A Complete Guide to the Future *

Frank S. de Boer [1], Dave Clarke [1], and Einar Broch Johnsen [2]

[1] CWI, Amsterdam, Netherlands
{frb,dave}@cwi.nl

[2] Dept. of Informatics, University of Oslo, Norway
einarj@ifi.uio.no

Abstract We present the semantics and proof system for an object-oriented language with active objects, asynchronous method calls, and futures. The language, based on Creol, distinguishes itself in that unlike active object models, it permits more than one thread of control within an object, though, unlike Java, only one thread can be active within an object at a given time and rescheduling occurs only at specific release points. Consequently, reestablishing an object's monitor invariant is possible at specific well-defined points in the code. The resulting proof system shows that this approach to concurrency is simpler for reasoning than, say, Java's multithreaded concurrency model. From a methodological perspective, we identify constructs which admit a simple proof system and those which require, for example, interference freedom tests.

1 Introduction

The increasing importance of distributed systems demands flexible communication forms between distributed processes. While object-orientation is a natural paradigm for distributed systems [17], the tight coupling between objects traditionally enforced by method calls may be criticized. *Asynchronous method calls* have been proposed to better combine object-orientation with distributed programming, with a looser coupling between a caller and a callee than in the tightly synchronized (remote) method invocation model. Return values from asynchronous calls are managed by so-called *futures* [4, 10, 13, 20, 26]. In this paper, we develop a kernel language for distributed concurrent objects in which asynchronous method calls is the basic communication construct. The model of asynchronously communicating objects is inherently concurrent, and synchronized communication and sequential execution appear as special cases. The proposed kernel language combines the concurrency model of Creol [18], an object-oriented language for concurrent objects, with first-class futures, presented in a Java-like syntax. Futures are not transparent but may be communicated between objects, so return values from asynchronous method calls may be shared. The paper presents an operational semantics for this kernel language, and introduces a novel proof system for concurrent objects with asynchronous method calls and futures.

* This research is in the context of the EU project IST-33826 CREDO: Modeling and analysis of evolutionary structures for distributed services (http://credo.cwi.nl).

In: R. De Nicola (Ed.): Proc. 16th European Symposium on Programming (ESOP'07).
To appear in LNCS, Springer 2007.

The adopted concurrency model is based on concurrent objects, each with its own processor. Inside an object, method activations are executed in an interleaved way. Thus execution in an object is reminiscent of monitors, but explicit signaling is avoided by introducing so-called *release points* at which control may change between different method activations competing for execution. The interleaved execution of method activations allows different activities to be pursued within the object; in particular, active and reactive object behavior are easily and dynamically combined. Whereas an active object usually relies on a preselected method to define its active behavior, we exploit asynchronous method calls as *triggers of concurrent activity*. Asynchronous method calls spawn activities in other objects while the caller proceeds with its execution. Futures extend this technique to include the forwarding and sharing of replies to method calls. Each object sharing a future may choose to either completely block or alternatively to release control while waiting for the reply associated with the future. Any method may be called both synchronously and asynchronously. In fact, synchronous calls are treated as a special case of asynchronous calls, for which execution immediately blocks while waiting for the reply. Thus, synchronous calls restrict the natural concurrency of the model by sequentializing activity.

Proof theories for multithreaded object systems are complicated by the interference problem for shared variables, which appears when threads operate concurrently in the same object. Reasoning about programs in this setting is highly complex [1]: Safety is by convention rather than by language design [3]. The simplicity of the proof system proposed in this paper, in contrast to that of, for example, multithreaded Java, is a major advantage of concurrent object models compared to multithread concurrency. The proposed proof system uses a local assertion language to describe the local state of an object in the pre- and postconditions of methods and in monitor invariants. On the other hand, a global assertion language is used for describing invariant properties of inter-object synchronization. In this paper, we present a novel view of an object as a maintainer of *multiple* local monitor invariants and a global synchronization constraint. The local invariants monitor the different release points of an object. These multiple monitor invariants require a novel proof system for their mutual dependencies to establish their invariance. This clear separation of concerns between intra- and inter-object synchronization is also reflected in the completeness proof for the proof theory. In fact, the completeness proof (only briefly discussed in this paper due to lack of space) is based on a semantic characterization of the global invariant in terms of futures and two local history variables. In addition to a local communication history, recording the externally observable behavior of each object as specified by its method calls, a local scheduling history records the internal scheduling in an object, which is completely encapsulated by its local invariants, recording snapshots of the corresponding release points.

Paper overview. Sect. 2 introduces the kernel language and its operational semantics and Sect. 3 provides an example. Sect. 4 introduces the assertion language, Sect. 5 a proof system for concurrent objects with asynchronous method calls and futures, Sect. 6 discusses related work and Sect. 7 concludes the paper.

$$
\begin{array}{lll}
P & ::= \overline{L}\ \{\overline{T}\ \overline{x}; sr\} & L ::= \texttt{class}\ C\ \texttt{extends}\ C\ \{\overline{T}\ \overline{f}; \overline{M}\} \\
M & ::= T\ m\ (\overline{T}\ \overline{x})\{\overline{T}\ \overline{x}; sr\} & e ::= v \mid e.\texttt{get} \mid e!m(\overline{e}) \mid \texttt{new}\ C() \mid \texttt{null} \\
v & ::= f \mid x & s ::= v := e \mid \texttt{await}\ g \mid s \,\square\, s \mid s \parallel s \mid \texttt{skip} \mid s; s \\
 & & \quad \mid \texttt{if}\ g\ \texttt{then}\ s\ \texttt{else}\ s\ \texttt{fi} \mid \texttt{release} \mid s \,/\!/\!/\, s \\
sr & ::= s; \texttt{return}\ e & g ::= \texttt{wait} \mid b \mid v? \mid g \wedge g \\
b & ::= \texttt{true} \mid \texttt{false} \mid v & T ::= C \mid \texttt{bool} \mid !T
\end{array}
$$

Figure 1. The language syntax. Variables v are fields (f) or local variables (x), and C is a class name.

2 The Language

A kernel language for distributed concurrent objects with asynchronous method calls and futures is now introduced, extending the syntax of Featherweight Java [16]. In contrast to Featherweight Java, each object encapsulates its state; i.e., external manipulation of the state is via the object's methods only. Furthermore different objects execute concurrently: each object has a thread dedicated to executing its processes, which correspond to activations of its methods. To preserve an object's invariants for reasoning control, execution is restricted so that only one process may be active in an object at a time; other processes in the object are *suspended*. We distinguish between *blocking* a process and *releasing* a process. Blocking suspends process execution, but does not relinquish control to a suspended process. Release stops process execution and reschedules another (suspended) process. Using release points within method bodies, an object may interleave the execution of several (non-terminating) processes.

Method calls are asynchronous and the result of a call is stored in a *future*. Rather than forcing the caller to *wait* for the call to return, which is unsatisfactory in a distributed setting where communications may disappear and permanently block the caller's process, return values are first accessed when required. Execution only blocks when attempting to read from a future without a return value. Futures may also be polled, enabling fine grained control of scheduling. In contrast to the read operation on a future, the polling operation never blocks.

The implicit control flow in an object can be influenced by means of release points expressed as Boolean guards, which may include the polling of futures. This way, processes may choose between blocking and releasing control while waiting for the reply to a method call. Release points can be used to combine active and reactive processes in an object; the object can behave both as client and server without requiring an active loop to interleave these different roles.

2.1 Syntax

The language syntax is given in Fig. 1. We emphasize the differences with Java. A program P is a list of class definitions followed by a method body. A class inherits from a superclass, which may be `Object`, extending it with additional

$$
\begin{array}{rlrl}
\mathit{config} & ::= \epsilon \mid \mathit{object} \mid \mathit{future} \mid \mathit{config}\ \mathit{config} & o & ::= (\mathit{oid}, C) \\
\mathit{object} & ::= (o, \mathit{processQ}, \mathit{fds}, \mathit{active}) & \mathit{fds} & ::= \overline{f}\ \overline{v} \\
\mathit{future} & ::= (\mathit{mid}, \mathit{mc}, \mathit{mode}, v) & \mathit{mc} & ::= \mathit{oid}.m(\overline{v}) \\
\mathit{active} & ::= \mathit{process} \mid \texttt{idle} & \mathit{process} & ::= (\overline{T}\ \overline{x}\ \overline{v}, \mathit{sr} : T) \\
\mathit{processQ} & ::= \epsilon \mid \mathit{process} \mid \mathit{processQ}\ \mathit{processQ} & v & ::= \mathit{oid} \mid \mathit{mid} \mid \texttt{null} \mid b
\end{array}
$$

Figure 2. The syntax for runtime configurations. Here, *oid* and *mid* denote identifiers for objects, and futures. Processes include both the types of local variables and the expected return type (which we often elide for simplicity of presentation).

fields f and methods M. Methods have read-only access to a variable `destiny` which is a reference to the future that will hold the result of the current method.

Expressions e are standard apart from the asynchronous method call $e!m(\overline{e})$ and the (blocking) read operation $v.\texttt{get}$. *Statements* s are standard apart from release points `await` g, non-deterministic choice $s_1 \,\square\, s_2$, and merge $s_1 \| s_2$ for the interleaved execution of branches s_1 and s_2. *Guards* g are conjunctions of `wait`, Boolean expressions b, and the polling operation $v?$ on a future v. When the guard in an `await` statement evaluates to `false`, the active process is released and another suspended process may be rescheduled. Otherwise, the process proceeds. Non-deterministic choice allows either branch to be selected. The branches of a merge are interleaved at release points, influencing the flow of control within a process without allowing other processes to execute. In addition, the intermediate statements `release` and $s_1 /\!/\!/ s_2$ appear during reduction. The `release` statement is introduced when the guard of an `await` statement reduces to `false`, and the $s_1 /\!/\!/ s_2$ statement corresponds to the activation of statement s_1 in the merge of statements s_1 and s_2, where statement s_2 is delayed.

Typing. The type system, omitted for space reasons, closely resembles that of Featherweight Java [16]. Let $!T$ denote the type of a future which will ultimately contain a value of type T. An asynchronous call to a method with return type T results in a future of type $!T$. If v has type $!T$, then $v.\texttt{get}$ has type T and $v?$ has type `bool`. Type soundness is easily established for this type system and the reduction semantics presented in Sect. 2.2 below.

2.2 Semantics

The semantics is a small-step reduction relation on *configurations* of objects and futures (see Fig. 2). *Objects* have an identifier, a class, a queue of suspended processes, fields, and an active process. The process `idle` indicates that no method is running in the object. A *future* captures the state of a method call: initially *sleeping*, the method call later becomes *active*, and finally, when *completed*, it stores its result in the future. The value $\mathit{mode} \in \{\texttt{s}, \texttt{a}, \texttt{c}\}$ represents these three future states. Types are given default values by the *default* function (e.g., $\mathit{default}(C) = \texttt{null}$, $\mathit{default}(\texttt{bool}) = \texttt{false}$, and $\mathit{default}(!T) = \texttt{null}$). The *initial configuration* of a program $\overline{L}\ \{\overline{T}\ \overline{x}; \mathit{sr}\}$ has one object $(o, \emptyset, \emptyset, (\overline{T}\ \overline{x}\ \mathit{default}(\overline{T}), \mathit{sr} : T))$.

Reduction takes the form of a relation $\mathit{config} \to \mathit{config}'$. Rules apply to partial configurations and may be applied in parallel. This differs from the semantics of object-oriented languages with a global store [11], but is consistent with the Creol's [18] executable semantics in Maude [5], and allows true concurrency in the distributed setting. The main rules are given in Fig. 3. The context reduction semantics decomposes a statement into a reduction context and a redex, and reduces the redex [9]. *Reduction contexts* are method bodies M, statements S, expressions E, and guards G with a single hole denoted by $\bullet$:

$$
\begin{aligned}
M &::= \bullet \mid S; \texttt{return}\ e \mid \texttt{return}\ E \\
S &::= \bullet \mid v := E \mid S; s \mid \texttt{if}\ G\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi} \mid S \mathbin{/\!/\!/} s \\
E &::= \bullet \mid E.\texttt{get} \mid E!m(\overline{e}) \mid v!m(\overline{v}, E, \overline{e}) \\
G &::= \bullet \mid E? \mid G \wedge g \mid b \wedge G
\end{aligned}
$$

Redexes reduce in their respective contexts; i.e., body-redexes in M, stat-redexes in S, expr-redexes in E, and guard redexes in G. Redexes are defined as follows:

$$
\begin{aligned}
\textit{body-redexes} &::= \texttt{return}\ v \\
\textit{stat-redexes} &::= x := v \mid f := v \mid \texttt{await}\ g \mid s \,\square\, s \mid \texttt{skip}; s \mid \texttt{if}\ b\ \texttt{then}\ s\ \texttt{else}\ s \\
&\quad \mid s \mathbin{|||} s \mid \texttt{skip} \mathbin{/\!/\!/} s \mid \texttt{release}; s \mathbin{/\!/\!/} s' \mid \texttt{release} \\
\textit{expr-redexes} &::= x \mid f \mid v.\texttt{get} \mid v.m!(\overline{v}) \mid \texttt{new}\ C() \\
\textit{guard-redexes} &::= \mathit{mid}? \mid b \wedge g \mid \texttt{wait}
\end{aligned}
$$

Filling the hole of a context M with an expression r is denoted $M[r]$. Before evaluating the expression e in the method body $s; \texttt{return}\ e$, the body will be reduced to $\texttt{skip}; \texttt{return}\ e$. For simplicity, we elide the `skip` and write just $\texttt{return}\ e$.

Expressions and guards. In (Red-Call), an asynchronous call adds a sleeping future to the configuration, returning its identifier to the caller. In (Red-Get), a read operation on a future variable blocks the active process until the future is in completed mode. Blocking does not reschedule a suspended process. Object creation in (Red-New) introduces a new instance of a class into the configuration, with default values for the new object's fields. Guards determine if a process should be released and another process rescheduled. In (Red-Poll), a future variable is polled to see if a call has been executed. In contrast to (Red-Get), polling a future at a release point (`await`) enables the release of the active process. In particular, `await wait` will always release the active process.

Statements and rescheduling. In (Red-Await), a process at a release point proceeds if its guard is true and otherwise releases. When a process is released, its guard is reused to reschedule the process. A guard with clause `wait` causes a process to release. When it becomes a candidate for rescheduling, `wait` is replaced by `true` so that the process can proceed. When an active process is released or terminates, it is replaced by the `idle` process, which allows a process from the process queue to be scheduled for execution in (Red-Reschedule).

(Red-Merge1)
$(o, pq, fds, (l, M[s \| s'])) \rightarrow (o, pq, fds, (l, M[s /// s']))$

(Red-Merge2)
$(o, pq, fds, (l, M[s \| s'])) \rightarrow (o, pq, fds, (l, M[s' /// s]))$

(Red-Merge-Skip)
$(o, pq, fds, (l, M[\texttt{skip} /// s])) \rightarrow (o, pq, fds, (l, M[s]))$

(Red-Call)
$$\frac{mid \text{ is fresh}}{(o, pq, fds, l, (M[oid!m(\overline{v})])) \rightarrow (o, pq, fds, (l, M[mid])) \; (mid, oid.m(\overline{v}), \texttt{s}, \texttt{null})}$$

(Red-New)
$$\frac{oid \text{ is fresh} \quad fds' = defaults(C)}{(o, pq, fds, (l, M[\texttt{new } C()])) \rightarrow (o, pq, fds, (l, M[oid])) \; ((oid, C), \epsilon, fds', (\epsilon, \texttt{skip}))}$$

(Red-Merge-Release1)
$$\frac{enabled(s', (fds, l), \mu)}{(o, pq, fds, (l, M[\texttt{release}; s /// s'])) \; \mu \rightarrow (o, pq, fds, (l, M[s' /// s])) \; \mu}$$

(Red-Merge-Release2)
$$\frac{\neg enabled(s', (fds, l), \mu)}{(o, pq, fds, (l, M[\texttt{release}; s /// s'])) \; \mu \rightarrow (o, pq, fds, (l, M[\texttt{release}; (s \| s')])) \; \mu}$$

(Red-Get)
$(o, pq, fds, (l, M[mid.\texttt{get}])) \; (mid, mc, \texttt{c}, v) \rightarrow (o, pq, fds, (l, M[v])) \; (mid, mc, \texttt{c}, v)$

(Red-Release)
$$\frac{M[\texttt{release}] \neq M'[\texttt{release}; s /// s']}{(o, pq, fds, (l, M[\texttt{release}])) \rightarrow (o, pq :: (l, M[\texttt{skip}], fds, \texttt{idle}))}$$

(Red-Poll)
$$\frac{b = (mode \equiv \texttt{c})}{(o, pq, fds, (l, M[mid?])) \; (mid, mc, mode, v) \rightarrow (o, pq, fds, (l, M[b])) \; (mid, mc, mode, v)}$$

(Red-Wait)
$(o, pq, fds, (l, M[\texttt{wait}])) \rightarrow (o, pq, fds, (l, M[\texttt{false}]))$

(Red-Await)
$$\frac{g' = g[\texttt{true}/\texttt{wait}]}{(o, pq, fds, (l, M[\texttt{await } g])) \rightarrow (o, pq, fds, (l, M[\texttt{if } g \texttt{ then skip else release; await } g' \texttt{ fi}]))}$$

(Red-Reschedule)
$(o, p :: pq, fds, \texttt{idle}) \rightarrow (o, pq, fds, p)$

(Red-Bind)
$$\frac{mbody(m, C) = (\overline{T} \; \overline{x}, \overline{U} \; \overline{y}, sr : T) \quad l = \overline{T} \; \overline{x} \; \overline{v}, \overline{U} \; \overline{y} \; default(\overline{U}), !T \; \texttt{destiny} \; mid \quad q = (l, sr : T)}{((oid, C), pq, fds, p) \; (mid, oid.m(\overline{v}), \texttt{s}, \texttt{null}) \rightarrow ((oid, C), pq :: q, fds, p) \; (mid, oid.m(\overline{v}), \texttt{a}, \texttt{null})}$$

(Red-Return)
$$\frac{l(\texttt{destiny}) = mid}{(o, pq, fds, (l, \texttt{return } v : T)) \; (mid, oid.m(\overline{v}), \texttt{a}, \texttt{null}) \rightarrow (o, pq, fds, \texttt{idle}) \; (mid, oid.m(\overline{v}), \texttt{c}, v)}$$

(Red-Context)
$$\frac{config \rightarrow config'}{config \; config'' \rightarrow config' \; config''}$$

(Red-Parallel)
$$\frac{config \; \mu \rightarrow config' \; \mu' \qquad config'' \; \mu \rightarrow config''' \; \mu'' \quad \mathrm{dom}(\mu) = \mathrm{dom}(\mu') = \mathrm{dom}(\mu'') \qquad \mathrm{dom}(config') \cap \mathrm{dom}(config''') = \emptyset}{config \; config'' \; \mu \rightarrow config' \; config''' \; \mu' \odot \mu''}$$

Figure 3. The context reduction semantics. μ denotes a configuration of futures.

Method invocation and return. A method call results in an activation on the callee's process queue. As the call is asynchronous, there is a delay between the call and its activation, represented by the sleeping mode of a future. Subsequent to the call, (Red-Bind) creates a process to run the method. This process is added to the process queue, and the future changes its mode to active, thus preventing multiple activations. When process execution is completed, the return value is stored by (Red-Return) in the future identified by the `destiny` variable. This future changes its mode to completed and the active process becomes `idle`.

Merge and release. Either branch of a merge may be selected for reduction, captured by (Red-Merge1) and (Red-Merge2). When a branch of a merge statement completes, (Red-Merge-Skip) schedules the other branch. If a release occurs inside a merge, the other branch of the merge is the first candidate for rescheduling — rescheduling is local to a process whenever possible. If both branches release, then the process is released. Let σ map fields and local variables to their values. Process release is based on the predicate *enabled* defined on guards, futures, and states which determines whether a guard will not directly release:

$$\begin{aligned} \mathit{enabled}(\texttt{wait}, \sigma, \mu) &= \mathit{false} \qquad\qquad \mathit{enabled}(b, \sigma, \mu) \;=\; b \\ \mathit{enabled}(v, \sigma, \mu) &= \mathit{enabled}(\sigma(v), \sigma, \mu) \\ \mathit{enabled}(\mathit{mid}?, \sigma, \mu) &= \mathit{mode} \equiv \texttt{c}, \text{ where } (\mathit{mid}, \_, \mathit{mode}, \_) \in \mu \\ \mathit{enabled}(g \wedge g', \sigma, \mu) &= \mathit{enabled}(g, \sigma, \mu) \wedge \mathit{enabled}(g', \sigma, \mu) \end{aligned}$$

The predicate is lifted to statements; $\mathit{enabled}(\texttt{await}\ g, \sigma, \mu) = \mathit{enabled}(g, \sigma, \mu)$ is the crucial case. In (Red-Merge-Release1), (Red-Merge-Release2) and (Red-Release), the contexts and redexes do not factor expressions involving `release` uniquely: these may be factored as both $M[\texttt{release}]$ and $M'[\texttt{release}; s /\!/\!/ s']$. A clause is added to (Red-Release) to ensure that $\texttt{release}; s /\!/\!/ s'$ is preferred.

Context and parallel reductions. A reduction applies to a subconfiguration by rule (Red-Context). In (Red-Parallel) futures may be shared between concurrent reductions, increasing the amount of concurrency expressible in the rules. As the futures witnessed by one process may be changed by another, they need to be recomposed in a consistent way. This is handled by a function $\mu \odot \mu'$ which collects futures from μ and μ' and resolves conflicting futures with the same *mid*. New futures are located in config' and config'''.

Synchronization and self-calls. Reading (`get`) a future is blocking and can introduce synchronization points in the code; for example, the statements $y = e!m(\overline{e}); y.\texttt{get}$ model the usual notion of synchronous method call, as this code blocks the active process after making a call to y until the call has completed. A minor problem arises when we wish to perform a synchronous call to self, $\texttt{this}.m(\overline{e})$: the statements $y = \texttt{this}!m(\overline{e}); y.\texttt{get}$ lead to deadlock. In order to execute a local method, the process needs to be released, as in the sequence $y = \texttt{this}!m(\overline{e}); \texttt{await}\ y?; z = y.\texttt{get}$. This sequence, however, does not capture the direct transfer of control as it enables any other blocked process in the object to be activated before the call to m. This ultimately means that the language needs an extension to handle synchronous self calls. A solution is proposed in [18].

3 An Example

We present a publisher-subscriber example wherein an event observed by a sensor is published to objects subscribed to a service. To avoid bottlenecks when publishing an event, the service delegates to a chain of proxy objects, where each proxy object informs both the next proxy and up to `limit` subscribing clients. We assume these classes exist: `Sensor` with method `detectEvent`, `Client` with method `signal`, and `List<T>`, parametric in type `T`, with method `add`.

```
class Service {
  Sensor sensor;  Proxy proxy;
  Service(int val) {                              // constructor
    sensor = new Sensor; proxy = new Proxy(val);
  }
  void subscribe(Client cl) { proxy.add(cl) }  // sync. call
  void process() {
    while (true) {
      !Event fut = sensor!detectEvent();
      proxy!publish(fut);                       // async. call
      await fut?;
} } }

class Proxy {
  List<Clients> myClients;  Proxy nextProxy;
  Event ev;  int limit;
  Proxy(int k) {                                  // constructor
    limit = k; myClients = new List();  nextProxy = null;
  }
  void add(Client cl) {
    if (myClients.length < limit) { myClients.add(cl); }
    else { if (nextProxy == null) nextProxy = new Proxy(limit);
           nextProxy.add(cl); }
  }
  void publish(!Event fut) {
    await fut?;
    if (nextProxy != null) { nextProxy!publish(fut); }
    ev = fut.get();
    for (Client client : myClients) { client!signal(ev); }
} }                                                  // notify clients
```

4 The Assertion Language

Assertions are used to specify (invariant) properties of the configurations occurring during computations generated by the operational semantics defined in Sect. 2.2. Assertions are constructed from expressions e of the following form:

$$e ::= z \mid z.f \mid ops(\bar{e})$$

Here z can be `this`, a local variable, or a *logical* variable. Logical variables are implicitly universally quantified. The expression $z.f$ denotes the value of the field f of the object denoted by the variable. By *ops* we mean an operation of some given abstract data type. Assertions are Boolean combinations of Boolean expressions.

In order to reason about the invocation and return of asynchronous method calls, futures are explicitly modeled as objects. Thus reasoning relies on an encoding of method calls $oid.m(\overline{e})$. Conceptually, a class representing futures is introduced for every method in the program. For every possibly inherited method m of a class C, we associate a class `Future_C_m`, with instance variables to store the callee, the actual parameters, the mode, and the return value of a call to the method. Given this class, the future $(mid, oid.m(\overline{e}), mode, v)$ corresponding to a method call of a method m of an object oid of class C, is denoted by the instance $((mid, \texttt{Future\_C\_m}), fds)$, where $fds = \texttt{callee} \mapsto oid, \overline{\texttt{arg}} \mapsto \overline{e}, \texttt{mode} \mapsto mode, \texttt{val} \mapsto v$. Note that we assume that some encoding of an enumerated type with elements sleeping, active, and completed exists.

As a simple example, the assertion $z.\texttt{mode} = \texttt{c} \rightarrow z.\texttt{v} > 0$, where z is an (implicitly) universally quantified logical variable ranging over all existing instances of `Future_C_m`, states that every completed instance of `Future_C_m` stores a positive integer, or, in other words, that every completed invocation of the method m (executed by an object of type C) has returned a positive integer. In a similar manner we can express invariant properties of the actual parameters (stored in the future objects).

5 The Proof System

The proof system consists of rules for proving that a local pre/postcondition specification $\{p\}s\{q\}$ is correct with respect to a global invariant I and a set of monitor invariants. Here p and q are *local* assertions which only use expressions that refer to the local state of an object via `this`; i.e., its fields and the local variables of one of its methods. Such assertions describe local properties of an object; i.e., properties which are invariant over the executions of the other objects. The global invariant describes invariant properties of the future objects, which form the shared data structure that models the (asynchronous) interaction between objects. The global invariant only refers to the future objects and their fields, and as such is only affected by operations on futures. Monitor invariants are local assertions associated with `await` statements that describe local properties of an object which hold whenever the (associated) `await` statement is scheduled.

In order to reason about statements involving futures, i.e., the basic statements $r := e!m(\overline{e})$, $v := r.\texttt{get}$, and `await` $r?$, we encode their operational semantics as described in detail below. This encoding allows the application of a (standard) weakest precondition calculus (as described by the first author [6], which takes into account aliasing and object creation).

The following proof rule derives the local specification of an asynchronous method invocation: $r := e!m(\bar{e})$:

$$\frac{\{p \wedge I\} r := \texttt{new Future\_C\_m}(e, \bar{e})\{q \wedge I\}}{\{p\} r := e!m(\bar{e})\{q\}}$$

The premise of this rule is a specification which additionally establishes invariance of the global invariant I over the statement $r := \texttt{new Future\_C\_m}(e, \bar{e})$. This statement consists of a call to the constructor method of the class `Future_C_m`, uniquely determined by the method m. This constructor method initializes the newly created future object such that $fds = \texttt{callee} \mapsto e, \overline{\texttt{arg}} \mapsto \bar{e}, \texttt{mode} \mapsto \texttt{s}, \texttt{val} \mapsto \texttt{null}$, encoding the reduction rule (Red-Call). As a simple example, the above assertion $z.\texttt{mode} = \texttt{c} \rightarrow z.\texttt{v} > 0$ is invariant because the value of `mode` of the newly created object is set to `s`.

The local specification of $v := r.\texttt{get}$ is captured by the following proof rule, corresponding to reduction rule (Red-Get):

$$\frac{\{p \wedge r.\texttt{mode} = \texttt{c} \wedge I\} v := r.\texttt{val}\{q \wedge I\}}{\{p\} v := r.\texttt{get}\{q\}}$$

The precondition of the premise additionally requires that the future r is indeed completed, and thus stores the return value. As an example, the assertion $r.\texttt{mode} = \texttt{c} \rightarrow r.\texttt{v} > 0$ can be used in conjunction with the additional information $r.\texttt{mode} = \texttt{c}$ to establish $v > 0$ as a postcondition to the `get` operation.

The following proof rule captures the specification of a statement $\texttt{await}\ r?$:

$$\frac{(i \wedge r.\texttt{mode} = \texttt{c} \wedge I) \rightarrow q}{\{i\}\texttt{await}\ r?\{q\}}$$

Here i denotes the monitor invariant (implicitly) associated with the `await` statement. The premise consists of an implication establishing the postcondition in case the return value is stored in the future object r.

The proof rule for deriving a local specification of a method definition is:

$$\frac{\begin{array}{c}\{p \wedge \texttt{this} = \texttt{d.callee} \wedge \texttt{d.mode} = \texttt{s} \wedge I\}\texttt{d.mode} := \texttt{a}; \overline{x} := \texttt{d.}\overline{\texttt{arg}}\{p' \wedge I\} \\ \{p'\}s\{q'\} \\ \{q' \wedge I\}\texttt{d.mode} := \texttt{c}; \texttt{d.val} := e\{q \wedge I\}\end{array}}{\{p\}s; \texttt{return}\ e\{q\}}$$

Here we denote by `d` the (distinguished) destiny variable of the method used to denote its future object. The rule establishes the invariance of I over the statements for retrieving the arguments to the call and for returning the value of e, encoded as $\texttt{d.mode} := \texttt{a}; \overline{x} := \texttt{d.}\overline{\texttt{arg}}$ and $\texttt{d.mode} := \texttt{c}; \texttt{d.val} := e$, encoding reduction rule (Red-Return). The additional information $\texttt{this} = \texttt{d.callee} \wedge \texttt{d.mode} = \texttt{s}$ ensures that this future object indeed records a sleeping (that is, not yet activated) method call to this callee. Note that in general the invariant I in the first premise will be used to validate the assumptions about the formal parameters expressed by the precondition p' of the method body.

The rules for the remaining statements are now briefly discussed. For example, we have the following assignment axiom $\{p[e/v]\}v := e\{q\}$. Since p and q are local assertions we do not have aliasing. Therefore the substitution $[e/v]$ simply replaces occurrences of v by e, provided e does not involve object creation. Assuming that in local assertions a variable v of type C can only be compared for equality, we can perform a simple contextual analysis of occurrences of v in the case of an assignment $v := \texttt{new}\ C()$. For example, the assertion $(v = e)[\texttt{new}/v]$ is false for every expression e (other than v) because e will denote an 'old' object (for details, see [6]). The rules for sequential composition, non-deterministic choice, and the conditional are standard. The rule for the merge statement involves an adaptation of the usual interference freedom test for shared variable concurrency (with the significant simplification that in our setting this test is local to the merge statement).

The proof system is used to prove that a class C maintains a set M of monitor invariants which describe its release points. The verification of these (local) monitor invariants involves a global invariant I and consists of proving that each $i \in M$ is invariant over each method body of C.[1] Formally, for every method body $s; \texttt{return}\ e$ we have to prove that the specification

$$\{i \wedge \texttt{d}' \neq \texttt{d}\}s; \texttt{return}\ e\{i\}$$

is correct with respect to the global invariant I and the following extended monitor invariants of its `await` statements: $i \wedge \texttt{d}' \neq \texttt{d} \wedge j$, where $j \in M$ is the monitor invariant of a given `await` statement in s. The additional information $\texttt{d}' \neq \texttt{d}$ expresses that we are dealing with two different method invocations, each with its own future represented by their local variables $\texttt{d}'$ and `d`.

5.1 Soundness and Completeness

For completeness we need to introduce the usual notion of *auxiliary* variables to validate global synchronization constraints. Auxiliary variables extend the local state of objects in order to record certain observations about internal scheduling, sending a message, setting a return value, and about the method activations themselves. A soundness proof then consists of a straightforward but tedious induction on the length of the computation which *atomically* executes the statements and their associated updates of the auxiliary variables. Conversely, completeness amounts to showing that assertions describing reachable configurations 'follow the rules' of the proof system. These assertions describe the external observable behavior of an object by means of a so-called *communication history variable*, an auxiliary variable which denotes the sequence of generated messages and which is updated by each method call, upon each method activation, and by each return statement. Furthermore, in order to reason about the internal process queue, we introduce a so-called *scheduling history variable*, an auxiliary

[1] To avoid name clashes between the local variables of i and the method body, we assume a variable convention wherein we rename the local variables x of i (and we denote its renamed local variables by x').

variable which records the local state (i.e., the current values of the local variables and the values of the fields) whenever the guard of an `await` statement is evaluated. Together these auxiliary history variables fix the internal computation of an object. This semantic property of these auxiliary variables forms the heart of the completeness proof (due to space limitations details are omitted).

6 Related Work

Futures were devised as a simple means for expressing concurrency in a manner that reduced the dependency on latency by enabling synchronization at the latest possible time. Futures were discovered by Baker and Hewitt in the 70s [14], and later rediscovered by Liskov and Shrira as Promises [20] and by Halstead in the context of MultiLisp [13]. Futures appear in languages like Alice [23], Oz-Mozart [24], Concurrent ML [22], C++ [19] and Java [25], often as libraries. Futures in these languages are essentially the same as in our language.

All implementations associate a future with the asynchronous execution of an expression in a new thread. The future is a placeholder object which is immediately returned to the calling site. From the perspective of the calling site, this placeholder is a read-only structure [21]. In some systems, this placeholder can be explicitly manipulated by the programmer in order to write the resulting data. In many implementations of futures, the placeholder can be accessed in both modes (CML, Alice, Java, C++, etc), though typically the design is such that both interfaces are presented separately — one to the caller and one to the callee. The calculus $\lambda(\mathsf{fut})$ [21] formalizes this distinction. Programming with promises explicitly is quite low-level, so our language ties writing the resulting value with method call return.

Futures can either be transparent or non-transparent. Transparent futures cannot be explicitly manipulated, the type of the future is the same as the expected result, and accesses made to the future transparently access the result stored in the future, possibly after waiting (e.g., in Multilisp). Non-transparent futures have a separate type to denote the future (e.g., $!T$ is a future of type T), and future objects can be manipulated (e.g., in CML, Alice, Java, C++, and our language). In addition, futures can also be dealt with lazily to give the effect of *call-by-need* computation, by delaying the invocation of the asynchronous computation until the moment when the future is accessed (e.g., in Alice).

Flanagan and Felleisen [10] present different semantic models of futures at various levels of abstraction in terms of an abstract machine. Their goal was to enable optimizations and program analyses. Their language was purely functional in contrast to ours, which is an imperative, object-oriented language.

Caromel, Henrio, and Serpett [4] present an imperative, asynchronous object calculus with transparent futures. Their active objects may have internal passive objects which can be passed between active objects by first deep copying the entire (passive) object graph. We do not provide this feature, which is orthogonal to the issue discussed in this paper. To manage the complexity of reasoning about distributed and concurrent systems, they restrict the language to ensure

that reduction is confluent and deterministic, whereas our focus is on preserving object invariants. No proof theory is presented for their calculus.

Actor systems [2] are concurrent processes which communicate exclusively through asynchronous messages. An actor encapsulates its fields, procedures that manipulate the state, and a single thread of control. Our objects are similar to actors, except that our methods return values which are managed by futures, and control can be released at specific points during a method execution. Messages to actors return no result and run to completion before another message can be handled. The lack of return makes programming with actors cumbersome.

Proof systems for actor languages exist [8], but these require explicit structures in the proof rules for reasoning about message queues, which our proof theory avoids. Previous work by the third author [7] on the verification of asynchronous method calls was performed in a language without first-class futures. The paper took a transformational approach by encoding the language into a sequential language with a non-deterministic assignment operator. However, the Hoare rules described only the custom semantics. Various proof systems for monitors exist [12, 15]. Our approach is distinct as we present a novel model of an object that maintains multiple local invariants monitoring its release points and a global invariant that describes its interaction with the other objects via futures. The model is formalized and has a sound and complete proof theory.

7 Discussion and Future Work

We developed a formal model for a distributed, concurrent object-oriented language with asynchronous method calls and futures. The model allows a novel view of concurrent objects as maintainers of multiple local monitor invariants and a global synchronization constraint. Having multiple monitor invariants allows a proper treatment of local process variables. In contrast, monitor invariants in existing models only refer to an object's fields, complicating reasoning about local variables in the context of the non-deterministic scheduling.

Although our language enables polling of futures (*var*? in an `if` statement), which may be used to release the active process to allow flexible internal scheduling in an object, to control interference of local proofs, our proof system restricts polling to guards in `await` statements. We argue that this proof system is significantly simpler than for the proof system for Java, based on previous work of the first author [1]. Java's proof system requires thread variables, and a general interference freedom test due to the arbitrary interleaving of threads.

In fact, one can compare proof theories for various sublanguages of the language presented here. Let us start with a base language without polling and without release points `await`. Thus, `get` is the only operation on futures.

Adding await statements: The language without `await` does not require the two history variables introduced in the completeness of our proof system; the local scheduling history becomes superfluous. This language would have no internal rescheduling, rather it would resemble an actor-based language with futures for managing the returns of asynchronous calls. The `await` statements in our lan-

guage allow a clear separation of concerns between an object as maintainer of its monitor invariants and as maintainer of the global synchronization constraint. This is clearly reflected in the completeness proof where the monitor invariance only describes the internal scheduling of the `await` statements, whereas the global synchronization constraint expresses the externally observable behavior.

Adding polling to conditionals significantly increases the complexity of the proof system. Interference freedom tests are necessary [1], because the required information about the absence of return values can be invalidated by other objects. In contrast, reasoning about `await` statement with polling only requires information about the presence of reply values, which cannot be invalidated.

This suggests that either the concurrency features of programming languages should be chosen to admit a simple proof system, or that the complexity of programming with such features should be measured in terms of the complexity of their proof system, and that this should be made known to programmers.

Futures are now a part of a programmer's toolbox: Java's `util.concurrent` library supports futures, and futures handle asynchronous calls to web services. To facilitate correct programming, it is important to guide the design of language features using proof theoretical considerations. To properly support language design, both the soundness and completeness of the proof system are paramount.

In the context of the EU IST project Credo we are currently extending the existing Creol implementation [18] with additional rewrite rules for modeling futures.

References

1. E. Ábrahám, F. S. de Boer, W. P. de Roever, and M. Steffen. An assertion-based proof system for multithreaded Java. *TCS*, 331(2-3):251–290, 2005.
2. G. Agha and C. Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In *Research Directions in Object-Oriented Programming*, pages 49–74. MIT Press, 1987.
3. P. Brinch Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34(4):38–45, Apr. 1999.
4. D. Caromel, L. Henrio, and B. Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM Symposium on Principles of Programming Languages (POPL'04)*, pages 123–134. ACM Press, 2004.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, Aug. 2002.
6. F. S. de Boer. A WP-calculus for OO. In W. Thomas, editor, *Proceedings of Foundations of Software Science and Computation Structure, (FOSSACS'99)*, volume 1578 of *Lecture Notes in Computer Science*, pages 135–149. Springer, 1999.
7. J. Dovland, E. B. Johnsen, and O. Owe. Verification of concurrent objects with asynchronous method calls. In *Proceedings of the IEEE International Conference on Software Science, Technology & Engineering (SwSTE'05)*, pages 141–150. IEEE Computer Society Press, Feb. 2005.

8. C. H. C. Duarte. Proof-theoretic foundations for the design of actor systems. *Mathematical Structures in Computer Science*, 9(3):227–252, 1999.
9. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.
10. C. Flanagan and M. Felleisen. The semantics of future and an application. *J. Funct. Program.*, 9(1):1–31, 1999.
11. M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 241–269. Springer, 1999.
12. R. Gerth and W. P. de Roever. A proof system for concurrent ada programs. *Sci. Comput. Program.*, 4(2):159–204, 1984.
13. R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.
14. B. Henry G., Jr. and C. Hewitt. The incremental garbage collection of processes. In *Proceeding of the Symposium on Artificial Intelligence Programming Languages*, number 12 in SIGPLAN Notices, page 11, August 1977.
15. C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.
16. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.
17. International Telecommunication Union. Open Distributed Processing — Reference Model parts 1–4. Technical report, ISO/IEC, Geneva, July 1995.
18. E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, Nov. 2006.
19. R. G. Lavender and D. C. Schmidt. Active object: an object behavioral pattern for concurrent programming. *Proc. Pattern Languages of Programs*, 1995.
20. B. H. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In D. S. Wise, editor, *Proceedings of the SIGPLAN Conference on Programming Lanugage Design and Implementation (PLDI'88)*, pages 260–267, Atlanta, GE, USA, June 1988. ACM Press.
21. J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364:338–356, 2006.
22. J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.
23. A. Rossberg, D. L. Botlan, G. Tack, T. Brunklaus, and G. Smolka. *Alice Through the Looking Glass*, volume 5 of *Trends in Functional Programming*, pages 79–96. Intellect Books, Bristol, UK, ISBN 1-84150144-1, Munich, Germany, Feb. 2006.
24. P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Mar. 2004.
25. A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA'05)*, pages 439–453, New York, NY, USA, 2005. ACM Press.
26. A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86). Sigplan Notices*, 21(11):258–268, Nov. 1986.