A Sound and Complete Reasoning System for Asynchronous Communication with Shared Futures ☆

Crystal Chang Din and Olaf Owe

Dept. of Informatics – Univ. 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. We consider the setting of concurrent objects communicating by *asynchronous method calls*. 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 Hoare style reasoning system for distributed objects based on a general concurrency and communication model focusing on asynchronous method calls and futures. The model facilitates invariant specifications over the locally visible communication history of each object. Compositional reasoning is supported, and each object may be specified and verified independently of its environment. The presented reasoning system is proven sound and (relatively) complete with respect to the given operational semantics.

Keywords: Program reasoning, Distributed systems, Object-orientation, Compositional reasoning, Hoare Logic, Concurrent objects, Asynchronous communication, Shared futures, Operational semantics, Communication history.

---

1. Introduction

Distributed systems play an essential role in society today. For example, distributed systems form the basis for critical infrastructure in different domains such as finance, medicine, aeronautics, telephony, and Internet services. It is of great importance that such systems work properly. However, quality assurance of distributed systems is non-trivial since they depend on unpredictable factors, such as different processing speeds of independent components. It is highly challenging to test such distributed systems after deployment under different relevant conditions. These challenges motivate 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

☆This work was done in the context of the EU project 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`).

components. Thereby, it is possible to deal with systems consisting of many components.

Object orientation is the leading framework for concurrent and distributed systems, recommended by the RM-ODP [1]. 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 [2]. Concurrent objects communicating by *asynchronous method calls*, which allows the caller to continue with its own activity without blocking while waiting for the reply, combine object-orientation and distribution in a natural manner, and therefore appear as a promising paradigm for distributed systems [3]. Moreover, the notion of *futures* [4, 5, 6, 7] 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.

ABS is a high-level imperative object-oriented modeling language, based on the concurrency and synchronization model of *Creol* [8]. It supports futures and concurrent objects with an asynchronous communication model suitable for loosely coupled objects in a distributed setting. In *ABS*, each concurrent object encapsulates its own state and processor, and internal interference is avoided as at most one process is executing. The concurrent object model of *ABS* without futures supports compositionality because there is *no direct access* to the internal state variables of other objects, and a method call leads to a new process on the called object. With futures, compositionality is more challenging.

In this paper, we consider the general communication model of *ABS* focusing on the future mechanism. A compositional reasoning system for *ABS* with futures has been presented in [9] based on local communication histories. We here present a revised and simplified version of this system and show that it is sound with respect to an operational semantics which incorporates a notion of global communication history. We also show a completeness result.

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 [10, 11]. At any point in time the communication history abstractly captures the system state [12, 13]. In fact, traces are used in the semantics for full abstraction results (e.g., [14, 15]). 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 [16].

In our reasoning system, we formalize object communication by an operational semantics based on five kinds of communication events, capturing shared (first class) futures, where each event is visible to only one object. Consequently, the local histories of two different objects share no common events. For each object, a history invariant can be derived from the class invariant by hiding the local state of the object. Modularity is achieved since history invariants can be established independently for each object, without interference, and composed at need. This results in behavioral specifications of dynamic system in an open environment. Such specifications allow objects to be specified independently of their internal implementation details, such as the internal state variables. In order to derive a global specification of a system composed of several components, one may compose the specification of different components. Global specifications can then be provided by describing the observable communication history between each component and its environment.

The main contribution of this paper is the presentation of a set of Hoare rules and a proof of soundness and relative completeness with respect to a revised operational semantics including a global communication history. The operational semantics is implemented in

Maude by rewriting rules and can be exploited as an executable interpreter for the language, such that execution traces can be automatically generated while simulating programs. In earlier work [17], a similar proof system is derived from a standard sequential language by means of a syntactic encoding. However, soundness with respect to the operational semantics was not considered. A challenge of the current work is that the presence of the global history and shared futures complicate compositional reasoning and also the soundness and completeness proof. We therefore focus the current work on the core communication model with futures and consider process suspension mechanism, but ignore other aspects such as inheritance. The work is relevant for the more general setting of concurrent objects with asynchronous methods and futures, and it can easily be extended to the full *ABS* setting.

An *ABS* reasoning system is currently being implemented within the KeY framework at Technische Universität Darmstadt. The tool support from KeY for (semi-)automatic verification is valuable for verifying *ABS* programs. A publisher-subscriber example will be used here to illustrate the language and the reasoning system.

Paper overview. Section 2 introduces and explains the core language syntax, Section 3 formalizes the observable behavior in the distributed systems, Section 4, presents the operational semantics, and Section 5 defines the proof system for local reasoning within classes and finally considers object composition. A publisher-subscriber example is presented in Section 2 and the corresponding proofs are shown in Section 6. Section 7 defines and proves soundness and relative completeness for our reasoning system. Section 8 shows how to extend the language with non-blocking queries on futures. Section 9 discusses the relevance of choices made in the considered language and formalization, and briefly discusses how some other approaches may affect the reasoning system. Section 10 discusses related and future work, and Section 11 concludes the paper.

2. A Core Language with Shared Futures

We consider concurrent objects interacting through method calls. 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 on an object at a time. Object communication is *asynchronous*, as there is no explicit transfer of control between the caller and the callee. In this setting a *future* represents a placeholder for the return value of a method call. Each future has an unique identity which is *generated* when the method is invoked, and a futures may be seen as a shared entity of information accessible by any object that knows its identity. A 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 resolving 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-class futures, inspired by *ABS* [18].

$$
\begin{array}{lcll}
In & ::= & \texttt{interface}\ I\ [\texttt{extends}\ I^+]^?\{S^*\} & \text{interface declaration}\\
Cl & ::= & \texttt{class}\ C([T\ cp]^*)\ [\texttt{implements}\ I^+] & \\
 & & \{[T\ w\ [:= e]^?]^*\ [s]^?\ M^*\} & \text{class definition}\\
M & ::= & S\ B & \text{method definition}\\
S & ::= & T\ m([T\ x]^*) & \text{method signature}\\
B & ::= & \{[\texttt{var}\ [T\ x\ [:= e]^?]^*;]^?\ [s;]^?\ \texttt{put}\ e\} & \text{method blocks}\\
T & ::= & I \mid \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{String} \mid \mathsf{Void} \mid \mathsf{Fut}{<}T{>} & \text{types}\\
v & ::= & x \mid w & \text{variables (local or field)}\\
e & ::= & \texttt{null} \mid \mathsf{this} \mid v \mid cp \mid f(\overline{e}) & \text{pure expressions}\\
s & ::= & v := e \mid fr := v!m(\overline{e}) \mid v := \texttt{get}\ e \mid v := \texttt{new}\ C(\overline{e}) & \text{statements}\\
 & & \mid \texttt{skip} \mid \texttt{if}\ e\ \texttt{then}\ s\ [\texttt{else}\ s]^?\ \texttt{fi} \mid s; s &
\end{array}
$$

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, repeated at least once and optional parts, respectively, and $\overline{e}$ is a (possibly empty) expression list. Expressions e and functions f are side-effect free.

Methods are organized in classes in a standard manner. A class C takes a list of formal parameters $\overline{cp}$, and defines fields $\overline{w}$, optional initialization statements $\overline{s}$ and methods $\overline{M}$. There is read-only access to class parameters $\overline{cp}$, method parameters $\overline{x}$, and implicit variables, such as this, referring to the current object, and the implicit method parameter future, referring to the future of the call. A method definition has the form $m(\overline{x})\{\texttt{var}\ \overline{y};\ s;\ \texttt{put}\ e\}$, when ignoring type information, where $\overline{x}$ is the list of formal parameters, $\overline{y}$ is an optional list of *method-local variables*, s is a sequence of statements, and the value of e is put in the future of the call upon termination (i.e., "resolving the future").

A future variable fr is declared by $\mathsf{Fut}<T>fr$, indicating that fr may refer to futures which may 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 $v := \texttt{get}\, fr$ 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.

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 programming with futures. In Section 8 we show how process release statements, including a releasing query statement, can be added as an extension of the present work. However, we first focus on a core language for futures, with a simple semantics, avoiding specialized features such as process control.

The core language contains additionally statements for assignment, **skip**, conditionals, and sequential composition. Object variables are typed by interfaces, and we assume that call and query statements are well-typed. If x refers to an object where m is defined with input types $\overline{S}$ and return type T, the following code is well-typed when $\overline{e}$ is of type $\overline{S}$: $\mathsf{Fut}<T>fr;\ T\ v;\ fr := x!m(\overline{e});\ v := \texttt{get}\, fr$, which represents a traditional synchronous and blocking method call. This call is abbreviated by the notation

$$v := x.m(\overline{e})$$

Note that the call $v := \mathsf{this}.m(\overline{e})$ will block, and thus a construct for local calls, say $v := m(\overline{e})$,

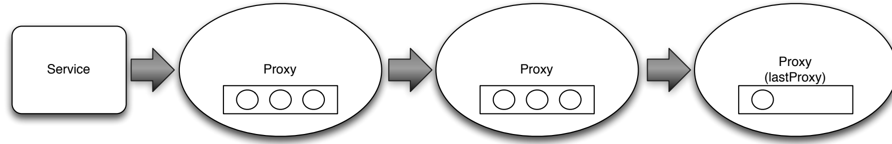

Figure 2: A Publisher-Subscriber Example with proxies handling at most three clients each.

```
1
2
3  interface ServiceI{
4      Void subscribe(ClientI cl);
5      Void produce()}
6  interface ProxyI{
7      ProxyI add(ClientI cl);
8      Void publish(Fut<News> fut)}
9  interface ClientI{
10     Void signal(News ns)}
11 interface ProducerI{
12     News detectNews()}
13
14 class Service(Int limit) implements ServiceI{
15     ProducerI prod; ProxyI proxy; ProxyI lastProxy;
16     {prod := new Producer(); proxy := new Proxy(limit,this); lastProxy := proxy; this!produce()}
17
18     Void subscribe(ClientI cl){lastProxy := lastProxy.add(cl)}
19
20     Void produce(){var Fut<News> fut; fut := prod!detectNews(); proxy!publish(fut)}}
21
22 class Proxy(Int limit, ServiceI s) implements ProxyI{
23     List<ClientI> myClients := Nil; ProxyI nextProxy;
24
25     ProxyI add(ClientI cl){
26         var ProxyI lastProxy := this;
27         if length(myClients) < limit then myClients := appendright(myClients, cl)
28         else if nextProxy = null then nextProxy := new Proxy(limit,s) fi;
29         lastProxy := nextProxy.add(cl) fi; put lastProxy}
30
31     Void publish(Fut<News> fut){
32         var News ns = None;
33         ns := get fut; myClients!signal(ns);
34         if nextProxy = null then s!produce() else nextProxy!publish(fut) fi}}
```

Figure 3: Implementation of the Publisher-Subscriber example. Notice that *proxy!publish(...)* is a unicast and *myClients!signal(...)* is a multicast. We have used *ABS* syntax for lists. See AppendixA for a full implementation including data type definitions (including *News*) and implementation of the *Producer* and *Client* classes.

would be useful. The core language ignores language features that are orthogonal to shared futures, such as inheritance. We refer to [19] for a treatment of this.

2.1. Publisher-Subscriber Example

To illustrate the language, and in particular the usage of shared futures, we consider an implementation of a version of the publisher-subscriber example in which clients may subscribe to a service, while the service object is responsible for generating news and distributing each news update to the subscribing clients. To avoid bottlenecks when publishing events, the service delegates publishing to a chain of *proxy* objects, where each proxy object handles a bounded number of clients, as illustrated in Fig. 2. The implementation of the classes *Service* and *Proxy* can be found in Fig. 3. We use this implementation later in the paper to illustrate our reasoning techniques: We will define class invariants and illustrate the proof system by verification of these invariants.

The example takes advantage of the future concept by letting the service object delegate publishing of news updates to the proxies without waiting for the result of the news update. This is done by the sequence $\mathit{fut} := \mathit{prod}!\mathit{detectNews}(); \mathit{proxy}!\mathit{publish}(\mathit{fut})$. Thus the service object is not blocking by waiting for news updates. Furthermore, the calls on *add* are blocking; however, this is harmless since the implementation of *add* may not deadlock and terminates efficiently. The other calls in the example are not blocking nor involving shared futures.

For convenience, we introduce the following notation for uni- and multicast to be used in the example. A call $\mathit{fr} := x!m(\overline{e})$ is abbreviated to $x!m(\overline{e})$ when the future (and result) is not needed. For this kind of simple *message passing* we also allow x to be a collection of objects (a list in the example), with the effect that the invocation is sent to all objects in the collection, thereby allowing us to program *multi-casting* (without explicit futures). A multicast to an empty collection has no effect. For simplicity, we here omit the (redundant) return statement of Void methods.

3. Observable Behavior

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 [10, 11].

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. 4. 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* $\langle \hookrightarrow\!\!\!\!\!\twoheadrightarrow o', u, m, \overline{e}\rangle$ is generated by o' once the method starts execution. When the method terminates, the object o' generates the *future event* $\langle \leftarrow\ o', u, e\rangle$. 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'', that object may also fetch the future value, reflected by the event $\langle o''\ \twoheadleftarrow, u, e\rangle$, generated by o''. The

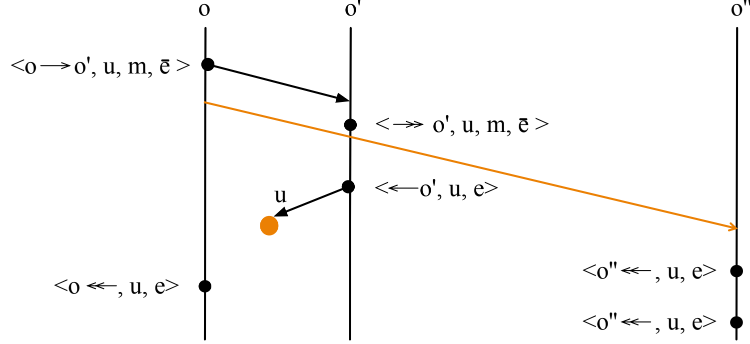

Figure 4: A method call cycle: object o calls a method m on object o' with *future* u. The events on the left-hand side are visible to o, those in the middle are visible to o', and the ones on the right-hand side are visible to o''. There is an arbitrary delay between message receiving and reaction. The message sending marked in orange from o to o'' represents that the *future* u is passed from o to o''.

object creation event $\langle o \uparrow o', C, \overline{e} \rangle$ represents object creation, and is generated by o when o creates a fresh object o'.

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

- Invocation events $\langle caller \rightarrow callee, future, method, args \rangle$, *generated by caller.*
- Invocation reaction events $\langle \twoheadrightarrow callee, future, method, args \rangle$, *generated by callee.*
- Future events $\langle \leftarrow\ callee, future, result \rangle$, *generated by callee.*
- Fetching events $\langle receiver\ \twoheadleftarrow, future, result \rangle$, *generated by receiver.*
- Object creation events $\langle caller \uparrow callee, class, args \rangle$, *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., $\langle \leftarrow\ o', u, e \rangle.result = e$.

For a method invocation with future u, the ordering of events depicted in Fig. 4 is described by the following regular expression (using $\cdot$ for sequential composition of events)

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

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 $_ \cdot _ : \mathsf{Seq}[T] \times T \to \mathsf{Seq}[T]$ (where "_" indicates an argument position). The choice of constructors gives rise to generate inductive function definitions, in the style of [13]. Projection, $_/_ : \mathsf{Seq}[T] \times \mathsf{Set}[T] \to \mathsf{Seq}[T]$ is defined inductively by $\varepsilon/s \triangleq \varepsilon$ and $(a \cdot x)/s \triangleq \mathbf{if}\ x \in s\ \mathbf{then}\ (a/s) \cdot x\ \mathbf{else}\ a/s\ \mathbf{fi}$, for $a : \mathsf{Seq}[T]$, $x : T$, and $s : \mathsf{Set}[T]$, restricting a to the elements in s. For sequences a and b, let $a\ \mathbf{ew}\ x$ denote that x is the last element of a, and $a \leqslant b$ denote that a is a prefix (initial sequence) of b, while $a \sqsubseteq b$ denotes that a is a subsequence of b (not necessarily a tight one). Let $[x_1, x_2, \ldots, x_i]$ denote the sequence of $x_1, x_2, \ldots, x_i$ for $i > 0$. For instance, $[x_1, x_2] \leqslant [x_1, x_2, x_3, x_4, x_5]$ and $[x_2, x_4] \sqsubseteq [x_1, x_2, x_3, x_4, x_5]$. Functions for event decomposition are lifted to sequences in the standard way, ignoring events for which the decomposition is not defined, e.g., $_.\mathit{result} : \mathsf{Seq}[\mathsf{Ev}] \to \mathsf{Seq}[\mathsf{Data}]$. 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

We define the operational semantics by a transition system, presented in the style of SOS. A system state g is here captured as a mapping. In our case a system state (configuration) consists of objects, messages, and futures, each identified by unique identifiers. A mapping is seen as a set of bindings from identifiers to units: the bindings $id \mapsto \mathbf{ob}(state, code)$, $id \mapsto \mathbf{msg}(callee, method, args)$, and $id \mapsto \mathbf{fut}(value)$, represent bindings of an object identifier to an object, a future identifier to an invocation message and a future identifier to a future containing a value, respectively. Here *code* denotes a sequence of statements followed by a **put** statement, or is *empty*. The special constant null may not bind to an object. A *substate* is given by the corresponding submapping. The local state of an object is given by two mappings, one for attributes a and one for local variables l, mapping variable names to values. We assume a given interpretation of data types and associated functions.

Notation. We use the following mapping notation. A mapping is seen as a set of bindings written $[z_i \mapsto value_i]$ for a set of disjoint identifiers z_i, the domain. Map look-up is written $M[z]$ where M is a mapping and z an identifier. The notation is lifted to expressions, letting $M[e]$ mean the expression e evaluated in the state given by M. Map composition is written $M + M'$ where bindings in M' override those in M for the same identifier. A map update, written $M[z \mapsto d]$, is the map M updated by binding z to the value d, i.e., $M[z \mapsto d]$ is the same as $M + [z \mapsto d]$. For an expression e, the notation $M[z := e]$ abbreviates $M[z \mapsto M[e]]$.

The state of an object is given by a twin mapping, written $(a|l)$, where a is the state of the field variables (including this, $\mathsf{nextFut}$, $\mathsf{nextObj}$, class parameters $\overline{cp}$, and the local history $\mathcal{H}$) and l is the state of the parameters and local variables (including the implicit parameter future). Look-up $(a|l)[z]$ is simply given by $(a+l)[z]$. The notation $(a|l)[v := e]$ abbreviates **if** v *in* l **then** $(a \mid l[v \mapsto (a|l)[e]])$ **else** $(a[v \mapsto (a|l)[e]] \mid l)$, where *in* is used for testing domain membership. We extract the local state and code of an object o from the global state g by $g[o].State$ and $g[o].Code$, respectively.

Definition 4 (Configuration Mappings). *A configuration of type Config is a mapping from object identities to objects of the form* $\mathbf{ob}(\delta, \overline{s})$*, from future identities to values of the form*

$$\mathbf{msg}(o, m, \overline{d}) \text{ or } \mathbf{fut}(d)$$

and possibly from the global history identifier H to a sequence of events h. The state δ of an object has the form of a twin mapping $(a|l)$. Let OB be the type of object bindings $o \mapsto \mathbf{ob}((a|l), \overline{s})$ such that $o \in Oid$, $o \neq null$, and $\mathsf{this} \mapsto o \in a$, FUT the type of future bindings $u \mapsto \mathbf{fut}(d)$ for $u \in Fid$, MSG the type of message bindings $u \mapsto \mathbf{msg}(o, m, \overline{d})$ for $u \in Fid$, and HIST the type of history bindings $H \mapsto h$ for $h \in Seq[Ev]$.

Thus in a global state we talk about objects, messages, futures, and possibly a representation of the global history, each with unique identities. An object identity o is mapped to the corresponding object, given as a pair of the state δ and the code $\overline{s}$ to be executed, written $\mathbf{ob}(\delta, \overline{s})$. A future identity u is mapped to a message containing the callee, the name of the called method and the actual parameters, written $\mathbf{msg}(o, m, \overline{d})$, or to the contained value, written $\mathbf{fut}(d)$. A class C could be bound to the attributes of the class and the set of methods, say $[C \mapsto \mathbf{class}(att, ms)]$. The method definitions in a class is of the form, $(m, \overline{p}, l, \overline{s})$ where m is the method name, $\overline{p}$ is the list of formal parameters, l contains the local variables (including default values), and $\overline{s}$ is the code. However, as classes here represent static information they are ignored in the operational semantics. The code of an object is simply a list of statements to be executed.

Generation of fresh object and future identifiers is modeled by the initial algebra given by the constructors:

- $initO$ taking an object identity (the generating object) and returning an (initial) object identity,
- $initF$ taking an object identity (the generating object) and returning an (initial) future identity,
- $next_{Fid}$ taking a future identity (say the last generated one) and returning a future identity,
- $next_{Oid}$ taking an object identity (say the last generated one) and returning an object identity.

When no ambiguity arises, we omit the index on the *next* function. A generator term, say $next(next(next(initF(o))))$, represents a unique future identity. Equality over generator terms is given by the syntactic equality, thus local uniqueness implies global uniqueness

skip: $o \mapsto \mathbf{ob}(\delta, \texttt{skip}; \overline{s})$
$$\xrightarrow{empty} o \mapsto \mathbf{ob}(\delta, \overline{s})$$

assign : $o \mapsto \mathbf{ob}(\delta, v := e; \overline{s})$
$$\xrightarrow{empty} o \mapsto \mathbf{ob}(\delta[v := e], \overline{s})$$

call : $o \mapsto \mathbf{ob}(\delta, fr := v!m(\overline{e}); \overline{s})$
$$\xrightarrow{\langle o \to \delta[v], \delta[\mathsf{nextFut}], m, \delta[\overline{e}] \rangle} o \mapsto \mathbf{ob}(\delta[fr := \mathsf{nextFut}, \mathsf{nextFut} := next(\mathsf{nextFut})], \overline{s})$$
$$\delta[\mathsf{nextFut}] \mapsto \mathbf{msg}(\delta[v], m, \delta[\overline{e}])$$

start : $u \mapsto \mathbf{msg}(o, m, \overline{d})$
$$o \mapsto \mathbf{ob}((a|l'), empty)$$
$$\xrightarrow{\langle \twoheadrightarrow o, u, m, \overline{d} \rangle} o \mapsto \mathbf{ob}((a|(l[\overline{p} \mapsto \overline{d}, \mathsf{future} \mapsto u])), \overline{s})$$
$\texttt{where}$ m is statically bound to $(m, \overline{p}, l, \overline{s})$

return : $o \mapsto \mathbf{ob}(\delta, \texttt{put}\ e)$
$$\xrightarrow{\langle \leftarrow o, \delta[\mathsf{future}], \delta[e] \rangle} o \mapsto \mathbf{ob}(\delta, empty)$$
$$\delta[\mathsf{future}] \mapsto \mathbf{fut}(\delta[e])$$

query : $u \mapsto \mathbf{fut}(d)$
$$o \mapsto \mathbf{ob}(\delta, v := \texttt{get}\ e; \overline{s})$$
$$\xrightarrow{\langle o \twoheadleftarrow, u, d \rangle} u \mapsto \mathbf{fut}(d)$$
$$o \mapsto \mathbf{ob}(\delta[v := d], \overline{s})$$
$\texttt{if}$ $\delta[e] = u$

new : $o \mapsto \mathbf{ob}(\delta, v := \texttt{new}\ C(\overline{e}); \overline{s})$
$$\xrightarrow{\langle o \uparrow \delta[\mathsf{nextObj}], C, \delta[\overline{e}] \rangle} o \mapsto \mathbf{ob}(\delta[v := \mathsf{nextObj}, \mathsf{nextObj} := next(\mathsf{nextObj})], \overline{s})$$
$$\delta[\mathsf{nextObj}] \mapsto \mathbf{ob}(\delta_{init}, init)$$

Figure 5: Operational rules reflecting small-step semantics. Variables are denoted by single characters (the uniform naming convention is left implicit). An object state δ has the form $(a|l)$. Method names are assumed to be unique for each class, and indexed with the class name during type analysis. Thus in the operational semantics we may assume that method names are unique in the system. Consequently, method binding can be done at static time. In Rule start, we assume that m is bound to a method with local state l (including default values), parameters $\overline{p}$, and code $\overline{s}$. Note that parameters and the implicit parameter future, which are read-only, are added to the local state in Rule start. In Rule new, δ_{init} denotes the initial state (including the binding this $\mapsto \delta[\mathsf{nextObj}]$, $\overline{cp} \mapsto \delta[\overline{e}]$), and default/initial values for the fields; and *init* denotes the initialization statements of class C.

since the generating object is encoded in the terms. The operational semantics uses an attribute nextFut, initialized to $initF(\mathsf{this})$, such that a fresh future identity is generated by $next(\mathsf{nextFut})$. Similarly, the attribute nextObj is initialized to $initO(\mathsf{this})$, such that a fresh object identity is generated by $next(\mathsf{nextObj})$.

4.1. Operational Rules

For our purpose, a configuration is a set of units such as concurrent objects, messages, futures, and possibly a representation of the global history. The units are uniquely identified and the configuration is formalized as a mapping. We use blank-space as the configuration constructor, allowing associativity, commutativity, identity (ACI) pattern matching. We later extend system configurations with an explicit representation of the global history ($H \mapsto h$). The history is included to define the interleaving semantics upon which we derive our history-based reasoning formalism.

For disjoint *substates* g_1 and g_2 (i.e. mappings with disjoint domains) we let $g_1||g_2$ denote the composition, and let $||$ be an ACI operator. Since we deal with distributed systems communicating asynchronously, g_1 will involve exactly one object, o, plus possibly messages and futures. The context rule

$$\frac{g_1 \xrightarrow{\alpha} g_1'}{g_1||g_2 \xrightarrow{\alpha} g_1'||g_2}$$

allows us to derive system transitions for composed systems, and the rule for sequential composition allows us to deal with sequences of statements inside each object.

The operational rules are summarized in Fig. 5. Objects are concurrent in the sense that their executions are interleaved, and in each object statements are executed sequentially. Method invocation is captured by the rule call. The generated future identity is locally unique, and also globally unique since the identity is given by a generator term embedding the parent object. The future identity generated by this rule is first bound to an invocation message, which is to be consumed by rule start. And a future unit is generated upon method completion reusing the same future identifier. When there is no active code in an object, denoted *empty*, a method call is selected for execution by rule start. The invocation message is removed from the configuration by this rule, and the future identity of the call is assigned to the implicit parameter future. Method execution is completed by rule return, and a future value is fetched by rule query. 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 objects to fetch the value of the same future. Object creation is captured by the rule new. The generated object identity is locally unique, and also globally unique since the identity is given by a generator term embedding the parent object. The object identity generated by this rule is then bound to the generated object. The given language fragment may be extended with constructs for inter object reentrance, process control and suspension, e.g., by using the *ABS* approach of [17].

4.2. Augmenting the Operational Semantics with a History

We have above formulated an operational semantics, representing interleaving semantics, by transitions of the form $g_1 \xrightarrow{\alpha} g_2$, which expresses a transition from the system (sub)state g_1 to the system (sub)state g_2 labeled by the event α (possibly empty). The set of sequences of events for all possible executions corresponds to the trace set.

The given operational semantics does not explicitly include a history. The history of a state is implicitly given as the sequence of events that has occurred in the execution leading to this state, initially being empty. We may include the history H explicitly by transforming each rule $g_1 \xrightarrow{\alpha} g_2$ to

$$g_1 + [H \mapsto h] \longrightarrow g_2 + [H \mapsto h \cdot \alpha]$$

$$\begin{array}{ll}
\textsf{call}: & \\
& o \mapsto \mathbf{ob}(\delta, fr := v!m(\overline{e}); \overline{s}), \\
& H \mapsto h \\
\longrightarrow & o \mapsto \mathbf{ob}(\delta[fr := u], \overline{s}) \\
& u \mapsto \mathbf{msg}(\delta[v], m, \delta[\overline{e}]) \\
& H \mapsto h \cdot \langle o \rightarrow \delta[v], u, m, \delta[\overline{e}] \rangle \\
\\
\mathtt{if}\ u \notin id(h) &
\end{array}
\qquad
\begin{array}{ll}
\textsf{new}: & \\
& o \mapsto \mathbf{ob}(\delta, v := \mathbf{new}\ C(\overline{e}); \overline{s}) \\
& H \mapsto h \\
\longrightarrow & o \mapsto \mathbf{ob}(\delta[v := o'], \overline{s}) \\
& o' \mapsto \mathbf{ob}(\delta_{init}, init) \\
& H \mapsto h \cdot \langle o \uparrow o', C, \delta[\overline{e}] \rangle \\
\\
\mathtt{if}\ o' \notin id(h) &
\end{array}$$

Figure 6: Abstract rule for call and object generation. As earlier δ_{init} and *init* denote the initial state (including the binding this $\mapsto o'$ and binding of the actual class parameters $\delta[\overline{e}]$) and the initialization statements of class C, respectively.

where h is (the value of) the history of the prestate and $h \cdot \alpha$ the history of the poststate, letting $h \cdot \alpha$ denote h when α is empty. In this way the special identifier H maps to the current history in a global state. Note that the event α may be omitted from the transition symbol in this version of the semantics, since it is redundant. Since we deal with predicates referring to the history, we will below use this history-explicit semantics. Furthermore, we have implemented the history-explicit semantics in Maude, which then allows run-time testing of properties over histories.

Assignments to the history is not possible with the given operation rule for assignments. It would require a specialized assignment rule. However, a program may not update the history by explicit assignments, since the history is hidden from the programmer.

4.2.1. An Abstract Semantics

The operational semantics above is executable and in particular includes an algorithm for generation of future identities and object identities. Local uniqueness will here imply global uniqueness. This is due to the rules call and new. By redefining these two rules we may give a more abstract semantics, based on non-determinism in generation of fresh identities. The abstract semantics uses history information, and we here therefore consider rules with explicit representation of the history. The abstract semantics is given in Fig. 6, in which the function *id* is overloaded, i.e., either $\mathsf{Seq[Ev]} \rightarrow \mathsf{Set[Id]}$ or $\mathsf{List[Data]} \rightarrow \mathsf{Set[Id]}$, where Id is the supertype of Oid and Fid. Namely, *id* extracts all the object/future identities occurring in a history and in the expression list $\overline{e}$, as follows:

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

where $\gamma : \mathsf{Ev}$. Remark that null is always included in $id(h)$. For a global history h, the projection $id(h)/\mathsf{Fid}$ returns all future identities in h, and $id(h/o)/\mathsf{Fid}$ returns the futures generated by o or received as parameters.

Note that the special variables nextFut and nextObj are not needed, nor are the semantical functions initF/initO and next for generating identities. Global uniqueness of object and future identities is here an explicit condition. Clearly the old version of the call and new rules is a specialization of the abstract semantics. Thus a history obtainable with the executable semantics is also a possible history of the abstract semantics.

4.3. Semantic Properties

We provide a notion of global and local wellformedness for histories corresponding to the abstract operational semantics, where the constructive approach to making fresh identities is abstracted away. (For soundness we could also use a version of wellformedness corresponding to the executable operational semantics.)

Definition 5. (Globally wellformed histories) *Let h : $\mathsf{Seq}[\mathsf{Ev}]$ be a non-empty history of a global object system S. The wellformedness predicate $wf : \mathsf{Seq}[\mathsf{Ev}] \to \mathsf{Bool}$ is defined by:*

$$
\begin{array}{lll}
wf(\varepsilon) & \triangleq & true \\
wf(\langle o \uparrow o, C, \overline{e} \rangle) & \triangleq & o \neq \mathsf{null} \wedge id(\overline{e}) = \emptyset \\
wf(h \cdot \langle o \rightarrow o', u, m, \overline{e} \rangle) & \triangleq & wf(h) \wedge o \in oid(new_{ob}(h)) \wedge (\{o'\} \cup id(\overline{e})) \subseteq id(h/o) \wedge u \notin id(h) \\
wf(h \cdot \langle \twoheadrightarrow o, u, m, \overline{e} \rangle) & \triangleq & wf(h) \wedge o \in oid(new_{ob}(h)) \wedge \ h/u \ \mathbf{ew} \ \langle _ \rightarrow o, u, m, \overline{e} \rangle \\
wf(h \cdot \langle \leftarrow \ o, u, e \rangle) & \triangleq & wf(h) \wedge o \in oid(new_{ob}(h)) \wedge id(e) \subseteq id(h/o) \wedge h/u \ \mathbf{ew} \langle \twoheadrightarrow o, u, _, _ \rangle \\
wf(h \cdot \langle o \ \twoheadleftarrow, u, e \rangle) & \triangleq & wf(h) \wedge o \in oid(new_{ob}(h)) \wedge u \in id(h/o) \wedge (h/u).result \ \mathbf{ew} \ e \\
wf(h \cdot \langle o \uparrow o', C, \overline{e} \rangle) & \triangleq & wf(h) \wedge o \in oid(new_{ob}(h)) \wedge id(\overline{e}) \subseteq id(h/o) \wedge o' \notin id(h)
\end{array}
$$

where non-interesting arguments are identified by _ in projections, and h/u abbreviates the projection of the history h to the future u, i.e. the subsequence of events γ such that $\gamma.future = u$. Similarly, $h.result$ is the sequence of result values extracted from the subsequence of h of events which has a result, i.e., future and fetching events. As defined below, $new_{ob}(h/o)$ extracts the objects *created by* o.

Wellformedness expresses that generated object and future identities are fresh ($\notin id(h)$ clauses), and otherwise no locally new identities are created (subset of $id(h/o)$ clauses); that active objects have been generated ($\in oid(new_{ob}(h)$ clauses); and that the ordering of method call cycles given in Figure 4 is respected (**ew** clauses). Note that the first event in a global history must be an object generation event, representing an externally generated initial object. The initial object has no accessible creator, and we use the convention that it is created by itself. The function new_{ob} : $\mathsf{Seq}[\mathsf{Ev}] \to \mathsf{Set}[\mathsf{Obj} \times \mathsf{Cls} \times \mathsf{List}[\mathsf{Data}]]$ returns the set of created objects (each given by its object identity, associated class and class parameters) in a history:

$$
\begin{array}{lll}
new_{ob}(\varepsilon) & \triangleq & \emptyset \\
new_{ob}(h \cdot \langle o \uparrow o', C, \overline{e} \rangle) & \triangleq & new_{ob}(h) \cup \{o' : C(\overline{e})\} \\
new_{ob}(h \cdot \boldsymbol{others}) & \triangleq & new_{ob}(h)
\end{array}
$$

where ***others*** matches all other events. The function oid : $\mathsf{Set}[\mathsf{Obj} \times \mathsf{Cls} \times \mathsf{List}[\mathsf{Data}]] \to \mathsf{Set}[\mathsf{Obj}]$ extracts object identities o from a set of elements of the form $\{o : C(\overline{e})\}$, like from the output of function new_{ob}.

For the core language considered, method bodies are executed sequentially, and it is possible to strengthen the notion of wellformedness to reflect this, i.e., ensuring

$$0 \leq \#(h/o/\{\twoheadrightarrow\}) - \#(h/o/\{\leftarrow\}) \leq 1$$

where # denotes sequence length. However, this would exclude addition of mechanisms for process release or non-blocking query of futures. Since we will consider an addition such statements in section 8, and since this property can be expressed by a local invariant, we do not let our notion of wellformedness reflect sequentially ordered method executions.

It follows directly from Def. 5 that a wellformed global history is monotone in the sense

$$h \leq h' \Rightarrow \mathit{wf}(h') \Rightarrow \mathit{wf}(h)$$

and that it satisfies the communication order pictured in Fig. 4, i.e.,

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

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 abstract operational semantics, is wellformed, $\mathit{wf}(h)$.*

This lemma can be proved by induction over the number of rule applications, proving the inductive property

$$\begin{array}{ll} & \mathit{wf}(g[H]) \\ \wedge & \mathit{Futures}(g[H]) = g/(\mathsf{MSG} \cup \mathsf{FUT}) \\ \wedge & \forall o \,.\, \mathit{id}(g[o].\mathit{State}) \subseteq \mathit{id}(g[H]/o) \end{array}$$

for any reachable configuration g, where $g/(\mathsf{MSG} \cup \mathsf{FUT})$ is the submapping of g consisting of bindings to messages and futures, and where $\mathit{Futures}(h)$ is the sequence of bindings to messages and futures calculated from the history. The second conjunct is needed to prove the event ordering information (given by **ew** of the wellformedness predicate). The third conjunct says that future/object identities found in the state of a given object have been observed in the local history. This is needed to prove that future identities appearing as *future* in the *future/fetching events*, object identities appearing as *callee* in the *invocation events*, and future/object identities appearing as *args* or *result* in the *invocation/future/new events* have been observed in the local history, given that our underlying programming language does not offer any functions producing (new) future or object identities. Note that freshness of identities and non-nullness of the objects generating the events follow by the definition of configuration.

Definition 6. (Locally wellformed histories) *Local wellformedness of a local history h for an object o, denoted $\mathit{wf}_o(h)$, is defined by*

$$\mathit{wf}_o(h) \triangleq \exists h' \,.\, \mathit{wf}(h') \wedge h = h'/o$$

Here h ranges over local histories and h' over global histories.

It follows that global wellformedness implies local wellformedness of the local history of an object o, i.e.,

$$\mathit{wf}(h) \Rightarrow \mathit{wf}_o(h/o)$$

Thus local wellformedness may be assumed in a given class. Moreover, it follows that local wellformedness reduces to

$$\begin{array}{lcl}
\mathit{wf}_o(\varepsilon) & \triangleq & \mathit{true} \\
\mathit{wf}_o(h \cdot \langle o \rightarrow o', u, m, \overline{e}\rangle) & \triangleq & \mathit{wf}_o(h) \wedge o \neq \mathsf{null} \wedge (\{o'\} \cup \mathit{id}(\overline{e})) \subseteq \mathit{id}(h) \wedge u \notin \mathit{id}(h) \\
\mathit{wf}_o(h \cdot \langle \twoheadrightarrow o, u, m, \overline{e}\rangle) & \triangleq & \mathit{wf}_o(h) \wedge o \neq \mathsf{null} \wedge h/u \ \leq\ \langle o \rightarrow o, u, m, \overline{e}\rangle \\
\mathit{wf}_o(h \cdot \langle \leftarrow\ o, u, e\rangle) & \triangleq & \mathit{wf}_o(h) \wedge o \neq \mathsf{null} \wedge \mathit{id}(e) \subseteq \mathit{id}(h) \wedge h/u\ \mathbf{ew}\ \langle \twoheadrightarrow o, u, _, _\rangle \\
\mathit{wf}_o(h \cdot \langle o\ \twoheadleftarrow, u, e\rangle) & \triangleq & \mathit{wf}_o(h) \wedge o \neq \mathsf{null} \wedge u \in \mathit{id}(h) \wedge (h/u \leq [\langle o \rightarrow _, u, _, _\rangle] \vee \\
 & & (h/u).\mathit{result}\ \mathbf{ew}\ e) \wedge h/u/\{\rightarrow o, \twoheadrightarrow, \leftarrow\} \neq \varepsilon \Rightarrow \langle \leftarrow\ o, u, e\rangle \in h \\
\mathit{wf}_o(h \cdot \langle o \uparrow o', C, \overline{e}\rangle) & \triangleq & \mathit{wf}_o(h) \wedge o \neq \mathsf{null} \wedge \mathit{id}(\overline{e}) \subseteq \mathit{id}(h) \wedge o' \notin \mathit{id}(h)
\end{array}$$

where h ranges over histories local to o. The clause $h/u/\{\rightarrow o, \twoheadrightarrow, \leftarrow\} \neq \varepsilon$ expresses that the object o is the callee of the call with future u. Note that the first event may or may not be a creation event, possibly an external creation represented by a self creation.

5. Program Verification

5.1. Local Reasoning

Local assertions express conditions on a state of a given object and the local history $\mathcal{H}$. Thus in a class C assertions may refer to the fields of C, as well as the class parameters cp, and this, which are constant. Inside a method the assertions may refer to the formal parameters (including future) and local variables of that method. Assertions may not refer to the run-time variables $\mathsf{nextFut}$ and $\mathsf{nextObj}$ used in the executable operational rules, nor the corresponding semantical functions for generating fresh names (*initF*, *initO* and *next*). For convenience, we let WF abbreviate $wf_{\mathit{this}}(\mathcal{H})$ since this is the only wellformedness predicate one may talk about inside a given class.

The reasoning rules for the core language are defined in Fig. 7. The triple $\{P\}\,\overline{s}\,\{Q\}$ expresses that if P holds prior to execution of the statement list $\overline{s}$ then Q holds after execution, provided it terminates. We write $\vdash \{P\}\,\overline{s}\,\{Q\}$ if $\{P\}\,\overline{s}\,\{Q\}$ is derivable by the rules. If the statement list $\overline{s}$ is empty we write $\vdash \{P\}\ \{Q\}$. Notice that the Rule imp together with sequential composition comp allows us to conclude $\{P\}\,\overline{s}\,\{Q\}$ from $\{P'\}\,\overline{s}\,\{Q'\}$ when $P \wedge WF \Rightarrow P'$ and $Q' \wedge WF \Rightarrow Q$. Standard rules for if-statements and adaptation are not included here. An adaptation rule would allow reuse of verification of triples, for instance when strengthening an invariant.

The rules for comp, skip, and assign, are standard. The imp rule is specialized to empty statement expressing that executions give wellformed states. The other rules deal with futures or object generation, and involve effects on the local history. The effects of a method call cycle is reflected by the rules call, method, return, and query, each introducing a call cycle event, an *invocation*, *invocation reaction*, *future*, and *fetching* event, respectively. Rule new extends the history of the creating object with an *object creation* event. The universal quantifiers in the precondition of call, query, and new reflect non-determinism of the introduced logical variables in the local reasoning. The use of $\overline{y}'$ in rule body reflects that $\overline{y}$ in P and Q does not refer to the local variables, $\overline{y}$.

As an example we consider reasoning about a blocking self call, i.e., $v := \mathsf{this}.m(\overline{e})$, which is an abbreviation of $\mathsf{Fut}{<}T{>}\ fr;\ fr := \mathsf{this}!m(\overline{e});\ v := \mathbf{get}\, fr$. We should be able to verify

$$\{true\}\ v := \mathsf{this}.m(\overline{e})\ \{false\} \tag{1}$$

since this call blocks and will never terminate, as there is no reentrance in the core language. By the imp rule it suffices to prove

$$\{true\}\, fr := \mathsf{this}!m(\overline{e});\ v := \mathbf{get}\, fr\, \{\, WF \Rightarrow false\}$$

By the rule for query and call, this reduces to proving the precondition

$$\forall fr, v\ .\ WF^{\mathcal{H}}_{\mathcal{H}\cdot\langle \mathsf{this} \rightarrow \mathsf{this}, fr, m, \overline{e}\rangle \cdot \langle \mathsf{this} \twoheadleftarrow, fr, v\rangle} \Rightarrow false$$

By definition of local wellformedness this can be reduced to $false \Rightarrow false$, which is $true$, since $\mathcal{H}/fr/\{\rightarrow \mathsf{this}, \twoheadrightarrow, \leftarrow\} \neq \varepsilon$ but $\langle \leftarrow\ \mathsf{this}, fr, e\rangle \in \mathcal{H}$ does not hold. Thus we have verified the triple (1). This example demonstrates the strength of the reasoning system, but it also

imp $$\frac{P \wedge WF \Rightarrow Q}{\{P\}\ \{Q\}}$$

comp $$\frac{\{P\}\ \overline{s_1}\ \{R\} \quad \{R\}\ \overline{s_2}\ \{Q\}}{\{P\}\ \overline{s_1}; \overline{s_2}\ \{Q\}}$$

skip $$\{Q\}\ \mathbf{skip}\ \{Q\}$$

assign $$\{Q^v_e\}\ v := e\ \{Q\}$$

call $$\{\forall fr' \,.\, Q^{fr,\mathcal{H}}_{fr',\mathcal{H}\cdot\langle \mathsf{this} \to o, fr', m, \overline{e}\rangle}\}\ fr := o!m(\overline{e})\ \{Q\}$$

method $$\frac{\{P^{\overline{y}}_{\overline{y}'} \wedge \overline{y} = \mathit{default}\}\ \overline{s}\ \{Q^{\overline{y}}_{\overline{y}'}\}}{\{P^{\mathcal{H}}_{\mathcal{H}\cdot\langle \twoheadrightarrow \mathsf{this}, \mathsf{future}, m, \overline{x}\rangle}\}\ (m(\overline{x})\ \{\mathbf{var}\ \overline{y}; \overline{s}\})\ \{Q\}}$$

return $$\{Q^{\mathcal{H}}_{\mathcal{H}\cdot\langle \leftarrow \mathsf{this}, \mathsf{future}, e\rangle}\}\ \mathbf{put}\ e\ \{Q\}$$

query $$\{\forall v' \,.\, Q^{v,\mathcal{H}}_{v',\mathcal{H}\cdot\langle \mathsf{this} \twoheadleftarrow, e, v'\rangle}\}\ v := \mathbf{get}\ e\ \{Q\}$$

new $$\{\forall v' \,.\, Q^{v,\mathcal{H}}_{v',\mathcal{H}\cdot\langle \mathsf{this} \uparrow v', C, \overline{e}\rangle}\}\ v := \mathbf{new}\ C(\overline{e})\ \{Q\}$$

Figure 7: Hoare style rules for the core language. Primed variables represent fresh logical variables, and WF is an abbreviation for $wf_{\mathsf{this}}(\mathcal{H})$.

indicates a weakness of the core language. In order to better support self calls, one could consider reentrance or release mechanisms. In section 8 we consider the latter and extend the language with constructs for non-blocking and terminating self calls.

Note that one might split the method rule into two rules, block and method', as follows:

block $$\frac{\{P^{\overline{y}}_{\overline{y}'} \wedge \overline{y} = \mathit{default}\}\ \overline{s}\ \{Q^{\overline{y}}_{\overline{y}'}\}}{\{P\}\ \{\mathbf{var}\ \overline{y}; \overline{s}\}\ \{Q\}}$$

method' $$\frac{\{P\}\ \mathit{body}\ \{Q\}}{\{P^{\mathcal{H}}_{\mathcal{H}\cdot\langle \twoheadrightarrow \mathsf{this}, \mathsf{future}, m, \overline{x}\rangle}\}\ (m(\overline{x})\ \mathit{body})\ \{Q\}}$$

Here block represents classical reasoning about blocks and method' represents the simple extension from blocks to annotated method declarations.

By applying the rule imp and sequential composition comp we may also derive $\{P\}\,\overline{s}\,\{Q\}$ from $\{P\}\,\overline{s}\,\{WF \Rightarrow Q\}$. Implication the other direction is trivial, since Q is stronger than $WF \Rightarrow Q$: We may derive $\{P\}\,\overline{s}\,\{WF \Rightarrow Q\}$ from $\{P\}\,\overline{s}\,\{Q\}$ by the imp rule. Accordingly, we have the following result:

Lemma 2 (Equivalence of Hoare triples).

$$\vdash \{P\}\,\overline{s}\,\{WF \Rightarrow Q\} \Leftrightarrow\ \vdash \{P\}\,\overline{s}\,\{Q\}$$

5.2. Invariant Reasoning

In interactive and non-terminating systems, it is difficult to specify and reason compositionally about object behavior in terms of pre- and post-conditions of the defined methods. Instead, pre- and post-conditions to method definitions are in our setting used to establish a so-called *class invariant*, i.e., a local assertion that holds after initialization of the class, and is maintained by all methods. The class invariant serves as a *contract* for the class: A method implements its part of the contract by ensuring that the invariant holds upon termination, assuming that the invariant holds initially. To facilitate compositional and component-based reasoning about programs, the class invariant is used to establish a *relationship between the internal state and the observable behavior of the class instance.* The internal state is given by the values of the class fields, whereas the observable behavior is expressed as a set of potential communication histories. By hiding the internal state, class invariants form a suitable basis for compositional reasoning about object systems. Assumptions to the environment may be reflected by invariants on the form of implications.

A *user-provided invariant* $I_C(\overline{w}, \mathcal{H})$ for a class C is a predicate over the fields $\overline{w}$ and the local history $\mathcal{H}$, as well as the formal class parameters $\overline{cp}$ and this, which are constant (read-only) variables.

5.3. Compositional Reasoning

A history invariant for instances of C is a predicate that only talks about the local history of that object and is satisfied at all times (in contrast to class invariants that may be temporarily violated inside a method). A history invariant can usually be derived from the class invariant (when prefix-closed). For an instance o of C with actual class parameter values $\overline{e}$, the history invariant $I_{o:C(\overline{e})}(h)$ is defined by hiding the internal state $\overline{w}$ and instantiating this and the class parameters $\overline{cp}$:

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

but in addition it must be proved that $I_{o:C(\overline{e})}(\mathcal{H})$ holds at all times, i.e., is maintained by each statement in the class C, possibly weakening the class invariant if needed. In practice this is trivial, when the history invariant is prefix closed (with respect to the history) [20].

We next consider systems with several objects and with an externally created initial object. The initial object may create some objects which again may create other objects and so on. We say that the system is *generated by* the externally created object. (We might generalize to several externally generated objects, using a reserved name to represent the environment of the program.)

The history invariant $I_S(h)$ for a system S given by an initial object, say $c : C(\overline{e})$, is then given by the conjunction of the history invariants of the initial and generated objects on their respective local histories:

$$I_S(h) \triangleq \langle c \uparrow c, C, \overline{e} \rangle \leq h \wedge wf(h) \bigwedge_{(o:C(\overline{e})) \in new_{ob}(h)} I_{o:C(\overline{e})}(h/o)$$

The externally created object will appear as an initial creation event in the global history, and thus be part of $new_{ob}(h)$. The wellformedness property serves as a connection between the local histories, relating events with the same future to each other. Note that the system invariant is obtained directly from the history invariants of the dynamically 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 [17]. Soundness of the composition rule is considered in [20].

6. Specification and Verification of the Publisher-Subscriber Example

In this example we consider object systems based on the classes found in Fig. 3. Different executions may lead to different global histories for this system, depending on the interleaving of the different object activities. However, we may state some general properties, like the following one: For every *signal* invocation from a proxy py to a client c with news ns, the client must have *subscribed* to a service v, which must have issued a *publish* invocation with a future u generated by a *detectNews* invocation, and then the proxy py must have received news ns from that future. This expresses that when clients get news it is only from services they have subscribed to, and the news is resulting from actions of the server. This global property can be formalized as follows:

$$\begin{aligned} H\ \mathbf{ew}\ \langle py \to c, u_0, signal, ns\rangle \implies\ & \exists v, u\,.\,(\langle \twoheadrightarrow v, _, subscribe, c\rangle, \langle py \twoheadleftarrow, u, ns\rangle) \sqsubseteq H\ \wedge \\ & (\langle v \to _, u, detectNews, \varepsilon\rangle, \langle v \to _, _, publish, u\rangle) \sqsubseteq H \end{aligned}$$

where non-interesting arguments are identified by _ rather than existentially quantified variables, for better readability.

We may derive this property within the proof system using the following class invariants, which focus on the order of the local events (while ignoring fields):

$$\begin{aligned} I_{Service(limit)}(\mathcal{H}) \triangleq\ & (\exists c\,.\,\langle \mathsf{this} \to _, _, add, c\rangle \sqsubseteq \mathcal{H} \\ & \quad \implies (\langle \twoheadrightarrow \mathsf{this}, _, subscribe, c\rangle, \langle \mathsf{this} \to _, _, add, c\rangle) \sqsubseteq \mathcal{H})\ \wedge \\ & (\exists u\,.\,\langle \mathsf{this} \to _, _, publish, u\rangle \sqsubseteq \mathcal{H} \\ & \quad \implies (\langle \mathsf{this} \to _, u, detectNews, \varepsilon\rangle, \langle \mathsf{this} \to _, _, publish, u\rangle) \sqsubseteq \mathcal{H}) \end{aligned}$$

$$\begin{aligned} I_{Proxy(limit,s)}(\mathcal{H}) \triangleq\ & (\exists u\,.\,\langle \mathsf{this} \to _, _, publish, u\rangle \sqsubseteq \mathcal{H} \\ & \quad \implies (\langle \twoheadrightarrow \mathsf{this}, _, publish, u\rangle, \langle \mathsf{this} \to _, _, publish, u\rangle) \sqsubseteq \mathcal{H})\ \wedge \\ & (\exists c, ns\,.\,\langle \mathsf{this} \to c, _, signal, ns\rangle \sqsubseteq \mathcal{H} \\ & \quad \implies \exists u\,.\,\langle \twoheadrightarrow \mathsf{this}, _, add, c\rangle, \langle \twoheadrightarrow \mathsf{this}, _, publish, u\rangle, \langle \mathsf{this} \twoheadleftarrow, u, ns\rangle, \\ & \qquad\quad \langle \mathsf{this} \to c, _, signal, ns\rangle \sqsubseteq \mathcal{H}) \end{aligned}$$

These invariants are straightforwardly verified in the above proof system. As explained, the corresponding invariants for the object instances $s : Service(limit)$ and $p : Proxy(limit, s)$ are obtained by substituting actual values for this and class parameters:

$$\begin{aligned} I_{s:Service(limit)}(\mathcal{H}) \triangleq\ & (\exists c\,.\,\langle s \to _, _, add, c\rangle \sqsubseteq \mathcal{H} \\ & \quad \implies (\langle \twoheadrightarrow s, _, subscribe, c\rangle, \langle s \to _, _, add, c\rangle) \sqsubseteq \mathcal{H})\ \wedge \\ & (\exists u\,.\,\langle s \to _, _, publish, u\rangle \sqsubseteq \mathcal{H} \\ & \quad \implies (\langle s \to _, u, detectNews, \varepsilon\rangle, \langle s \to _, _, publish, u\rangle) \sqsubseteq \mathcal{H}) \end{aligned}$$

$$\begin{aligned} I_{p:Proxy(limit,s)}(\mathcal{H}) \triangleq\ & (\exists u\,.\,\langle p \to _, _, publish, u\rangle \sqsubseteq \mathcal{H} \\ & \quad \implies (\langle \twoheadrightarrow p, _, publish, u\rangle, \langle p \to _, _, publish, u\rangle) \sqsubseteq \mathcal{H})\ \wedge \\ & (\exists c, ns\,.\,\langle p \to c, _, signal, ns\rangle \sqsubseteq \mathcal{H} \\ & \quad \implies \exists u\,.\,\langle \twoheadrightarrow p, _, add, c\rangle, \langle \twoheadrightarrow p, _, publish, u\rangle, \langle p \twoheadleftarrow, u, ns\rangle, \\ & \qquad\quad \langle p \to c, _, signal, ns\rangle \sqsubseteq \mathcal{H}) \end{aligned}$$

The global invariant of a system S with one server object $s : Service(limit)$ and some clients, created by an initial object, say $c : C(\overline{e})$, is then

$$\begin{aligned} I_S(h) \triangleq\ & \langle c \uparrow c, C, \overline{e}\rangle \leq h \wedge wf(h) \wedge I_{s:Service(limit)}(h/s) \\ & \bigwedge_{(p:Proxy(limit,s))\in new_{ob}(h)} I_{p:Proxy(limit,s)}(h/p) \end{aligned}$$

where wellformedness allows us to relate the different object histories. Note that invariants of other objects are ignored as we have not given invariants for other classes (and are by default *true*). From this global invariant we may inductively derive the system property defined above, by means of induction with respect to the length of the history, similarly to the buffer example in [17]. By strengthening the class invariant of *Proxy*, we can also prove other properties such as:

> For each proxy, if *nextProxy* is null, the number of contained clients is less or equal to *limit*, otherwise equal to *limit*.

7. Soundness and Completeness

We say that a reasoning system is sound if any provable property is valid, i.e.,

$$\vdash \{P\}\,\overline{s}\,\{Q\} \Rightarrow \models \{P\}\,\overline{s}\,\{Q\}$$

To prove that a reasoning system is sound, we need to show that all axioms of the system are valid and that all inference rules are sound, in the sense that they preserve validity. Validity of Hoare triples, denoted $\models \{P\}\,\overline{s}\,\{Q\}$, is defined by means of the operational semantics. We base the semantics on the operational semantics augmented with histories, 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 $g_1 \xrightarrow{o} g_2$ to indicate that the execution step is made by object o. When exactly one statement s is executed by o and we wish to highlight this statement, we write $g_1 \xrightarrow{o:s} g_2$:

Definition 7 (Explicit execution step).

$$g_1 \xrightarrow{o} g_2 \triangleq g_1 \rightarrow g_2 \wedge \forall o' \,.\, (o \neq o') \Leftrightarrow (g_1[o'] = g_2[o'])$$

expresses a transition from system (sub)state g_1 to system (sub)state g_2 due to an execution step made by object o (while all other objects are unchanged).

$$g_1 \xrightarrow{o:s} g_2 \triangleq g_1 \xrightarrow{o} g_2 \wedge g_1[o].Code = s; g_2[o].Code$$

expresses a transition from the system (sub)state g_1 to the system (sub)state g_2 due to an execution step made by object o through execution of statement s.

The latter relation is lifted to sequences of statements $\overline{s}$, executed by the same object o, by

$$g_1 \xrightarrow{o:\, s;\overline{s}} g_2 \triangleq \exists g' \,.\, g_1 \xrightarrow{o:s} g' \wedge g' \xrightarrow{o:\overline{s}} g_2$$

letting $g_1 \xrightarrow{o:\varepsilon} g_2 \triangleq g_1 = g_2$.

In Section 5 the axioms and inference rules are defined locally for each statement. Therefore, to express the soundness of our reasoning system, a notion of local state transitions are needed. Accordingly, we develop a proof structure to extract local state transition from the rewriting rules as shown in Fig. 8.

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 $P \xrightarrow[o]{o':s} Q$ express that if the condition P holds for object o before execution of s by object o', then Q holds for o after the execution. For convenience, we add a similar notation without s. This is defined as follows:

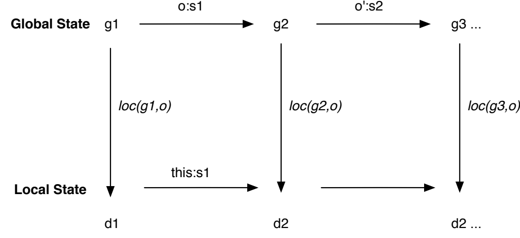

Figure 8: Projection from the global states to the local states. Statement s_2 in object o' follows statement s_1 in object o. Local state d_2 is maintained by the transition $g_2 \xrightarrow{o':s_2} g_3$.

Definition 8 (Validity of pre/post-conditions over execution steps).

$$P \xrightarrow[o]{o'} Q \triangleq \forall g, g', \overline{z} \,.\, wf(g[H]) \wedge wf(g'[H]) \wedge g \xrightarrow{o'} g' \wedge loc(g,o)[P] \Rightarrow loc(g',o)[Q]$$

$$P \xrightarrow[o]{o':\overline{s}} Q \triangleq \forall g, g', \overline{z} \,.\, wf(g[H]) \wedge wf(g'[H]) \wedge g \xrightarrow{o':\overline{s}} g' \wedge loc(g,o)[P] \Rightarrow loc(g',o)[Q]$$

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

$$loc(g,o) \triangleq g[o].State + [\mathcal{H} \mapsto g[H]/o]$$

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. Then $loc(g,o)[x]$ is the value bound to variable x in the local state $loc(g,o)$. We also use the notation $loc(g,o)[e]$ to evaluate expressions e in the local state of o; and in particular $loc(g,o)[P]$ is the truth-value of the condition P in the local state of o, made by replacing the free variables in P by the corresponding values given by the local state and the given history.

Due to the local understanding of histories in local assertions and disjointness of alphabets for different objects, we have the following non-interference result:

Lemma 3 (Non-interference).

$$o \neq o' \Rightarrow P \xrightarrow[o]{o'} P$$

expressing that an assertion on object o is not affected by execution of other objects (o'). This means that local reasoning can be done locally.

The proof is straight forward by considering the rules of the operational semantics: Each rule involves exactly one object, and no object o' may change the state of object o (assuming $o \neq o'$). Object o' may change the global history but not the projection on o since each object has disjoint alphabets.

The above non-interference result allows the following local understanding of validity of Hoare triples for statement lists:

Definition 9 (Validity of Hoare triples for statement lists).

$$\models \{P\}\,\overline{s}\,\{Q\} \triangleq \forall o\,.\,P \xrightarrow[o]{o:\overline{s}} Q$$

Here o is the executing object and the object on which P and Q are interpreted. Thus a Hoare triple is valid if for any object o executing $\overline{s}$, the postcondition holds in the poststate, provided the precondition holds in the prestate, assuming wellformedness. As seen this is not affected by the environment of o.

Due to Def. 9 which incorporates global wellformedness, and the entailment of local wellformedness from global wellformedness, we have the following result:

Lemma 4 (Equivalence of Validity).

$$\models \{P\}\,\overline{s}\,\{\mathit{WF} \Rightarrow Q\} \Leftrightarrow \models \{P\}\,\overline{s}\,\{Q\}$$

A Hoare axiom is said to be sound if it is valid. A proof rule is sound if validity of the premises imply validity of the conclusion.

Note that global validity, say of a global invariant I over H, could be defined directly on global states (with explicit H), i.e., the meaning of $\{I(H)\}\,s\,\{I(H)\}$ is that $I(g[H]) \Rightarrow I(g'[H])$ for all transitions $g \xrightarrow{o:s} g'$.

Theorem 1. (Soundness)

$$\vdash \{P\}\,\overline{s}\,\{Q\} \Rightarrow \models \{P\}\,\overline{s}\,\{Q\}$$

Theorem 2. (Relative completeness)

$$\models \{P\}\,\overline{s}\,\{Q\} \Rightarrow \vdash \{P\}\,\overline{s}\,\{Q\}$$

assuming completeness of the underlying logic for verification conditions.

The proofs for soundness and completeness are considered below. The proof is done by identifying weakest liberal preconditions for each construct, according to the given semantics, and prove that these are derivable in the system. In fact for each statement the given proof rules express the weakest liberal preconditions (modulo wellformedness). Thus the system is complete in the sense of Cook since the proof of completeness satisfies Cook's condition on expressiveness [21].

We first state a lemma useful for reducing the problem at hand. Due to Lemma 2 and Lemma 4, we derive the following result:

Lemma 5 (Proof Alternatives). *For proving* $\vdash \{P\}\,\overline{s}\,\{Q\} \;\Leftrightarrow\; \models \{P\}\,\overline{s}\,\{Q\}$*, it suffices to prove* $\vdash \{P\}\,\overline{s}\,\{\mathit{WF} \Rightarrow Q\} \;\Leftrightarrow \models \{P\}\,\overline{s}\,\{\mathit{WF} \Rightarrow Q\}$.

In other words, to prove that the reasoning system for $\{P\}\,\overline{s}\,\{Q\}$ is sound and complete, it suffices to prove that the reasoning system is sound and complete for postconditions of the form $\mathit{WF} \Rightarrow Q$.

7.1. Proof of Soundness and Completeness

The proof of soundness is by induction on the proof structure. It suffices to prove that each axiom is valid and that each inference rule is sound. Below we consider validity of each axiom and soundness of the imp rule and method rule.

The proof of completeness is by induction on the number of applications of operational rules. We show below the base case corresponding to executing one basic statement. Each base case will involve at least one application of an operational rule. In the inductive step we may assume $\models \{P\}\,\overline{s}\,\{Q\}$ implies $\vdash \{P\}\,\overline{s}\,\{Q\}$ for n execution steps and it is sufficient to prove $\models \{P\}\,\overline{s}; s\,\{Q\}$ implies $\vdash \{P\}\,\overline{s}; s\,\{Q\}$ for any basic statement s.

We first show soundness and completeness for empty statement lists. For each basic statement s we will show that validity of $\{P\}\,s\,\{Q\}$ can be reduced to an implication $P \Rightarrow Q'$ and that $\{Q'\}\,s\,\{Q\}$ is a Hoare axiom, i.e., Q' is the weakest possible precondition according to validity. Thus soundness follows, and completeness also follows when Q' is the precondition of the axiom, since $\vdash \{P\}\,s\,\{Q\}$ can be obtained by the imp rule, using $P \Rightarrow Q'$. We show below for each basic statement s that the axiom for s expresses exactly the weakest possible precondition according to validity. Thus by the imp rule, completeness follows for s as above, proving $\vdash \{P\}\{Q'\}\,s\,\{Q\}$.

For sequential composition we notice first that by Lemma 5 and the imp rule, one may keep assertions on the form $WF \Rightarrow P$. And for these kinds of assertions the validity of sequential composition reduces to the standard one. Thus this case is not so interesting. Standard rules for if-statements and loops are not considered here. At the end we consider soundness and completeness for reasoning about annotated method declarations.

The Implication rule.
For the implication rule we observe that by definition $\models \{P\}\ \{Q\}$ reduces to

$$\forall o, g, \overline{z}\,.\, wf(g[H]) \wedge loc(g,o)[P] \Rightarrow loc(g,o)[Q]$$

which by definition of local wellformedness reduces to $WF \wedge P \Rightarrow Q$ (for all free variables involved), which is exactly the condition of the implication rule. Thus the given Hoare rule is sound.

For completeness of reasoning about empty sequence lists, we have that $\models \{P\}\ \{Q\}$ implies $\vdash \{P\}\ \{Q\}$ since $\models \{P\}\ \{Q\}$ gives $WF \wedge P \Rightarrow Q$ which again gives $\vdash \{P\}\ \{Q\}$, and since we assume completeness of the underlying logic.

7.1.1. Basic statements

Before we look at the different basic statements, we first notice that given a semantic rule of form $o \mapsto \mathbf{ob}(\delta, s; \overline{s}) \xrightarrow{\alpha} o \mapsto \mathbf{ob}(\delta', \overline{s})$, we may reduce $\models \{P\}\,s\,\{Q\}$ to

$$wf(h) \wedge wf(h \cdot \alpha) \wedge \delta[P^{\mathcal{H}}_{h/o}] \Rightarrow \delta'[Q^{\mathcal{H}}_{h/o\cdot\alpha}]$$

(for all o, h, and other free variables) which by prefix closure of wf is

$$wf(h \cdot \alpha) \wedge \delta[P^{\mathcal{H}}_{h/o}] \Rightarrow \delta'[Q^{\mathcal{H}}_{h/o\cdot\alpha}]$$

which by definition of local wellformedness is

$$wf_o(\mathcal{H} \cdot \alpha) \wedge \delta[P] \Rightarrow \delta'[Q^{\mathcal{H}}_{\mathcal{H}\cdot\alpha}]$$

for all local histories $\mathcal{H}$ and other free variables, since h/o spans all possible local histories, i.e., for all local histories h we have that $h/o = h$, and using the definition of local wellformedness. This can be reformulated as

$$\delta[P] \Rightarrow (wf_o(\mathcal{H} \cdot \alpha) \Rightarrow \delta'[Q^{\mathcal{H}}_{\mathcal{H}\cdot\alpha}])$$

which is the same as

$$\delta[P] \Rightarrow \delta'[WF \Rightarrow Q]^{\mathcal{H}}_{\mathcal{H}\cdot\alpha}$$

According to lemma 5, this can be simplified to

$$\delta[P] \Rightarrow \delta'[Q^{\mathcal{H}}_{\mathcal{H}\cdot\alpha}]$$

(for all free variables) since we may assume that the postcondition Q already has WF as a condition. We use this general reduction result when considering the different statements below.

Skip statement.
The operational semantics of the skip statement is given by

$$o \mapsto \mathbf{ob}(\delta, \mathtt{skip}; \overline{s}) \xrightarrow{empty} o \mapsto \mathbf{ob}(\delta, \overline{s})$$

As explained above, $\models \{P\}\, skip\, \{Q\}$ reduces to $\delta[P] \Rightarrow \delta[Q]$. For a local state δ consisting of substitutions $a_i \mapsto v_i$ and $l_j \mapsto u_j$ (for a_i and l_j ranging over fields and method local variables) we may rename v_i and u_j to a_i and l_j respectively, and obtain

$$P \Rightarrow Q$$

for all a_i and l_j and $\mathcal{H}$. For a given postcondition Q, the weakest possible precondition satisfying this is Q. Therefore the axiom $\{Q\}\, \mathtt{skip}\, \{Q\}$ defines the weakest precondition. Thus soundness follows and completeness also follows since the precondition of the axiom is the weakest possible according to validity, and since any stronger precondition can be obtained by the imp rule.

Assignment statement.
The operational semantics of assignment statement is given by

$$o \mapsto \mathbf{ob}(\delta, v := e; \overline{s}) \xrightarrow{empty} o \mapsto \mathbf{ob}(\delta[v := e], \overline{s})$$

As explained above, $\models \{P\}\, v := e\, \{Q\}$ reduces to $\delta[P] \Rightarrow \delta[v := e][Q]$ which (as explained for **skip**) reduces to

$$P \Rightarrow Q^v_e$$

(for all free variables). Thus, for postcondition Q the weakest possible precondition is Q^v_e which is exactly what is stated in the axiom $\{Q^v_e\}\, v := e\, \{Q\}$.

Return statement.
The operational semantics of the return statement is given by

$$o \mapsto \mathbf{ob}(\delta, \mathtt{put}\ e) \xrightarrow{\langle \leftarrow o, \delta[\mathsf{future}], \delta[e] \rangle} \begin{array}{l} o \mapsto \mathbf{ob}(\delta, empty) \\ \delta[\mathsf{future}] \mapsto \mathbf{fut}(\delta[e]) \end{array}$$

As explained above, $\models \{P\}\, \texttt{put}\ e\, \{Q\}$ reduces to

$$\delta[P] \Rightarrow \delta[\mathcal{H} := \mathcal{H} \cdot \langle \leftarrow\ o, \delta[\mathsf{future}], \delta[e]\rangle][Q]$$

which is

$$P \Rightarrow Q^{\mathcal{H}}_{\mathcal{H}\cdot\langle\leftarrow\ \mathsf{this},\mathsf{future},e\rangle}$$

Thus, for postcondition Q the weakest possible precondition is

$$Q^{\mathcal{H}}_{\mathcal{H}\cdot\langle\leftarrow\ \mathsf{this},\mathsf{future},e\rangle}$$

which is exactly the same as the precondition of the axiom $\{Q^{\mathcal{H}}_{\mathcal{H}\cdot\langle\leftarrow\ \mathsf{this},\mathsf{future},e\rangle}\}\, \texttt{put}\ e\, \{Q\}$.

Query statement.
The operational semantics of query statement is given by

$$\begin{array}{l} u \mapsto \mathbf{fut}(d) \\ o \mapsto \mathbf{ob}(\delta, v := \texttt{get}\ e; \overline{s}) \\ \texttt{if}\ \delta[e] = u \end{array} \xrightarrow{\langle o\ \twoheadleftarrow, u, d\rangle} \begin{array}{l} u \mapsto \mathbf{fut}(d) \\ o \mapsto \mathbf{ob}(\delta[v := d], \overline{s}) \end{array}$$

As explained above, $\models \{P\}\, v := \texttt{get}\ e\, \{Q\}$ reduces to

$$\delta[P] \Rightarrow \delta[v := d, \mathcal{H} := \mathcal{H} \cdot \langle o\ \twoheadleftarrow, e, d\rangle][Q]$$

which, as explained above, reduces to

$$P \Rightarrow Q^{v,\mathcal{H}}_{d,\mathcal{H}\cdot\langle\mathsf{this}\ \twoheadleftarrow,e,d\rangle}$$

(for all d, $\mathcal{H}$, and other free variables) which means

$$\forall d\, .\, P \Rightarrow Q^{v,\mathcal{H}}_{d,\mathcal{H}\cdot\langle\mathsf{this}\ \twoheadleftarrow,e,d\rangle}$$

Since d may not occur in P, this reduces to

$$P \Rightarrow \forall d\, .\, Q^{v,\mathcal{H}}_{d,\mathcal{H}\cdot\langle\mathsf{this}\ \twoheadleftarrow,e,d\rangle}$$

Thus, for postcondition Q the weakest possible precondition is

$$\forall d\, .\, Q^{v,\mathcal{H}}_{d,\mathcal{H}\cdot\langle\mathsf{this}\ \twoheadleftarrow,e,d\rangle}$$

which is exactly the same as the precondition of the axiom
$\{\forall v'\, .\, Q^{v,\mathcal{H}}_{v',\mathcal{H}\cdot\langle\mathsf{this}\ \twoheadleftarrow,e,v'\rangle}\}\, v := \texttt{get}\ e\, \{Q\}$.

Asynchronous method call statement.
The abstract operational semantics of asynchronous method call is given by

$$\begin{array}{l} o \mapsto \mathbf{ob}(\delta, fr := v!m(\overline{e}); \overline{s}) \\ H \mapsto h \\ \texttt{if}\ \ u \notin id(h) \end{array} \rightarrow \begin{array}{l} o \mapsto \mathbf{ob}(\delta[fr := u], \overline{s}) \\ u \mapsto \mathbf{msg}(\delta[v], m, \delta[\overline{e}]) \\ H \mapsto h \cdot \langle o \rightarrow \delta[v], u, m, \delta[\overline{e}]\rangle \end{array}$$

As explained above, $\models \{P\}\, fr := v!m(\overline{e})\, \{Q\}$ reduces to

$$\delta[P]\ \wedge\ u \notin id(\mathcal{H}) \Rightarrow \delta[fr := u, \mathcal{H} := \mathcal{H} \cdot \langle o \rightarrow \delta[v], u, m, \delta[\overline{e}]\rangle][Q]$$

which is, assuming Q has WF as a condition according to Lemma 5,

$$P \Rightarrow \forall u \,.\, Q^{fr,\mathcal{H}}_{u,\mathcal{H}\cdot\langle \mathsf{this} \twoheadrightarrow v,u,m,\overline{e}\rangle}$$

since WF implies uniqueness of future identities in invocation events. Thus, for postcondition Q the weakest possible precondition is

$$\forall fr' \,.\, Q^{fr,\mathcal{H}}_{fr',\mathcal{H}\cdot\langle \mathsf{this} \twoheadrightarrow v,fr',m,\overline{e}\rangle}$$

which is exactly the same as the precondition of the axiom
$\{\forall fr' \,.\, Q^{fr,\mathcal{H}}_{fr',\mathcal{H}\cdot\langle \mathsf{this} \twoheadrightarrow v,fr',m,\overline{e}\rangle}\}\ fr := v!m(\overline{e})\ \{Q\}$.

New statement.
The abstract operational semantics of object creation is given by

$$\begin{array}{lcl} o \mapsto \mathbf{ob}(\delta, v := \mathtt{new}\ C(\overline{e}); \overline{s}) & & o \mapsto \mathbf{ob}(\delta[v := o'], \overline{s}) \\ H \mapsto h & \rightarrow & o' \mapsto \mathbf{ob}(\delta_{init}, init) \\ \mathtt{if}\ \ o' \notin id(h) & & H \mapsto h \cdot \langle o \uparrow o', C, \delta[\overline{e}]\rangle \end{array}$$

As explained above, $\models \{P\}\ v := \mathtt{new}\ C(\overline{e})\ \{Q\}$ reduces to

$$\delta[P] \ \wedge\ o' \notin id(\mathcal{H}) \Rightarrow \delta[v := o', \mathcal{H} := \mathcal{H} \cdot \langle o \uparrow o', C, \overline{e}\rangle][Q]$$

which is, assuming Q has WF as a condition according to Lemma 5,

$$P \Rightarrow \forall v' \,.\, Q^{v,\mathcal{H}}_{v',\mathcal{H}\cdot\langle o\uparrow v',C,\overline{e}\rangle}$$

since WF implies uniqueness of object identities in object creation events. Thus, for postcondition Q the weakest possible precondition is

$$\forall v' \,.\, Q^{v,\mathcal{H}}_{v',\mathcal{H}\cdot\langle o\uparrow v',C,\overline{e}\rangle}$$

which is exactly the same as the precondition of the axiom
$\{\forall v' \,.\, Q^{v,\mathcal{H}}_{v',\mathcal{H}\cdot\langle o\uparrow v',C,\overline{e}\rangle}\}\ v := \mathtt{new}\ C(\overline{e})\ \{Q\}$.

For each basic statement, we have now shown that the reasoning rule expresses the weakest possible precondition according to the semantics. As explained, this ensures soundness and completeness. We have paid special attention to all statements that involve futures and histories, apart from method declarations which are treated next.

7.1.2. Annotated Method Declarations

We first define validity of annotated method declarations, which has not been considered so far. Then we consider soundness and completeness for reasoning about annotated method declarations.

Definition 10 (Validity of annotated method declarations).

$$\begin{array}{r} \models \{P\}\ (m(\overline{x})\{\mathtt{var}\ y; \overline{s}\})\ \{Q\} \triangleq \forall o, g, g'', \overline{y}', \overline{z} \,.\, wf(g[H]) \wedge wf(g''[H]) \wedge \\ g' \xrightarrow[o]{o:\overline{s}} g'' \wedge loc(g,o)[P^{\overline{y}}_{\overline{y}'}] \Rightarrow loc(g'',o)[Q^{\overline{y}}_{\overline{y}'}] \end{array}$$

where $\overline{y}'$ are fresh logical variables and g' denotes $g[H := H \cdot \langle \twoheadleftarrow o, \mathsf{future}, m, \overline{x}\rangle, \overline{y} := \mathit{default}]$. Here *default* represents the default values of the appropriate types, and $\overline{z}$ is the list of logical variables other than $\overline{y}'$. The use of $\overline{y}'$ reflects that $\overline{y}$ in P and Q does not refer to the local variables $\overline{y}$ of the method.

To prove that the reasoning rule

$$\mathsf{method}\quad \frac{\{P^{\overline{y}}_{\overline{y}'} \wedge \overline{y} = \mathit{default}\}\ \overline{s}\ \{Q^{\overline{y}}_{\overline{y}'}\}}{\{P^{\mathcal{H}}_{\mathcal{H}\cdot\langle\twoheadleftarrow \mathsf{this},\mathsf{future},m,\overline{x}\rangle}\}\ (m(\overline{x})\{\mathbf{var}\ \overline{y}; \overline{s}\})\ \{Q\}}$$

is sound, we need to show that the validity of the premise implies the validity of the conclusion. The validity of the conclusion reduces to

$$\forall o, g', g'', \overline{y}', \overline{z}\,.\, \mathit{wf}(g'[H]) \wedge \mathit{wf}(g''[H]) \wedge g'[\overline{y} \mapsto \mathit{default}] \xrightarrow[o]{o:\overline{s}} g'' \wedge \mathit{loc}(g', o)[P^{\overline{y}}_{\overline{y}'}] \Rightarrow \mathit{loc}(g'', o)[Q^{\overline{y}}_{\overline{y}'}]$$

which is exactly the validity of the premise. We therefore have

$$\models \{P^{\mathcal{H}}_{\mathcal{H}\cdot\langle\twoheadleftarrow \mathsf{this},\mathsf{future},m,\overline{x}\rangle}\}\ (m(\overline{x})\{\mathbf{var}\ \overline{y}; \overline{s}\})\ \{Q\}\ \Leftrightarrow \models \{P^{\overline{y}}_{\overline{y}'} \wedge \overline{y} = \mathit{default}\}\ \overline{s}\ \{Q^{\overline{y}}_{\overline{y}'}\}$$

Thus the given Hoare rule is sound.

For completeness, we must prove

$$\vdash \{R\}\ (m(\overline{x})\{\mathbf{var}\ \overline{y}; \overline{s}\})\ \{Q\}$$

for any R and Q assuming

$$\models \{R\}\ (m(\overline{x})\{\mathbf{var}\ \overline{y}; \overline{s}\})\ \{Q\} \tag{2}$$

Taking P of the above equivalence as $R^{\mathcal{H}}_{\mathit{pop}(\mathcal{H})}$ we get

$$\models \{(R^{\mathcal{H}}_{\mathit{pop}(\mathcal{H})})^{\mathcal{H}}_{\mathcal{H}\cdot\langle\twoheadleftarrow \mathsf{this},\mathsf{future},m,\overline{x}\rangle}\}\ (m(\overline{x})\{\mathbf{var}\ \overline{y}; \overline{s}\})\ \{Q\}\ \Leftrightarrow \models \{(R^{\mathcal{H}}_{\mathit{pop}(\mathcal{H})})^{\overline{y}}_{\overline{y}'} \wedge \overline{y} = \mathit{default}\}\ \overline{s}\ \{Q^{\overline{y}}_{\overline{y}'}\}$$

which reduces to

$$\models \{R\}\ (m(\overline{x})\{\mathbf{var}\ \overline{y}; \overline{s}\})\ \{Q\}\ \Leftrightarrow \models \{(R^{\mathcal{H}}_{\mathit{pop}(\mathcal{H})})^{\overline{y}}_{\overline{y}'} \wedge \overline{y} = \mathit{default}\}\ \overline{s}\ \{Q^{\overline{y}}_{\overline{y}'}\}$$

since $(R^{\mathcal{H}}_{\mathit{pop}(\mathcal{H})})^{\mathcal{H}}_{\mathcal{H}\cdot\langle\twoheadleftarrow \mathsf{this},\mathsf{future},m,\overline{x}\rangle}$ is the same as R. By (2) and the induction hypothesis (ie., that completeness applies to structurally simpler programs) we have

$$\vdash \{(R^{\mathcal{H}}_{\mathit{pop}(\mathcal{H})})^{\overline{y}}_{\overline{y}'} \wedge \overline{y} = \mathit{default}\}\ \overline{s}\ \{Q^{\overline{y}}_{\overline{y}'}\}$$

And the method proof rule gives us

$$\vdash \{R\}\ (m(\overline{x})\{\mathbf{var}\ \overline{y}; \overline{s}\})\ \{Q\}$$

Consequently, we have completeness for reasoning about annotated method declarations.

8. Addition of Non-Blocking Queries and Process Control

We consider here an extension of the considered core language by constructs for process control, allowing conditional and unconditional process release and release while waiting for a future to be resolved, thereby avoiding blocking.

suspend	unconditional release
await b	conditional release
$x :=$ **await** future	releasing query
$x :=$ **await** $o.m(\overline{e})$	releasing call

$$
\begin{array}{lrl}
\textsf{await bool}: & & o \mapsto \mathbf{ob}((a|l), \texttt{await}\ b; \overline{s}, q) \\
& \xrightarrow{empty} & o \mapsto \mathbf{ob}((a|empty), empty, q \cdot (l, \texttt{await}\ b; \overline{s})) \\
\\
\textsf{await fut}: & & o \mapsto \mathbf{ob}((a|l), x := \texttt{await}\ e; \overline{s}, q) \\
& \xrightarrow{empty} & o \mapsto \mathbf{ob}((a|empty), empty, q \cdot (l, x := \texttt{await}\ e; \overline{s})) \\
\\
\textsf{choose bool}: & & o \mapsto \mathbf{ob}((a|l'), empty, (l, \texttt{await}\ b; \overline{s}) \cdot q) \\
& \xrightarrow{empty} & o \mapsto \mathbf{ob}((a|l), \overline{s}, q) \\
& \texttt{if} & (a|l)[b] = true \\
\\
\textsf{choose fut}: & & u \mapsto \mathbf{fut}(d) \\
& & o \mapsto \mathbf{ob}((a|l'), empty, (l, x := \texttt{await}\ e; \overline{s}) \cdot q) \\
& \xrightarrow{empty} & o \mapsto \mathbf{ob}((a|l), x := d; \overline{s}, q) \\
& & u \mapsto \mathbf{fut}(d) \\
& \texttt{if} & (a|l)[e] = u
\end{array}
$$

Figure 9: Operational rules for process control.

Conditional release allows releasing the processor while waiting for a condition b to be satisfied. Unconditional release may be defined in terms of conditional release as follows:

$$\texttt{suspend} \triangleq \texttt{await}\ true$$

A releasing query allows releasing the processor while waiting for the future to be resolved. Similarly, a releasing call statement releases while waiting for the call to be completed. A releasing call is defined in terms of releasing query, waiting for the associated future:

$$x := \texttt{await}\ o.m(\overline{e}) \triangleq v := o!m(\overline{e}); x := \texttt{await}\ v$$

where v is a fresh future variable.

Thus it suffices to formalize conditional release and releasing query.

8.1. Operational Semantics

The operational semantics is given in Fig. 9. The old rules for the core language are unchanged, apart from adding the queue element to the objects considered in the rules. The old notion of wellformedness is already compatible with the added statements. Notice that an idle object (i.e., an object with an empty statement list) may choose to continue processing a suspended method activation from the queue, provided it is enabled, or start a new method activation by the old start rule.

8.2. Axiomatic Semantics

Let e is an expression denoting a future. For partial correctness reasoning, we may define $x := \texttt{await}\ e$ by

$$\texttt{suspend};\ x := \texttt{get}\ e$$

since the effect of temporary blocking is not visible within partial correctness. Thus, $\{P\}\, x := \texttt{await}\ e\, \{Q\}$ is the same as $\{P\}\, \texttt{await}\ true; x := \texttt{get}\ e\, \{Q\}$. Hence, it remains to give reasoning rules for conditional release.

Reasoning over conditional release points can be done by means of the local invariant I. The invariant I must be proved to hold after initialization and that it is maintained by each method. The rule below ensures that the invariant is reestablished at release points.

$$\{I \wedge L \wedge h_0 = \mathcal{H}\}\, \mathbf{await}\ b\, \{b \wedge I \wedge L \wedge h_0 \leq \mathcal{H}\}$$

where L is an assertion referring to method parameters, local variables, and logical variables only. Thus the invariant provides information about the fields (and class parameters), and the local assertion L allows reasoning about local variables and parameters. A logical variable h_0 is used to express that the history may only be appended. Note that the triple $\{h_0 = \mathcal{H}\}\ s\ \{h_0 \leq \mathcal{H}\}$ can be derived for all statements s.

The reasoning rule is somewhat simpler than in [22]. The present version of conditional release will release even when the condition is satisfied.

8.3. Example

We reconsider the previous example, and show a more efficient implementation of the *publish* method, using a releasing query statement to avoid blocking the proxy object. This allows a higher degree of concurrent activity.

```
1  Void publish(Fut<News> fut){
2     var News ns = None;
3     ns := await fut;
4     myClients!signal(ns);
5     if nextProxy = null then s!produce() else nextProxy!publish(fut) fi}
```

We are still able to prove the same local and global invariants as before, because the local invariants are defined by means of subsequences of the history. In particular, the local history of the proxy object may grow at the process releasing point which is between $\langle \twoheadrightarrow \mathsf{this}, _, publish, u\rangle$ and $\langle \mathsf{this}\ \twoheadleftarrow, u, ns\rangle$. Since the class invariant defined in Sec. 6 does not require tight subsequence of history, the class invariant is maintained here.

9. Discussion

The setting of concurrent objects communicating solely by asynchronous method calls allows compositional reasoning in a way which resembles sequential reasoning. The main complication compared to sequential reasoning is the manipulation of the local history variable. Disjointness of alphabets for disjoint objects implies that local histories are disjoint, in the sense that processing steps made by one object do not affect the invariants of other objects. This ensures local reasoning inside each class, and composition of concurrent objects amounts to conjunction of invariants and relating local histories to the global history. We have extended this framework to futures, supporting local reasoning about futures based on locally visible events involving futures. Thus in order to obtain global knowledge about futures one relies on the composition rule. We find that specification of futures is tightly connected to that of the objects creating and operating on the futures. Our approach makes this possible.

We have shown how to extend the framework to the high-level process control statements (suspension and await statements) of the Creol/ABS languages, including constructs for fetching futures without blocking. This is useful to obtain more efficient computations and to avoid deadlocks. It also opens up for local calls of the form $x := \mathbf{await}\ \mathsf{this}.m(\ldots)$ which

are then executed before x is assigned the resulting future. In contrast a local call of the form $x := \mathsf{this}.m(\ldots)$ would deadlock since our semantics would block, as shown in section 5.1. Reentrant handling of such local calls could be done as in Creol [8]. However, efficient implementation and specialized reasoning of local calls are omitted here for simplicity reasons as we focus on object interaction mechanisms.

Our setting does not allow remote access to fields, as found in several mainstream object-oriented languages including Java and Spec$^\sharp$ [23]. This would necessitate constructs for programming critical regions, for instance by means of locks using a thread-based concurrency model. Remote field access would severely complicate our setting. In order to do compositional reasoning with histories one would need to consider read and write operations to shared variables [24, 25], as well as reasoning problems related to aliasing. Non-observable events would then be reflected in the histories, making both specification and reasoning much more low-level and much more complicated. As an alternative to history-based specifications, several approaches for Java and Spec$^\sharp$ use specifications with model variables. However, one needs more fine-grained control of when a class invariant holds. Rather than invariants maintained by each methods body (and reestablished at release points when considering suspend and await statements), one would need to consider invariants maintained by all atomic statements, as in [24], or include explicit control of when an invariant holds, for instance based on packing and unpacking of invariants as in [26].

Aliasing occurs when more than one variable refer to the same object. Therefore, if both variables v_1 and v_2 refer to the same object o, modifications of v_1 on o's fields affect the value of the variable v_2 as well. Thus with remote field assignments the classical assignment axiom (as in figure 7) would not be sound. We consider a programming language in which access to the internal state variables of other objects is only possible through remote method calls. This is the reason why the classical assignment axiom is sound in our case.

Our setting extended with await statements allows *callbacks*, i.e., inside a method body one may make calls to the caller object or other objects received as parameters. The implicit *caller* parameter (typed by a cointerface as in Creol) opens up for callbacks to the caller object without passing that object as an explicit parameter [8]. An object o making a non-blocking call, say $x :=$ **`await`** $r.m(\overline{e})$, is free to handle such callbacks, since the execution of the caller is suspended. The callback from r can then be executed as soon as any ongoing method execution of the object o reaches the end or a release point. In this way callbacks are possible without reentry mechanisms. (As before, calls for which the future is not needed by the caller, also allow callbacks, as was illustrated in the *produce* method of class *Service* in Figure 3.) In contrast a callback from r while the object o is blocking for some future would lead to deadlock. Therefore deadlock can be avoided if all queries are non-blocking, i.e., using the **`await`** mechanism.

Furthermore, a blocking query made by an object o will not cause deadlock if the callee does not cause a query to o. In order to get a syntactic guarantee for deadlock freedom, one could consider a partial ordering implying that there is no circular query chain starting with a blocking query (considering the implicit queries defined by calls with dot-notation). For simplicity we assume that each class has exactly one interface and we ignore complications with interface inheritance. We may then consider a strict partial ordering of interfaces, and restrict (blocking and nonblocking) queries to smaller interfaces according to the ordering. The body of a method used to implement a certain interface may query a future if the associated callee has a smaller interface according to the ordering. A complication is that a query on a future may not identify the callee (the object producing the future value). However, if the call introducing the future is in the same body, one may statically associate

a callee with the future. For futures appearing as formal parameters it suffices to consider all possible callees associated with the corresponding actual parameters – in the whole program. (A similar discussion applies to futures appearing as method results.) This gives a static approximation of the possible callees. When the callee is of the same interface as the caller we may use the partial order implied by ownership, i.e., we may allow calls to (statically known) child objects.

The example in Figure 3 can be seen to be deadlock-free according to this strategy. Class *Service* has one blocking query in method *subscribe*, given by the call *lastProxy.add*. This query is OK with *ProxyI* less than *ServiceI*. Class *Proxy* has one blocking query in method *add*, given by the call *nextProxy.add*. This query is made to a child object of the same interface. In method *publish* there is a query on *fut*, which is OK with *ProducerI* less than *ProxyI*, since *prod* is the (only) associated callee of the corresponding actual parameter. From the appendix, we see that the remaining classes are OK with *NewsProducerI* less than *ProducerI*. Clearly, the interface ordering required is a proper partial order.

10. Related Work

Models for asynchronous communication without futures have been explored for process calculi with buffered channels [11], for agents with message-based communication [27], for method-based communication [28], and in particular for Java [29]. 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 concurrent, object-oriented systems makes tool-based verification difficult in practice. A recent survey of these challenges can be found in [30]. The present approach follows the line of work based on communication histories to model object communication events in a distributed setting [30, 10, 31, 17, 32, 33, 11, 34, 25]. Objects are concurrent and interact solely by method calls and futures, and remote access to object fields are forbidden. Object generation is reflected in the history by means of object creation events. This enables compositional reasoning of concurrent systems with dynamic generation of objects and aliasing (while avoiding alias reasoning problems).

Futures, first introduced in Multilisp [5] are language constructs that improve concurrency and data flow synchronization in a natural and transparent way. Some frameworks allow futures to be passed to other processes. Such shared futures are called *first class futures*, which offer greater flexibility in application design and can significantly improve concurrency in object-oriented paradigms. A reasoning system for asynchronous method communication without futures is introduced in [17], from which we redefine the six-event semantics to reflect actions on first class futures, ending up with a five-event semantics. The semantics provides 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 simplifies the model and the accompanied proof system, thereby reducing the gap between reasoning about sequential systems and concurrent object-oriented systems. Especially, when reasoning about a class, it is not necessary to explicitly account for the activity of objects in the environment. 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 [17], which is inspired by previous approaches [34, 25].

By creating unique references for method calls, the *label* construct of Creol [8] resembles futures, as callers may postpone reading result values. Verification systems capturing Creol labels can be found in [30, 32]. However, a label reference is local to the caller, and cannot be shared with other objects. In [30], a compositional verification system for *Creol* is presented.

A reasoning system for asynchronous method calls and futures has been presented in [35], 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, rather than by compositional rules. Thus the environment of a class must be known at verification time. The completeness proof is based on a semantic characterization of the global invariant in terms of futures and two history variables. One denotes the sequence of generated communication events, which is updated by method calls, upon each method activation, and by each return statement. The other records the local state in order to reason about the internal process queue.

A compositional reasoning system for *ABS* with futures has been presented in our earlier work [9] based on local communication histories. We here show that a revised and simplified version of this system is sound and (relatively) complete with respect to a revised version of an operational semantics which incorporates a notion of global communication history. The present system is based on disjointness of events and uses five kinds of events, i.e., four related to futures and one for object creation, which is a simplification compared to earlier work with disjoint events. Soundness of the composition rule is studied in [20].

A compositional Hoare Logic for concurrent processes (objects) is presented in [36]. Soundness and relative completeness are proved with respect to the operational semantics. Communication histories capture the sequences of output messages, input messages and the generated object identities. History information is used in both the operational semantics and the reasoning system. In contrast to the present work, communication is by message passing rather than by method interaction, the objects communicate through FIFO channels, and futures are not considered.

In [37], a reasoning system for a subset of Eiffel is presented. Soundness and completeness of the reasoning system is proved with respect to the given operational semantics. However, concurrency is not considered as the language is sequential. Object orientation is the main focus, and in particular exception handling (but not futures) is included. In Eiffel, so-called *once routines* cache the first execution result for the later received invocations, where the arguments are irrelevant. Therefore, a "once routine" has less flexibility than the concept of shared futures in *ABS*.

11. Conclusion

In this paper we present a sound and relative complete compositional reasoning system for distributed objects with shared futures, based on a general concurrency and communication model centered around concurrent objects, asynchronous methods calls, and futures. This model is chosen due to advantages with respect to program reasoning, while integrating asynchronous interaction and object orientation in a natural manner. Compositional reasoning is facilitated by expressing object properties in terms of observable interaction between the object and its environment, recorded on communication histories. Object generation is reflected in the history by means of object creation events. A method call cycle

with multiple future readings is reflected by four kinds of events, giving rise to disjoint communication alphabets for different objects. Specifications in terms of history invariants may then be derived independently for each object and composed in order to derive properties for concurrent object systems.

At the class level, invariants define relationships between class attributes and the observable communication of class instances. The presented Hoare style system is proven sound and relatively complete with respect to the given operational semantics. This system is easy to apply in the sense that class reasoning is similar to standard sequential reasoning, but with the addition of effects on the local history for statements involving futures. In particular, reasoning inside classes is not affected by the complexity of concurrency and synchronization; and one may express assumptions about inputs from the environment when convenient.

At the global level, the notion of wellformedness allows us to connect the information in the different local invariants of each object, according to the natural event ordering (as expressed in Fig. 4). The notion of wellformedness and event kinds are simpler than in earlier work, and thus local reasoning as well as global reasoning are simplified. The presented reasoning system is illustrated by a version of the Publisher-Subscriber example. As seen in the example the presence of future implies that specifications typically involve existentially quantified future identities to express causal relationships in communication patterns. The example also shows that histories are highly expressive, letting projections and functions over the history express minimal requirements.

An interpreter for the considered core language based on the operational semantics has been implemented in Maude, including calculation of the global history. With the underlying tool support of Maude one can prototype and model check programs written in the language with respect to properties involving local and global histories. An *ABS* reasoning system is currently being implemented within the KeY framework at Technische Universität Darmstadt. The tool support from KeY for (semi-)automatic verification is valuable for verifying *ABS* programs. The current axiomatic system has been integrated in the tool. In [38] we compare initial experiments with this extension of the KeY system with the developed runtime checker for *ABS* programs with futures.

[1] International Telecommunication Union, Open Distributed Processing - Reference Model, parts 1–4, Tech. rep., ISO/IEC, Geneva (Jul. 1995).

[2] A. Ahern, N. Yoshida, Formalising Java RMI with explicit code mobility, *Theoretical Computer Science* 389 (3) (2007) 341 – 410.

[3] O.-J. Dahl, O. Owe, Formal methods and the RM-ODP, Tech. Rep. Research Report 261, Dept. of Informatics, Univ. of Oslo, Full version of a paper presented at NWPT'98: Nordic Workshop on Programming Theory, Turku (1998).

[4] H. G. Baker Jr., C. Hewitt, The incremental garbage collection of processes, in: Proceedings of the 1977 symposium on Artificial intelligence and programming languages, ACM, New York, NY, USA, 1977, pp. 55–59.

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

[6] B. H. Liskov, L. Shrira, Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems, in: D. S. Wise (Ed.), Proc. SIGPLAN Conference on

Programming Language Design and Implementation (PLDI'88), ACM Press, 1988, pp. 260–267.

[7] A. Yonezawa, J.-P. Briot, 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) (1986) 258–268.

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

[9] C. C. Din, J. Dovland, O. Owe, Compositional reasoning about shared futures, in: G. Eleftherakis, M. Hinchey, M. Holcombe (Eds.), Proc. International Conference on Software Engineering and Formal Methods (SEFM'12), Vol. 7504 of LNCS, Springer-Verlag, 2012, pp. 94–108.

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

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

[12] O.-J. Dahl, Object-oriented specifications, in: Research directions in object-oriented programming, MIT Press, Cambridge, MA, USA, 1987, pp. 561–576.

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

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

[15] A. S. A. Jeffrey, J. Rathke, Java Jr.: Fully abstract trace semantics for a core Java language, in: Proc. European Symposium on Programming, Vol. 3444 of LNCS, Springer-Verlag, 2005, pp. 423–438.

[16] B. Alpern, F. B. Schneider, Defining liveness, Information Processing Letters 21 (4) (1985) 181–185.

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

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

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

[20] C. C. Din, O. Owe, Compositional and sound reasoning about active objects with shared futures, Research Report 437, Dept. of Informatics, University of Oslo, In review for the FAC journal (Feb. 2014).
URL `http://urn.nb.no/URN:NBN:no-41224`

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

[22] C. C. Din, J. Dovland, E. B. Johnsen, O. Owe, Observable behavior of distributed systems: Component reasoning for concurrent objects, Research Report 401, Dept. of Informatics, University of Oslo, available at `http://folk.uio.no/crystald/` (Oct. 2010).

[23] M. Barnett, K. R. M. Leino, W. Schulte, The Spec$^\sharp$ programming system: An overview, in: Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, CASSIS'04, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 49–69.
URL `http://dx.doi.org/10.1007/978-3-540-30569-9_3`

[24] O. Owe, Axiomatic treatment of processes with shared variables revisited, Formal Asp. Comput. 4 (4) (1992) 323–340.

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

[26] K. M. Leino, P. Müller, A verification methodology for model fields, in: P. Sestoft (Ed.), Programming Languages and Systems, Vol. 3924 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2006, pp. 115–130.
URL `http://dx.doi.org/10.1007/11693024_9`

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

[28] B. Morandi, S. S. Bauer, B. Meyer, SCOOP – A Contract-Based Concurrent Object-Oriented Programming Model, in: P. Müller (Ed.), Advanced Lectures on Software Engineering, LASER Summer School 2007/2008, Vol. 6029 of LNCS, Springer, 2008, pp. 41–90.

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

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

[31] O.-J. Dahl, Can program proving be made practical?, in: M. Amirchahy, D. Néel (Eds.), Les Fondements de la Programmation, Institut de Recherche d'Informatique et d'Automatique, Toulouse, France, 1977, pp. 57–114.

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

[33] J. Dovland, E. B. Johnsen, O. Owe, Observable behavior of dynamic systems: Component reasoning for concurrent objects, in: D. Goldin, F. Arbab (Eds.), Proc. Workshop on the Foundations of Interactive Computation (FInCo'07), Vol. 203 of Electr. Notes Theor. Comput. Sci., Elsevier, 2008, pp. 19–34.

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

[35] F. S. de Boer, D. Clarke, E. B. Johnsen, A complete guide to the future, in: R. de Nicola (Ed.), Proc. 16th European Symposium on Programming (ESOP'07), Vol. 4421 of LNCS, Springer-Verlag, 2007, pp. 316–330.

[36] F. S. de Boer, A Hoare logic for dynamic networks of asynchronously communicating deterministic processes, *Theoretical Computer Science* 274 (2002) 3–41.

[37] M. Nordio, C. Calcagno, P. Müller, B. Meyer, Soundness and completeness of a program logic for Eiffel, Tech. Rep. 617, ETH Zurich (2009).

[38] C. C. Din, O. Owe, R. Bubel, Runtime assertion checking and theorem proving for concurrent and distributed systems, in: Proceedings of the 2nd Intl. Conf. on Model-Driven Engineering and Software Development, *Modelsward'14*, SCITEPRESS, 2014, pp. 480–487, DOI: 10.5220/0004877804800487.

AppendixA. Complete Code of Publisher-Subscriber Example

```
1  data News = E1 | E2 | E3 | E4 | E5 | None;
2
3  interface ServiceI{
4          Void subscribe(ClientI cl);
5          Void produce()}
6
7  interface ProxyI{
8          ProxyI add(ClientI cl);
9          Void publish(Fut<News> fut)}
10
11 interface ProducerI{
12         News detectNews()}
13
14 interface NewsProducerI{
15         Void add(News ns);
16         News getNews();
17         List<News> getRequests()}
18
19 interface ClientI{
20         Void signal(News ns)}
21
22
23 class Service(Int limit, NewsProducerI np) implements ServiceI{
24     ProducerI prod; ProxyI proxy; ProxyI lastProxy;
25     {prod := new Producer(np); proxy := new Proxy(limit,this); lastProxy := proxy; this!produce()}
26
27     Void subscribe(ClientI cl){lastProxy := lastProxy.add(cl)}
28
29     Void produce(){var Fut<News> fut := prod!detectNews(); proxy!publish(fut)}}
30
31
32 class Proxy(Int limit, ServiceI s) implements ProxyI{
33     List<ClientI> myClients := Nil; ProxyI nextProxy;
34
35     ProxyI add(ClientI cl){
36       var ProxyI lastProxy = this;
37       if length(myClients) < limit then myClients := appendright(myClients, cl)
38       else if nextProxy == null then nextProxy := new Proxy(limit,s) fi;
39       lastProxy := nextProxy.add(cl) fi; put lastProxy}
40
41     Void publish(Fut<News> fut){
42       var News ns = None;
43       ns = fut.get; myClients!signal(ns);
44       if nextProxy == null then s!produce() else nextProxy!publish(fut) fi}}
45
46 class Producer(NewsProducerI np) implements ProducerI{
47         News detectNews(){
48           var List<News> requests := Nil; News news := None;
49           requests := np.getRequests();
50           while requests == Nil do requests := np.getRequests() od
51           news := np.getNews(); put news}}
52
53 class NewsProducer implements NewsProducerI{
54         List<News> requests := Nil;
55         Void add(News ns){requests := appendright(requests,ns)}
56         News getNews(){var News firstNews := head(requests); requests := tail(requests); put firstNews}
57         List<News> getRequests(){put requests}}
58
59 class Client implements ClientI{
60         News news := None;
61         Void signal(News ns){news := ns}}
```

We have here augmented the given core language with ABS syntax for data types.