CoBoxes: Unifying Active Objects and Structured Heaps

Jan Schäfer* and Arnd Poetzsch-Heffter

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

Abstract. Concurrent programming in object-oriented languages is a notoriously difficult task. We propose coboxes – a novel language concept which combines and generalizes active objects and techniques for heap structuring. CoBoxes realize a data-centric approach that guarantees mutual-exclusion for groups of objects. This is important for more complex data structures with invariants ranging over several objects. CoBoxes support multiple entry objects, several cooperating tasks in a cobox, and nesting of coboxes for composition and internal parallelism. Communication between coboxes is handled by asynchronous method calls with futures, which is in particular suitable for distributed programming. In this paper, we explain how aspects of concurrent programming can be solved by coboxes. We develop a core language for coboxes and present a formal operational semantics for this language.

1 Introduction

Today's programming languages support concurrency mainly by multi-threading. Especially in object-oriented settings, multi-threaded programming is notoriously hard. The programmer has to control the shared state of threads which is difficult in object-oriented programming because state sharing is a dynamic property depending on the reference structure in the heap. Another problem for thread safety is the fact that programming invariants often depend on several objects. Thus, after violating an invariant by modifying some object X, a thread needs to work on other objects to reestablish the invariant before another thread can access X. As threads are based on preemptive scheduling and every thread may be suspended at any time leading to an arbitrary interleaving of threads, the programmer must explicitly prevent certain interleavings by using locking, a fairly primitive and error-prone programming construct.

Whereas preemptive scheduling causes problems within a single process, it is successful for processes on the operating system level. The reason is simple: processes do not share state and communicate via message passing. However, to substitute all method calls in OO programming by asynchronous message passing would throw out the baby with the bath water. The state space of a process would be restricted to a single object and the well-understood sequential

* Supported by the Deutsche Forschungsgemeinschaft (German Research Foundation)

reasoning that can be used for thread-safe parts of programs with synchronous method calls could no longer be applied.

In this paper, we present a model that tries to combine the best of two worlds. It builds upon and integrates techniques for active objects [7] and for structuring the heap into groups of objects, so-called *boxes*, which have only been presented in a sequential setting [37] (*sequential boxes*). We call this symbiosis *concurrent boxes* or *coboxes*. Like sequential boxes, coboxes hierarchically partition the heap into runtime components consisting of multiple objects. To use coboxes, the programmer simply has to declare certain classes as cobox classes. Instantiating such a class creates a cobox together with its *main* object (also called *owner* object). A cobox may have multiple *tasks* which are scheduled cooperatively and where only one at a time may be active. Thus, within a cobox programs are executed sequentially with programmer-defined points of suspension and scheduling. CoBoxes run concurrently and communicate via message passing. To support composition of boxes and a restricted form of internal concurrency, coboxes can be nested, i.e. coboxes can contain other coboxes.

Contributions and Overview. This paper extends the sequential box concept that we developed in [37] to concurrency. The novel concurrency model generalizes active objects with asynchronous messages and futures to multiple object components. The resulting concurrent programming model enables the programmer to develop instantiable and composable components of scalable sizes. Although a cobox may expose several objects to the environment the model guarantees that tasks within a cobox are free of data-races and are executed atomically between suspension points.

After a discussion of related work, we explain and illustrate our programming model (Sect. 3). As central technical contribution, we present a formal small-step operational semantics of a core language for concurrent boxes and investigate some of its properties (Sect. 4). We conclude the paper after a discussion of the current limitations of our approach as well as possible solutions (Sect. 5).

2 Related Work and Motivation

Heap Structuring. A heap in OOLs is an unstructured graph of objects. The missing structure makes it difficult to reason about the behavior of object-oriented programs. Different approaches have been proposed to handle this problem. Ownership Types [11, 6, 34] and variants [2, 36, 38] statically structure the heap into ownership contexts. This can be used, for example, to ease program verification [13, 35] or to statically guarantee data-race and deadlock freedom in multi-threaded programs [5]. The concept of object ownership can also be encoded in a programming logic to achieve data-race [30] and deadlock [31] freedom by program verification. In a previous work, we used hierarchical structuring of the heap to modularly describe the behavior of sequential object-oriented runtime components [37].

Actors. The actor [1, 33] or active object model [7] treats every object as an actor. Actors run in parallel and communicate via asynchronous messages. To allow result values for asynchronous method calls, futures [33] are used. In general, actors only have a single thread of control. This makes it difficult to implement multiple interleaved control flows within an actor, as this has to be explicitly implemented. This problem has been solved by Creol [32, 12], for example, which supports multiple control flows in objects. Another problem is the flat object model of actors. In general, actors cannot have a complex aggregate state. Some approaches allow actors to have local or passive objects [7, 8], which are deep copied between actors or cannot be referenced by other actors at all. To the best of our knowledge, no actor model has been proposed yet which allows multiple entry objects or treats hierarchical nesting.

Monitors. Monitors [20, 27] couple data with procedures, which are guaranteed to run in mutual exclusion. They support multiple control flows coordinated by condition variables. The original monitor concept has been designed without pointers and the aliasing problem [28] in mind. The realization of monitors in mainstream OO-languages typically has two shortcomings. It merely supports internal concurrency and only protects single objects.

Several Java modifications have been proposed to overcome these shortcomings. Guava [3] distinguishes between monitors and objects and allows monitors with complex aggregate state consisting of multiple objects, but without multiple entry objects for a single monitor. JAC [22] is an approach to specify concurrency mechanisms in Java in a declarative way, but only on the granularity of single objects. *Sequential Object Monitors* (SOM) [9] separate scheduling of requests from objects that handle requests. SOM does not allow multiple entry objects, nor multiple tasks in a SOM. *Parallel Object Monitors* (POM) [10] generalize SOM, by allowing multiple concurrently running threads in a single POM. In addition, multiple objects can be controlled by a single POM. In contrast to SOM, POM does not guarantee data-race freedom.

Transactions. When talking about (software) transactions one must distinguish the implementation side and the language side. There are many proposals how to implement transactions, either using ordinary locks [25], or implementing a transactional memory in software only [39], in hardware [23], or both [24]. A still open question is how transactional behavior could be integrated into object-oriented languages. Atomic blocks [21] are one idea, another are atomicity annotations [15]. The problem of these pure code-centric approaches is that still high-level inconsistencies can occur if invariants range over several objects [41]. Atomic sets [41] are one answer to overcome this problem. Communication of threads as well as other waiting conditions within transactions are addressed in [40]. Although transactions can be implemented distributively, transactional memory is not a concept which can easily be generalized to the distributed setting. Thus we see transactional memory as an orthogonal concept which may be integrated in our model to allow truly parallel tasks within a single cobox, for

```
cobox class Consumer {
  Producer prod;
  Consumer(Producer p) { prod = p; }
  void consume() {
    Fut<Product> f = prod!produce();
    ... // do something else
    Product p = f.get; // wait for result
    ... // consume p
} }
```

```
interface Producer { Product produce(); }

cobox class SimpleProducer
  implements Producer {
  Product produce() {
    Product p = ... // produce p
    return p
  }
}
```

Fig. 1. Producer-Consumer example

example. An interesting starting point would be the work of Smaragdakis et al. [40].

3 Programming with Concurrent Boxes

The novel concept of our programming language are coboxes. Like objects, coboxes are runtime entities. CoBoxes are containers for objects. Every object exactly belongs to a single cobox for its entire lifetime. CoBoxes are also containers for cooperatively scheduled tasks, where at most one task is active and all other tasks are suspended. The important difference to other active object approaches is that activity is not bound to a single object, but to a set objects, namely the objects that belong to the same cobox. This also means that coboxes generalize active objects as active objects can be simulated by coboxes containing only single objects. All objects of a cobox can be referenced by other coboxes without any restriction. Moreover, coboxes can be nested, i.e. coboxes can be contained in other coboxes, which allows cobox-internal concurrency and reuse of cobox classes.

To explain our cobox concept we use a Java-like language called JCoBox. JCoBox is Java where the standard concurrency mechanisms are replaced by the following ones:

- Annotation of classes as cobox classes
- Asynchronous method invocation with futures as results
- Task coordination by await, get and yield expressions.

If none of these constructs is used, a JCoBox program behaves like a corresponding sequential Java program.

3.1 CoBoxes as Active Objects

If a cobox only consists of one object, it behaves like active objects in Creol [12]. We illustrate this with a simple producer-consumer example shown in Fig 1. Besides a Producer interface, the example contains a Consumer and SimpleProducer

```
cobox class PipelineProducer
  implements Producer {
  PreProducer pp = new PreProducer();
  FinalProducer fp = new FinalProducer();
  Product produce() {
    PreProduct p1 = pp!produce().await;
    Product p = fp!produce(p1).await;
    return p;
} }

cobox class PreProducer {
  PreProduct produce() { ... }
}

cobox class FinalProducer {
  Product produce(PreProduct p) {
    ...
} }
```

Fig. 2. Producer with a pipelined production

class, which both are declared as cobox classes. This means that objects of these classes always live in separate coboxes, as instantiating a cobox class creates a new cobox together with its *main* object. The Consumer object has a reference to a Producer object. To get a new Product, the consumer asynchronously invokes the produce method of the producer. That call immediately returns a future object of type Fut<Product>. While the producer is busy producing the product, the consumer can do something else in parallel. Eventually, the consumer uses get on the future object. This blocks the active task of the consumer until the future value is available, which is the case when the producer returns from the produce method.

Different coboxes can run concurrently. Thus, multiple consumers can concurrently access the same producer. The cobox semantics serializes such accesses. Each cobox manages an internal set of *tasks*. Whenever a method is invoked on an object of a cobox, a new task is created. At most one task is *active* in a cobox, all other tasks are *suspended*. An active task terminates if the execution of the invoked method terminates. It suspends itself by executing a yield statement or an await on a future. If no task is active, a scheduler selects one of the suspended tasks.

3.2 Nested CoBoxes and Internal Concurrency

To support concurrency within a cobox and to implement new cobox classes using existing ones, it is possible to create *nested* coboxes. A cobox is always nested in the cobox which created it. For example, the developer of the Producer class recognizes that the production process can be divided into two steps. The first step produces a PreProduct which can then be used in a second step to produce the final product. Thus, it has a production pipeline, which doubles the throughput of the producer. The pipelined producer implementation is shown in Fig. 2. The implementation of the produce method, asynchronously invokes the produce methods of the nested producers and uses await to wait for the result. Thus, the task that executes the produce method suspends itself after it has sent the first message and waits for the result. While the nested coboxes concurrently work on the production, the Producer can accept other produce

```
class LinkedList<V> { ... }

cobox class ProducerPool {
  LinkedList<Producer> prods;
  ProducerPool() {
    prods =
      new LinkedList<Producer>();
    // ... fill pool with producers
  }
  Producer getProducer() {
    return new ProducerProxy(this);
} }
```

```
class ProducerProxy implements Producer {
  ProducerPool pool;
  ProducerProxy(ProducerPool p) {
    pool = p;
  }
  Product produce() {
    while (pool.prods.isEmpty()) yield;
    Producer p = pool.prods.removeFirst();
    Product res = p!produce().await;
    pool.prods.append(p);
    return res;
} }
```

Fig. 3. Producer pool example

messages and activate tasks, resulting in an interleaved execution of multiple tasks in the producer, as well as a parallel execution of the nested pre- and final producers. Notice that the use of nested coboxes is transparent to users of PipelinedProducer objects.

3.3 Multiple Entry Objects

The granularity of an object is often insufficient to model complex components. In general, a component has a complex internal state consisting of multiple objects as well as multiple objects which act as entry objects to the component, extending its external interface. For example, in the producer-consumer scenario, it might be useful to improve parallelism by introducing a pool of producers. The producer pool implementation is shown in Fig. 3. The ProducerPool cobox class has a field prods holding a list of currently free producers. A client can use the getProducer method to obtain a new Producer instance. The getProducer method returns an instance of a ProducerProxy. As the ProducerProxy class is not a cobox class, all its instances are contained in the ProducerPool cobox that created them. To better understand the cobox structure that appears at runtime, we show a runtime snapshot of the producer pool example in Fig. 4. It shows the nesting of the coboxes as well as which objects belong to which coboxes.

A ProducerPool is an example of a more realistic cobox. It consists of the main object of type ProducerPool, further ProducerProxy entry objects, the objects implementing the LinkedList, and the nested PipelinedProducer coboxes. The interesting part is the implementation of the produce method. First, it ensures that the list of free producers is not empty. This is done in a while loop, suspending the running task with a yield statement, until the list is not empty anymore. This illustrates a non-blocking wait. (So far, we did not include orthogonal concepts like guards or wait conditions to the language; cf. Sect. 5 for a discussion). As the active task always has exclusive access to all objects of its cobox until it explicitly suspends itself, the code in the produce method of class ProducerProxy

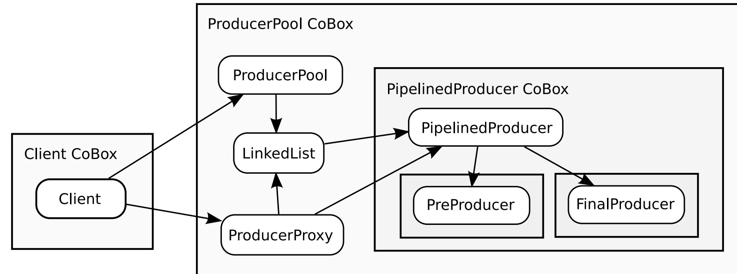

Fig. 4. A runtime snapshot of the producer pool example. Rectangles denote coboxes, rounded rectangles denote objects, edges with arrows denote references.

can safely remove the first entry of the free producers list, as the list belongs to the same cobox as the ProducerProxy object. After removing the first element, the task asynchronously calls the produce method of the producer and suspends itself until the result is available by using the await operation. Finally, the used producer is appended to the list of free producers again.

3.4 Controlling Reentrant Calls

An important aspect of coboxes is that coboxes can control reentrant calls. Reentrant calls even happen in a purely sequential setting and complicate the reasoning about object-oriented programs. Figure 5 shows a typical subject-observer scenario. A Subject has an internal state and allows to register Observers which get notified about state changes. When notified, the Observer calls back to the Subject to get the new state. If this scenario would be implemented in sequential Java, the method call o.changed(this) would result in a reentrant call by the called observer. While the observers are notified about the state change, theoretically arbitrary calls to the Subject could happen. In the cobox implementation, the observers are notified by an asynchronous call. As the notifying task does not suspend itself, no other method can be activated in the cobox of the Subject until the task has finished. Thus, the s.getState() calls of the observers are delayed until all observers have been notified. This guarantees that the Subject cannot be changed while the observers are notified. Note that even though we are in a concurrent setting, we get stronger guarantees than in the sequential Java setting, which eases behavioral reasoning.

The Java behavior can be simulated in JCoBox by calling await on the result of the o!changed(this) call, as in this case the current task suspends itself until the future value is available, i.e. until the called method has been finished. In between, other tasks can be activated, for example, tasks created by reentrant calls. Using a get instead of an await would result in a deadlock, because the active task waits for the result of the future without allowing other tasks to be activated – it *blocks* the cobox. Thus tasks created by reentrant calls cannot be activated. Note that this only applies if the future value is calculated by a task

```
cobox class Subject {
  LinkedList<Observer> obs =
    new LinkedList<Observer>();
  State state = ... // initialize state
  State getState() { return state; }
  void updateState(State newState) {
    state = newState; notifyObservers();
  }
  void notifyObservers() {
    for (Observer o : obs) { o!changed(this); }
  }
  void register(Observer o) { ... }
}

cobox class Observer {
  void changed(Subject s) {
    // callback
    State state = s.getState();
    // ... do something
  }
}
```

Fig. 5. Subject-Observer example

of a different cobox. If the future was the result of a self-call, that is, a call on an object of the same cobox, then such dependencies are resolved by activating the task responsible for calculating the future value.

4 Formal Semantics

In this section we present a formal dynamic semantics for a core language of coboxes, called JCoBox$^{\text{C}}$. JCoBox$^{\text{C}}$ shares ideas from Featherweight Java [29], ClassicJava [16], Creol [12] and a previous formalization of a sequential language with boxes by the authors [37]. JCoBox$^{\text{C}}$ only contains the core object-oriented language constructs for inheritance with method overriding and synchronous method calls as well as the new features for cobox classes, asynchronous method calls, futures, and task suspension. Other language features could be added as usual. Typing for JCoBox$^{\text{C}}$ is similar to Java. The only necessary adaptation is the integration of futures which is straightforward.

4.1 Syntax

Figure 6 shows the abstract syntax of our language. Lower-case letters represent meta-variables, capital letters represent subsets of syntactic categories, and overbars indicate sequences of elements ($\bullet$ is the empty sequence, $\circ$ is concatenation of sequences). A program p is a set of class declarations D. To be executable a program must contain a cobox class Main with a main method. Such a program is then executed by creating an instance of Main and executing the main method. A class declaration d consists of a class name c, a super class name c', a list of field declarations $\overline{c\ f}$, and a set of method declarations H. We assume a predefined class Object with no fields, no methods and no super class. A class can be declared to be a cobox class by using the cobox keyword. Methods return the value of their body expression as result. New objects are created by the new c

$$
\begin{aligned}
p \in \textbf{Prog} &::= D \\
d \in D \subseteq \textbf{ClassDecl} &::= [\mathsf{cobox}]\ \mathsf{class}\ c\ \mathsf{extends}\ c'\ \{\ \overline{c\ f};\ H\ \} \\
h \in H \subseteq \textbf{MethDecl} &::= c\ n(\overline{c\ x})\{e\} \\
e \in E \subseteq \textbf{Expr} &::= x \mid \mathsf{null} \mid \mathsf{new}\ c \mid e.f \mid e.f = e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \\
&\quad\ \ e.n(\overline{e}) \mid e!n(\overline{e}) \mid e.\mathsf{get} \mid e.\mathsf{await} \mid \mathsf{yield} \mid e; e'
\end{aligned}
$$

$$c \in \text{class names}, n \in \text{method names}, f \in \text{field names}, x \in \text{variable names}$$

Fig. 6. Abstract syntax of JCoBox$^{\mathrm{C}}$

expression, all fields are initialized to null, constructors are not supported. let expressions introduce new local variables and can be used for sequential composition. An asynchronous method invocation, indicated by an exclamation mark, always results in a reference to a future. A future is an object of the special class Fut (details below). The value of a future can be obtained by either using get or await expressions with different blocking semantics. A yield suspends the currently active task allowing other tasks to proceed.

In the syntax above, we listed synchronous calls and sequential composition to stress that they are covered by the core language. However, in the following, we treat them as syntactic sugar to further compactify the semantics: $e.n(\overline{e}) \equiv e!n(\overline{e}).\mathsf{get}$ and $e; e' \equiv \mathsf{let}\ x = e\ \mathsf{in}\ e'$, where $x \notin FV(e')$.

4.2 Semantic Entities

A new aspect of the cobox semantics is that the heap is explicitly partitioned into the subheaps corresponding to coboxes. In particular, each cobox has its own objects. Compared to a formalization with a global heap and additional mappings capturing the box structure and the information to which box an object belongs, an explicit partitioning allows the creation of objects in a box-local way without knowing the global state. This simplifies modular treatment of box implementations (see [37]) and reflects distributed object-oriented programming.

The necessary semantic entities are shown in Figure 7. The state b of a cobox is represented by a tuple $\mathsf{B}\langle w, O, B, T, M, t_\epsilon, m_\epsilon\rangle$, consisting of a cobox reference w, a set of objects O, a set of nested coboxes B, a set of *suspended* tasks T, a set of incoming messages M, an optional *active* task t_ϵ and an optional current message m_ϵ. To simplify wording, we will not distinguish between a cobox and its state when it is clear from the context. A cobox reference is a sequence of cobox identifiers ι_b describing the path of the cobox in the nesting hierarchy of coboxes. This means that if a cobox has reference w then all its directly nested coboxes have references of the form $w.\iota_b$, whereas the *root* cobox has a single ι_b as reference.

Objects and their states are represented by triples $\mathsf{O}\langle \iota_o, c, \overline{v}\rangle$ with a box-unique object identifier ι_o, a class name c, and the list of field values $\overline{v}$. An object o always belongs to a certain cobox. If w is the reference of this cobox

$$
\begin{array}{rll}
b \in B \subseteq \textbf{CoBox} ::= & \mathsf{B}\langle w, O, B, T, M, t_\epsilon, m_\epsilon \rangle & \text{coboxes} \\
o \in O \subseteq \textbf{Obj} ::= & \mathsf{O}\langle \iota_o, c, \overline{v} \rangle & \text{objects} \\
 | & \mathsf{O}\langle \iota_o, \mathsf{Fut}, v_\epsilon \rangle & \text{future objects} \\
t \in T \subseteq \textbf{Task} ::= & \mathsf{T}\langle r, e \rangle & \text{tasks} \\
m \in M \subseteq \textbf{Msg} ::= & \mathsf{M}\langle r, r', n(\overline{v}) \rangle & \text{call messages} \\
 | & \mathsf{M}\langle r, v \rangle & \text{return messages} \\
v \in V \subseteq \textbf{Value} ::= & r \mid \mathsf{null} & \text{values} \\
r \in R \subseteq \textbf{Ref} ::= & w.\iota_o & \text{object references} \\
w \in W \subseteq \textbf{CoRef} ::= & w.\iota_b \mid \iota_b & \text{cobox references} \\
e \in E \subseteq \textbf{Expr} ::= & \ldots \mid v & \text{extended expressions} \\
l \in \textbf{Lab} ::= & \uparrow m \mid \downarrow m \mid \tau & \text{labels} \\
\iota_o \in \textbf{ObjId} & & \text{object identifiers} \\
\iota_b \in \textbf{BoxId} & & \text{cobox identifiers}
\end{array}
$$

Fig. 7. Semantics entities of JCoBox$^{\text{C}}$. Optional terms are indicated by an ϵ as index and may be ϵ. Small capital letters like B or K are used as "constructors" to distinguish the different semantic entities syntactically.

and ι_o the object identifier of o, the globally unique name of o is $w.\iota_o$.[1] The fields of an object can only be accessed by objects of the same cobox. Objects of other coboxes must use method calls.

Every cobox has a set of suspended tasks T and an optional active task t_ϵ. A task t corresponds to a single method activation and is represented by a pair $\mathsf{T}\langle r, e \rangle$ where r references the future that will receive the result of the call and where e is the expression to be reduced. CoBoxes communicate by exchanging messages. A message is either a call or return message. A call message $\mathsf{M}\langle r, r', n(\overline{v}) \rangle$ consists of a future reference r, which waits for the result of the call, a receiver reference r' together with the called method name n, and argument values $\overline{v}$. A return message $\mathsf{M}\langle r, v \rangle$ consists of the target future reference r and the result value v.

Futures are modeled as objects of the predefined class Fut with the following implementation:

```
class Fut extends Object { Object await() { this.await } }
```

Instances of the Fut class cannot be created explicitly, but are implicitly created by an asynchronous method call. A future object has a single *optional* value v_ϵ, which is ϵ until the value of the future is available. This value cannot be set by field updates and may only be read by e.get or e.await expressions. Future references can be used like object references, in particular they can be passed to and used by other coboxes. Therefore, the Fut class contains an await() method that is not invoked explicitly, but used whenever a get or await operation is executed on a future object, not belonging to the current cobox. In that case,

[1] This is similar to the naming scheme in a tree-structured file system: coboxes correspond to directories, objects to files.

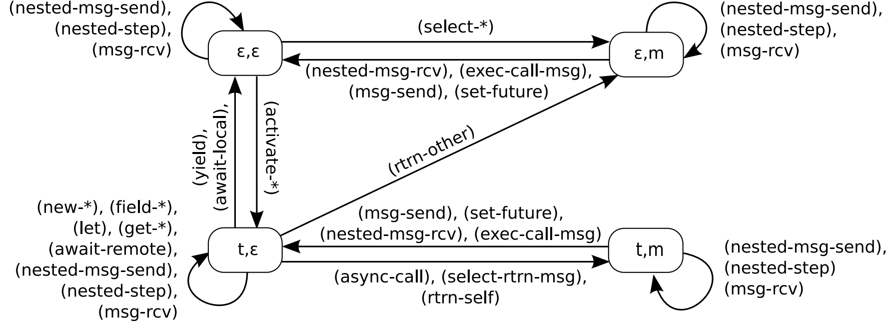

Fig. 8. Overview of the different reduction rules, showing the possible state transitions.

the await() method is asynchronously called on the future with an immediate get or await, respectively, resulting in a new future object, which acts as a proxy for the former future.

Evaluation Contexts. For a compact presentation we define some evaluation contexts [14]. An evaluation context is a term with a "hole" $\Box$ at a certain position. By writing $e_\Box[e']$ that hole is replaced by term e'. In our syntax, holes can only appear at positions where expressions are expected.

$$\begin{aligned} e_\Box &::= \Box \mid e_\Box.f \mid e_\Box.f = e \mid v.f = e_\Box \mid e_\Box.\mathsf{get} \mid e_\Box.\mathsf{await} \mid \\ &\quad\; \mathsf{let}\ x = e_\Box\ \mathsf{in}\ e \mid e_\Box!m(\overline{e}) \mid v!m(\overline{v}, e_\Box, \overline{e}) \\ t_\Box &::= \mathsf{T}\langle r, e_\Box \rangle \end{aligned}$$

4.3 Transition Rules

The dynamic semantics of JCoBox$^{\mathrm{C}}$ is defined in terms of a labeled transition system as a relation on coboxes and labels:

$$\longrightarrow \subseteq \mathbf{CoBox} \times \mathbf{Lab} \times \mathbf{CoBox}\,.$$

The notation $b \xrightarrow{l} b'$ means that cobox b can be reduced to b' with label l. We distinguish three kinds of transitions: internal steps ($l = \tau$), receiving a message ($l = \downarrow m$), and sending a message ($l = \uparrow m$). The transition rules are shown in Figures 10 and 12, which use auxiliary predicates and functions defined in Figures 9 and 11. To better understand in which order the different rules may be executed, we give an overview in terms of an automaton shown in Figure 8. The automaton states represent the currently active task and the current message of a cobox. The edges are labeled with the rules that can be applied. Objects, nested boxes, suspended tasks, and incoming messages are not shown in the automaton.

$$\frac{\textsf{cobox class } c \ \ldots}{cobox(c)} \qquad \frac{}{fields(\textsf{Object}) = \bullet} \qquad \frac{_ \textsf{ class } c \textsf{ extends } c' \ \{ \ \overline{c'' \ f}; \ H \ \}}{fields(c) = fields(c') \circ \overline{f}}$$

$$\frac{_ \textsf{ class } c \ \ldots \ \{ \ \ldots \ _ \ n(\overline{_ \ x})\{e\} \ \ldots \ \}}{mbody(c,n) = (\overline{x})e} \qquad \frac{_ \textsf{ class } c \textsf{ extends } c' \ \{ \ \ldots \ ; H\} \qquad n \text{ not in } H}{mbody(c,n) = mbody(c',n)}$$

Fig. 9. Auxiliary predicates and functions extracting information from the (underlying) program code

Expressions. We start by explaining how expressions are reduced, assuming that some task is active in the current cobox and that no current message exists (state t, ϵ). let expressions are handled by standard capture-avoiding substitution (LET). New objects of non-cobox classes are created in the current cobox with all values initialized to null (NEW-OBJ). If a class is annotated with cobox, a new nested cobox is created and the new object is created in that cobox instead of the current cobox (NEW-COBOX). Field selects and field updates are only allowed on objects of the current cobox (FIELD-SELECT, FIELD-UPDATE). The rules select/update the ith value of the corresponding ith field and reduce to the old/new value. An asynchronous call is reduced to a reference of a new future object, which is added to the object set of the cobox. In addition, a call message is created and set as current message (ASYNC-CALL). A yield expression moves the active task into the set of suspended tasks, giving other tasks the possibility to be activated (YIELD).

Future Expressions. When a future reference r is used, two cases can be distinguished: *local futures* and *remote futures*. Local futures belong to the current cobox ($r = w.\iota_o$, where w is the reference of the current cobox), remote futures belong to a different cobox ($r \neq w.\iota_o$). A get expression on a local future is reduced to the value of the future if the value is available (GET-LOCAL-SUCCESS), otherwise the current task is blocked and remains active until the value becomes available. However, to prevent deadlocks, we have to treat synchronous self-calls as a special case. If a future is computed by a task of the same cobox ($t_\square[w.\iota_o.\textsf{get}] \in T$) then that task is activated, and the currently active task is suspended (GET-LOCAL-SELF) and later reactivated by (RTRN-SELF). Like a yield, an await on a local future suspends the currently active task (AWAIT-LOCAL). On a remote future reference r an await or get expression is transformed into an asynchronous method call of the await() method on that future with an immediate await or get, i.e. into a $r!await()$.await or $r!await()$.get expression, respectively (AWAIT-REMOTE, GET-REMOTE). These calls create corresponding local futures in the current cobox that are handled by the rules explained above.

Task Termination. A task terminates if its expression has been reduced to a single value. If the task was created due to a self-call and another task in the current cobox is blockingly waiting for the result ($t_\square[r.\textsf{get}] \in T$), that task is di-

(LET)

$$\mathsf{B}\langle w, O, B, T, M, t_\Box[\mathsf{let}\ x = v\ \mathsf{in}\ e], \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T, M, t_\Box[[v/x]e], \epsilon\rangle$$

(NEW-OBJ)

$$\frac{\neg cobox(c) \qquad \iota_o \text{ fresh in } O \qquad \overline{v} = \overline{\mathsf{null}} \qquad |\overline{v}| = |fields(c)| \qquad o = \mathsf{O}\langle \iota_o, c, \overline{v}\rangle}{\mathsf{B}\langle w, O, B, T, M, t_\Box[\mathsf{new}\ c], \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O \cup \{\, o \,\}, B, T, M, t_\Box[w.\iota_o], \epsilon\rangle}$$

(NEW-COBOX)

$$\frac{\begin{array}{c} cobox(c) \qquad \iota_b \text{ fresh in } B \\ \overline{v} = \overline{\mathsf{null}} \qquad |\overline{v}| = |fields(c)| \qquad o = \mathsf{O}\langle \iota_o, c, \overline{v}\rangle \qquad b = \mathsf{B}\langle w.\iota_b, \{\, o \,\}, \varnothing, \varnothing, \varnothing, \epsilon, \epsilon\rangle \end{array}}{\mathsf{B}\langle w, O, B, T, M, t_\Box[\mathsf{new}\ c], \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B \cup \{\, b \,\}, T, M, t_\Box[w.\iota_b.\iota_o], \epsilon\rangle}$$

(FIELD-SELECT)

$$\frac{\mathsf{O}\langle \iota_o, c, \overline{v}\rangle \in O \qquad fields(c) = \overline{f}}{\begin{array}{c}\mathsf{B}\langle w, O, B, T, M, t_\Box[w.\iota_o.f_i], \epsilon\rangle \\ \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T, M, t_\Box[v_i], \epsilon\rangle\end{array}}$$

(FIELD-UPDATE)

$$\frac{\begin{array}{c} O = O' \uplus \{\, \mathsf{O}\langle \iota_o, c, \overline{v}\rangle \,\} \qquad fields(c) = \overline{f} \\ O'' = O' \cup \{\, \mathsf{O}\langle \iota_o, c, [v/v_i]\overline{v}\rangle \,\} \end{array}}{\begin{array}{c}\mathsf{B}\langle w, O, B, T, M, t_\Box[w.\iota_o.f_i = v], \epsilon\rangle \\ \xrightarrow{\tau} \mathsf{B}\langle w, O'', B, T, M, t_\Box[v], \epsilon\rangle\end{array}}$$

(ASYNC-CALL)

$$\frac{\begin{array}{c} \iota_o \text{ fresh in } O \\ m = \mathsf{M}\langle w.\iota_o, r, n(\overline{v})\rangle \qquad o = \mathsf{O}\langle \iota_o, \mathsf{Fut}, \epsilon\rangle \end{array}}{\begin{array}{c}\mathsf{B}\langle w, O, B, T, M, \mathsf{T}\langle r', e_\Box[r!n(\overline{v})]\rangle, \epsilon\rangle \\ \xrightarrow{\tau} \mathsf{B}\langle w, O \cup \{\, o \,\}, B, T, M, \mathsf{T}\langle r', e_\Box[w.\iota_o]\rangle, m\rangle\end{array}}$$

(YIELD)

$$\frac{t = t_\Box[\mathsf{yield}]}{\begin{array}{c}\mathsf{B}\langle w, O, B, T, M, t, \epsilon\rangle \\ \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T \cup \{\, t \,\}, M, \epsilon, \epsilon\rangle\end{array}}$$

(GET-LOCAL-SUCCESS)

$$\frac{\mathsf{O}\langle \iota_o, \mathsf{Fut}, v\rangle \in O}{\begin{array}{c}\mathsf{B}\langle w, O, B, T, M, t_\Box[w.\iota_o.\mathsf{get}], \epsilon\rangle \\ \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T, M, t_\Box[v], \epsilon\rangle\end{array}}$$

(GET-LOCAL-SELF)

$$\frac{t = t_\Box[w.\iota_o.\mathsf{get}] \qquad t' = \mathsf{T}\langle w.\iota_o, e\rangle}{\begin{array}{c}\mathsf{B}\langle w, O, B, T \uplus \{\, t' \,\}, M, t, \epsilon\rangle \\ \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T \cup \{\, t \,\}, M, t', \epsilon\rangle\end{array}}$$

(AWAIT-LOCAL)

$$\frac{t = t_\Box[w.\iota_o.\mathsf{await}]}{\begin{array}{c}\mathsf{B}\langle w, O, B, T, M, t, \epsilon\rangle \\ \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T \cup \{\, t \,\}, M, \epsilon, \epsilon\rangle\end{array}}$$

(GET-REMOTE)

$$\frac{r \neq w.\iota_o}{\begin{array}{c}\mathsf{B}\langle w, O, B, T, M, t_\Box[r.\mathsf{get}], \epsilon\rangle \\ \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T, M, t_\Box[r!await().\mathsf{get}], \epsilon\rangle\end{array}}$$

(AWAIT-REMOTE)

$$\frac{r \neq w.\iota_o}{\begin{array}{c}\mathsf{B}\langle w, O, B, T, M, t_\Box[r.\mathsf{await}], \epsilon\rangle \\ \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T, M, t_\Box[r!await().\mathsf{await}], \epsilon\rangle\end{array}}$$

(RTRN-OTHER)

$$\frac{t_\Box[r.\mathsf{get}] \notin T}{\begin{array}{c}\mathsf{B}\langle w, O, B, T, M, \mathsf{T}\langle r, v\rangle, \epsilon\rangle \\ \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T, M, \epsilon, \mathsf{M}\langle r, v\rangle\rangle\end{array}}$$

(RTRN-SELF)

$$\frac{t' = t_\Box[r.\mathsf{get}]}{\mathsf{B}\langle w, O, B, T \uplus \{\, t' \,\}, M, \mathsf{T}\langle r, v\rangle, \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T, M, t', \mathsf{M}\langle r, v\rangle\rangle}$$

Fig. 10. Expression reduction rules

rectly activated (RTRN-SELF). Together with rule (GET-LOCAL-SELF) this simulates a stack-like behavior for synchronous self-calls. If no task in the current cobox is blockingly waiting for the result, the active task is set to ϵ, i.e. it vanishes (RTRN-OTHER). In both cases a return message with the corresponding result value is set as current message. That message is treated in a next step by a message handling rule.

Tasks and Message Handling. Each task corresponds to a method activation. As method calls are initiated by call messages, a new task is created whenever the current message is a call message targeting an object of the current cobox (EXEC-CALL-MSG). The new task gets the reference of the future, which will hold the return value and the body expression of the called method, where formal parameters are substituted by actual parameters. A yield statement is prepended, so that newly created tasks are treated like yielded tasks in the set of suspended tasks. If the current message is a return message targeting a future of the current cobox, the value of the future is set to the value of the return message (SET-FUTURE).

A cobox is *idle* if it has no active task and no current message (state ϵ, ϵ). In an idle cobox, a task can be activated if it either was suspended by a yield expression (ACTIVATE-YIELDED-TASK), or it was suspended by an await expression on a future whose value is available (ACTIVATE-WAITING-TASK). An idle cobox can as well select a message from the set of incoming messages and make it its new current message (SELECT-*). Whereas a call message can only be selected by an idle cobox, return messages can also be selected if a current task is active. This design decision reflects the goal to control all incoming calls, but allow return messages to reach their destination in nested coboxes without being blocked by active tasks in the surrounding box.

Messages are routed following the nesting structure. The parent coboxes are responsible for handling messages coming from and going to nested coboxes. This allows the parent cobox to control external communication of nested coboxes. If the current message is not targeting the current cobox, nor any nested cobox, it is *emitted* by rule (MSG-SEND). Messages can be received by a cobox if they target the cobox itself or any nested cobox (MSG-RCV). Messages can be received in any state and are added to the set of incoming messages. This reflects the asynchronous character of coboxes. If the current message targets a nested cobox, i.e. a nested cobox can receive that message, that message is send to the corresponding cobox by (NESTED-MSG-RCV). If a nested cobox emits a message that message is added to the set of incoming message of the parent cobox by (NESTED-MSG-SEND).

Internal steps of nested coboxes are propagated to its parent cobox and indirectly to the outermost cobox at the root of the box tree by (NESTED-STEP). Both rules are completely independent of the state of the current cobox and thus may be applied at any time, modeling concurrent behavior.

$$\frac{m = \mathsf{M}\langle r, v\rangle}{\mathit{isReturn}(m)} \qquad \frac{m = \mathsf{M}\langle r, r', n(\overline{v})\rangle}{\mathit{target}(m) = r'} \qquad \frac{m = \mathsf{M}\langle r, v\rangle}{\mathit{target}(m) = r}$$

Fig. 11. Auxiliary predicates and functions

(EXEC-CALL-MSG)
$$\frac{\mathsf{O}\langle \iota_o, c, _\rangle \in O \qquad \mathit{mbody}(c, n) = (\overline{x})e \qquad t = \mathsf{T}\langle r, \mathsf{yield}; [w.\iota_o/\mathit{this}, \overline{v}/\overline{x}]e\rangle}{\mathsf{B}\langle w, O, B, T, M, t_\epsilon, \mathsf{M}\langle r, w.\iota_o, n(\overline{v})\rangle\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T \cup \{\, t \,\}, M, t_\epsilon, \epsilon\rangle}$$

(SET-FUTURE)
$$\frac{m = \mathsf{M}\langle w.\iota_o, v\rangle}{\mathsf{B}\langle w, O \uplus \{\, \mathsf{O}\langle \iota_o, \mathsf{Fut}, \epsilon\rangle \,\}, B, T, M, t_\epsilon, m\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O \cup \{\, \mathsf{O}\langle \iota_o, \mathsf{Fut}, v\rangle \,\}, B, T, M, t_\epsilon, \epsilon\rangle}$$

(ACTIVATE-WAITING-TASK)
$$\frac{\mathsf{O}\langle \iota_o, \mathsf{Fut}, v\rangle \in O}{\mathsf{B}\langle w, O, B, T \uplus \{\, t_\Box[w.\iota_o.\mathsf{await}] \,\}, M, \epsilon, \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T, M, t_\Box[v], \epsilon\rangle}$$

(ACTIVATE-YIELDED-TASK)
$$\frac{}{\mathsf{B}\langle w, O, B, T \uplus \{\, t_\Box[\mathsf{yield}] \,\}, M, \epsilon, \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T, M, t_\Box[\mathsf{null}], \epsilon\rangle}$$

(SELECT-CALL-MSG)
$$\frac{\neg \mathit{isReturn}(m)}{\mathsf{B}\langle w, O, B, T, M \uplus \{\, m \,\}, \epsilon, \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T, M, \epsilon, m\rangle}$$

(SELECT-RTRN-MSG)
$$\frac{\mathit{isReturn}(m)}{\mathsf{B}\langle w, O, B, T, M \uplus \{\, m \,\}, t_\epsilon, \epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B, T, M, t_\epsilon, m\rangle}$$

(MSG-SEND)
$$\frac{\mathit{target}(m) \neq w.r}{\mathsf{B}\langle w, O, B, T, M, t_\epsilon, m\rangle \xrightarrow{\uparrow m} \mathsf{B}\langle w, O, B, T, M, t_\epsilon, \epsilon\rangle}$$

(MSG-RCV)
$$\frac{\mathit{target}(m) = w.r}{\mathsf{B}\langle w, O, B, T, M, t_\epsilon, m_\epsilon\rangle \xrightarrow{\downarrow m} \mathsf{B}\langle w, O, B, T, M \cup \{\, m \,\}, t_\epsilon, m_\epsilon\rangle}$$

(NESTED-MSG-RCV)
$$\frac{b \xrightarrow{\downarrow m} b'}{\mathsf{B}\langle w, O, B \uplus \{\, b \,\}, T, M, t_\epsilon, m\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B \cup \{\, b' \,\}, T, M, t_\epsilon, \epsilon\rangle}$$

(NESTED-MSG-SEND)
$$\frac{b \xrightarrow{\uparrow m} b'}{\mathsf{B}\langle w, O, B \uplus \{\, b \,\}, T, M, t_\epsilon, m_\epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B \cup \{\, b' \,\}, T, M \cup \{\, m \,\}, t_\epsilon, m_\epsilon\rangle}$$

(NESTED-STEP)
$$\frac{b \xrightarrow{\tau} b'}{\mathsf{B}\langle w, O, B \uplus \{\, b \,\}, T, M, t_\epsilon, m_\epsilon\rangle \xrightarrow{\tau} \mathsf{B}\langle w, O, B \cup \{\, b' \,\}, T, M, t_\epsilon, m_\epsilon\rangle}$$

Fig. 12. Message and task handling rules as well as nested box reduction rules

4.4 Program Execution

A program with a cobox class Main is executed by nesting a cobox instance b_{main} of Main in a special root or environment cobox b_{env} and setting the current message of b_{env} to a call message m_{main}, which invokes the main method of the main object ι'_o of b_{main}:

$$\begin{aligned} b_{env} &\equiv \mathsf{B}\langle \iota_b, \{\, \mathsf{O}\langle \iota_o, \mathsf{Fut}, \epsilon \rangle \,\}, \{\, b_{main} \,\}, \varnothing, \varnothing, \epsilon, m_{main} \rangle \\ b_{main} &\equiv \mathsf{B}\langle \iota_b.\iota'_b, \{\, \mathsf{O}\langle \iota'_o, \mathsf{Main}, \overline{\mathsf{null}} \rangle \,\}, \varnothing, \varnothing, \varnothing, \epsilon, \epsilon \rangle \\ m_{main} &\equiv \mathsf{M}\langle \iota_b.\iota_o, \iota_b.\iota'_b.\iota'_o, \mathsf{main}(\bullet) \rangle \end{aligned}$$

Program execution is performed by τ-transitions on the environment cobox:

$$b_{env} \xrightarrow{\tau} \cdots \xrightarrow{\tau} b'_{env}$$

When the main cobox finishes its execution of the main method, it will answer with a return message, whose result value is stored in the future object ι_o of the environment cobox.

4.5 Properties

JCoBox^{C} guarantees data-race freedom. As tasks can only access fields of objects of the same cobox and only one task in a cobox can be active, data-races can never occur. Furthermore, a task has atomic access to the objects of its cobox until it explicitly suspends itself.

Internal parallelism by nested coboxes is transparent to clients as all messages to nested coboxes are controlled by the parent cobox. This allows introducing parallelism within a cobox without changing its externally visible behavior.

A task can prevent reentrancy while waiting for a result value of a different cobox using get or synchronous method calls, as then the currently active task is not suspended. Consequently, no other task can be activated in the cobox. The only exception are tasks that have to be activated due to self-calls. Note again that reentrancy is not restricted to single objects, but can be handled for groups of objects.

A JCoBox^{C} program exactly behaves like a corresponding sequential Java program if no additional features of JCoBox^{C} are used and the only cobox is the initial Main cobox. This means that this cobox contains all objects that are created. As in sequential Java only synchronous method calls can be used, these calls are translated into asynchronous method calls with an immediate get. All these calls are self-calls, which means that tasks are created and executed in a stack-like way, simulating the call-stack of sequential Java.

5 Conclusions

We presented coboxes as a data-centric concurrency mechanism for object-oriented programs. CoBoxes hierarchically structure the object-heap into concurrently

running multi-object components. CoBoxes may run completely parallel. Access to nested coboxes is controlled by the parent cobox. A cobox can have multiple *boundary* objects allowing the implementation of complex components with multiple entry objects. A cobox can have multiple tasks, which are scheduled cooperatively and only interleave at explicit release points. This guarantees that each task has exclusive access to the state of its cobox until it explicitly suspends itself. CoBoxes communicate via asynchronous method calls with futures, which allows modeling of synchronous method invocations as a special case. Our concurrency concept generalizes the active object model to hierarchically structured, parallel running groups of objects. It guarantees the absence of data-races and simplifies maintaining invariants ranging over groups of objects. We show how to use coboxes to write concurrent object-oriented programs and present a formal semantics for a Java-like core language with coboxes.

Current Limitations. The presented work is the first step towards generalizing the active object model to dynamically instantiable components supporting multiple internal and multiple boundary objects as well as component nesting. In the following we discuss current shortcomings and limitations.

In the given semantics, scheduling of incoming messages is nondeterministic. In practice, one could implement a default FIFO behavior, or allow the programmer to specify the scheduling algorithm [7, 9, 10]. Join patterns [17, 4, 18] could also be introduced as a scheduling and synchronization mechanism for incoming messages. Regarding coordination of tasks, in the presented model, tasks are activated nondeterministically when they are ready. A more fine-grained control is desirable, especially when waiting for conditions to become true. Concepts that can be used to achieve this are for example: guards [26], events [19], and condition variables [27].

A limitation of our current model is that objects and nested coboxes can only be created in the current cobox. We adopted this restriction to focus on the essential cobox concepts. More flexibility would be desirable, in particular the possibility to create objects or nested coboxes in other coboxes. One way to do this is to extend the `new` expression by an argument where the new object should be created.

Future Work. One of our future goals is to use the cobox model for distributed object-oriented programming such that each remote site corresponds to a cobox. In such scenarios, but also larger local systems, it is desirable to distinguish between value objects and objects passed by reference. Value objects are passed by creating a deep copy in the receiving cobox. This way the number of non-local messages can be drastically reduced. From a conceptual point of view, objects would be distinguished whether they represent data or not, like it is done in the Java RMI mechanism and other approaches [7, 3].

Acknowledgments. We thank Ina Schaefer and the anonymous reviewers for their helpful comments.

References

[1] Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge, MA, USA (1986)

[2] Aldrich, J., Chambers, C.: Ownership domains: Separating aliasing policy from mechanism. In Odersky, M., ed.: ECOOP 2004. Volume 3086 of LNCS., Springer (2004) 1–25

[3] Bacon, D.F., Strom, R.E., Tarafdar, A.: Guava: a dialect of java without data races. In: OOPSLA 2000, ACM Press (2000) 382–400

[4] Benton, N., Cardelli, L., Fournet, C.: Modern concurrency abstractions for C#. In Magnusson, B., ed.: ECOOP 2002. Volume 2374 of LNCS., Springer (2002) 415–440

[5] Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: Preventing data races and deadlocks. In: OOPSLA 2002, ACM Press (November 2002) 211–230

[6] Boyapati, C., Liskov, B., Shrira, L.: Ownership types for object encapsulation. In: POPL '03, ACM Press (2003) 213–223

[7] Caromel, D.: Towards a method of object-oriented concurrent programming. Comm. ACM **36**(9) (1993) 90–102

[8] Caromel, D., Henrio, L., Serpette, B.P.: Asynchronous and deterministic objects. In: POPL '04, ACM Press (2004) 123–134

[9] Caromel, D., Mateu, L., Éric Tanter: Sequential object monitors. In Odersky, M., ed.: ECOOP 2004. Volume 3086 of LNCS., Springer (2004) 316–340

[10] Caromel, D., Mateu, L., Pothier, G., Éric Tanter: Parallel object monitors. Concurrency and Computation: Practice and Experience (2007) To appear.

[11] Clarke, D., Potter, J., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA '98, ACM Press (1998) 48–64

[12] de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: ESOP 2007. Volume 4421 of LNCS. (March 2007) 316–330

[13] Dietl, W., Müller, P.: Universes: Lightweight ownership for JML. Journal of Object Technology **4**(8) (2005) 5–32

[14] Felleisen, M., Hieb, R.: The revised report on the syntactic theories of sequential control and state. Theorectical Computer Science **103**(2) (1992) 235–271

[15] Flanagan, C., Freund, S.N., Lifshin, M.: Type inference for atomicity. In: TLDI '05, ACM Press (2005) 47–58

[16] Flatt, M., Krishnamurthi, S., Felleisen, M.: A programmer's reduction semantics for classes and mixins. Formal Syntax and Semantics of Java **1523** (1999) 241–269

[17] Fournet, C., Gonthier, G.: The reflexive CHAM and the join-calculus. In: POPL '96, ACM Press (1996) 372–385

[18] Haller, P., Cutsem, T.V.: Implementing joins using extensible pattern matching. Technical Report LAMP-REPORT-2007-004, EPFL (August 2007)

[19] Hansen, P.B.: Structured multiprogramming. Comm. ACM **15**(7) (1972) 574–578

[20] Hansen, P.B.: 7.2 Class Concept. In: Operation System Principles. Prentice Hall (1973) 226–232

[21] Harris, T., Fraser, K.: Language support for lightweight transactions. In Crocker, R., Jr., G.L.S., eds.: OOPSLA 2003, ACM Press (2003) 388–402

[22] Haustein, M., Löhr, K.P.: Jac: declarative Java concurrency. Concurrency and Computation: Practice and Experience **18**(5) (2006) 519–546

[23] Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proc. Int. Symp. on Computer Architecture. (1993)

[24] Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory. In Tarr, P.L., Cook, W.R., eds.: OOPSLA 2006, ACM Press (2006) 253–262

[25] Hindman, B., Grossman, D.: Atomicity via source-to-source translation. In: MSPC '06, ACM Press (2006) 82–91

[26] Hoare, C.A.R.: Towards a theory of parallel programming. In: Operating System Techniques, Academic Press (1972) 61–71

[27] Hoare, C.A.R.: Monitors: An operating system structuring concept. Comm. ACM **17**(10) (1974) 549–577

[28] Hogg, J., Lea, D., Wills, A., de Champeaux, D., Holt, R.: The Geneva convention on the treatment of object aliasing. SIGPLAN OOPS Messenger **3**(2) (1992) 11–16

[29] Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. TOPLAS **23**(3) (2001) 396–450

[30] Jacobs, B., Piessens, F., Leino, K.R.M., Schulte, W.: Safe concurrency for aggregate objects with invariants. In Aichernig, B.K., Beckert, B., eds.: SEFM, IEEE Computer Society (2005) 137–147

[31] Jacobs, B., Smans, J., Piessens, F., Schulte, W.: A statically verifiable programming model for concurrent object-oriented programs. In Liu, Z., He, J., eds.: ICFEM. Volume 4260 of LNCS., Springer (2006) 420–439

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

[33] Lieberman, H.: Concurrent object-oriented programming in Act 1. In Yonezawa, A., Tokoro, M., eds.: Object-Oriented Concurrent Programming, MIT Press (1987) 9–36

[34] Lu, Y., Potter, J.: On ownership and accessibility. In Thomas, D., ed.: ECOOP 2006. Volume 4067 of LNCS., Springer (2006) 99–123

[35] Lu, Y., Potter, J., Xue, J.: Validity invariants and effects. In Ernst, E., ed.: ECOOP 2007. Volume 4609 of LNCS., Springer (2007) 202–226

[36] Müller, P., Poetzsch-Heffter, A.: A type system for controlling representation exposure in Java. In Drossopoulou, S., Eisenbach, S., Jacobs, B., Leavens, G.T., Müller, P., Poetzsch-Heffter, A., eds.: Formal Techniques for Java Programs, Technical Report 269–5, Fernuniversität Hagen (2000)

[37] Poetzsch-Heffter, A., Schäfer, J.: A representation-independent behavioral semantics for object-oriented components. In Bonsangue, M.M., Johnsen, E.B., eds.: FMOODS 2007. Volume 4468 of LNCS., Springer (2007) 157–173

[38] Schäfer, J., Poetzsch-Heffter, A.: A parameterized type system for simple loose ownership domains. Journal of Object Technology **5**(6) (2007) 71–100

[39] Shavit, N., Touitou, D.: Software transactional memory. In: PODC. (1995) 204–213

[40] Smaragdakis, Y., Kay, A., Behrends, R., Young, M.: Transactions with isolation and cooperation. In Gabriel, R.P., Bacon, D.F., Lopes, C.V., Steele Jr., G.L., eds.: OOPSLA 2007, ACM Press (2007) 191–210

[41] Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. In: POPL '06, ACM Press (2006) 334–345