarrow [s]\phi \Rightarrow \models \psi \Rightarrow [s]\phi$$

Theorem 1 *The proof system for reasoning about classes is sound:*

$$\vdash_C I(\mathcal{H}) \Rightarrow \models_C I(\mathcal{H})$$

Proof of lemma 3. We focus on the rules for statements involving futures and object generation, and consider therefore the rules invoc, fetch and new, as given in figure 5. The axioms given in figure 6 represent standard statements not involving futures, and we omit the soundness proof of these.

Asynchronous method call statement

We prove that the invoc rule preserves validity. The validity of the conclusion is $\models [fr := v!m(\overline{e});\ s]\phi$. Consider now a given G and o, and let ϕ' denote $[s]\phi$. According to Definition. 7, the validity can be written as

$$wf(G.hist) \Rightarrow [\![true \xrightarrow{o:fr:=v!m(\overline{e})} \phi']\!]_{G,o}$$

which by Definition. 6 is

$$\forall G', \overline{z} \,.\, wf(G.hist) \wedge wf(G'.hist) \wedge G \xrightarrow{o:fr:=v!m(\overline{e})} G' \Rightarrow loc(G', o)[\phi']$$

By the operational semantics of call and assign, we have that G' is G with MSG and $G'.hist = G.hist \cdot \text{MSG}$, where MSG denotes $\langle o \rightarrow loc(G, o)[v], ft(o, n), m, loc(G, o)[\overline{e}] \rangle$,

and such that the object state $G'[o].l$ is $(G[o].l)[\mathit{fr} \mapsto \mathit{ft}(o,n)]$ if $\mathit{fr} \in G[o].l$, and otherwise $G'[o].a$ is $(G[o].a)[\mathit{fr} \mapsto \mathit{ft}(o,n)]$. Here n is the counter value of $G[o]$ (same as in $G'[o]$). Other parts of the object state are unchanged.
Thus $loc(G',o)[\phi']$ can be reduced to $loc(G,o)[\phi'\,{}^{\mathit{fr},\mathcal{H}}_{\mathit{ft}(o,n),\mathcal{H}\cdot\langle\mathsf{this}\to v,\mathit{ft}(o,n),m,\overline{e}\rangle}]$
since $loc(G,o)[\langle\mathsf{this} \to v, \mathit{ft}(o,n), m, \overline{e}\rangle] = \textsc{msg}$, and it suffices to prove

$$\forall\overline{z}\,.\,\mathit{wf}(G.hist) \Rightarrow loc(G,o)[\phi'\,{}^{\mathit{fr},\mathcal{H}}_{\mathit{ft}(o,n),\mathcal{H}\cdot\langle\mathsf{this}\to v,\mathit{ft}(o,n),m,\overline{e}\rangle}]$$

The validity of the premise is

$$\models \forall u\,.\,\{\mathcal{H} := \mathcal{H}\cdot\langle\mathsf{this} \to v, u, m, \overline{e}\rangle || \ \mathit{fr} := u\}\ \phi'$$

which by Definition 7 is

$$\forall\overline{z}, u\,.\,\mathit{wf}(G.hist) \Rightarrow loc(G,o)[\phi'\,{}^{\mathit{fr},\mathcal{H}}_{u,\mathcal{H}\cdot\langle\mathsf{this}\to v,u,m,\overline{e}\rangle}]$$

Clearly this is sufficient to ensure validity of the conclusion, since the universal quantifier on u covers the value given by $\mathit{ft}(o,n)$.

Query statement
We prove that the fetch rule preserves validity. The validity of the conclusion is $\models [v := e?;\ s]\phi$. Consider now a given G and o, and let ϕ' denote $[s]\phi$. According to Definition. 7, the validity can be written as

$$\mathit{wf}(G.hist) \Rightarrow [\![\mathit{true} \xrightarrow{o:v:=e?} \phi']\!]_{G,o}$$

which by Definition. 6 is

$$\forall G', \overline{z}\,.\,\mathit{wf}(G.hist) \wedge \mathit{wf}(G'.hist) \wedge G \xrightarrow{o:v:=e?} G' \Rightarrow loc(G',o)[\phi']$$

By the operational semantics of query and assign, we have that G' is G with $\textsc{msg}$ and $G'.hist = G.hist\cdot\textsc{msg}$, where $\textsc{msg}$ denotes $\langle o \twoheadleftarrow, loc(G,o)[e], d\rangle$ and such that the object state $G'[o].l$ is $(G[o].l)[v \mapsto d]$ if $v \in G[o].l$, and otherwise $G'[o].a$ is $(G[o].a)[v \mapsto d]$. Other parts of the object state are unchanged.
Thus $loc(G',o)[\phi']$ can be reduced to $loc(G,o)[\phi'\,{}^{v,\mathcal{H}}_{d,\mathcal{H}\cdot\langle\mathsf{this}\twoheadleftarrow,e,d\rangle}]$
since $loc(G,o)[\langle\mathsf{this} \twoheadleftarrow, e, d\rangle] = \textsc{msg}$, and it suffices to prove

$$\forall\overline{z}\,.\,\mathit{wf}(G.hist) \Rightarrow loc(G,o)[\phi'\,{}^{v,\mathcal{H}}_{d,\mathcal{H}\cdot\langle\mathsf{this}\twoheadleftarrow,e,d\rangle}]$$

The validity of the premise is

$$\models \forall v'\,.\,\{\mathcal{H} := \mathcal{H}\cdot\langle\mathsf{this} \twoheadleftarrow, e, v'\rangle || \ v := v'\}\ (\phi' \wedge \exists\overline{w}\,.\,I(\overline{w},\mathcal{H}))$$

which by Definition 7 is

$$\forall\overline{z}, v'\,.\,\mathit{wf}(G.hist) \Rightarrow loc(G,o)[(\phi' \wedge \exists\overline{w}\,.\,I(\overline{w},\mathcal{H}))\,{}^{v,\mathcal{H}}_{v',\mathcal{H}\cdot\langle\mathsf{this}\twoheadleftarrow,e,v'\rangle}]$$

Clearly this is sufficient to ensure validity of the conclusion, since the universal quantifier on v' covers the value given by d. Note that the invariant is not required here. But it will be needed later to prove soundness of the class rule.

Object creation statement
We prove that the new rule preserves validity. The validity of the conclusion is $\models [v := \mathit{new}\ C(\overline{e});\ s]\phi$. Consider now a given G and o, and let ϕ' denote $[s]\phi$. According to Definition. 7, the validity can be written as

$$\mathit{wf}(G.\mathit{hist}) \Rightarrow [\![\mathit{true} \xrightarrow{o:v:=\mathit{new}\ C(\overline{e})} \phi']\!]_{G,o}$$

which by Definition. 6 is

$$\forall G', \overline{z}\,.\, \mathit{wf}(G.\mathit{hist}) \wedge \mathit{wf}(G'.\mathit{hist}) \wedge G \xrightarrow{o:v:=\mathit{new}\ C(\overline{e})} G' \Rightarrow \mathit{loc}(G', o)[\phi']$$

By the operational semantics of new and assign, we have that G' is G with $\textsc{msg}$ and $G'.\mathit{hist} = G.\mathit{hist} \cdot \textsc{msg}$, where $\textsc{msg}$ denotes $\langle o \xrightarrow{\mathit{new}} \mathit{ob}(C, n), C, \mathit{loc}(G, o)[\overline{e}]\rangle$ and such that the object state $G'[o].l$ is $(G[o].l)[v \mapsto \mathit{ob}(C, n)]$ if $v \in G[o].l$, and otherwise $G'[o].a$ is $(G[o].a)[v \mapsto \mathit{ob}(C, n)]$. Here n is the counter value of $G[C]$ (same as in $G'[C]$). Other parts of the object state are unchanged.
Thus $\mathit{loc}(G', o)[\phi']$ can be reduced to $\mathit{loc}(G, o)[\phi'\ {}^{v,\mathcal{H}}_{\mathit{ob}(C,n),\mathcal{H}\cdot\langle \mathsf{this} \xrightarrow{\mathit{new}} \mathit{ob}(C,n),C,\overline{e}\rangle}]$ since $\mathit{loc}(G, o)[\langle \mathsf{this} \xrightarrow{\mathit{new}} \mathit{ob}(C, n), C, \overline{e}\rangle] = \textsc{msg}$, and it suffices to prove

$$\forall \overline{z}\,.\, \mathit{wf}(G.\mathit{hist}) \Rightarrow \mathit{loc}(G, o)[\phi'\ {}^{v,\mathcal{H}}_{\mathit{ob}(C,n),\mathcal{H}\cdot\langle \mathsf{this} \xrightarrow{\mathit{new}} \mathit{ob}(C,n),C,\overline{e}\rangle}]$$

The validity of the premise is

$$\models \forall v'\,.\, \{\mathcal{H} := \mathcal{H} \cdot \langle \mathsf{this} \xrightarrow{\mathit{new}} v', C, \overline{e}\rangle ||\ v := v'\}\ \phi'$$

which by Definition 7 is

$$\forall \overline{z}, v'\,.\, \mathit{wf}(G.\mathit{hist}) \Rightarrow \mathit{loc}(G, o)[\phi'\ {}^{v,\mathcal{H}}_{v',\mathcal{H}\cdot\langle \mathsf{this} \xrightarrow{\mathit{new}} v',C,\overline{e}\rangle}]$$

Clearly this is sufficient to ensure validity of the conclusion, since the universal quantifier on v' covers the value given by $\mathit{ob}(C, n)$.

Proof of Theorem 1. The theorem follows by lemma 3 above and by proving that if one can prove $\vdash_C I'(\mathcal{H})$ by the class rule, then $\models_C I'(\mathcal{H})$, letting $I'(\mathcal{H})$ denote $\exists \overline{w}\,.\, I(\overline{w}, \mathcal{H})$.
Consider the rule class. We may assume that the premises of the rule are valid. By definition, the validity of $I'(\mathcal{H})$ is

$$\forall G, o\,.\, \mathit{init}_{\overline{Cl}} \longrightarrow^* G \wedge G.\mathit{hist} = \mathcal{H} \wedge o \in G.\mathit{obj} \wedge G[o].\mathit{class} = C \Rightarrow I'(\mathcal{H}/o)$$

We first prove that this holds for all C objects o in states G such that $G[o].\mathrm{Pr} = \mathit{empty}$. With the given operational semantics, Pr is *empty* for an object o when o has finished a method, or init_C, and it can only start a new method when Pr is *empty*. By lemma 1 we only need to consider states with a wellformed history. We need to show that the invariant $I(\overline{w}, \mathcal{H})$ holds after the initialization and is maintained by every methods of class C, considering any interleaved execution according to the operational semantics. The validity of the second premise gives

$$\forall G, o\,.\, \mathit{wf}(G.\mathit{hist}) \Rightarrow [\![\mathcal{H} = \varepsilon \xrightarrow{o:\mathit{init}_C} (\mathit{wf}_{\mathsf{this}}(\mathcal{H}) \Rightarrow I(\overline{w}, \mathcal{H}))]\!]_{G,o}$$

which by lemma 2 is the same as

$$\forall G, o\,.\, \mathit{wf}(G.\mathit{hist}) \Rightarrow [\![\mathcal{H} = \varepsilon \overset{o:\mathit{init}_C}{\rightrightarrows} (\mathit{wf}_{\mathsf{this}}(\mathcal{H}) \Rightarrow I(\overline{w}, \mathcal{H}))]\!]_{G,o}$$

which by definition is

$$G \overset{o:init_C}{\rightrightarrows} G' \wedge loc(G,o)[\mathcal{H} = \varepsilon] \Rightarrow loc(G',o)[\mathit{wf}_{\mathsf{this}}(\mathcal{H}) \Rightarrow I(\overline{w},\mathcal{H})]$$

for all states G' with wellformed histories, and $\overline{z}$. This states that the class invariant $I(\overline{w},\mathcal{H})$ holds after the initialization of class C, conditioned by local wellformedness. The condition on local wellformedness follows from the global wellformedness $\mathit{wf}(G'.hist)$. The condition $loc(G,o)[\mathcal{H} = \varepsilon]$ follows by induction on the length of an execution showing that no object can generate events before its initial code has started.
Similarly, the validity of the third premise gives that

$$G \overset{o:body_{C,m}}{\rightrightarrows} G' \wedge loc(G,o)[\mathit{wf}_{\mathsf{this}}(\mathcal{H}) \wedge I(\overline{w},\mathcal{H})] \Rightarrow loc(G',o)[\mathit{wf}_{\mathsf{this}}(\mathcal{H}) \Rightarrow I(\overline{w},\mathcal{H})]$$

for all states G, G' with wellformed histories, and all o, $\overline{z}$. This states that the class invariant is maintained by a method m of C under the assumption of local wellformedness. As before local wellformedess follows from global wellformedess. Thus $I(\overline{w},\mathcal{H})$, and therefore also $I'(\mathcal{H})$, hold for all C objects o in reachable states G with empty $G[o].\mathrm{Pr}$.
It remains to show that the invariant also holds in states G where $G[o].\mathrm{Pr}$ is nonempty. By the first premise, we have that I' is prefix-closed with respect to the history. Thus all states in between those where $G[o].\mathrm{Pr}$ is empty will also satisfy I'. In order to ensure I' in case of nonterminating methods (or *init*), we must consider loops and other sources of non-termination. For loops it suffices to let I' be required at the beginning of each loop iteration, which we do require in the while axiom. The other source of nonterminating methods is the query statement; however, here the proof rule fetch insists that we verify I'. Thus any proof of that method (or *init*) must establish I' at his point. By lemma 3 we have that I' is valid. We may conclude that reasoning about classes is sound.
We remark that it would be sufficient to verify I' for queries where the caller of the future equals this as reasoning is local, and independent of the behavior of other objects. But this would require notation for expressing the caller of a future (say $u.\mathsf{caller}$, defined by $ft(o,n).\mathsf{caller} = o$) in the specification (and possibly programming) language. However, the verification cost of having I' in the rule for query and in the axiom for *while*, is not great since one is obliged to prove $I(\overline{w},\mathcal{H})$ at the end of the body.

5.3 Compositional Reasoning

The class invariant $I(\overline{w},\mathcal{H})$ for some class C is a predicate over the fields $\overline{w}$, the local history $\mathcal{H}$, as well as the formal class parameters $\overline{cp}$ and this, which are constant (read-only) variables. *History invariants* $I_C(\mathcal{H})$ for instances of C, expressed as a predicate over the local history, can be derived from the class invariant by hiding fields, i.e., $\exists \overline{w} \,.\, I(\overline{w},\mathcal{H})$.

$$I_C(\mathcal{H}) \triangleq \exists \overline{w} \,.\, I(\overline{w},\mathcal{H})$$

Notice that the history invariants should be prefix-closed since according to the definition in Section 4.2 C-local invariant property must be satisfied by all reachable states. Consequently, $\exists \overline{w} \,.\, I(\overline{w},\mathcal{H})$ should be weakened if needed in order to obtain prefix-closedness. Then we assume from now on that $I_C(\mathcal{H})$ is prefix-closed.
For an instance o of C with actual parameter values $\overline{e}$, the *object invariant* $I_{o:C(\overline{e})}(h)$ is defined by the class invariant applied to the local projection of the history and instantiating this and the class parameters:

$$I_{o:C(\overline{e})}(h) \triangleq I_C(h/o)^{\mathsf{this},\overline{cp}}_{o,\overline{e}}$$

where I_C is a prefix-closed class invariant as above, with hidden internal state $\overline{w}$. We consider a composition rule for a system S of objects $o : C(\overline{e})$ together with dynamically generated objects by S. The history invariant $I_S(h)$ for such a system is then given by combining the history invariants of the composed objects:

$$I_S(h) \triangleq \mathit{wf}(h) \bigwedge_{(o:C(\overline{e}))\in S\cup ob(h)} I_{o:C(\overline{e})}(h)$$

where the function $ob : \mathsf{Seq[Ev]} \to \mathsf{Set[Obj} \times \mathsf{Cls} \times \mathsf{List[Data]]}$ returns the set of created objects (each given by its object identity, associated class and class parameters) in a history:

$$\begin{array}{lcl} ob(\varepsilon) & \triangleq & \{main : Main(\varepsilon)\} \\ ob(h \cdot \langle o \xrightarrow{new} o', C, \overline{e}\rangle) & \triangleq & ob(h) \cup \{o' : C(\overline{e})\} \\ ob(h \cdot \boldsymbol{others}) & \triangleq & ob(h) \end{array}$$

(where ***others*** matches all other events). By choosing S as $\{main : Main(\varepsilon)\}$ we may reason about a global system by means of $I_{\{main:Main(\varepsilon)\}}(h)$.
The wellformedness property serves as a connection between the local histories. Note that the system invariant is obtained directly from the history invariants of the composed objects, without any restrictions on the local reasoning, since the local histories are disjoint. This ensures compositional reasoning. The composition rule is similar to [18], which also considers dynamically created objects.

5.4 Soundness Proof of Compositional Reasoning

The proof rule for composition is:

$$\textsf{composition}\ \frac{\vdash_C I_C(h),\ \text{for each C in } \overline{Cl}}{\overline{Cl} \vdash \mathit{wf}(h) \bigwedge_{(o:C(\overline{e}))\in ob(h)} I_{o:C(\overline{e})}(h)}$$

Note that $\vdash_C I_C(h)$ is trivial for $I_C(h) \triangleq \mathit{true}$, thus one may provide invariants for a subset of the classes and using *true* as default invariant for the rest.

Theorem 2 *The object composition rule is sound.*

Proof. We show that the composition rule preserves soundness. For each class C we may then assume $\models_C I_C(h)$ which by definition is

$$\forall G, o \,.\, init_{\overline{Cl}} \longrightarrow^* G \wedge G.hist = h \wedge o \in G.obj \wedge G[o].class = C \Rightarrow I_{o:C(\overline{e})}(h)$$

Next we prove $\models I_{o:C(\overline{e})}(h)$ for all C-objects in h, i.e.,

$$\forall G \,.\, init_{\overline{Cl}} \longrightarrow^* G \wedge G.hist = h \Rightarrow \bigwedge_{(o:C(\overline{e}))\in ob_C(h)} I_{o:C(\overline{e})}(h)$$

letting $ob_C(h)$ denote the set of all C-objects in h. This reduces to proving that each C-object in $G.hist$ is found in $G.obj$. This can be proved by induction on the length of an execution. Finally by Lemma 1 we have $\models \mathit{wf}(h)$; and since conjunction commutes with validity we have $\models \mathit{wf}(h) \bigwedge_{(o:C(\overline{e}))\in ob(h)} I_{o:C(\overline{e})}(h)$.

5.5 Example

In this example we consider object systems based on the classes found in Listing. 1 and 2. Assume that the global system consists of the objects $w_1 : \mathit{Worker}$, $w_2 : \mathit{Worker}$, $w_3 : \mathit{Worker}$, $mr : \mathit{MapReduce}(wp)$, and $m : \mathit{Main}(mr)$, where the only visible activity of m is that it invokes *mapReduce* method on the object mr. The semantics may lead to several global histories for this system, depending on the interleaving of the different object activities. For convenience, below we abbreviate the method names *mapReduce* to mR, *invokeMap* to ivM, and *invokeReduce* to ivR. One global history h caused by a call to mR on mr from m is as follows:

$$\begin{aligned}
&[\langle m \rightarrow mr, u_1, \mathsf{mR}, \overline{e_1}\rangle, \langle m \twoheadrightarrow mr, u_1, \mathsf{mR}, \overline{e_1}\rangle,\\
&\langle mr \rightarrow w_1, u_2, \mathsf{ivM}, \overline{e_2}\rangle, \langle mr \rightarrow w_2, u_3, \mathsf{ivM}, \overline{e_3}\rangle,\\
&\langle mr \twoheadrightarrow w_2, u_3, \mathsf{ivM}, \overline{e_3}\rangle, \langle mr \twoheadrightarrow w_1, u_2, \mathsf{ivM}, \overline{e_2}\rangle,\\
&\langle \leftarrow w_1, u_2, \mathsf{ivM}, e_2\rangle, \langle \leftarrow w_2, u_3, \mathsf{ivM}, e_3\rangle, \langle mr \twoheadleftarrow, u_2, e_2\rangle, \langle mr \twoheadleftarrow, u_3, e_3\rangle,\\
&\langle mr \rightarrow w_2, u_4, \mathsf{ivR}, \overline{e_4}\rangle, \langle mr \twoheadrightarrow w_2, u_4, \mathsf{ivR}, \overline{e_4}\rangle, \langle mr \rightarrow w_1, u_5, \mathsf{ivR}, \overline{e_5}\rangle,\\
&\langle mr \rightarrow w_3, u_6, \mathsf{ivR}, \overline{e_6}\rangle, \langle mr \twoheadrightarrow w_3, u_6, \mathsf{ivR}, \overline{e_6}\rangle, \langle \leftarrow w_2, u_4, \mathsf{ivR}, e_4\rangle,\\
&\langle \leftarrow w_3, u_6, \mathsf{ivR}, e_6\rangle, \langle mr \twoheadrightarrow w_1, u_5, \mathsf{ivR}, \overline{e_5}\rangle, \langle \leftarrow w_1, u_5, \mathsf{ivR}, e_5\rangle,\\
&\langle mr \twoheadleftarrow, u_4, e_4\rangle, \langle mr \twoheadleftarrow, u_6, e_6\rangle, \langle mr \twoheadleftarrow, u_5, e_5\rangle, \langle \leftarrow mr, u_1, \mathsf{mR}, e_1\rangle]
\end{aligned}$$

It follows that the Reduce phase will starts only after the Map phase has been completed. In addition, none of the requests sent out to the workers is uncompleted when the call to mR on mr is finished. We may derive these properties within the proof system from the following class invariants:

$$\begin{aligned}
I_{\mathit{Worker}}(\mathcal{H}) \;\triangleq\;& \mathcal{H} \leqslant [\langle c \twoheadrightarrow \mathsf{this}, u_1, \mathsf{ivM}, \overline{e_1}\rangle, \langle \leftarrow \mathsf{this}, u_1, \mathsf{ivM}, e_1\rangle |\\
&\langle c \twoheadrightarrow \mathsf{this}, u_2, \mathsf{ivR}, \overline{e_2}\rangle, \langle \leftarrow \mathsf{this}, u_2, \mathsf{ivR}, e_2\rangle \,.\, \texttt{\textbf{some}}\ c, u_1, u_2, \overline{e_1}, e_1, \overline{e_2}, e_2]^*\\
I_{\mathit{MapReduce}(wp)}(\mathcal{H}) \;\triangleq\;& \mathcal{H} \leqslant [\langle c \twoheadrightarrow \mathsf{this}, d, \mathsf{mR}, \overline{e_1}\rangle, \langle \mathsf{this} \rightarrow \_, \_, \mathsf{ivM}, \_\rangle^a,\\
&\langle \mathsf{this} \twoheadleftarrow, \_, \_\rangle^a, \langle \mathsf{this} \rightarrow \_, \_, \mathsf{ivR}, \_\rangle^b, \langle \mathsf{this} \twoheadleftarrow, \_, \_\rangle^b,\\
&\langle \leftarrow \mathsf{this}, d, \mathsf{mR}, e_1\rangle \,.\, \texttt{\textbf{some}}\ c, d]^*
\end{aligned}$$

Here we use regular expression notation to express patterns over the history, letting $|$ denote choice, letting superscript b specify b repetitions of a pattern, and $h \leqslant p^*$ express that h is a prefix of a repeated pattern p where additional variables occurring in p (after **some**) may change for each repetition. Notice that the class invariant of *MapReduce* ensures that for each of the invocation event in $\langle \mathsf{this} \rightarrow \_, \_, \mathsf{ivM}, \_\rangle^a$, there is a corresponding fetch event in $\langle \mathsf{this} \twoheadleftarrow, \_, \_\rangle^a$ by the same future identity. Same approach is applied to $\langle \mathsf{this} \rightarrow \_, \_, \mathsf{ivR}, \_\rangle^b$ and $\langle \mathsf{this} \twoheadleftarrow, \_, \_\rangle^b$. These class invariants are straightforwardly verified in the above proof system.
The corresponding object invariants for $w_1 : \mathit{Worker}$, $w_2 : \mathit{Worker}$, $w_3 : \mathit{Worker}$ and $mr : \mathit{MapReduce}(wp)$ are obtained by substituting actual values for this and class parameters:

$$\begin{aligned}
I_{w_i:\mathit{Worker}}(h) \;\triangleq\;& h/w_i \leqslant [\langle \_ \twoheadrightarrow w_i, u_1, \mathsf{ivM}, \overline{e_1}\rangle, \langle \leftarrow w_i, u_1, \mathsf{ivM}, e_1\rangle |\\
&\langle \_ \twoheadrightarrow w_i, u_2, \mathsf{ivR}, \overline{e_2}\rangle, \langle \leftarrow w_i, u_2, \mathsf{ivR}, e_2\rangle \,.\, \texttt{\textbf{some}}\ u_1, u_2, \overline{e_1}, e_1, \overline{e_2}, e_2]^*\\
I_{mr:\mathit{MapReduce}(wp)}(h) \;\triangleq\;& h/mr \leqslant [\langle \_ \twoheadrightarrow mr, d, \mathsf{mR}, \overline{e_1}\rangle, \langle mr \rightarrow \_, \_, \mathsf{ivM}, \_\rangle^a,\\
&\langle mr \twoheadleftarrow, \_, \_\rangle^a, \langle mr \rightarrow \_, \_, \mathsf{ivR}, \_\rangle^b, \langle mr \twoheadleftarrow, \_, \_\rangle^b,\\
&\langle \leftarrow mr, d, \mathsf{mR}, e_1\rangle \,.\, \texttt{\textbf{some}}\ d]^*
\end{aligned}$$

The global invariant of a system S with the objects, $w_1 : \mathit{Worker}$, $w_2 : \mathit{Worker}$, $w_3 : \mathit{Worker}$, $mr : \mathit{MapReduce}(wp)$ and $m : \mathit{Main}(mr)$ is then

$$I_S(h) \;\triangleq\; \mathit{wf}(h) \wedge I_{m:\mathit{Main}(mr)}(h) \wedge I_{mr:\mathit{MapReduce}(wp)}(h) \bigwedge_{i \in \{1,2,3\}} I_{w_i:\mathit{Worker}}(h)$$

where wellformedness allows us to relate the different object histories. From this global invariant we may derive that the Reduce phase will starts only after the Map phase has been completed. Besides, none of the requests sent out to the workers is uncompleted when the call to mR on mr is finished.

As a special case, we consider a system where the instance of *Main* invokes mR only once, i.e. $I_{m:Main(mr)}(h) \triangleq h/m \preccurlyeq [\langle m \rightarrow mr, u, \mathsf{mR}, \overline{e}\rangle\,.\,\texttt{some}\ u]$. History wellformedness then ensures that the cycles defined by the remaining invariants are repeated at most once, and that variables in the patterns are connected, i.e., the future u in $I_{m:Main(mr)}$ is identical to the future d in $I_{mr:MapReduce(wp)}$. The global invariant then reduces to the following:

$$
\begin{array}{rcl}
I_S(h) & \triangleq & wf(h) \wedge h/m \preccurlyeq [\langle m \rightarrow mr, u, \mathsf{mR}, \overline{e}\rangle] \wedge \\
& & h/w_1 \preccurlyeq [\langle mr \twoheadrightarrow w_1, u_1, \mathsf{ivM}, \overline{e_1}\rangle, \langle \leftarrow w_1, u_1, \mathsf{ivM}, e_1\rangle | \\
& & \langle mr \twoheadrightarrow w_1, u_2, \mathsf{ivR}, \overline{e_2}\rangle, \langle \leftarrow w_1, u_2, \mathsf{ivR}, e_2\rangle\,.\,\texttt{some}\ u_1, u_2, \overline{e_1}, e_1, \overline{e_2}, e_2]^* \wedge \\
& & h/w_2 \preccurlyeq [\langle mr \twoheadrightarrow w_2, u_3, \mathsf{ivM}, \overline{e_3}\rangle, \langle \leftarrow w_2, u_3, \mathsf{ivM}, e_3\rangle | \\
& & \langle mr \twoheadrightarrow w_2, u_4, \mathsf{ivR}, \overline{e_4}\rangle, \langle \leftarrow w_2, u_4, \mathsf{ivR}, e_4\rangle\,.\,\texttt{some}\ u_3, u_4, \overline{e_3}, e_3, \overline{e_4}, e_4]^* \wedge \\
& & h/w_3 \preccurlyeq [\langle mr \twoheadrightarrow w_3, u_5, \mathsf{ivM}, \overline{e_5}\rangle, \langle \leftarrow w_3, u_5, \mathsf{ivM}, e_5\rangle | \\
& & \langle mr \twoheadrightarrow w_3, u_6, \mathsf{ivR}, \overline{e_6}\rangle, \langle \leftarrow w_3, u_6, \mathsf{ivR}, e_6\rangle\,.\,\texttt{some}\ u_5, u_6, \overline{e_5}, e_5, \overline{e_6}, e_6]^* \wedge \\
& & h/mr \preccurlyeq [\langle m \twoheadrightarrow mr, u, \mathsf{mR}, \overline{e}\rangle, \langle mr \rightarrow \_, \_, \mathsf{ivM}, \_\rangle^a, \\
& & \langle mr \twoheadleftarrow, \_, \_\rangle^a, \langle mr \rightarrow \_, \_, \mathsf{ivR}, \_\rangle^b, \langle mr \twoheadleftarrow, \_, \_\rangle^b, \\
& & \langle \leftarrow mr, u, \mathsf{mR}, e\rangle]
\end{array}
$$

This invariant allows a number of global histories, depending on the interleaving of the activities in the different objects. The history h presented first in this section satisfies the invariant, and represents one particular interleaving.

6 Related Work and Conclusion

Models for asynchronous communication without futures have been explored for process calculi with buffered channels [25], for agents with message-based communication [2], for method-based communication [32], and in particular for Java [22]. Behavioral 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 survey of these challenges can be found in [4]. Soundness of the parallel composition rules for shared-variable concurrency and synchronous message passing are proved in [16]. A Hoare Logic for concurrent processes (objects) is presented in [14]. The Hoare Logic is compositional, and soundness and relative completeness are proven. In contrast to our work, communication is by message passing rather than by futures, and the objects communicate through FIFO channels.

The present approach follows the line of work based on communication histories to model object communication events in a distributed setting [8, 10, 25]. Objects are concurrent and interact solely by method calls and futures, and remote access to object fields are forbidden. By creating unique references for method calls, the *label* construct of Creol [29] resembles futures, as callers may postpone reading result values. Verification systems capturing Creol labels can be found in [4, 21]. However, a label reference is local to the caller, and cannot be shared with other objects. A reasoning system for futures has been presented in [15], 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.

A reasoning system for asynchronous methods in ABS without futures is presented in [18]. We here define a five-event semantics reflected actions on shared futures and object creation. The semantics gives a clean separation of the activities of the different objects, which leads to disjointness of local histories. Thus, object behavior can be specified in terms of the observable interaction of the current object only. This is essential for obtaining a simple reasoning system. In related approaches, e.g., [4, 21], events are visible to more than one object. The local histories must then be updated with the activity of other objects, resulting in more complex reasoning systems. Based on the five-event semantics, we present a compositional reasoning system for distributed, concurrent objects with asynchronous method calls. A class invariant defines a relation between the inner state and the observable communication of instances, and can be verified independently for each class. The class invariant can be instantiated for each object of the class, resulting in a history invariant over the observable behavior of the object. Compositional reasoning is ensured as history invariants may be combined to form global system specifications. The composition rule is similar to [18], which is inspired by previous approaches [33, 34]. This work is an extension of our former paper [20]. Here we analyze a larger case study using futures, extend the language and semantics. Also, soundness proofs for class reasoning and object composition are provided.

A result of this paper is that sound composition requires the `query` rule to have a condition related to the invariant, not found in earlier papers. We consider here global history invariants that are continuously satisfied, in the sense that any reachable global configuration of an object system must satisfy the invariant. The condition on the `query` rule would not be needed with a weaker notion of global history invariants stating that the global invariant holds as long as all objects are live (not blocked). Verification-wise the condition on the `query` rule, is somewhat similar to a that of a query statement releasing the processor, as for instance the **`await`** *future* statement of the ABS language. Semantically, a blocking query has the advantage that it does not change the state, whereas a non-blocking query gives a state satisfying the local invariant. Thus the combination of the query and processor release mechanisms will not add significant verification complexity, and is also attractive from a programming perspective.

In order to focus on the future mechanism, this paper considers a core language with shared futures. The report version [19] considers a richer language, including constructs for inter-object process control and processor release. The verification system is suitable for an implementation within the KeY framework. With support for (semi-)automatic verification, such an implementation will be valuable when developing larger case studies. It is also natural to investigate how our reasoning system would benefit from extending it with rely/guarantee style reasoning [16]. Assumptions about callee behavior may, for instance, be used to express properties of return values. More sophisticated techniques may also be used, e.g., [13,28] adapts rely/guarantee style reasoning to history invariants. However, such techniques requires more complex composition rules.

References

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

[2] G. Agha, S. Frolund, W. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and modularity mechanisms for concurrent computing. *Parallel Distributed Technology: Systems Applications, IEEE*, 1(2):3 –14, may 1993.

[3] A. Ahern and N. Yoshida. Formalising Java RMI with Explicit Code Mobility. Theoretical Computer Science, 389(3):341 – 410, 2007.

[4] W. Ahrendt and M. Dylla. A system for compositional verification of asynchronous objects. Science of Computer Programming, 77(12):1289–1309, Oct. 2012.

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

[6] H. G. Baker Jr. and C. Hewitt. The Incremental Garbage Collection of Processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 55–59, New York, NY, USA, 1977. ACM.

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

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

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.

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

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

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

[13] O.-J. Dahl and O. Owe. Formal methods and the RM-ODP. Research Report 261, May 1998.

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

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

[16] W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency verification: introduction to compositional and non-compositional methods.* Cambridge University Press, New York, NY, USA, 2001.

[17] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[18] C. C. Din, J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. *Journal of Logic and Algebraic Programming*, 81(3):227–256, 2012.

[19] C. C. Din, J. Dovland, and O. Owe. An approach to compositional reasoning about concurrent objects and futures. Research Report 415, Dept. of Informatics, University of Oslo, Feb. 2012. Available at `http://folk.uio.no/crystald/`.

[20] C. C. Din, J. Dovland, and O. Owe. Compositional reasoning about shared futures. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *Proc. International Conference on Software Engineering and Formal Methods (SEFM'12)*, volume 7504 of *LNCS*, pages 94–108. Springer-Verlag, 2012.

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

[22] K. E. K. Falkner, P. D. Coddington, and M. J. Oudshoorn. Implementing asynchronous remote method invocation in java. In *6th Australian Conference on Parallel and Real-Time Systems*, pages 22–34. Springer-Verlag, 1999.

[23] R. H. Halstead Jr. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, Oct. 1985.

[24] Full ABS Modeling Framework (Mar 2011). Deliverable 1.2 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

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

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

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

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

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

[30] B. H. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In D. S. Wise, editor, *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88)*, pages 260–267. ACM Press, June 1988.

[31] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science, 96:73–155, 1992.

[32] B. Morandi, S. S. Bauer, and B. Meyer. SCOOP – A Contract-Based Concurrent Object-Oriented Programming Model. In P. Müller, editor, *Advanced Lectures on Software Engineering, LASER Summer School 2007/2008*, volume 6029 of *LNCS*, pages 41–90. Springer, 2008.

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

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

[35] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86). Sigplan Notices*, 21(11):258–268, Nov. 1986.