Verifiable Programming of Object-Oriented and Distributed Systems*

Olaf Owe

University of Oslo, Department of Informatics, Oslo, Norway, and
University of California, Santa Cruz, Department of Computer Science, USA

April 2015

Abstract

Distributed and concurrent object-oriented systems are difficult to analyze due to the complexity of their concurrency, communication, and synchronization mechanisms. This paper explores a programming paradigm based on active, concurrent objects communicating by so-called asynchronous method calls giving rise to efficient interaction by means of non-blocking method calls, implemented by means of message passing. The paradigm facilitates invariant specifications over the locally visible communication history of each class. Compositional reasoning is supported by rules comparable to those of sequential programming, and global properties may be derived from local specifications. Reasoning about inheritance is not limited by behavioral subtyping, but allowing free reuse of code, considering also multiple inheritance. A small, illustrating example is considered.

Keywords distributed systems, object-orientation, program reasoning, compositional reasoning, concurrent objects, asynchronous communication, communication history.

*This is an early version of a paper appearing as Chapter 5 in the book From Action System to Distributed Systems: The Refinement Approach, A tribute to Kaisa Sere. Editors: Luigia Petre and Emil Sekerinski, Chapman and Hall/CCRC Book Press, online April 2016
1 Introduction

Today distributed systems form an essential basis of modern infrastructure. Object-oriented programming is a leading programming paradigm for such systems. It is in general required that object-oriented, distributed systems are of high quality and behave properly. However, quality assurance of object-oriented, distributed systems is a non-trivial and challenging topic. There are a number of approaches for different kinds of quality assurance. Formal methods play an important role in the systematic studies of such systems and in their semantics. Several formal approaches consider functional or single-assignment languages, which are semantically simpler than the setting of imperative, object-oriented programs. To make the result applicable to common imperative object-oriented languages, one may then consider refinement approaches. The approach considered by Kaisa Sere among others [4] is based on refinement, starting by programming in the mathematically simpler formalism of Action Systems [5], based on the paradigms of single assignment and guarded command, and making a number of refinement steps, ending up with a correct-by-construction distributed object system. This line of work focuses on refinement theory, including concepts for increasing concurrency, like superposition [5], rather than reasoning on imperative, distributed programs.

In this paper we will not consider refinements between designs at different abstraction levels, but rather consider imperative, object-oriented, distributed systems directly; and in particular formal reasoning and verification of such systems, focusing on concurrency aspects and inheritance aspects. The concurrency model has some similarities to that of Action Systems. Both are based on an execution model where each concurrent unit handles one action/process at a time and where guards may lead to several enabled actions/processes, non-deterministically chosen. Object-oriented Action Systems allow non-terminating local actions, like active objects of our concurrency model. Incremental development by refinement in Action Systems has similarities to incremental program design by inheritance in our work.

The challenges in imperative, object-oriented, concurrent program reasoning are related to several factors including the programming language and constructs chosen, the specification language, and the verification system, and last but not least, tool support. The goal of this paper is to strive for simplicity in reasoning about object-oriented, distributed systems. In order to approach this goal we will make some recommendations with respect to language constructs, in particular object-oriented constructs and reasoning techniques. We will keep tool support in mind by emphasizing reasoning approaches that allow mechanized analysis.

For object-oriented systems the notions of encapsulation, inheritance, late binding, and object creation, are central, and a proper treatment of these is essential. As class hierarchies are open-ended, reasoning should be incremental with respect to addition of subclasses, rather then based on a closed-world assumption. Since distributed systems involve concurrency, it is essential to have a reasoning system that can handle concurrent units and to consider programming constructs that allow efficient interaction of concurrent units.

Simplicity of reasoning does not only concern the choice of specification language and verification system, but also the choice of programming language constructs and their semantics. In particular one should choose a programming language with compositional semantics that allows compositional reasoning. We therefore focus on programming mechanisms that have semantical advantages and allow efficient interaction. The paper builds on research results from the language development and reasoning activity around the paradigm of so-called active concurrent objects developed through the languages OUN [36, 35], Creol [22, 26, 25, 23, 20], and partly ABS [21]. The contribution of this paper is to reconsider rele-
vant results from the work on active concurrent objects, while making further improvements with respect to simplicity of verifiability.

The paper is organized as follows: The next section discusses basic programming constructs for concurrent objects, introducing a core language, ending up with an example. Sec. 3 motivates and explains invariants over communication histories. Sec. 4 discusses reasoning challenges for single and multiple inheritance. Sec. 5 suggests a Hoare style verification system for classes, based on invariants over communication histories, reconsidering the example. Sec. 6 discusses extensions to future and delegation mechanisms. Finally, Sec. 7 comments on related work and makes a conclusion.

2 Basic Programming Constructs

For distributed systems the notion of concurrent unit together with interaction and cooperation mechanisms are crucial. The Actor model [2] is semantically appealing giving rise to compositional semantics, unifying interaction and cooperation mechanisms through message passing. It is clearly simpler than the shared variable setting and the general thread-based model where a thread may execute code on several objects, and where non-trivial interference complicates the semantics. For object-oriented systems the natural unit of interaction is the object; in fact the term “activity” was used before the terms “class” and “object” appeared in Simula 67 [10]. Concurrent objects extend the Actor paradigm to the object-oriented setting, allowing two-way interaction by means of remote method calls rather than one-way interaction by messages. Inheritance of code is meaningful since program code is organized in methods.

Consider a remote method call, say $v := o.m(\bar{e})$, where the callee $o$ is an external object supporting a (type-correct) method $m$, $\bar{e}$ is the list of actual parameters, and $v$ is the variable receiving the return value. In the setting of concurrent objects the default way of performing such a method call is that the calling object blocks while the callee object $(o)$ is executing the method when free to do so [11]. One may improve efficiency by adding a list of statements $s$ that the caller can do while waiting, say by the syntax $v := o.m(\bar{e})\{s\}$, where the caller blocks after performing $s$ and only if the return value has not arrived.

However, efficient interaction of concurrent objects requires some kind of non-blocking call mechanism allowing the caller object to do something else (like handling another incoming call) while waiting for the call to the external object to finish. With a suspension mechanism a method invocation can be suspended and placed in a queue of suspended processes while another (enabled) process may continue instead. Enabledness can be controlled by guards, either a Boolean condition $b$ on the state of an object or the presence of a return value of a call. The guarded await statement $\text{await } b$ suspends while waiting for $b$ to become true. The await statement $\text{await } v := o.m(\bar{e})$ initiates the call and suspends while waiting for the call to complete [22]. This allows an object to handle a number of processes, being incoming calls or continuations of such calls, or self calls representing self activity. Local methods intended for non-terminating self activity need to have release points in order to allow periods with reactive behavior. Method invocations and method results are realized by message passing between the caller and callee objects in an asynchronous manner.

We consider objects with their own virtual processor; thus at most one process in an object can be active at a given time. This allows sequential reasoning inside a class, and suspension control is programmed within each method. Thus neither notification nor signaling from other objects or processes is needed, which simplifies reasoning. For instance
in concurrent Java programs, some of the hardest bugs to locate are missing or misplaced
notify statements. The await statement can be efficiently implemented since enabledness of
waiting processes is only checked when no other process is active, i.e., at a suspension point
or end of a method invocation. This await mechanism gives rise to passive waiting.

Concurrent objects with suspension control may be reactive, responding to method calls
from the environment, or active, performing their own activity. Initial active behavior is
programmed by the initialization code, for instance by calling a non-terminating local run
method with suspension points allowing reactive behavior to be mixed with active behavior.
By dynamic object creation, new concurrent units can be generated at any time, starting
with their run method, either by themselves or by the parent object. Non-blocking method
calls give independence of objects, however, sometimes synchronization may be desirable, in
which case one may use the default object-oriented mechanism for blocking calls. For the
case that the result of a call is not needed by the caller, we use the syntax $o.m(e)$, letting the
caller continue with the next statement after sending the invocation message to the callee.

Concurrent objects interact by means of remote method calls only, and remote access
to fields is not allowed. If allowed, this would create aliasing problems in reasoning. An
assignment statement has the form $v := e$ where $v$ is a simple variable (either a field variable
or a local variable of the method) and $e$ is a pure expression. Even though we have aliasing
in the sense that $v$ may refer to the same object as another variable of the same process, the
classical Hoare rule for assignment is satisfied, i.e., $[Q^v] v := e [Q]$ where the precondition is
obtained by substituting $e$ for $v$. Assertions are here enclosed in square brackets (while curly
brackets will occur in code). Thus reasoning is not affected by semantic alias considerations,
since the (pointer) value of $v$ is updated and not the object referred to by $v$.

Object-oriented languages have different approaches to class encapsulation and hiding. In
addition, reasoning about object-oriented systems requires some kind of abstraction mecha-
nism. Behavioral interfaces provide abstraction as well as encapsulation and hiding [9]. We
let behavioral interfaces control which methods are visible, while hiding all fields, and let
interfaces contain specification of the object behavior by means of the local communication
history. Class encapsulation is then enforced by declaring object variables by interfaces,
rather than by classes. Local data structures in an object can be defined by data types.
Thus objects represent independent units, while data types are used for values that can be
copied. A variable declared of a data type represents a value of that type, for instance a list
of object identities can be represented as a data type.

The setting of concurrent objects gives rise to many logical concurrent units. However
they are easily mapped to a given number of physically concurrent CPUs, since they are
working asynchronously. The objects may reflect activity on parallel hardware architectures
as well as distributed machines working in parallel.

A Core Language

A core language based on the concurrency model above is given in Fig. 1. Program code
is organized in classes. Classes and interfaces may include specifications in the form of in-
variants and pre/postconditions of methods. To control code and specification reuse, we let
the keyword inherits indicate code reuse, while the keyword extends indicate reuse of
both code and specification. A class may implement a number of interfaces, specified
by an implements clause. The code of a class (including inherited code) must satisfy its
specifications (including inherited specifications), and must satisfy the implements clauses
(including inherited ones caused by extends clauses). This gives rise to verification obliga-
The language supports the mechanisms for remote calls and suspension described in Sec. 2, including guarded suspension `await b`, suspending call `await v := o.m(\overline{r})`, interleaved call `v := o.m(\overline{r})\{s\}`, as well as simple call `o.m(\overline{r})`. A simple call does not wait for a result, and is useful when the result is not needed. A blocking call `v := o.m(\overline{r})` is equivalent to an interleaved call with an empty statement list. The syntax `v := m(\overline{r})` represents a late-bound local call and `v := C : m(\overline{r})` a static local call, taking the method `m` of class `C`. Both are implemented in the usual stack-based manner. Dot-notation in a call, `v := o.m(\overline{r})`, may only
interface Bank {
  Bool sub(Nat x) [return= true]
  Int bal() [return= sum(h)]
  where sum(empty) = 0, sum(h: ←add(x;true)= sum(h)+x,
           h: ←sub(x;true)=sum(h)−x, sum(h:others=)=sum(h) }
interface PerfectBank extends Bank {Bool sub(Nat x) [return= true]}
interface BankPlus extends Bank {inv sum(h)>=0 }

class BANK implements PerfectBank {Int bal=0;
  Bool upd(Int x) {bal:=bal+x; return true} [inv, bal=sum(h)+x and return=true]
  Bool add(Nat x) {Bool ok:=upd(x); return ok} [return= true]
  Bool sub(Nat x) {Bool ok:=upd(−x); return ok} [return= true]
  Int bal() {return bal}
  inv bal=sum(h) }

class BANKPLUS implements BankPlus inherits BANK{
  Bool upd(Int x) {Bool ok:=(bal+x>=0); if ok then ok:= BANK:upd(x) fi; return ok}
                 [inv, bal>=0 and if return then bal=sum(h)+x else bal=sum(h)]
  inv BANK:inv and bal >=0 }

class BANKWAIT implements BankPlus extends BANK {
  Bool upd(Int x){await bal+x>=0 ;bal:=bal+x; return true} [bal >=0]
  inv BANKPLUS:inv }

Figure 2: A simple bank example. In assertions, inv refers to the current invariant, while C: inv refers to the invariant of class C. The auxiliary function sum calculates the balance from the local history h by means of past return events of completed add and sub calls.

be used when the declared interface of o supports a (type-correct) method m. It is possible to make a call to a null object, but no return value will ever be received by the caller in such a case. A simple call statement to null will terminate, but not a blocking call to null, while a suspended call to null will never be enabled. (Exceptions are not part of the core language.)

A class or interface may specify an invariant and add pre/postconditions to method declarations. Specifications may refer to the local communication history h, as discussed in Sec. 3. A pre- and postcondition pair is written [P, Q] where P is the precondition and Q the postcondition. The trivial precondition true may be omitted. An inherited method may be re-specified by simply adding the method declaration with the added pre/postcondition. The keyword inv identifies invariants and where identifies auxiliary function definitions. Auxiliary functions may be defined inductively, letting others in a left hand side match any remaining cases, and letting _ denote arguments in a left-hand side pattern that are not needed in the right-hand side.
interface Pin(Text pin) {
  Bool open(Text code) [return=(user(h)=caller and code=pin)]
  Bool close() [return=(user(h)=null)]
  where user(empty)=null, user(h·o←open(code;_))=
      if code=pin and user(h)=null then o else user(h),
      user(h·o←close;_))=if user(h)=o then null else user(h),
      user(h·others)=user(h) }

class PIN(Text pin) implements Pin(pin) { Any u=null;
  Bool open(Text code) {if code=pin and u=null then u:=caller fi;
      return (u=caller and code=pin)}
  Bool close() {if u=caller then u:=null fi; return (u=null)}
  inv u = user(h) }

class PINBANK(Text pin) implements Bank extends PIN(pin) inherits BANK {
  Bool sub(Nat x) {Bool ok:=false; if u=caller then ok:=upd(-x) fi; return ok}
  [return=(u=caller)]
  inv bal=sum(h) and u=user(h) }

Figure 3: A simple example of multiple inheritance, reusing the bank example. The interface Any is the most general interface, supported by any class.

An Example

Fig. 2 and 3 show a simple bank example illustrating history-based specification, suspension, static and late bound calls, as well as single and multiple inheritance. The methods add, sub, and upd, return a Boolean value indicating whether the transaction was successfully performed. Both add and sub are implemented by means of upd. Interface Bank, exporting methods add, sub, and bal, says that bal returns the sum calculated from the successful add and sub transactions of the local communication history h. Function sum is defined inductively over the history, according to the approach in Sec. 3. Intuitively it says that the sum is the sum of amounts added minus amounts subtracted, counting only successful method returns (returning true). The Bank interface does not require that sub always succeeds. There are two subinterfaces of Bank, PerfectBank, which requires that all transactions succeed, and BankPlus, which requires that the sum is never negative.

All class implementations are based on the BANK invariant expressing that the field bal equals the calculated sum, bal = sum(h). Both subclasses of BANK ensure non-negative balance, BANKPLUS by letting sub fail (return false) in case of insufficient balance, and BANKWAIT by reimplementing upd such that it suspends until the balance is large enough.

Code reuse here is demonstrated by inhering the bal, add, and sub methods of class BANK in the two subclasses of BANK, while redefining the upd method. Specification reuse is demonstrated by the class BANKWAIT, in which upd inherits the specification of BANK, resulting in the specification [bal = sum(h), bal = sum(h)+x∧bal >= 0∧return = true].

As discussed below, a challenge with respect to reasoning is that the BANK specification (namely the postcondition of upd and support of PerfectBank) is violated by the subclass BANKPLUS. We will suggest a way to solve this challenge; and in Sec. 5 we show how to verify parts of the example.
Class PINBANK in Fig. 3 demonstrates multiple inheritance. It inherits code from both class BANK and PIN, and inherits specifications from class PIN. It implements interface Bank and also interface Pin (implicitly through the `extends` clause).

3 Class Invariants

Class invariants are commonly used for specifying, and reasoning about, object-oriented systems, and are especially useful in a distributed setting with non-terminating concurrent objects[8, 34]. In a global state of such a system, we may assume that each object satisfies its invariant. This gives rise to sound compositional reasoning [39]. Interface invariants reflect parts of class invariants that are relevant to the environment. Encapsulation of inner state means that remote field access should not be allowed since several classes may implement the same interface, each with different fields. In order to capture an abstraction of the object state, one may use model variables [28] or ghost variables such as the communication history [8]. Communication histories have the advantage that they are expressive, since they include all visible events, and one need not invent specific model/ghost variables for each class. One may define functions over the history to extract information relevant in an interface or class specification.

History variables give an abstraction of the state in terms of communication events only, and distinguish different sequences of communication events. Class invariants can be formulated as predicates over the local history of the object as well as field variables, this, and class parameters. Interface invariants can be formulated as predicates over the local history of the object as well as this and interface parameters, but not field variables, since these are not visible in an interface. Interface invariants are supposed to hold at all times and should therefore be prefix closed with respect to the history, whereas class invariants need only hold at suspension and method return points.

A history is a sequence of visible events. We consider the following events:

\( o \rightarrow o'.newC(\bar{e}) \), o creates a new C object \( o' \) with actual class parameters \( \bar{e} \).

This creation event is visible to o. The events for method interaction are:

\( o \rightarrow o'.m(\bar{e}) \), denoting a call to method \( m \) with actual parameters \( \bar{e} \), with o as caller and \( o' \) as callee. This event is caused by o and is visible to o only.

\( o \rightarrow o'.m(\bar{e}) \), denoting start of processing of the call \( o'.m(\bar{e}) \) with o as caller. This event is caused by the callee and is visible to \( o' \) only.

\( o \leftarrow o'.m(\bar{e}; e) \), denoting the generation of the return value \( e \) resulting from the call \( o'.m(\bar{e}) \).

This event is caused by the callee and visible to \( o' \) only.

\( o \leftarrow o'.m(\bar{e}; e) \), denoting the reception of the return value \( e \) by o of the call \( o'.m(\bar{e}) \). This event is performed by the caller and is visible to o only.

Thus the following events are caused by an object o: \( o \rightarrow o'.m(\bar{e}) \), \( o' \rightarrow o.m(\bar{e}) \), \( o' \leftarrow o.m(\bar{e}; e) \), \( o \leftarrow o'.m(\bar{e}; e) \), and \( o \rightarrow o'.newC(\bar{e}) \). The events are referred to as call, start, return, get, and creation events, respectively. All four communication events are needed since method communication is asynchronous and the events above may happen at different times. However, for each call the events must happen in the order given above. This ordering is captured by a notion of global wellformedness, formalized in Sec. 3.
Local specification and reasoning in a class or interface are done with local histories, i.e., the sequence of events visible to this object. In contrast, global specification and reasoning are done in terms of the global history, i.e., the sequence of all events. It follows that the local histories of distinct objects are disjoint in the sense that they do not share events. This allows independent reasoning of each class, and compositional reasoning by means of a simple composition rule stating that the invariant of a global system is the conjunction of all the invariants of the objects involved, adding wellformedness [15].

In specifications, auxiliary functions may be defined inductively over the local history \( h \), using \( \text{empty} \) and append right (\( \cdot \)) as the history constructors. In this way the last event, which reflects the current activity, is explicit. For instance the ends-with predicate (\( \text{ew} \)) can be defined by the two cases (where \( x \) and \( y \) range over events):

\[
\text{empty} \ \text{ew} \ x \ y = \text{false}, \quad \text{h} \cdot \ x \ \text{ew} \ y = (x = y)
\]

using infix notation. The history without the last event can be defined for non-empty histories by \( \text{old}(\text{h} \cdot \ x) = \text{h} \). Projection of the history by a set of events \( s \), denoted \( \text{h}/s \), is defined by

\[
\text{empty}/s = \text{empty}, \quad (\text{h} \cdot \ x)/s = \text{if} \ x \in s \ \text{then} (\text{h}/s) \ \text{else} \ \text{h}/s
\]

We let \( \text{h}/o \) denote the projection of \( \text{h} \) to events caused by \( o \), and we let \( \text{h}/F \) denote the projection of \( \text{h} \) to the alphabet of an interface \( F \), i.e., restricting \( \rightarrow \) and \( \leftarrow \) events to methods of \( F \). Similarly, \( \text{h}/C \) denotes the projection of \( \text{h} \) to the alphabet of a class \( C \), i.e., restricting \( \rightarrow \) and \( \leftarrow \) events to methods of \( C \). And \( \text{h}/(o:F) \), denoting the history of \( o \) as seen through the interface \( F \), is a shorthand for \( \text{h}/o/F \); and \( \text{h}/(o:C) \), denoting the history of \( o \) as seen through the class \( C \), is a shorthand for \( \text{h}/o/C \). In practice, return and get events are essential in invariant specifications, describing output and input, respectively, of the specified object, while call and start events are often not needed (unless synchronization aspects are specified), as in the Bank example. In an interface or class specification we may write \( o \leftarrow m(\pi; e) \) rather than \( o \leftarrow \text{this.m}(\pi; e) \), skipping the redundant this.

### Invariant Refinement and Satisfaction

The invariant of a class \( C \) is written \( I_C(\text{h}, \pi) \) where \( \text{h} \) is the local history and \( \pi \) its fields. The invariant of an interface \( F \) is written \( I_F(\text{h}) \) where \( \text{h} \) is the local history. An interface invariant \( I_F(\text{h}) \) will only restrict events in its own alphabet. In a subinterface or class with a wider alphabet, the invariant is therefore understood as \( I_F(\text{h}/F) \), which ensures that the invariant does not restrict events outside its alphabet. Thus the invariant \( I_F(\text{h}) \) of an interface \( F \) is inherited as \( I_F(\text{h}/F) \) in a subinterface of \( F \). For a class \( C \) implementing \( F \), one must verify that \( I_C(\text{h}, \pi) \Rightarrow I_F(\text{h}/F) \), in which case \( C \) is said to satisfy \( F \). Similarly, a class invariant \( I_C(\text{h}, \pi) \) is inherited as \( I_C(\text{h}/C, \pi) \) in a subclass extending \( C \). Furthermore, for an inductively defined function \( f(h) \), the equation \( f(\text{h} \cdot \text{others}) = f(h) \) is added to make the definition complete and to adjust for any alphabet extension when the function is used in a subinterface or subclass.

For each class \( C \) one must prove that the class invariant is established by the initializing code, that it is maintained by each method exported through a \( C \) interface, that the invariant holds at each suspension point, and that the class satisfies each interface of the class.

In an interface or class, postconditions can easily be expressed by invariants over the history. A postcondition \( [Q(h, \text{return}, \text{caller}, \pi)] \) of a method \( m(\pi) \) abbreviates the invariant \( h \ \text{ew} \ \text{call} \leftrightarrow \text{this.m}(\pi; \text{return}) \Rightarrow Q(h, \text{return}, \text{caller}, \pi) \) where \( \text{return} \) is a special variable to
be used in postconditions to refer to the returned value of a method. In a class postconditions may refer to fields. In an interface, (abstract) fields may be extracted by means of auxiliary functions over the history. In the examples we use postconditions as a notational convenience. Auxiliary function definitions are inherited downwards in subclasses, subinterfaces, and implementing classes, so that they are available for specification and reasoning.

Composition

For simplicity we here ignore interface and class parameters in the discussion. Given an invariant \( I_F \) over the local history of an interface \( F \), we define the object invariant of an object \( o \) as seen through \( F \) by

\[
I_{o:F}(H) \triangleq I_F(H/(o : F))_{\text{this}}
\]

where the projection \( H/(o : F) \) denotes the history \( H \) reduced to the set of events generated by \( o \) and visible through \( F \). Next the object invariant of an object \( o \) of class \( C \) is defined as

\[
I_{o:C}(H) \triangleq \bigwedge_{F \in C.\text{implements}} I_{o:F}(H)
\]

where \( C.\text{implements} \) is the list of interfaces implemented by \( C \) according to the class definition. Finally the global invariant of a system with dynamically created objects initiated by an initial object \( \text{system} : \text{System} \) (where the \( \text{System} \) interface may include minimal requirements and primitives of the underlying operating system) is defined by

\[
I_{\text{global}}(H) = \text{wf}(H) \land \bigwedge_{(o, C) \in \text{ob}(H)} I_{o:C}(H)
\]

where \( \text{ob}(H) \) is the set of object identifiers created in \( H \) by a \( \text{new} \) event (including the initial object):

- \( \text{ob}(\text{empty}) = \{\text{system} : \text{System}\} \)
- \( \text{ob}(H \cdot (o' \rightarrow o.\text{new} C(\overline{e}))) = \text{ob}(H) \cup (o : C)\)
- \( \text{ob}(H \cdot \text{others}) = \text{ob}(H) \)

and where \( \text{wf}(H) \) is the welldefinedness predicate expressing that for each call the communication events obey the natural order \( \text{call}, \text{start}, \text{return}, \text{get} \), that generated object identifiers are fresh, and that all identifiers used by an object \( o \) have been seen by the object:

\[
\begin{align*}
\text{wf}(\text{empty}) & \triangleq \text{true} \\
\text{wf}(H \cdot (o \rightarrow o'.\text{new} C(\overline{e}))) & \triangleq \text{wf}(H) \land o \in \text{ob}(H) \land \text{id}(\overline{e}) \subseteq \text{id}(H/o) \land o' \notin \text{id}(H) \\
\text{wf}(H \cdot (o \rightarrow o'.m(\overline{e}))) & \triangleq \text{wf}(H) \land o \in \text{ob}(H) \land (\{o'\} \cup \text{id}(\overline{e})) \subseteq \text{id}(H/o) \\
\text{wf}(H \cdot (o' \rightarrow o.m(\overline{e}))) & \triangleq \text{wf}(H) \land o \in \text{ob}(H) \\
& \land \#H/\{o' \rightarrow o.m(\overline{e})\} < \#H/\{o' \rightarrow o.m(\overline{e})\} \\
\text{wf}(H \cdot (o' \leftarrow o.m(\overline{e}))) & \triangleq \text{wf}(H) \land o \in \text{ob}(H) \land \text{id}(\overline{e}) \subseteq \text{id}(H/o) \\
& \land \#H/\{o' \leftarrow o.m(\overline{e})\} < \#H/\{o' \leftarrow o.m(\overline{e})\} \\
\text{wf}(H \cdot (o \leftarrow o'.m(\overline{e}))) & \triangleq \text{wf}(H) \land o \in \text{ob}(H) \\
& \land \#H/\{o \leftarrow o'.m(\overline{e})\} < \#H/\{o \leftarrow o'.m(\overline{e})\}
\end{align*}
\]

where \# denotes sequence length and the \( \text{id} \) function gives the set of all identifiers (including \( \text{null} \)) appearing in an expression or history. (In the set \( \{o \leftarrow o'.m(\overline{e})\} \_ \) may be any value.)
4 Inheritance

The semantics of inheritance and late binding has been a research topic for many years, with [40] as an early investigation. Class invariants are used to specify the semantics of a class, and a (logically) strong invariant is usually called for in order to verify that the class satisfies the specifications of the given interfaces [9]. Inheritance is an essential element of object-oriented programming, giving flexible reuse of code. Reasoning is often restricted to a form of behavioral subtyping, implying that a superclass invariant must be respected and maintained by all subclasses [30]. The advantage is that behavioral subtyping allows modular reasoning about late bound calls using the specification of the class of the callee, or the enclosing class in case of a local call. However, this severely limits programming; for instance a perfect bank class, like BANK in Fig. 2, cannot be used to derive a bank class with negative balance protection, like BANKPLUS in the example (nor versions with interest or charges), since this represents a non-conservative extension violating the BANK invariant. Hence flexibility of code reuse is lost if behavioral subtyping is accepted, unless class invariants are very weak. Thus behavioral subtyping is not an acceptable form of reasoning about object-oriented systems in our setting.

The notion of lazy behavioral subtyping [17, 18, 19] allows better flexibility, by talking about two kinds of properties for each class, $S$ (for specifications) and $R$ (for requirements), letting $S_C$ denote properties about class $C$, and $R_C$ minimal properties to be respected by all subclasses of $C$. And $S_C$ is used to verify interface satisfaction as explained. In order to prove that the class satisfies $S_C$, one must in general reason about late-bound local calls in the method bodies of the class, in which case $R_C$ requirements are needed, and $R_C$ must then be enriched with minimal properties of these calls. Lazy behavioral subtyping allows reasoning support for extensible class hierarchies. Each class can be analyzed separately as long as its superclasses have been analyzed earlier. The invariant and pre/postconditions in $C$ form the $S_C$ specifications, while the $R_C$ requirements (initially empty) are generated by adding minimal properties as needed for reasoning about late-bound local calls in the class. Both $S_C$ and $R_C$ properties must be verified in $C$, and by adding all verified properties to $S_C$, the property set $R_C$ will be a subset of $S_C$, i.e., $R_C \subseteq S_C$.

By adding language support for statically bound local calls, which are in general useful for explicit code reuse, fewer $R$ requirements are needed since reasoning about these can be done by $S$ specifications. A static local call, say $v := C : m(\overline{e})$, which binds to the method $m$ of $C$, is meaningful in a subclass of $C$, say $C'$. We may reason about this call using $S_C$, and we may enrich $S_C$ if needed. Thus in the presence of statically bound local calls, there is less need for late-bound local calls. And for programs without late-bound local calls, there is no longer a need for $R$ requirements, we only need interface specifications and class specifications. Separation logic also offers $S$- and $R$-like requirements for the setting of sequential Java or JML style programs with non-trivial aliasing but without history specifications [7, 31, 37]. This approach is not able to handle the challenge of the Bank example.

For programs with late-bound local calls, we still have a problem with code reuse when a redefined method violates an $R$ requirement. To solve this problem, we insist that all object variables are typed by interfaces. At run-time an object variable will refer to an object of a class implementing its declared interface, when not null [35, 26]. This property, called the interface substitution principle, is guaranteed by static type checking. Thus a callee is typed by an interface rather than a class, and reasoning relies on the interface specification. We distinguish reuse of code from reuse of specifications: Interface specifications are inherited by subinterfaces, i.e., $I_F(h)$ is inherited as $I_F(h/F)$ in a subinterface of $F$. Class invariants
need not be inherited, and a subclass of a class implementing interface \( F \) may violate \( F \). We let class invariants and interface support be stated independently for each class; however, if a class implements \( F \), it implicitly implements any superinterface of \( F \).

A subclass \texttt{class} \( C_1 \) \texttt{extends} \( C_2 \) inherits all code and specifications from the class \( C_2 \) (with the usual projection of histories), thus supporting all interfaces that \( C_2 \) implements. This corresponds to (a kind of) behavioral subtyping. A subclass \texttt{class} \( C_1 \) \texttt{inherits} \( C_2 \) inherits all code, but not specifications, from \( C_2 \). This allows free code reuse, as long as one can verify the stated interfaces clauses of \( C_1 \). One may inspect the proof outlines of \( C_2 \) and see which ones are valid for \( C_1 \), and the case of lazy behavioral subtyping occurs when all pre- and postconditions used in a proof outline for a late-bound local call in \( C_2 \) are supported by any redefined version of the called methods. Moreover, there is no need to record \( R \) requirements, since subclasses no longer need to respect \( R \) requirements. Instead of pushing requirement downwards in the class hierarchy, one needs to look upwards in the class hierarchy when verifying a class, reconsidering inherited code. Thus we omit the downward inheritance of requirements and obtain more fine-grained verification control. This approach to reasoning can be called \textit{behavioral interface subtyping}. The interface substitution principle is satisfied, and our approach is able to handle the challenge of the Bank example.

**Lemma 1** Our language satisfies the interface substitution principle, and reasoning by means of behavioral interface subtyping is sound, i.e., one may reason about an object variable \( v \) declared of an interface \( F \) by means of the \( F \) invariant, assuming each class \( C \) is verified as described above, possibly involving superclasses but not any subclasses of \( C \).

**Proof outline.** We prove that our language, which includes late binding and free (type-correct) method redefinition, satisfies the interface substitution principle, and that each object referred to at run-time by a variable \( v \) declared of an interface \( F \) satisfies the \( F \) specification. Each object variable \( v \) is declared of an interface \( F \). At run-time the initial value of \( v \) is \texttt{null} or the value of an actual parameter, and its value may be changed by \texttt{new} statements, assignments, and call statements assigning the result of the call to the variable. Static type checking ensures that all these statements assign to \( v \) a value which is of type \( F \) or a subinterface of \( F \), or is \texttt{null}, and that each actual object parameter is of the interface (or a subinterface) of the corresponding formal parameter. Given an operational semantics, this can be proved more formally by means of subject reduction, as in [26] (which also deals with the complications of call labels). An object variable will therefore at run-time be \texttt{null} or refer to an object of a class with an interface being \( F \) or a subinterface. It remains to prove that such a class satisfies \( F \), and that this involves only \( C \) and superclasses.

We may assume that all classes are verified. For simplicity we assume that all pre/post-conditions are expressed through the invariant. And we assume a sound reasoning system for proving class invariants. For each class \( C \) it is required to verify that the invariant is established and maintained, and to verify the interfaces of \( C \), using the invariant. A \( C \) object may at run-time perform methods defined in \( C \) or inherited methods, called remotely, and these may make local calls to methods in \( C \) or a superclass, including static calls, which may only bind to superclass methods, statically known. Also the binding of late-bound method calls in \( C \) or superclass methods are statically known when the executing object is a \( C \) object. Thus \( C \) satisfaction of \( F \) depends only on methods in \( C \) and its superclasses, and the binding of all local calls can be resolved at verification time. It is therefore possible to prove satisfaction of \( F \) by normal static analysis, ensuring that all methods in \( C \) as well as inherited ones maintain the \( C \) invariant, and that suspension points in locally called superclass methods respect the \( C \) invariant. If needed any local calls to superclass methods can be
reverified to deal with these calls. Thus the verification depends on $C$ and its superclasses, and not on any subclass of $C$. This ensures that any $C$ object will satisfy the interfaces of $C$. Since an interface $F$ inherits all requirements of any superinterface, we have that a class satisfying $F$ also satisfies a superinterface of $F$. We may conclude that at run-time an object variable $v$ will be `null` or refer to an object of a class satisfying its declared interfaces.

**Inheritance in the Bank Example**

In the Bank example, class `BANKWAIT` extends `BANK`, while class `BANKPLUS` inherits `BANK` without claiming to respect its specifications. For class `BANKPLUS` it is easy to see that only the postcondition `return = true` of method `sub` and that of `upd` are violated. And it is easy to see that the verification of the invariant $bal = sum(h)$ in `BANK` is valid in class `BANKPLUS`. The new specification of `upd` and the added invariant conjunct $bal \geq 0$ must be verified, which is straightforward. Support of interface `BankPlus` (and thereby interface `Bank`) follows from this. Notice that with lazy behavioral subtyping, class `BANKPLUS` would not satisfy the $R_{BANK}$ requirement; thus a reasoning approach requiring inheritance of $R$ requirements would not be able to handle this kind of code reuse. We avoid verification problems in this case, as opposed to the case of previous work on lazy behavioral subtyping as well as the work on separation logic.

**Multiple Inheritance**

The notion of multiple inheritance has been an ideal in object-oriented programming due to its expressive power with respect to code reuse. However, programming with multiple inheritance can be confusing [32], and reasoning about multiple inheritance is challenging, especially if not restricted. In order to control programming with multiple inheritance and late binding, `binding healthiness` [19] ensures that a call textually occurring in a given class $C$ may only bind to a class related to $C$, either below $C$ or above $C$. To solve the diamond binding problem, i.e., to bind a method in case of several alternative definitions, we use the ordering given by the order of the inheritance list. In addition, explicit class qualification of late-bound local calls, similar to that for static binding, allows fine-grained control. Synchronization interference between multiple superclasses is avoided, due to local concurrency control. With healthiness, lazy behavioral subtyping extends to the case of multiple inheritance. When analyzing a given class $C$, added $R$ specifications concern $C$ and future subclasses of $C$, whereas $S$ specifications may be added to $C$ or some superclass of $C$. Classes are analyzed in some order consistent with the subclass order, typically the order in which they are defined. As before, $R$ specifications are not needed when restricting local calls to static ones, and in this case healthiness is guaranteed.

Reasoning with behavioral interface subtyping also extends to multiple inheritance. Healthiness is not required, since reasoning is done for the case that the executing object is of the class considered. Any subclass must be considered separately, reusing reasoning results when possible, as in the case of single inheritance. Fig. 3 gives an example where class `PINBANK` inherits both class `PIN` and `BANK`, not respecting the specifications of the latter, since interface `PerfectBank` is violated. Thus the challenge of Fig. 2 reappears.
Figure 4: Hoare style rules for non-standard constructs. Primed variables represent fresh logical variables, and fresh(v', h) expresses that v' does not occur in h. L denotes a condition on local variables. For each class one must verify that the class invariant I holds after initialization and is maintained by all methods exported through an interface. The await rules ensure that it holds upon suspension.

5 Local Reasoning

The disjointness of local histories of distinct objects allows sequential style local reasoning inside a class, while global reasoning is possible with the composition rule. In particular, the Hoare rules for basic statements are standard, including assignment, if-statements and skip, assuming expressions are side-effect free and welldefined. Standard consequence, conjunction, and adaptation rules apply. The consequence rule may be adjusted to assume local wellformedness of local histories, which implies that any caller and this are non-null. One may reason about local calls as normal, using the relevant class specifications. Thus for a local static call we have

\[ \left[ P^b \land y = \text{default} \right] s \left[ Q^b \land \exists v. \text{return} e \right] \]

where L is a predicate not referring to fields of C, h, or v, and given that method m(\(\pi\)) of C (possibly inherited) satisfies the pre- and postcondition pair \([P, Q]\). The quantifier on \(v'\) is only needed if Q refers to v. Handling of recursive calls can be done as usual. For a late-bound local call occurring in a class C reasoning can be done in the same manner, i.e., a call \(v := m(\pi)\) (as well as \(v := this.m(\pi)\)) is treated as the static call \(v := C : m(\pi)\). And the same rule applies to a remote call \(v := o.m(\pi)\) given that the pre/post pair \([P, Q]\) follows from the specification of the interface of o, and that \([P, Q]\) does not refer to the (disjoint) history of that interface. Note that all methods exported through an interface must maintain the invariant. Dot-notation is not allowed on non-exported methods; and these need not satisfy the invariant. Note that this means that suspending calls (which require the invariant) are not available for non-exported methods.
Rule history in Fig. 4 expresses that the local history is monotonic, i.e., the past can never change. Rule await guard expresses that the statement await b will not change local variables (but during suspension fields may be updated by other processes), and when it terminates the waiting condition is satisfied and the invariant has been reestablished provided it held before. The rule for simple call expresses that the partial correctness semantics of the call o.m(\(\pi\)) is equivalent to that of the assignment \(h := h \cdot (\text{this} \rightarrow o.m(\pi))\). The rule call-while expresses that the partial correctness semantics of a non-local call \(v := o.m(\pi)\{s\}\) is equivalent to that of the statements

\[
h := h \cdot (\text{this} \rightarrow o.m(\pi)); s; v' := \text{some}; h := h \cdot (\text{this} \leftarrow o.m(\pi; v')); v := v'
\]

where \(v'\) is a fresh variable representing the locally unknown result, letting \(v' := \text{some}\) denote a non-deterministic assignment. The call-while rule may be derived from this using the rule \([\forall v. Q]v := \text{some}[Q]\). The blocking call \(v := o.m(\pi)\) is equivalent to \(v := o.m(\pi)\{\text{skip}\}\). Thus we may derive the rule

\[
[\forall v'. o \neq \text{this} \land Q^v_{v',h.\text{this} \rightarrow o.m(\pi); (\text{this} \leftarrow o.m(\pi,v'))}] v := o.m(\pi) [Q]
\]

With respect to partial correctness, the suspending call await \(v := o.m(\pi)\) is equivalent to calling while suspending, i.e., \(v := o.m(\pi)\{\text{await true}\}\), and the await call rule can be derived from this. The method rule expresses that the body of a method \(T m(T \bar{x})\{T \bar{y}; s; \text{return e}\}\) can be seen as the statements

\[
h := h \cdot (\text{caller} \rightarrow \text{this}.m(\pi)); \{U \bar{y}; s; \text{return := e}\}; h := h \cdot (\text{caller} \leftarrow \text{this}.m(\pi); \text{return})
\]

(with return and caller as local variables). In the rule, \(\bar{y}\) is a list of logical variables used to handle possible name clashes between local variables \(\bar{y}\) and variables occurring in \(P\) or \(Q\), and uninitialized local variables are initialized by default values of the respective types (if \(\bar{y}\) default). The rule for object creation expresses the obvious extension of the local history and (local) freshness of the identity of the new object (and \(v'\) is needed if \(v\) occurs in \(e\)).

### Verification of the Bank Example

Consider the inherited method sub of class BANKWAIT. We need to verify the invariant and the postcondition:

\[
[I(h)] \text{Bool ok} := \text{upd}(-x); \text{return ok} [I(h) \land \text{return} = \text{true}]
\]

where \(I(h)\) is the class invariant \((\text{bal} = \text{sum}(h) \land \text{bal} \geq 0)\). Left-constructive reasoning according to the method rule gives

\[
[I(h)] \text{ok} := \text{upd}(-x) [I(h \cdot (\text{this} \leftarrow \text{this}.\text{sub}(x; \text{ok}))) \land \text{ok} = \text{true}]
\]

which by definition of \(\text{sum}\) reduces to

\[
[I(h)] \text{ok} := \text{upd}(-x) [\text{bal} = \text{sum}(h) - x \land \text{bal} \geq 0 \land \text{ok} = \text{true}]
\]

It suffices that \(\text{upd}(x)\) satisfies the specification

\[
[\text{bal} = \text{sum}(h), \text{return} = \text{true} \land \text{bal} = \text{sum}(h) + x \land \text{bal} \geq 0]
\]

which is easily verified by the rule for local calls, using the redefined \(\text{upd}\) (with \(\text{ok}\) receiving the method result). Moreover, these verification tasks could easily be automated by a tool.

A client object may call methods on a bank object \(b\). If \(b\) is declared to be of interface PerfectBank, we obtain \([\text{true}] \text{ok} := b.\text{sub}(a) [\text{ok} = \text{true}]\), and \([\text{true}] v := b.\text{bal}() [v \geq 0]\) if \(b\)
is of interface BankPlus. Reasoning with the compositional rule is required to obtain further information about the calls.

**Verification of the PinBank Example.** For class PIN it is straightforward to verify that the invariant is maintained by each method, and that it holds initially. We may also show that the postcondition given in interface Pin for close is established by the implementation of close in PIN, given the invariant as precondition. And the one for open is trivial to verify.

Consider the method sub of class PINBANK. We need to verify that the body of sub satisfies its postcondition and that it maintains the given invariant. In the first verification task we rely on the postcondition return = true of upd from BANK. The invariance proof of bal = sum(h) reduces to the one in BANK, and the conjunct u = user(h) is maintained since it is not affected by sub.

6 Discussion of Future-related Mechanisms

The notion of future has been suggested as a mechanism to increase concurrency and reduce waiting in method bodies [41, 29]. A future may be seen as a reference to a location where a method result will be stored when available. Futures may be first-order in the sense that they may be passed as parameters. Thereby information sharing is possible and one may return a future rather than waiting for the result itself. Reasoning about ABS with first-order futures is studied in [13, 14]. Futures are in particular useful when a callee depends on remote calls to other objects. However, there is a cost of using futures, both at the programming level and at the specification and reasoning level. At the programming level, an interface must decide if and where to use futures, since this affects the signatures of the interface methods. These decisions are not easy to make at an early stage, and it is difficult to change these decisions at a later stage in the programming, since they affect other classes as well. At the specification and reasoning level, one will need to talk about future identities and it is in general not trivial to make the connection between a call event and the corresponding get event with the result of the call. Thus the simple call-response paradigm is no longer syntactically reflected in the histories. Reasoning rules must deal with future identities. And as seen in [14] there is a cost with respect to compositional reasoning, related to the “get” rule for accessing a future value.

Our experience is that first-order futures are often not needed, and therefore the drawbacks mentioned are quite expensive in a reasoning perspective since futures appear in the communication events even when not used in the program. The mechanism of delegation is in several ways similar to futures. In the body of a method $m(\pi)$ one may delegate the rest of the method body to another call (possibly remote), letting for instance the statement delegate $o.n(\pi)$ mean that the current call terminates without producing a result, while delegating to the remote call $o.n(\pi)$ to send a result back to the caller of $m$. Type checking must ensure that the result type of $n$ is the same, or better (i.e., a subinterface or subtype), than that of $m$. However, interface declarations are not affected by issues related to delegation. Thus delegation may be used in code when suitable without prior planning in interfaces.

Our current setting may accommodate delegation with some adjustments of the communication events, adding more information to global events, and strengthening the notion of wellformedness to make up for the difference in local and global events. In local reasoning, events may be as before, except that a delegation call must be indexed with the current call, and in this case no return event should occur. In global reasoning, all communication events
could be indexed with the initial call, starting with a call event \((o \rightarrow o.1.m(\tau))_{(o \rightarrow o.1.m(\tau))}\) and ending with a get event \((o \leftarrow o.1.m(\tau); r)_{(o \rightarrow o.1.m(\tau))}\). Thus \((o \rightarrow o.1.m(\tau))\), may be followed by \((o \rightarrow o.1.m(\tau))_c\), and \((o \rightarrow o.1.m(\tau))_c\) may be followed by either a delegation event \((o \rightarrow o.2.n(\bar{\tau}))\), or a return event \((o \leftarrow o.1.m(\tau); r))_c\). The delegation event is like a call event and may be followed by \((o \rightarrow o.2.n(\bar{\tau}))_c\), while a return event \((o \leftarrow o.1.m(\tau); r))_{(o \rightarrow o.1.m(\tau))}\) may be followed by \((o \leftarrow o.1.m(\tau); r))_{(o \rightarrow o.1.m(\tau))}\), closing the cycle. A normal 4 event call cycle is now generalized to a call cycle of length \(4 + 2n\) where \(n\) is the number of delegations in the cycle. If the initial call is redundant (i.e., the caller, callee, method and actual input parameters are given by the main event), the index may be omitted. Thus if delegation is not used, events are written as in the setting without delegation. We conclude that the cost of delegation is less than that of the future mechanism.

A simpler approach is to simulate delegation by the statements \textbf{await} dummy := o.n(\tau); \textbf{return} dummy, where dummy is a fresh local variable, not used in the invariant \(I\) nor postcondition \(Q\). This is like a delegation except that the delegating object needs to pass on the result. However, this may be efficiently implemented. We denote these statements \textbf{delegate} o.n(\tau), whereas the statements \textbf{dummy} := o.n(\tau); \textbf{return} dummy, denoted \textbf{return} o.n(\tau), represent a blocking version of delegation. Return may also be generalized to local calls. For flexible programming, we modify our language so that the last statement in a method body is either a return or delegation statement, or a (nested) if construct where each branch ends with a return or delegation statement. For instance, method \textbf{upd} of class BANKPLUS can be written \{\textbf{if} bal + x >= 0 \textbf{then} \textbf{return} BANK; \textbf{else} \textbf{return} false \}. Reasoning rules can be derived from the definition above:

\[
delegation \quad \frac{hew (this \rightarrow o.1.n(\tau)) \land I \Rightarrow \forall return. Q^h_{h \cdot (this \rightarrow o.1.n(\tau); return)) \cdot (caller \leftarrow this. m(\tau; return))}{[I^h_{h \cdot (this \rightarrow o.1.n(\tau))}] \textbf{delegate} o.1.n(\tau) [Q]}
\]

where \(m\) is the enclosing method and \(\tau\) the formal parameters. For return-call we obtain

\[
[o \neq this \land Q^h_{h \cdot (this \rightarrow o.1.n(\tau)) \cdot (this \rightarrow o.n(\tau); return)) \cdot (caller \leftarrow this. m(\tau; return))]} \textbf{return} o.1.n(\tau) [Q]
\]

for remote calls (local calls are similar to earlier). This allows reasoning with a delegation-like construct without changing the structure of events nor the notion of wellformedness.

7 Conclusion

We have motivated and presented the main elements of a concurrency model and reasoning framework based on active, concurrent objects. From a programming point of view the model allows efficient and simple programming of distributed and multi-core systems, resulting in a high degree of parallelism, and with semantically simple synchronization mechanisms, allowing local synchronization control within each method. Reasoning about such systems has been studied in [12, 38, 18, 3, 15, 16, 13]; [27, 3] survey research outside this framework. The current work makes several improvements on this setting: In the compositional reasoning our treatment is based on interface invariants rather than directly on class invariants, simplifying the treatment of prefix closure of class invariants. Secondly, the axiomatic semantics is simplified by the statement for interleaved call introduced here. Finally, we have shown how lazy behavioral subtyping can be replaced by what we call behavioral interface subtyping, avoiding inheritance of requirements to subclasses, thereby allowing free code reuse (as long as the specified interfaces are supported by each class). As seen in the example, this makes a significant difference in practice.
The approach is modular since each class can be analyzed separately, verifying the given specifications and implements-claims, and subclasses can be added incrementally. In a subclass one may need to verify new properties of inherited or statically called methods of a superclass $C$. By separating code inheritance from specification inheritance we obtain control of code reuse. Behavioral subtyping corresponds to inheriting while respecting superclass specifications, lazy behavioral subtyping corresponds to inheriting while respecting only pre/postconditions of redefined methods, and free code reuse corresponds to inheriting without respecting superclass specification.

We have avoided the use of futures/call labels in order to make the language more high-level. This has the advantage that the connection between postconditions and history-based invariants is direct, not depending on future identities. The interleaved call statement has allowed us to present a label-free version of the language with the expressiveness of normal (i.e., pair-wise) use of call labels. The presented rules for the different call mechanisms are simpler than in earlier work on future/label-free communication [18], due to the treatment of interleaved call and guarded suspension. Reasoning is as for sequential programs, apart from side-effects on the local history. Some topics have not been considered, like typing considerations, dynamic class updates [36, 25] and constructs for object grouping, which allow clusters of concurrent objects to be seen as a single object from the outside [24].

We have not included soundness and completeness of the Hoare style reasoning system, but the basic primitives are modeled by (partly non-executable) assignments where all updates on the history are explicit. The given rules can be derived from this model.

A number of tools have been developed for the presented concurrency model, mainly under the HATS project, including compilers and a reasoning framework in KeY [1]. In order to exploit the paradigm presented here in Java, a Java library has been developed allowing Java programming with the described communication primitives and concurrency model [33].

Acknowledgments

The author has known Kaisa Sere since the late 80ies and is indebted to her for many years of discussions on Action Systems and the concurrency model of this paper, and its predecessors, especially through the yearly NWPT workshops.

The reviewers and Charlie McDowell have contributed constructively to the presentation and discussions of the paper. This work relates to the EU projects FP7-610582 Envisage: Engineering Virtualized Services (http://www.envisage-project.eu) and FP7-ICT-2013-X UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations (http://www.upscale-project.eu).
References


