JCoBox: Generalizing Active Objects to Concurrent Components

Jan Schäfer* and Arnd Poetzsch-Heffter

University of Kaiserslautern
{jschaefer,poetzsch}@cs.uni-kl.de

Abstract. Concurrency in object-oriented languages is still waiting for a satisfactory solution. For many application areas, standard mechanisms like threads and locks are too low level and have shown to be error-prone and not modular enough. Lately the actor paradigm has regained attention as a possible solution to concurrency in OOLs.
We propose JCoBox: a Java extension with an actor-like concurrency model based on the notion of concurrently running object groups, so-called *coboxes*. Communication is based on asynchronous method calls with standard objects as targets. Cooperative multi-tasking within coboxes allows for combining active and reactive behavior in a simple and safe way. Futures and promises lead to a data-driven synchronization of tasks.
This paper describes the concurrency model, the formal semantics, and the implementation of JCoBox, and shows that the performance of the implementation is comparable to state-of-the-art actor-based language implementations for the JVM.

1 Introduction

The Internet and the broad availability of multi-processors radically influence the way software has to be written [57,20]. On the one hand standard desktop programs have to deal with distribution aspects like network transmission delay and failure, on the other hand they must utilize multiple cores to scale in the future.

The object-oriented programming paradigm is widely used in practice and supported by industry-strength languages like Java and C#. These languages have built-in mechanisms for multi-threaded and distributed programming, but support for structuring and encapsulation of the state space, higher-level communication mechanisms and a common model for local and distributed concurrency is missing. Concurrency in these languages is introduced by dividing the control flow over a number of concurrently running threads working on a shared state. Threads are scheduled preemptively, allowing any thread to be suspended or activated at any time. To prevent threads from unwanted interleavings, low-level, basic synchronization concepts, like locks, have to be used. Experience shows that software written in such a way is prone to errors, difficult to debug, hard to maintain and to extend

* This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Methods

[57,31,50]. In addition, threads can freely cross component boundaries, making it difficult to modularly describe the behavior of a component and to maintain its invariants.

To motivate our approach and to relate it to other work (see Sect. 6 more details), we shortly discuss solutions that have been developed to overcome the mentioned problems.

As a core concurrency concept, the actor model [33,3,42] has regained attention as, in contrast to thread-based concurrency, it encapsulates control flow and data. Prominent examples are Erlang [6] based on a functional language background and Scala actors [29] integrated into a modern OOL. Combining actors with object-orientation comes with some challenging problems. One problem is that actors communicate by message passing instead of invoking methods. Thus, resulting systems are written with two incompatible communications mechanisms. In Scala actors, it is for example unsafe to simply call a method on an actor, as method calls are not protected by the actor. In addition, messages are often only dynamically typed, or require additional mailboxes to become statically type safe, further complicating the communication mechanism. Instead of using messages, asynchronous method calls provide a type safe communication mechanism, compatible with standard method calls, which is adopted by many active object approaches [13,14,11,10].

Object-oriented components are often realized by groups of interacting objects. Similar to many other component models, an OSGi component [60], for example, provides a number of so-called *services*, where each service can be realized by a different object. In contrast, typical active object approaches have single objects as the unit of concurrency, which makes it difficult to implement such components, as each service object must be an active object again, which cannot share state with the main component object. In some actor approaches, e.g. ASP [14], the state of an actor can be represented by multiple objects. These objects, however, cannot be referenced by other actors. This issue is solved in the E programming language [44] and AmbientTalk [61], where multiple objects can be referenced.

Most actor and active object approaches have in common that a single thread is responsible for executing the code inside an actor. This makes it difficult to have multiple independent control flows within an actor, which is important for two reasons: first, it is not easy to combine active with reactive behavior, and second, waiting for certain messages requires the actor to completely block the actor for other activities. Creol proposes a solution, namely to have multiple cooperatively scheduled tasks within a single active object [11].

Contributions. We developed a concurrency model and language that unifies existing solutions for the design problems with the following features:

- A concurrency model suitable for local and distributed concurrency.
- An appropriate integration of synchronous and asynchronous communication.
- A scaling support for components, in particular for components with several services provided by distinct objects.
- A partitioning of the object space into so-called coboxes such that each cobox can control local concurrency and maintain its invariants.

- A formal semantics that extends and simplifies that of an earlier version [55].
- An implementation of the language as an extension to a Java that can compete with existing actor implementations.

Overview. The paper continues with the description of the concurrency model and its realization as an extension of the sequential subset of Java (Sect. 2.1). Section 3 presents the formal semantics of the core fragment of JCoBox. Section 4 briefly explains the implementation of JCoBox. Section 5 evaluates the performance of JCoBox and summarizes practical experiences. Section 6 discusses related work, limitations, and future work. Finally, Section 7 concludes.

2 The Core Concepts

This section describes and motivates the core concepts of the cobox model, shows how they are realized as a Java extension, the so-called JCoBox language, and sketches additional language constructs provided by JCoBox.

2.1 The CoBox Model: Informal Description

The central concept of the cobox model is the *cobox*. A cobox can be considered as a container for objects. CoBoxes are concurrently running, isolated, object-oriented components. Figure 1 presents a schematic view of the cobox model. A cobox encapsulates both state and behavior and is in this sense similar to active objects [62,4,13]. The state of a cobox is constituted of a heap of objects. The cobox *owns* these objects for their entire lifetime. The behavior of a cobox is constituted of a set of *cooperative tasks*, which, again, are owned by the cobox for their entire lifetime.

CoBox-Local Computations. Inside a cobox, computations are similar to sequential object-oriented programming. All objects of a cobox can be directly accessed by accessing their fields or by invoking methods. Such *direct method calls* are immediately executed by the calling task in the standard stack-based way. To realize concurrency, a cobox supports multiple, possibly interleaved control flows, called *tasks*. Tasks are created when methods are asynchronously called on objects of a cobox and are responsible for executing the called method.

Tasks are scheduled *cooperatively*. This means that at most one task can be *active* in a cobox at a time, and that the active task has to give up its control explicitly to allow other tasks to become active. While a task is active, it has exclusive access to the cobox-local state. This allows the active task to reestablished invariants ranging over several objects of the cobox before it gives up control. If the active task never gives up control until it terminates, sequential execution of the task is enforced.

Tasks never leave a cobox; they stay in a cobox until they terminate. This is an important aspect, which differs from typical thread-based approaches, where threads are not bound to a component, but are allowed to leave and reenter components, which makes the externally visible behavior of such components very complex as it depends on concrete thread identifiers [2].

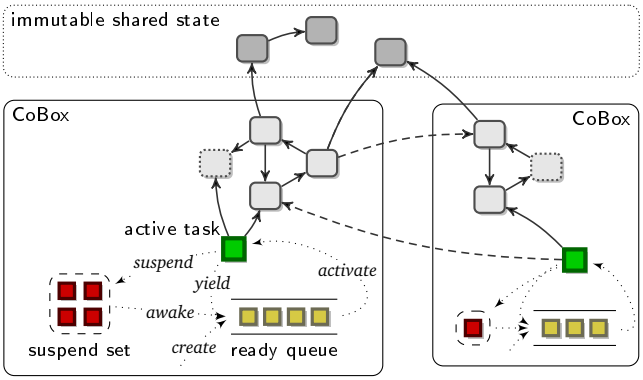

Fig. 1. Schematic view of the cobox model. Legend: standard objects, transfer objects, immutable objects, active task, ready tasks, suspended tasks, local reference, far reference task scheduling.

If a task is not active, it is either *ready* or *suspended*. Initially a task is in the ready state. Ready tasks are organized in a FIFO queue (the *ready queue*). If the active task gives up control the next task from the ready queue becomes active. A task can give up control in three ways: it can terminate, it can *yield*, which immediately adds it to the end of the ready queue, or it can *suspend*, waiting for some condition to be satisfied. When the condition of a suspended task becomes satisfied, the task is *awaked* and added to the ready queue, for eventual execution.

Inter-CoBox Communication. As a cobox is only represented by its objects, inter-cobox communication means that objects of one cobox communicate with objects of another cobox. To do so, an object in one cobox needs to reference objects of other coboxes. Such references are called *far references* (following the naming of E [44]), in contrast to *local references*, which refer to objects of the same cobox. Dually, we speak of *far* and *local objects*. Far references are the analogue of remote references used in RMI.

Far references can only be used as targets for asynchronous method calls. It is not possible to directly invoke methods or access fields on far references. Asynchronous method calls return *futures* [8,30,62,42,47,1]. Futures are place holders for the results of asynchronous method calls. Tasks can synchronize in the cobox model by waiting on futures. Using futures to wait for the results of asynchronous method calls leads to a data-driven synchronization, where tasks can only wait for data. Futures are first-class values and can be passed to other coboxes.

Futures cannot only be resolved implicitly by asynchronous method calls, but also explicitly using *promises* [43,44,47,1][1]. A promise is a special object that can be explicitly resolved once and can have multiple associated futures. Promises can be shared between coboxes. They are a safe, flexible communication and coordination

[1] Note that the terms future and promise are not used consistently in the literature. This paper uses definitions similar to Ábrahám et al. [1] and Niehren et al. [47].

```
interface Client {
  void onChatMsg(Msg m); }
interface Server {
  Session connect(Client c); }
```

```
interface Session {
  void publish(Msg m);
  void keepAlive();
  void close(); }
```

Fig. 2. The interfaces of the different components of a simple chat application.

mechanism and can be used to model many synchronization patterns, like condition variables, for example.

Data Transfer and Object Sharing. To combine encapsulation, data transfer between coboxes, and sharing of objects among coboxes, the cobox model distinguishes three kinds of objects. *Standard objects* in the cobox model are passed by reference between coboxes. CoBoxes can only interact with standard objects of other coboxes by using asynchronous method calls via far references. To transfer data between coboxes without exposing objects of a cobox by passing it via a far reference, *transfer objects* can be used. Transfer objects are always local to a cobox and can never be referenced by other coboxes using far references. They are deep-copied when passed to another cobox. The target cobox then gets a local reference to the object copy instead of a far reference to the original object. Transfer objects are like *passive objects* in ASP [14], and *isolates* in AmbientTalk [61]. They can also be compared to non-remote, serializable objects in Java [59].

As transferring data using transfer objects can be inefficient due to the copying overhead, it is possible to directly share state between coboxes using *immutable objects* [28,18]. Immutable objects never change their state, and it is thus safe to access this state concurrently. Like standard objects, immutable objects are passed by reference between coboxes. Conceptually, immutable objects are not owned by any cobox. Nevertheless references to immutable objects are treated as local references. It is thus be possible to directly call methods and to directly access the fields of immutable objects. Immutable objects can only reference standard objects or other immutable objects. Transfer objects cannot be referenced by immutable objects.

2.2 Motivating Example: A Chat Application

To illustrate and motivate central design decisions of our model, we use a simple IRC-like chat application as an example. The chat scenario consists of multiple clients communicating with a single server. For simplicity, there is only one public chat room which all clients use. Whenever a client sends a message to the server it is broadcast to all other clients. All clients should see the messages in the same order as they appeared at the server. A user should interact with its client instance via a graphical user interface. The implementation should work in a distributed environment, and thus must deal with network latencies. For simplicity, we assume no network failures, but allow clients to silently disconnect from the server.

In the typical usage scenario, a client sends a connect message to the server, together with a callback object, which is used by the server later to send published

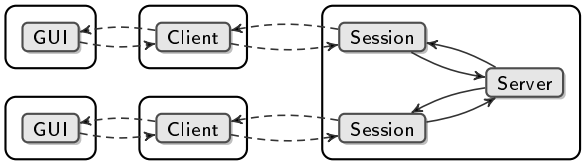

Fig. 3. Runtime view of the chat application in the CoBox model, after two clients have connected to the server.

messages to the client. The server answers with a reference to a session object, which is a typical service object. The session object can then be used by the client to send messages to the server. Using a separate session object for each client simplifies the server logic for tracking client-specific state. Whenever the server gets a message from a client it broadcasts the message to all connected clients. The server also assumes that a client constantly sends alive messages to the server. Figure 2 shows how possible interfaces of these components may look in Java.

Implementing such an application in an object-oriented language has some challenges. On the client side, the GUI and the network-related parts should be decoupled and run concurrently, to prevent network latencies from affecting GUI responsiveness. Frameworks for graphical user interfaces are in general not thread-safe, further complicating concurrency handling in desktop applications. Futhermore, the client has to deal with asynchronous messages from the server and has to constantly send alive messages to the server.

The server component has to deal with concurrently accessing clients. In addition, these clients interact with different objects of the server. Standard OO monitors cannot deal with such *multi-object interfaces*. In addition, the server has to separate the communications of the different clients from each other, to prevent slow clients from affecting the communication with other clients. In addition, the ordering of messages must be guaranteed by the server.

Using the cobox model, the chat application could be structured as shown in Fig. 3. The client is split into two coboxes, one for running the GUI-related code and one for the client logic. Both parts thus run concurrently and communicate asynchronously. The server is realized by a single cobox, which owns the main server object as well as the session objects for each client. Thus the state of the server belongs to a single cobox and programming within the server is done using cooperative multitasking. The server runs concurrently to all clients and communicates with the clients asynchronously. Messages would be realized as either transfer or immutable objects.

2.3 The Core JCoBox Language

A programming model alone cannot be used to write programs. JCoBox is an extension of sequential Java that implements the cobox model to realize concurrency. As JCoBox extends Java, it is a class-based object-oriented language. The extension is conservative, which means that a well-formed sequential Java program is a well-

formed JCoBox program. There is only one exception, which is that static fields in JCoBox have some restrictions (see Sect. 2.4). The syntax of Java is only minimally extended by an additional operator for asynchronous method calls and an extended `new` expression. All other extensions are done by standard Java annotations and using special purpose classes and interfaces. We assume that the reader is familiar with sequential Java and thus we only explain the concepts introduced by the JCoBox extension. This subsection explains the core constructs of JCoBox in some detail; in particular it describes how

- coboxes are created and objects are assigned to coboxes;
- the different object kinds (standard, transfer, immutable) are distinguished;
- asynchronous method calls and futures are addressed;
- the mechanisms for task cooperation is realized.

Creating CoBoxes. Coboxes are not first-class citizens of the language; a cobox is only represented by the objects it contains. A cobox is *implicitly* created when a *cobox class* is instantiated. Instances of cobox classes are standard objects in the cobox model, but with the guarantee that they are always created in a *new* cobox. They are thus the first object of a cobox and can then be used to interact with the cobox. CoBox classes are declared by the **`@CoBox`** annotation. An implementation of the `Server` interface from the chat application could thus be done by writing a cobox class[2]:

```
@CoBox class AServer implements Server {
   List<Session> sessions = new ArrayList<Session>();
   Session connect(Client c) { return new ASession(c); }
   void broadcast(Msg m) { ... }
   class ASession implements Session { ... } }
```

Assigning Objects to CoBoxes. Objects of cobox classes are always created in a new cobox. In contrast, objects of non-cobox classes are created by default in the cobox of the creating task. The `ASession` class, for example, could be implemented as follows:

```
class ASession implements Session {
   Client client; Date lastActivity;
   ASession(Client c) { client = c;
      sessions.add(this); keepAlive(); this!checkAliveness(); }
   void publish(Msg m) { broadcast(m); keepAlive(); }
   void keepAlive() { lastActivity = new Date(); }
   void close() { sessions.remove(this); }
   void checkAliveness() { ... } }
```

The class has no annotation and is thus called a *plain class*. As it is a non-cobox class, its objects are created in the cobox of the creating task. In the example, it is always created by the server cobox and thus can access fields of server objects and can

[2] For brevity we ignore access modifiers in example code.

directly call methods on the server object. There is no mechanism in the language that statically specifies the cobox of an object. Such a guarantee could, however, be given by extending the language with an ownership type system [17,18].

It is possible to create non-cobox objects in other coboxes than the current cobox by using an extended **new** expression that allows for specifying a target cobox. For example, `new A() in (b)` creates an object of `A` in the cobox that owns object `b`.

Transfer and Immutable Objects. Standard objects in JCoBox are instances of cobox classes or plain classes. Transfer and immutable objects are instances of *transfer* and *immutable classes*, respectively, declared by **@Transfer** and **@Immutable** annotations. For example, the `Msg` class of the chat application could be implemented as:

```
@Transfer class Msg { String user; String content; }
```

To guarantee immutability of immutable objects, JCoBox currently adopts a simple mechanism, namely to make all fields of immutable classes implicitly `final`. In addition, fields may not refer to transfer classes. This is a rather restrictive definition, but is, nevertheless, sufficient in many cases.

Asynchronous Method Calls. Asynchronous method calls are expressed by using the `!` operator instead of a dot. For example, `a!m()` asynchronously invokes the method `m()` on object `a`. Any method of plain or cobox classes can be invoked asynchronously in JCoBox. Methods of immutable and transfer classes can only be called directly.

Asynchronous method calls are partially ordered. Partially means that two subsequent calls, from the same cobox, targeting objects of the same cobox, are executed in the given order. For example, the methods sequence `x!m(); y!n()` are executed in the given order if `x` and `y` refer to objects owned by the same cobox. As in a distributed setting, guaranteeing ordering of messages can impose additional costs, JCoBox also supports an unordered variant of an asynchronous method call indicated by the `!!` operator.

Futures. Futures in JCoBox are instances of the special interface `Fut<V>`, where `V` is the type of the future value. The following code invokes an asynchronous method and stores the result in a future variable:

```
Fut<Session> fut = server!connect(this);
```

In order to get the value of a future, it has to be explicitly *claimed* [1]. Similar to the Creol approach [11], claiming a future in JCoBox can be done in two different ways: a blocking and a cooperative one. Claiming a future blockingly is done by using the `get()` method. Blockingly means that the active task waits for the future without giving up control, thus preventing other tasks of the same cobox from being activated. This guarantees that no other task can modify the state of the cobox while waiting for the future value.

A task can also wait cooperatively for a future using the `await()` method. Cooperatively means that, when the future is not ready yet, the waiting task gives up control and is added to the suspend set of the cobox, allowing other tasks to be executed. When the future is ready, the waiting task is added to the ready queue again. The following code cooperatively waits for a future using `await()`:

```
Session session = fut.await();
```

Whether a future should be claimed by `get()` or by `await()` depends on the given scenario. As a general rule, `get()` should only be used if the invoking task cannot establish the invariant of its cobox without knowing the value of the claimed future. In addition, a `get()` can only be used if the waited future can be resolved by the called method without a callback to the waiting cobox. As otherwise a (deterministic) deadlock would happen. When using `await()`, the claiming task must establish the invariant of the cobox before the call of `await()`, because other tasks could be activated in between.

Synchronous communication can be simulated by asynchronous method calls with an immediate `get()` or `await()`. In fact, a synchronous call `x.m()` is treated as `x!m().get()`, when `x` refers to a far object.

Task Cooperation. Beside using `await()` to give up control to another task, it is also possible to directly yield control to the next ready task by using the `JCoBox.yield()` method. Optional time parameters can be used to specify a period of time to wait before the task is added to the ready queue again. For example, the `ensureAliveness` method of the client could constantly send `keepAlive` messages to the server session:

```
void ensureAliveness() {
   while (!stopped) {
      session!keepAlive();
      yield(1, TimeUnit.SECONDS); }}
```

Start Up and Termination of JCoBox Programs. JCoBox programs are started by executing the standard `main` method. The main method is executed in a special *main* cobox, which is created at start up. A JCoBox program terminates when all tasks of all coboxes have terminated.

2.4 Further Features

JCoBox has several additional features, which are important for writing practical applications. This section briefly describes some of them.

Futures. Beside explicitly claiming a future, it is possible to asynchronously invoke methods on futures, which are invoked on the value of the future, when it becomes ready. It is also possible to register event handlers at futures, which are executed when the future is ready. This is similar to `when` expressions in E [44].

Exceptions, Arrays, Static Fields, and Static Methods. Uncatched exceptions, which are thrown during the execution of asynchronous method calls, are rethrown when the future of the call is claimed. Arrays are treated like transfer objects and are copied when passed to another cobox. JCoBox restricts the usage of static fields similar to Kilim [56]. Static fields are implicitly `final` and may not refer to transfer classes. As global state is thus immutable, static methods can be executed by any cobox and, in particular, in parallel.

$$
\begin{aligned}
p \in \textbf{Prog} &::= D\ e \\
d \in D \subseteq \textbf{ClassDecl} &::= [(\text{cobox}|\text{transfer})]\ \text{class}\ c\ \text{extends}\ c'\ \{\ \overline{\tau\ f};\ H\ \} \\
h \in H \subseteq \textbf{MethDecl} &::= \tau\ m(\overline{\tau\ x})\{e\} \\
e \in E \subseteq \textbf{Expr} &::= x \mid \text{null} \mid \text{let}\ x = e\ \text{in}\ e' \mid e.f \mid e.f = e' \mid \text{new}\ c\ [\text{in}\ e] \mid e.m(\overline{e}) \mid \\
&\quad\ e!m(\overline{e}) \mid e.\text{get} \mid e.\text{await} \mid \text{yield} \mid \text{promise}\ \tau \mid e.\text{fut} \mid e.\text{resolve}(e') \\
\tau \in \textbf{Type} &::= c \mid \mathsf{F}\langle\tau\rangle \mid \mathsf{P}\langle\tau\rangle \\
c \in \textbf{ClassName}, m &\in \textbf{MethName}, f \in \textbf{FieldName}, x \in \textbf{VarName}
\end{aligned}
$$

Fig. 4. Abstract syntax of JCoBox^c.

Java Interoperability. For pragmatical reasons, objects created from standard Java classes can be arbitrarily used in JCoBox. The programmer has to ensure that only thread-safe objects are shared between coboxes, otherwise objects can be wrapped by a dynamic proxy object to be shared by coboxes. JCoBox has special support for Swing applications. CoBox classes can be annotated with the `@Swing` annotation, in which case all tasks of that cobox are executed by the Swing event handling thread.

Distributed Programming. JCoBox has prototypical support for writing distributed applications using RMI, where the unit of distribution is a cobox. This allows for asynchronous communication via RMI with futures and promises.

3 Formal Semantics

This section presents a formal calculus for a core of JCoBox, called JCoBox^c. The calculus serves as a precise definition of the semantics of JCoBox. The calculus focuses on the key features of the language, namely: cobox classes, asynchronous method calls, cooperative multitasking, futures, promises, and transfer classes. Immutable objects are not included as they can be regarded as transfer objects optimized for performance reasons. The semantics of JCoBox^c is implemented in the Maude rewriting framework [19], which allows JCoBox^c programs to be executed, and can be obtained from [35]. We only present a dynamic semantics for JCoBox. The type system for JCoBox is a straightforward extension of a standard Java type system, which mainly adds the typing of asynchronous method calls, futures, and promises. The sequential part of the dynamic semantics is based on *Featherweight Java* [34] and *ClassicJava* [24]. The combination of futures and cooperative multitasking is similar to that of Creol [11]. The treatment of transfer objects is similar to that of passive objects in ASP [14]. The combination of promises and asynchronous method calls is similar to Ábrahám et al. [1]. The formalization is a modification and simplification of previous work [55], which treats hierarchical coboxes and which does neither feature transfer classes nor promises.

3.1 Syntax

The abstract syntax of JCoBox^c is shown in Fig. 4. Terms enclosed by brackets are optional; capital letters denote sets; an overbar indicates a sequences. • denotes the

$$
\begin{array}{rll}
k \in K \subseteq \textbf{Config} & ::= b \mid p & \text{configurations} \\
b \in B \subseteq \textbf{CoBox} & ::= \mathtt{B}\langle \iota_b, O, T, \overline{t} \rangle & \text{coboxes} \\
p \in P \subseteq \textbf{Prom} & ::= \mathtt{P}\langle \iota_p, O, v_\epsilon \rangle & \text{promises} \\
o \in O \subseteq \textbf{Obj} & ::= \mathtt{O}\langle \iota_o, c, \overline{v} \rangle \mid \mathtt{F}\langle \iota_o, \iota_p, v_\epsilon \rangle & \text{objects and futures} \\
t \in T \subseteq \textbf{Tsk} & ::= \mathtt{T}\langle e \rangle & \text{tasks} \\
v \in V \subseteq \textbf{Value} & ::= r \mid \mathsf{null} & \text{values} \\
r \in R \subseteq \textbf{Ref} & ::= \iota_g.\iota_o \mid \iota_p & \text{references} \\
\iota_g \in \textbf{GlobalId} & ::= \iota_b \mid \iota_p & \text{global identifiers} \\
e \in E \subseteq \textbf{Expr} & ::= \ldots \mid v & \text{extended expressions}
\end{array}
$$

$$\iota_o \in \textbf{ObjId}, \iota_b \in \textbf{CoBoxId} \cup \{\,\mathsf{main}\,\}, \iota_p \in \textbf{PromId}$$

Fig. 5. Semantic entities of JCoBox^e. Optional terms are indicated by an ϵ as index and may be ϵ. Small capital letters like B or O are used as "constructors" to distinguish the different semantic entities syntactically.

empty sequence, $\cdot$ adds an element to a sequence, and $\circ$ concatentates sequences. In addition, we often implicitly treat single elements as sequences or sets of size one when needed. Most of the syntax should be clear from the description in Sect. 2. A program p is a pair $D\ e$ consisting of set of class declarations D and a main expression e. A promise for a value of type τ is created by the promise τ expression. A future can be retrieved from a promise with e.fut. The expression e.resolve(e') resolves a promise e to the evaluated value of e'. A type τ, can either be a class name c, a future type $\mathtt{F}\langle\tau\rangle$, or a promise type $\mathtt{P}\langle\tau\rangle$.

3.2 Dynamic Semantics

The dynamic semantics is formulated as a small step operational semantics. The semantic entities used by the semantics are shown in Fig. 5. The state of a program is represented by a set K of *configurations* k, which can either be coboxes b or promises p. A cobox $\mathtt{B}\langle \iota_b, O, T, \overline{t} \rangle$ consists of a cobox identifier ι_b, a set of objects O, a set of suspended tasks T and a sequence of tasks $\overline{t}$. The head of $\overline{t}$ represents the active task of the cobox, the tail represents the ready queue. Promises are "degenerated" coboxes, that do not have any tasks. Instead, a promise $\mathtt{P}\langle \iota_p, O, v_\epsilon \rangle$ has an optional value v_ϵ, which is ϵ as long as the promise is not resolved. A promise also has a set of objects O, which is initially empty, but which represents all transfer objects reachable by v, when the promise is resolved to v. Objects $\mathtt{O}\langle \iota_o, c, \overline{v} \rangle$ consist of an object identifier ι_o, a class name c, and a sequence of values $\overline{v}$ representing its state. Futures $\mathtt{F}\langle \iota_o, \iota_p, v_\epsilon \rangle$ are similar to objects, but always have a reference to a promise and have an optional value v_ϵ, which is ϵ until the future is resolved. A task $\mathtt{T}\langle e \rangle$ only consists of a single expression e, which is in general of the form ι_p.resolve(e'), where ι_p is the promise that is resolved by the task. Only the initial task of a program has no associated promise. Values are either references r or null, where a reference can either be an object reference $\iota_g.\iota_o$ or a promise identifier ι_p. Note that object references in general are of the form $\iota_b.\iota_o$, $\iota_p.\iota_o$ references can only appear in objects of promises.

$$
\begin{aligned}
\mathit{oids}(O) &\stackrel{\text{def}}{=} \{\, \iota_o \mid \mathsf{O}\langle \iota_o, \_, \_\rangle \in O \vee \mathsf{F}\langle \iota_o, \_, \_\rangle \in O \,\} \\
\mathit{init}(c) &\stackrel{\text{def}}{=} \overline{\mathsf{null}} \quad \text{, where the length is equal to the number of fields of } c \\
\mathit{coboxcl}(c) &\stackrel{\text{def}}{=} c \text{ is defined as a cobox class} \\
\mathit{transfercl}(c) &\stackrel{\text{def}}{=} c \text{ is defined as a transfer class} \\
\mathit{mexpr}(c, m, r, \overline{v}) &\stackrel{\text{def}}{=} [r/\mathit{this}, \overline{v}/\overline{x}]e \quad \text{, where } \overline{x}.e = \mathit{body}(c.m) \\
\mathit{body}(c.m) &\stackrel{\text{def}}{=} \overline{x}.e \quad \text{, where } \tau\ m(\overline{\tau\ x})\{e\} \text{ is the declaration of } m
\end{aligned}
$$

$$
\begin{aligned}
\mathit{reach}(, \overline{v}) &\stackrel{\text{def}}{=} \\
\mathit{reach}(O, \bullet) &\stackrel{\text{def}}{=} \\
\mathit{reach}(O \uplus o, \overline{v} \cdot \iota_g.\iota_o) &\stackrel{\text{def}}{=} \mathit{reach}(O, \overline{v} \circ \overline{v}') \cup \{\, o \,\} && \text{if } o = \mathsf{O}\langle \iota_o, c, \overline{v}' \rangle \wedge \mathit{transfercl}(c) \\
\mathit{reach}(O \uplus o, \overline{v} \cdot \iota_g.\iota_o) &\stackrel{\text{def}}{=} \mathit{reach}(O, \overline{v}) \cup \{\, \mathsf{F}\langle \iota_o, \iota_p, \epsilon \rangle \,\} && \text{if } o = \mathsf{F}\langle \iota_o, \iota_p, v_\epsilon \rangle \\
\mathit{reach}(O, \overline{v} \cdot v) &\stackrel{\text{def}}{=} \mathit{reach}(O, \overline{v}) && \text{else}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{copy}(\iota_g, O, \overline{v}, \iota'_g, O') \stackrel{\text{def}}{=} (\sigma O'', \sigma \overline{v}) \\
&\qquad \text{where } O'' = \mathit{reach}(O, \overline{v}) \\
&\qquad\quad \text{and } \sigma = \{\, \iota_g.\iota_o \mapsto \iota'_g.\iota'_o,\ \iota_o \mapsto \iota'_o \mid \iota_o \in \mathit{oids}(O'') \wedge \iota'_o \text{ fresh} \,\}
\end{aligned}
$$

Fig. 6. Auxiliary functions and predicates

Evaluation Contexts. To abstract from the context of an expression and to define the evaluation order of expressions we use evaluation contexts [22]. An evaluation context is an expession with a "hole" $\square$ at a certain position. By writing $e_\square[e]$ that hole is replaced by expression e.

$$
\begin{aligned}
e_\square ::= {} & \square \mid e_\square.f \mid e_\square.f = e \mid v.f = e_\square \mid \mathsf{new}\ c\ \mathsf{in}\ e_\square \\
& \mid \mathsf{let}\ x = e_\square\ \mathsf{in}\ e \mid e_\square.n(\overline{e}) \mid v.n(\overline{v}, e_\square, \overline{e}) \mid e_\square!n(\overline{e}) \mid v!n(\overline{v}, e_\square, \overline{e}) \\
& \mid e_\square.\mathsf{get} \mid e_\square.\mathsf{await} \mid e_\square.\mathsf{fut} \mid e_\square.\mathsf{resolve}(e) \mid v.\mathsf{resolve}(e_\square)
\end{aligned}
$$

Auxiliary Functions. Figure 6 gives a definition of auxiliary functions and predicates used by the rules. As many of these functions are technically simple, we only provide an informal description for them. For a precise definition we refer to the Maude formalization [35].

Data Transfer. Transfer objects and futures are copied between coboxes/promises by the *copy* function. The *copy* function uses the $\mathit{reach}(O, \overline{v})$ function to extract all transfer objects of O reachable from $\overline{v}$. In addition, futures are extracted, but their value is reset to ϵ and not regarded for the reachability of objects. This point is important for a deterministic transfer of futures, independent of their actual resolving status. Whenever a future is transferred to another cobox it has to be resolved again by the associated promise. $\mathit{copy}(\iota_g, O, \overline{v}, \iota'_g, O')$ creates a copy of all transfer objects from cobox/promise ι_g, reachable by $\overline{v}$, with fresh identifiers for cobox/promise ι'_g.

Evaluation Rules. The operational semantics is defined in terms of a relation on sets of configurations, $K \longrightarrow K'$. It is implicitly parametrized by a fixed underlying

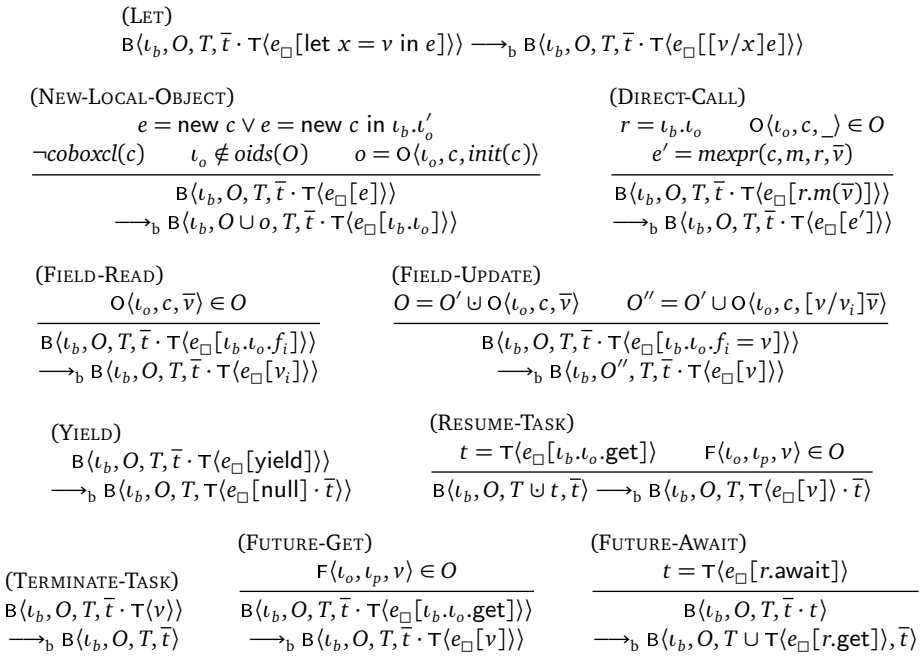

Fig. 7. CoBox-local rules of JCoBox^c.

program, which we omit for conciseness. The rules defining the relation are split into two parts: cobox-local rules and global rules. Splitting up the rules in this way makes it explicit which steps can be executed in isolation and which require interaction between coboxes and/or promises.

CoBox-Local Rules. The cobox-local rules are shown in Fig. 7. The relation is denoted by $\longrightarrow_b$ and is defined on coboxes. These rules essentially model programming inside a cobox. The sequential programming rules are more or less standard. The important aspect is that the target of field reads, updates, and direct method calls must be objects of the same cobox. In addition, cooperative task scheduling and future claiming is covered by the cobox-local rules.

Global Rules. The global rules are shown in Fig. 8. The (New-Far-Object) rule is essentially equal to (New-Local-Object), but creates the new object in a different cobox and does not allow to create transfer classes. (New-CoBox) creates a new cobox with an initial object, whose reference is the result of the new-expression. A new cobox has no tasks. Asynchronous method calls are distinguished into local (Local-Async-Call) and far calls (Far-Async-Call). The local one addresses the current cobox and does not copy the method parameters. The far one addresses a different cobox and copies the parameters. Both calls create a new promise for holding the result of the call and add a new task, which executes the body expression of the

$$\frac{\neg coboxcl(c) \qquad \neg transfercl(c) \qquad K = K' \uplus \mathsf{B}\langle \iota'_b, O', T', \overline{t}' \rangle \qquad \iota_o \notin oids(O') \qquad K'' = K' \uplus \mathsf{B}\langle \iota'_b, O' \cup \mathsf{O}\langle \iota_o, c, init(c) \rangle, T', \overline{t}' \rangle}{K \uplus \mathsf{B}\langle \iota_b, O, T, \overline{t} \cdot \mathsf{T}\langle e_\Box[\mathsf{new}\ c\ \mathsf{in}\ \iota'_b.\iota'_o] \rangle \rangle \longrightarrow K'' \uplus \mathsf{B}\langle \iota_b, O, T, \overline{t} \cdot \mathsf{T}\langle e_\Box[\iota'_b.\iota_o] \rangle \rangle}\ (\textsc{New-Far-Object})$$

$$\frac{coboxcl(c) \qquad \iota'_b\ \mathsf{fresh} \qquad b' = \mathsf{B}\langle \iota'_b, \{\mathsf{O}\langle \iota_o, c, init(c) \rangle\}, , \bullet \rangle}{K \uplus \mathsf{B}\langle \iota_b, O, T, \overline{t} \cdot \mathsf{T}\langle e_\Box[\mathsf{new}\ c] \rangle \rangle \longrightarrow K \cup \mathsf{B}\langle \iota_b, O, T, \overline{t} \cdot \mathsf{T}\langle e_\Box[\iota'_b.\iota_o] \rangle \rangle \cup b'}\ (\textsc{New-CoBox})$$

$$\frac{r = \iota_b.\iota_o \qquad \mathsf{O}\langle \iota_o, c, \_ \rangle \in O \qquad \iota_p\ \mathsf{fresh} \qquad t' = \mathsf{T}\langle \iota_p.\mathsf{resolve}(mexpr(c, r, m, \overline{v})) \rangle}{K \uplus \mathsf{B}\langle \iota_b, O, T, \overline{t} \cdot \mathsf{T}\langle e_\Box[r!m(\overline{v})] \rangle \rangle \longrightarrow K \cup \mathsf{B}\langle \iota_b, O, T, t' \cdot \overline{t} \cdot \mathsf{T}\langle e_\Box[\iota_p.\mathsf{fut}] \rangle \rangle \cup \mathsf{P}\langle \iota_p, , \epsilon \rangle}\ (\textsc{Local-Async-Call})$$

$$\frac{r = \iota'_b.\iota_o \qquad K = K' \uplus \mathsf{B}\langle \iota'_b, O', T', \overline{t}' \rangle \qquad \mathsf{O}\langle \iota_o, c, \_ \rangle \in O' \qquad (O'', \overline{v}') = copy(\iota_b, O, \overline{v}, \iota'_b, O') \qquad \iota_p\ \mathsf{fresh} \qquad t' = \mathsf{T}\langle \iota_p.\mathsf{resolve}(mexpr(c, r, m, \overline{v}')) \rangle \qquad K'' = K' \cup \mathsf{B}\langle \iota'_b, O' \cup O'', T', t' \cdot \overline{t}' \rangle}{K \uplus \mathsf{B}\langle \iota_b, O, T, \overline{t} \cdot \mathsf{T}\langle e_\Box[r!m(\overline{v})] \rangle \rangle \longrightarrow K'' \cup \mathsf{B}\langle \iota_b, O, T, \overline{t} \cdot \mathsf{T}\langle e_\Box[\iota_p.\mathsf{fut}] \rangle \rangle \cup \mathsf{P}\langle \iota_p, , \epsilon \rangle}\ (\textsc{Far-Async-Call})$$

$$\frac{\mathsf{P}\langle \iota_p, \_, \_ \rangle \in K \qquad \iota_o \notin oids(O)}{K \uplus \mathsf{B}\langle \iota_b, O, T, \overline{t} \cdot \mathsf{T}\langle e_\Box[\iota_p.\mathsf{fut}] \rangle \rangle \longrightarrow K \cup \mathsf{B}\langle \iota_b, O \cup \mathsf{F}\langle \iota_o, \iota_p, \epsilon \rangle, T, \overline{t} \cdot \mathsf{T}\langle e_\Box[\iota_b.\iota_o] \rangle \rangle}\ (\textsc{New-Future})$$

$$\frac{\iota_p\ \mathsf{fresh} \qquad K' = K \cup \mathsf{P}\langle \iota_p, , \epsilon \rangle}{K \uplus \mathsf{B}\langle \iota_b, O, T, \overline{t} \cdot \mathsf{T}\langle e_\Box[\mathsf{promise}\ \tau] \rangle \rangle \longrightarrow K' \cup \mathsf{B}\langle \iota_b, O, T, \overline{t} \cdot \mathsf{T}\langle e_\Box[\iota_p] \rangle \rangle}\ (\textsc{New-Promise})$$

$$\frac{O = O' \uplus \mathsf{F}\langle \iota_o, \iota_p, \epsilon \rangle \qquad \mathsf{P}\langle \iota_p, O'', v \rangle \in K \qquad (O''', v') = copy(\iota_p, O'', v, \iota_b, O)}{K \uplus \mathsf{B}\langle \iota_b, O, T, \overline{t} \rangle \longrightarrow K \cup \mathsf{B}\langle \iota_b, O''' \cup O' \cup \mathsf{F}\langle \iota_o, \iota_p, v' \rangle, T, \overline{t} \rangle}\ (\textsc{Resolve-Future})$$

$$\frac{(O', v') = copy(\iota_b, O, v, \iota_p,) \qquad K = K' \uplus \mathsf{P}\langle \iota_p, , \epsilon \rangle \qquad K'' = K' \cup \mathsf{P}\langle \iota_p, O', v' \rangle}{K \uplus \mathsf{B}\langle \iota_b, O, T, \overline{t} \cdot \mathsf{T}\langle e_\Box[\iota_p.\mathsf{resolve}(v)] \rangle \rangle \longrightarrow K'' \cup \mathsf{B}\langle \iota_b, O, T, \overline{t} \cdot \mathsf{T}\langle e_\Box[\mathsf{null}] \rangle \rangle}\ (\textsc{Resolve-Promise})$$

Fig. 8. Global reduction rules of JCoBox^c

corresponding method, to the end of the task queue. Both call expressions are reduced to a ι_p.fut expression to obtain a future from the new promise. A future to a promise is simply added to the local object heap of the cobox (New-Future). A future is resolved when the associated promise is resolved. The objects of the promise are then copied to the cobox of the future. The value of the future is set to a copy from the promise. A promise is resolved by copying the value and all reachable transfer objects and futures to the promise. Note that this means that the result of an asynchronous method call is essentially copied twice: first to the promise and then to the cobox of the future. This is needed, as futures can be passed to other coboxes and require to get a copy of the original result value, stored in the promise. In practice, this double copying can be avoided in many cases, where it is statically clear that the future is not passed to another cobox.

Finally, rule (CONGRUENCE) integrates the cobox-local relation into the global relation:

$$\frac{b \longrightarrow_{\mathrm{b}} b'}{K \uplus b \longrightarrow K \cup b'} \quad \text{(CONGRUENCE)}$$

Initial Configuration. A JCoBox^c program is $D\ e$ is started by executing e in the initial main cobox: $\mathtt{B}\langle \mathrm{main}, , , \mathtt{T}\langle e \rangle \rangle$.

Properties. When proposing a new concurrency model, the two most interesting properties are properties concerning data-races and deadlocks. JCoBox is by design data-race free, which is directly reflected in the operational semantics by distributing the object heap over the coboxes. As the active task of a cobox can only access fields of its own cobox, it is immediately clear that data-races cannot occur. Deadlocks are possible in JCoBox by blockingly waiting for futures. In many cases, deadlocks result from direct data-dependencies between coboxes, which, in general, lead to deterministic deadlocks, which can easily be found by standard testing. Deadlocks can be avoided by imposing a partial order (e.g. a hierarchy) on coboxes and only blockingly wait for futures of smaller coboxes, according to that order.

4 Implementation

We implemented JCoBox on top Java[3]. The JCoBox implementation consists of two parts: the compiler and the runtime system. The JCoBox compiler (JCoBoxC) is implemented as an extension of Polyglot 1.3.5 [52,49]. Support for Java 5 features, e.g. generics, is realized by the JL5 extension [36]. JCoBoxC takes JCoBox source files and Java class files as input and generates Java source files, which are then compiled to JVM bytecode by a standard Java compiler.

CoBoxes. Every cobox at runtime is represented by a cobox object, which is created when an instance of a cobox class is created. To realize the relationship between objects and coboxes, every object gets an additional field pointing to its owning cobox, which is set when the object is created. This additional field is added by inheriting all classes without an explicit super class, implicitly from `CoObjectClass`, which has such a field. Transfer, immutable and classes inheriting from standard Java classes do not inherit from `CoObjectClass` and thus do not have this field. Whenever a method is invoked on an object of `CoObjectClass`, it is checked whether the invoking cobox is equal to the cobox of the object. If this is not the case the call is passed to the scheduler of the cobox for eventual execution. The invoking cobox is determined by the invoking thread. Every thread has a thread-local field referring to the cobox of the task which it currently executes. The check is cheap as no synchronization is required, so that cobox local calls only have a small overhead. In addition, the compiler can optimize cases, where it is statically clear that the cobox is not left, e.g. for calls on `this`.

[3] The implementation can be obtained from the JCoBox website [35]

Asynchronous Method Calls. Asynchronous calls are realized by task objects invoking the corresponding method synchronously. For each asynchronous call a separate class is generated that contains code for copying of parameters, as well as the invocation of the corresponding method. Each task object is assigned a promise object for holding the result of the call, which is resolved when the call has finished. Asynchronous method calls return a future obtained from the assigned promise. If it is statically clear that the future is not needed, the compiler uses an optimized task object, without promises and futures.

Synchronous Method Calls. For efficiency reasons, synchronous method calls targeting far objects are not realized by asynchronous method calls and an immediate `get()`, but by standard synchronous calls. To guarantee that they are semantically equivalent, the JCoBox compiler generates wrapper methods for every method of a class that gives that guarantee.

Task Scheduling. The tasks of a cobox are managed by a scheduler. The scheduler manages the ready queue of the cobox. The ready queue contains either asynchronous task objects, or `Thread` objects, which are used to represent synchronous calls. The scheduler executes the tasks in FIFO order. Asynchronous tasks are executed by passing them to a global thread pool for execution. Synchronous threads are blocked until they reach the head of the task queue. This all happens without the need of an additional scheduler thread. So that a cobox without any running task does not require a separate thread.

Asynchronous tasks are executed by a thread pool. The size of the thread pool is dynamically adapted to ensure that there are always threads that make progress. The thread pool is increased, for example, when a task waits for an unresolved future. In addition, liveness is ensured by monitoring progress and adding additional threads when needed.

Transfer Classes. For each transfer class a copy method is generated, which recursively copies all fields of the class. This is similar to the default implementation of serializable classes, but is much faster, because no intermediate byte stream is used. In the distributed setting, the standard serialization mechanism of RMI is used.

5 Evaluation

5.1 Performance

A programming model is only useful in practice if it can be efficiently implemented. To evaluate the performance of the JCoBox implementation, we compare JCoBox with two industry-strength languages that run on the JVM and have some kind of actor abstraction: Scala and Clojure[4].

[4] We also planned to include Kilim [56] in our performance evaluation, but until the end, Kilim suffered from a data-race, which made it impossible to get reliable measurements.

Scala features an actor library, which belongs to the fastest actor implementations on the JVM [38]. Clojure is a Lisp-dialect targeting the JVM and has the concept of *agents* to allow for the asynchronous execution of functions. We used Scala v2.7.7-final and Clojure-1.0.0 in all of our benchmarks. We also included Scala v2.7.5-final, because it uses a different actor backend.

As each language has different names for similar things, we use in the following the term *actor* for coboxes in JCoBox, actors in Scala, and agents in Clojure; and we speak of *message sending*, when talking about asynchronous invocation of methods in JCoBox, message sending in Scala, and asynchronous invocation of functions in Clojure.

Benchmark Programs. As there is currently no standard benchmark for measuring actor-like languages, we measure performance by three micro-benchmarks, The first two are taken from the Computer Language Benchmark Game [58], the third one is an example, which is used in Srinivasan and Mycroft [56].

Thread Ring creates a ring of n actors. A single message is then send around the ring m times, resulting in a total of $n * m$ sent messages. We consider two different configurations. The first configuration (*ringmsgs*) sets n to 1 000 and increases m from 1 000 up to 10 000. This configuration measures mainly message sending and receiving performance and is similar to the configuration used in [58]. The second configuration (*ringnodes*) sets m to 10 and increases n from 100 000 up to 1 000 000. This configuration mainly measures the creation and handling of a large amount of actors. Both configurations have no concurrency, no contention, and there always exists only one unreceived message at a time.

The *Chameneos* example [58], creates n (first 3 then 10) chameneos, which all try to meet another chameneos at a single mall to complement their colors. The number of meets m is increased from 200 000 up to 2 000 000. This benchmark measures performance of a high frequency of messages under high contention, but the message load is relatively low and the number of actors is also very low. There is a small potential for parallel execution.

The *BigPingPong* (pingpong) example [56] creates n actors, which sends each other actor a single message, so that n^2 messages are sent and received. This benchmark measures performance under a high load of simultaneous messages, under low contention, with a medium number of actors and with a high potential of parallel execution.

Setup. We ran all benchmarks on five different hardware platforms: An Intel Atom N270 1.6GHz CPU and 1GB RAM (Atom). An Intel Core 2 Duo T7400 2.16GHz CPU and 2GB RAM (Core2Duo). An AMD Athlon dual-core 4850e CPU with 2.5Ghz and 4GB RAM (Athlon). An AMD machine with two dual-core AMD Opteron 270 2GHz CPUs and 4GB RAM (Opteron). An Intel Xeon X3220 quad-core CPU with 4GB RAM (Xeon).

The Atom, Core2Duo and Athlon platforms run a 32-bit Linux 2.6.31, the Opteron platform run in a XEN virtual machine with 64-bit Linux 2.6.16-xen, and the Xeon platform run a 64-bit Linux 2.6.25. All benchmarks were executed on the Sun JDK

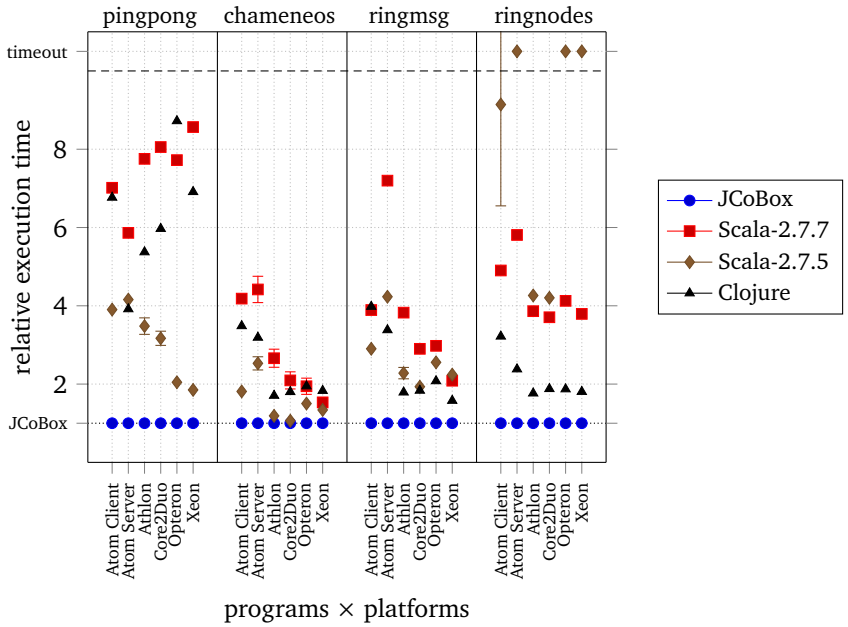

Fig. 9. Relative execution time of the different benchmark execution, each with the maximum input value. The times are relative to the corresponding JCoBox times on each platform.

version 1.6.16. On the Core2Duo and the Athlon platform the 32-bit Server VM, and on the the Opteron and Xeon platform the 64-bit Server VM was used. On the Atom platform we used the 32-bit Client (Atom Client), which is the default on this platform, as well as the Server VM (Atom Server). All benchmarks were executed with `java -Xmx1024M`, thus with a maximal memory of 1GB.

We followed the advice of Georges et al. [26] and executed each benchmark k times within n JVM invocations. For each JVM invocation we took the arithmetic mean, $\overline{x}_i$, of the last 10 of k benchmark runs, where k was at most $10+5$, but could be less if the JVM reached a steady state earlier. A steady state was assumed if the CoV[5] of the last 10 runs was less or equal to 0.02. The time of a benchmark run was measured by using the Java `System.nanoTime()` method. We then calculated the mean and the 95% confidence interval of the n means, $\overline{x}_i$, where n was either 10 or less if the size of the 95% confidence interval fell below 3% earlier.

Results. The chart in Fig. 9 gives an overview over all benchmark runs with maximal input parameters. Each point represents the execution time of a single language-program-platform combination relatively to the corresponding JCoBox execution time. The y-axis can also be read as the speedup of JCoBox compared to the other languages. This chart allows for comparing the different languages as well as the

[5] CoV is defined as the standard deviation s divided by the mean $\overline{x}$, see [26] for details.

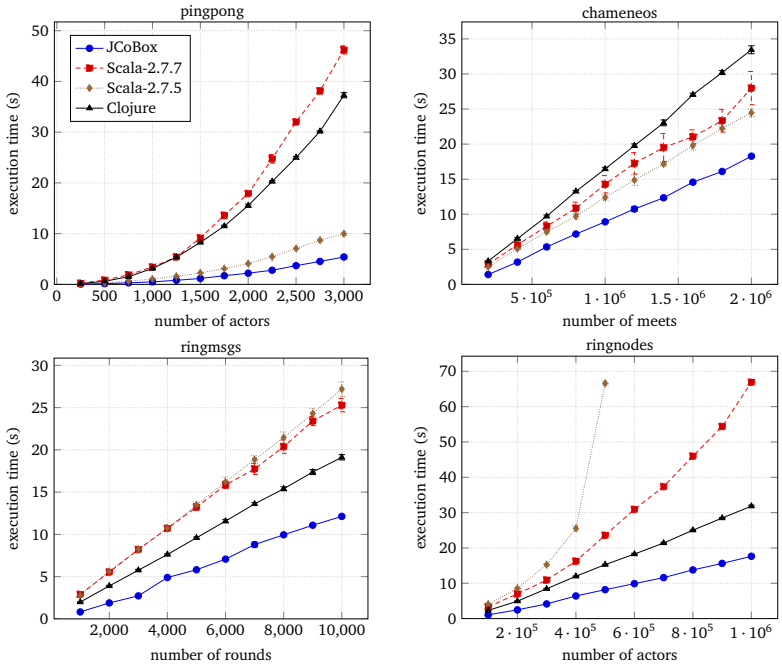

Fig. 10. The different benchmark executions on the Xeon-Platform

different platforms. Figure 10 exemplarily shows the different benchmarks runs on the Xeon platform with increasing input parameters.

Discussion. In all our benchmarks JCoBox outperforms Scala as-well-as Clojure. The largest speedups (between 2 and 9) are in the pingpong example, which shows that JCoBox can deal with a high load of messages. It also shows the effect of additional cores. Scala v2.7.5 seems to profit, whereas Clojure and Scala v2.7.7 are slowed down, compared to JCoBox. The lowest speedups (≈1–4) are in the chameneos example, which focuses on high-contention, which shows that a JVM limit might be reached by all languages. The ringmsg benchmark shows that message sending and receiving is fast in JCoBox (speedups between 1.5 and 7). The ringnodes benchmark shows that coboxes are very cheap and scale up to millions of coboxes. Even though it was not the focus of the performance tests, it can be noticed that Clojure is surprisingly fast, even though it is a dynamically typed language. There is also a significant difference between Scala v2.7.5 and v2.7.7, where the latter have been significantly slower in 3 of 4 of our benchmarks, and the former ran out of memory or timed out in the ringnodes benchmark on some platforms, a known bug, which has been fixed in v2.7.7.

It is always critical to use micro-benchmarks to compare the performance of different languages. It cannot be concluded from these benchmarks that in practice there will be a significant difference in speed between the different languages, as the

dominant factor of an application might not lie in the actor framework. However, we believe that in practice, JCoBox will be at least as fast as the compared languages.

5.2 Example Applications

Among several small examples and applications, we implemented four mid-size desktop applications in JCoBox. A connect four game, which also supports a computer player, which utilizes multiple cores and can run on a different machine. The CoCoME example [53], which implements a distributed trading system, which was implemented by a master student. A disk usage visualizer, which incrementally visualizes the contents of a file system and allows for navigating through the view, while it is built. Finally, we implemented a distributed chat application as discribed in Sect. 2, but with multiple chat rooms. Except the CoCoME example, all applications have been written by the authors.

Our experience with JCoBox is very promising. The cobox model matches very well with the typical object-oriented programming style and does not require a radical new way of thinking. CoBoxes naturally appear in typical OO applications. Asynchronous method calls are also a natural communication mechanism and lead to loosely coupled systems. The cooperative multi-tasking inside coboxes proved to be very useful for combining active and reactive behavior and for simulating synchronous communication with flexible reentrancy control. JCoBox prevents data-races by design and makes it difficult to create non-deterministic deadlocks, which greatly simplifies the development of concurrent OO applications.

6 Related and Future Work

JCoBox does not introduce radically new ideas or features. It rather unifies existing ideas into a single formalized programming model and provides a practical implementation on top of Java.

6.1 Related Work

There exists a vast amount of work on how to combine concurrency with object-orientation (see, for example, [51,12] for surveys). This subsection necessarily concentrates on the closest work.

The principle idea of a data-centric concurrency model, i.e. a model where concurrency is structured by the data, stems from the actor model [33,3,42]. The pure actor model is defined in a functional setting, without mutable state, with Erlang [6] being the most prominent implementation of the recent past. The combination of stateful objects and actors as *active objects* was first done in ABCL/1 [62], POOL2 [4], and Eiffel// [13]. These approaches have single objects as the unit of concurrency. Hybrid [48] generalizes this to *groups* of objects called *domains*, which, however, communicate via synchronous method calls. ASP [14] groups objects into so-called *activities*. Activities, however, have only one distinguished object, the *active object*, which can be referenced by other activities. Multiple service objects for one activity

are thus not possible. ASP also introduced the notion of *passive objects*, which can only be referenced inside a single activity and are deep-copied when passed to other activities. Passive objects correspond to transfer objects in JCoBox. The ASP concurrency model is implemented in ProActive [7].

The E programming language [44] introduces a concurrency model called *communicating event loops*. The unit of concurrency in E is a *vat*, which hosts a group of objects. All objects of a vat can be referenced by other vats, allowing multiple service objects, equally to coboxes. Computations inside a vat, however, are only executed by a single thread of control. This leads to an event-based programming model, where the control flows has to be spread over event handlers. The E programming model can be simulated in JCoBox by never suspending or yielding a task. Ordering of messages in E is only with respect to single objects, where in JCoBox it is with respect to coboxes, which we believe is what a programmer in general expects. Like JCoBox, E can execute a vat by the event handling thread of Swing, to allow for a seamless GUI interaction. The E programming model has been lately adopted by AmbientTalk [61]. AmbientTalk also integrates a notion of transfer objects called *isolates*.

The idea of using cooperative multitasking for concurrency inside of active objects stems from Creol [11,37]. In addition, combining cooperative multitasking with futures, was also pioneered by Creol. Creol, however, has single objects as the unit of concurrency, which does not allow for multiple service objects. The Creol model can be simulated by the cobox model by putting every object in a unique cobox. In Symbian OS [45] active objects are scheduled cooperatively within the same active scheduler, which are thread-local. Each active object only has a single thread of control. Symbian OS also shows how to combine active objects with GUI programming. Kilim [56] can schedule tasks cooperatively if the assigned scheduler is configured to be single-threaded. Rodriguez and Rossetto [54] combine cooperative multitasking with asynchronous RMI in the Lua language.

Thorn [10] is an actor approach, which combines message sending and asynchronous method calls. Unlike JCoBox, Thorn does not support multiple tasks within a process. However, Thorn has a special `splitSync` construct, which can be used to solve the problems of the single-threading of Thorn components in some cases. Only methods declared to be asynchronous can be invoked in an asynchronous way, where in JCoBox, any method can be invoked asynchronously.

6.2 Current Limitations and Future Work

Although JCoBox showed to be usable in practice, there are some limitations and improvable aspects.

Data-Parallel Programming. JCoBox can be used for many parallelization problems, namely ones, where the problem state can be partitioned into separate parts and be worked on in isolation. However, there are some parallelization problems, which cannot be efficiently addressed in the cobox model, namely data-parallel algorithms, where parallel threads work on shared mutable state. For example, a parallel in-place sorting of an array will not be possible to implement in the cobox model. We

argue that JCoBox should be used as a high-level programming model, where it is possible in certain cases to "escape" from the high-level model and use special purpose libraries that address these issues [41,40]. In addition, parallel access to shared state is possible in JCoBox if the state is immutable.

Receiver Coordination. Currently, a cobox serves all method calls it gets in the same order as they appear. There is no direct mechanism to define a certain communication protocol. Instead, a programmer has to use promises or monitor-like condition variables. There exist several different solutions for receiver-side coordination. One are guards at method declarations [32] or at suspension points [11]. Another are join patterns [25,9]. An actor like mechanism, where a body explicitly specifies a communication protocol [4,5,14,39,10] could be integrated into JCoBox. Finally, one may specify the scheduling of messages in a separate object as done in SOM [16] and POM [15]. This is already partly supported by JCoBox as the scheduler object is independent from the cobox object, a feature which is used to implement the Swing support.

Data Transfer. Currently, data in JCoBox is transferred between coboxes either by copying or by using a simple form of class immutability. The current form of immutability is very strict. We are currently working on an implementation of a more flexible immutability notion based on [27,18,28]. Sometimes neither copying nor immutability is an option. A network connection object, for example, can neither be immutable nor can be copied. It is thus desirable to be able to safely transfer mutable objects by reference. Several solutions to this problem exist [18,56,21], which could be added to JCoBox.

Cooperative Multitasking. The strongest limitation of the current JCoBox implementation is the fact that suspending a task can lead to a suspension of the underlying thread and may require to increase the thread pool to prevent starvation. This can be an expensive operation especially if the number of simultaneously suspended tasks is large. This problem is effectively solved by continuation frameworks for Java [56,23], which we plan to integrate in our implementation.

Distributed Programming. The cobox model is well-suited for distributed programming, supported by the fact that languages explicitly designed for distribution have a similar programming model [44,7,61]. The current JCoBox implementation only features a proof-of-concept implementation based on RMI. It can already be used to write distributed programs in an asynchronous style, where the underlying programming model is, despite network failures, identical in the local and distributed case.

7 Conclusions

Concurrency in OOP is an ongoing research topic. This paper presents JCoBox: a language that unifies several concepts for OOP-concurrency. The concurrency model

of JCoBox is based on *coboxes*, concurrently running, isolated object-oriented components. The language integrates asynchronous method calls for a loosely coupled communication, suitable for distributed systems. Futures and promises allow for a data-driven synchronization between tasks. The behavior of a cobox is constituted by a set of cooperatively scheduled tasks, enabling the easy combination of active and reactive behavior. Data can be transferred either by copy or by reference using immutable objects.

The semantics of JCoBox is precisely described by a formal core calculus. JCoBox is implemented on top of sequential Java and was successfully used to write several concurrent and distributed desktop applications with graphical user interfaces. The JCoBox implementation is a research prototype, nevertheless, its performance is comparable to other state-of-the-art actor implementations for the JVM.

Acknowledgments. The authors thank Martin Steffen for a discussion on the programming model, Ilham Kurnia and Yannick Welsch for proof-reading, and the anonymous ECOOP reviewers for their helpful comments.

References

1. Ábrahám, E., Grabe, I., Grüner, A., Steffen, M.: Behavioral interface description of an object-oriented language with futures and promises. Journal of Logic and Algebraic Programming 78(7), 491–518 (Aug 2009)
2. Ábrahám, E., Grüner, A., Steffen, M.: Abstract interface behavior of object-oriented languages with monitors. Theor. Comput. Sci. 43(3), 322–361 (2008)
3. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press (1986)
4. America, P.: Issues in the design of a parallel object-oriented language. Form. Asp. Comput. 1(4), 366–411 (1989)
5. Andrews, G.R., Coffin, M., Elshoff, I., Nilson, K., Townsend, G., Olsson, R.A., Purdin, T.: An overview of the SR language and implementation. TOPLAS 10(1), 51–86 (1988)
6. Armstrong, J.: Making reliable distributed systems in the presence of software errors. Ph.D. thesis, The Royal Institute of Technology, Stockholm, Sweden (2003)
7. Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., Quilici, R.: Programming, deploying, composing, for the Grid. In: Cunha, J.C., Rana, O.F. (eds.) Grid Computing: Software Environments and Tools. Springer (Jan 2006)
8. Baker, Jr., H.G., Hewitt, C.: The incremental garbage collection of processes. In: Proceedings of the 1977 symposium on Artificial intelligence and programming languages. pp. 55–59. ACM (Aug 1977)
9. Benton, N., Cardelli, L., Fournet, C.: Modern concurrency abstractions for C#. In: Magnusson, B. (ed.) ECOOP. LNCS, vol. 2374, pp. 415–440. Springer (Jun 2002)
10. Bloom, B., Field, J., Nystrom, N., Östlund, J., Richards, G., Strniša, R., Vitek, J., Wrigstad, T.: Thorn: robust, concurrent, extensible scripting on the JVM. SIGPLAN Not. 44(10), 117–136 (2009)
11. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: Nicola [46], pp. 316–330
12. Briot, J.P., Guerraoui, R., Lohr, K.P.: Concurrency and distribution in object-oriented programming. ACM Comput. Surv. 30(3), 291–329 (1998)

13. Caromel, D.: Towards a method of object-oriented concurrent programming. Communications of the ACM 36(9), 90–102 (1993)
14. Caromel, D., Henrio, L., Serpette, B.P.: Asynchronous and deterministic objects. In: Jones, N.D., Leroy, X. (eds.) POPL. pp. 123–134. ACM (2004)
15. Caromel, D., Mateu, L., Pothier, G., Éric Tanter: Parallel object monitors. Concurrency and Computation: Practice and Experience 20(12), 1387–1417 (2008)
16. Caromel, D., Mateu, L., Éric Tanter: Sequential object monitors. In: Odersky, M. (ed.) ECOOP. LNCS, vol. 3086, pp. 316–340. Springer (Jun 2004)
17. Clarke, D., Potter, J., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA. pp. 48–64. ACM Press (Oct 1998)
18. Clarke, D., Wrigstad, T., Östlund, J., Johnsen, E.B.: Minimal ownership for active objects. In: Ramalingam, G. (ed.) APLAS. LNCS, vol. 5356. Springer (2008)
19. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, LNCS, vol. 4350. Springer (2007)
20. Creeger, M.: Multicore cpus for the masses. Queue 3(7), 64–ff (2005)
21. Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J.R., Levi, S.: Language support for fast and reliable message-based communication in Singularity OS. SIGOPS Oper. Syst. Rev. 40(4), 177–190 (2006)
22. Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theor. Comput. Sci. 103(2), 235–271 (1992)
23. Fischer, J., Majumdar, R., Millstein, T.: Tasks: language support for event-driven programming. In: Ramalingam, G., Visser, E. (eds.) PEPM. pp. 134–143. ACM (2007)
24. Flatt, M., Krishnamurthi, S., Felleisen, M.: A programmer's reduction semantics for classes and mixins. Formal Syntax and Semantics of Java 1523, 241–269 (1999)
25. Fournet, C., Gonthier, G.: The reflexive CHAM and the join-calculus. In: POPL. pp. 372–385. ACM (Jan 1996)
26. Georges, A., Buytaert, D., Eeckhout, L.: Statistically rigorous Java performance evaluation. In: Gabriel, R.P., Bacon, D.F., Lopes, C.V., Jr., G.L.S. (eds.) OOPSLA. pp. 57–76. ACM (2007)
27. Haack, C., Poll, E.: Type-based object immutability with flexible initialization. In: Drossopoulou, S. (ed.) ECOOP. LNCS, vol. 5653, pp. 520–545. Springer (2009)
28. Haack, C., Poll, E., Schäfer, J., Schubert, A.: Immutable objects for Java-like languages. In: Nicola [46], pp. 347–362
29. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theor. Comput. Sci. 410(2-3), 202–220 (2009)
30. Halstead, Jr., R.H.: Multilisp: a language for concurrent symbolic computation. ACM Trans. Program. Lang. Syst. 7(4), 501–538 (1985)
31. Hasselbring, W.: Programming languages and systems for prototyping concurrent applications. ACM Comput. Surv. 32(1), 43–79 (2000)
32. Haustein, M., Löhr, K.P.: JAC: declarative Java concurrency. Concurrency and Computation: Practice and Experience 18(5), 519–546 (2006)
33. Hewitt, C., Bishop, P., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: IJCAI. pp. 235–245. William Kaufmann (Aug 1973)
34. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. 23(3), 396–450 (May 2001)
35. JCoBox website. http://softech.cs.uni-kl.de/~jcobox (2010)
36. Jl5 polyglot extension. http://www.cs.ucla.edu/~todd/research/polyglot5.html
37. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and Systems Modeling 6(1), 35–58 (Mar 2007)

38. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the JVM platform: a comparative analysis. In: PPPJ. pp. 11–20. ACM (2009)
39. Keen, A.W., Ge, T., Maris, J.T., Olsson, R.A.: JR: Flexible distributed programming in an extended java. ACM Trans. Program. Lang. Syst. 26(3), 578–608 (2004)
40. Lea, D.: A Java fork/join framework. In: JAVA '00: Proceedings of the ACM 2000 conference on Java Grande. pp. 36–43. ACM (2000)
41. Leijen, D., Schulte, W., Burckhardt, S.: The design of a task parallel library. SIGPLAN Not. 44(10), 227–242 (2009)
42. Lieberman, H.: Concurrent object-oriented programming in Act 1. In: Yonezawa, A., Tokoro, M. (eds.) Object-Oriented Concurrent Programming. pp. 9–36. MIT Press (1987)
43. Liskov, B., Shrira, L.: Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In: PLDI. pp. 260–267 (1988)
44. Miller, M.S., Tribble, E.D., Shapiro, J.S.: Concurrency among strangers. In: Nicola, R.D., Sangiorgi, D. (eds.) TGC. LNCS, vol. 3705, pp. 195–229. Springer (2005)
45. Morris, B.: Cactive and friends. Symbian Developer Network (Jun 2008)
46. Nicola, R.D. (ed.): Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings, LNCS, vol. 4421. Springer (2007)
47. Niehren, J., Schwinghammer, J., Smolka, G.: A concurrent lambda calculus with futures. Theor. Comput. Sci. 364(3), 338–356 (Nov 2006)
48. Nierstrasz, O.M.: Active objects in Hybrid. In: OOPSLA. pp. 243–253. ACM Press (1987)
49. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An extensible compiler framework for Java. In: Hedin, G. (ed.) CC. LNCS, vol. 2622, pp. 138–152. Springer (2003)
50. Ousterhout, J.: Why threads are a bad idea (for most purposes) (1996), invited talk at the 1996 USENIX Conference
51. Philippsen, M.: A survey of concurrent object-oriented languages. Concurrency – Practice and Experience 12(10), 917–980 (2000)
52. Polyglot website. http://www.cs.cornell.edu/projects/polyglot/ (Mar 2010)
53. Rausch, A., Reussner, R., Plasil, F., Mirandola, R. (eds.): The Common Component Modeling Example: Comparing Software Component Models, LNCS, vol. 5153. Springer (2008)
54. Rodriguez, N., Rossetto, S.: Integrating remote invocations with asynchronism and cooperative multitasking. Parallel Processing Letters 18(1), 71–85 (2008)
55. Schäfer, J., Poetzsch-Heffter, A.: CoBoxes: Unifying active objects and structured heaps. In: Barthe, G., de Boer, F.S. (eds.) FMOODS. LNCS, vol. 5051, pp. 201–219. Springer (2008)
56. Srinivasan, S., Mycroft, A.: Kilim: Isolation-typed actors for java. In: Vitek, J. (ed.) ECOOP. LNCS, vol. 5142, pp. 104–128. Springer (Jul 2008)
57. Sutter, H.: The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb's Journal 30(3) (Mar 2005)
58. The computer language benchmarks game. http://shootout.alioth.debian.org
59. The Java RMI Specification. http://java.sun.com/products/jdk/rmi/ (Dec 2009)
60. The OSGi specification release 4 version 4.2. http://www.osgi.org (Sep 2009)
61. Van Cutsem, T., Mostinckx, S., Boix, E.G., Dedecker, J., Meuter, W.D.: Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In: SCCC. pp. 3–12. IEEE Computer Society (2007)
62. Yonezawa, A., Briot, J.P., Shibayama, E.: Object-oriented concurrent programming ABCL/1. In: Meyrowitz, N. (ed.) OOPSLA. pp. 258–268. ACM Press (1986)