




Active Objects with Deterministic Behaviour

Ludovic Henrio¹ , Einar Broch Johnsen² , and Violet Ka I Pun^{2,3} 

¹ Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, France

`ludovic.henrio@ens-lyon.fr`

² University of Oslo, Norway

`{einarj,violet}@ifi.uio.no`

³ Western Norway University of Applied Sciences, Norway

`Violet.Ka.I.Pun@hvl.no`

Abstract. Active objects extend the Actor paradigm with structured communication using method calls and futures. Active objects are, like actors, known to be data race free. Both are inherently concurrent, as they share a fundamental decoupling of communication and synchronisation. Both encapsulate their state, restricting access to one process at a time. Clearly, this rules out low-level races between two processes accessing a shared variable. However, is that sufficient to guarantee deterministic results from the execution of an active object program?

In this paper we are interested in so-called high-level races caused by the fact that the arrival order of messages between active objects can be non-deterministic, resulting in non-deterministic overall behaviour. We study this problem in the setting of a core calculus and identify restrictions on active object programs which are sufficient to guarantee deterministic behaviour for active object programs. We formalise these restrictions as a simple extension to the type system of the calculus and prove that well-typed programs exhibit deterministic behaviour.

1 Introduction

Concurrent programs are characterised by multiple threads executing over a program’s state space, possibly in parallel on multicore or distributed hardware. Concurrency introduces non-determinism in the programs, which makes it hard to reason about program behaviour and easy to inadvertently introduce errors. Two major causes for errors in concurrent programs are *deadlocks* and *races*. One has to choose between making programs *more synchronous*, which makes them exhibit less behaviour but also makes them more deadlock-prone, and making program *more asynchronous*, which enables more behaviour and makes them less deadlock-prone. However, allowing more behaviour also allows more races to occur between the program threads.

Active object languages [1], which extend the Actor [2, 3] model of concurrency with asynchronous method calls and synchronisation using futures, naturally lend themselves to an asynchronous program style because they decouple communication from synchronisation. Asynchronous method calls can be dispatched without any transfer of control between the active objects. Although

asynchrony generally leads to non-determinism, languages based on the Actor model are known to be free from data races (e.g., [4]). This is because actors (and active objects) encapsulate internal state and restrict local state access to one method at a time, which eliminate such low-level races. However, these systems are prone to *high-level communication races* which result in a non-deterministic order of execution for methods on an actor in the system. These races may be triggered by asynchronous method calls (e.g., they are the only races in ASP [5]), by the synchronisation on the associated futures (e.g., [6, 7]) to retrieve the return values from these method calls, and by cooperative scheduling inside the active objects (e.g., [8]). The occurrence of high-level races gives rise to the following question: *under which conditions are active object programs guaranteed to be deterministic?* That is, the programs always produce the same output given a particular input.

This paper studies the problem of active objects with guaranteed deterministic behaviour. Deterministic behaviour for a concurrent program boils down to confluence properties between execution steps. We formalise the execution of active objects systems in a core calculus to study their confluence properties. We combine certain characteristics of the underlying communication network and the local scheduling policy of each active object with restrictions on the program's topology, and show that these together suffice to prove confluence. We identify characteristics that can ensure determinacy, and show how to restrict existing languages to make them partially deterministic. We further show that a simple typing discipline suffices to statically enforce this topology and relate our findings to existing calculi and languages to shed light on how to approach the problem of designing a deterministic active object system in different languages.

The main contributions of the paper can be summarised as follows: We extend previous work on deterministic active object systems, which enforce a tree-shaped object graph, to handle explicit futures and cooperative scheduling, and show that a simple type system is sufficient to guarantee deterministic behaviour even when futures can be shared among objects in the tree-shaped topology.

Paper overview. Section 2 motivates the problem addressed in this paper through an example. Section 3 introduces the active object calculus in which we study the problem, including its operational semantics and basic type system. Section 4 defines and proves confluence properties for our calculus. Section 5 addresses the problem of statically guaranteeing a tree structure in the program topology. Section 6 discusses related work, and in particular to what extent existing active object calculi and languages can guarantee deterministic behaviour. Section 7 concludes the paper.

2 Motivation and Example

An actor is a single-threaded unit of distribution that communicates with other actors by asynchronous message sending. The absence of multi-threading inside an actor and the fact that data is handled by a single actor prevents data races.

However, race conditions can appear when two actors send messages to the same receiver, or when an actor chooses the next message to be processed. Thus, actors are a programming abstraction that limits non-determinism, but does not prevent it. Different adaptations of the actor principles entail different sources of non-deterministic behaviour for programs. To motivate our work on deterministic behaviour for active objects, which are actors synchronising on futures, we review below two classical parallel programming patterns implemented using active objects and discuss the races they exhibit.

```

1 Worker {
2   Array work(int i) { .... } // omitted
3 }
4 { // main
5   Worker w1 = new Worker(); // creation of active objects
6   Worker w2 = new Worker();
7   Fut<Array> f1 = w1!work(1); // asynchronous invocation
8   Fut<Array> f2 = w2!work(2);
9   Array r1 = get f1; // synchronisation
10  Array r2 = get f2; // synchronisation
11  average = (sum(r1) + sum(r2)) / (length(r1) + length(r2))
12 }

```

Fig. 1. Implementation with a master-worker pattern.

```

1 Map {
2   int work(int i, Reducer red) {
3     .... // computation omitted
4     Fut<int> c = red!reduce(computedArray);
5     return 0
6   }
7 }
8 Reducer {
9   int expectedResults; // number of expected results
10  OutputObject out; // result is sent to out
11  int partialNb, partialAvg;
12  int NbWorks; // number of received results
13
14  int reduce (Array oneResult) { // reduce computing partial average
15    int newPartialNb = partialNb+length(oneResult);
16    partialAvg = (partialAvg*partialNb+sum(oneResult))/newPartialNb;
17    partialNb = newPartialNb;
18    NbWorks = NbWorks + 1;
19    if (NbWorks == expectedResults) { out!send(partialAvg) };
20    return partialAvg
21  }
22 }
23 { // main. We suppose out is an active object expecting the result
24   Reducer red = new Reducer(2,out,0,0,0); //reducer creation with
25     initial values for fields
26   Worker m1 = new Map();
27   Worker m2 = new Map();
28   Fut<int> f1 = m1!work(1,red); // asynchronous invocation
29   Fut<int> f2 = m2!work(2,red)

```

Fig. 2. Implementation with a map-reduce pattern.

We consider two implementations of a program which computes the average over a sequence of values. Figure 1 shows an implementation using a master-worker pattern based on active objects. Two workers `w1` and `w2` are called asynchronously (Lines 7 and 8) to perform some work task, the main object then synchronises on the returns from the two invocations (Lines 9 and 10 use a `get`-statement to retrieve the return values) before it computes the average in Line 11. The implementation is presented in the core calculus of Section 3 using an additional basic type `Array` with `sum` and `length` operators.

Figure 2 shows an implementation of the same problem using a map-reduce pattern. In this implementation, partial results are reduced as they arrive. The workers send their results to a `Reducer` active object who computes the partial average of the results as they arrive and forwards the average to a receiving active object `out` (we omit its implementation). We see that the asynchronous method calls to the workers (Lines 27 and 28) are not associated with futures in this implementation, but include a reference to the `Reducer` instance so the partial results can be passed on directly. The computed result would be deterministic with a commutative and associative reduction operator—but this is not the case in our example. Observe that if the first partial average is computed over an empty array, a division-by-zero error will be triggered. This bug might only appear in some executions because messages are received in a non-deterministic order, which makes the reducer difficult to debug. In contrast, the master-worker implementation behaves deterministically; if a division-by-zero bug would occur in that implementation, it would occur in every execution.

Map-reduce is a popular pattern that is supported by powerful runtime frameworks like Hadoop. In the sequel, we identify why patterns such as map-reduce are potentially non-deterministic and design a type-system that ensures deterministic behaviour for active objects. This type system can type the master-worker implementation, but not the map-reduce one.

3 An Active Object Language

In this section we propose a core language for active objects. We adopt a Java-like syntax that is similar to ABS [8].

Notations. \bar{T} denotes a list of elements T , unless stated otherwise this list is ordered. In the syntax x, y, u range over variable names, m method names, α, β active object identifiers, f future identifiers, and `Act` class names. The set of binary operators on values is represented by an abstract operator \oplus , it replaces all the classical operations on integer and booleans. Mappings are denoted $[\bar{x} \mapsto \bar{a}]$ which builds a map from the two lists \bar{x} and \bar{a} of identical length, $m[x \mapsto a]$ updates a map, associating the value a to the entry x , and $+$ merges two maps (taking values in the rightmost one in case of conflict). $\bar{q}\#q$ (resp. $q\#\bar{q}$) is the FIFO enqueue (resp. dequeue) operation.

$P ::= \overline{\text{Act}}\{\overline{T\ x\ M}\} \{\overline{T\ x\ s}\}$	program
$M ::= T\ m(\overline{T\ x}) \{\overline{T\ x\ s}\}$	method
$s ::= \text{skip} \mid x = z \mid \text{if } v \{s\} \text{ else } \{s\} \mid s ; s$ $\quad \mid \text{return } v \mid \text{await } e$	statements
$z ::= e \mid v!m(\overline{v}) \mid \text{new Act}(\overline{v}) \mid \text{get } v$	rhs of assignments
$e ::= v \mid v \oplus v$	expressions
$v ::= x \mid \text{null} \mid \text{integer-and-boolean-values}$	atoms
$B ::= \text{Int} \mid \text{Bool} \mid \text{Act}$	basic type
$T ::= B \mid \text{Fut}(B)$	type

Fig. 3. Static syntax of the core language.

3.1 Syntax and Semantics

We design a simple active object model with one thread per object and where all objects are active (uniform active object model). Interested readers are referred to [1] for a complete description of the different request scheduling strategies in active object languages.

Figure 3 shows the syntax of our language. A program P is made of a set of classes, each having a set of fields and a set of methods, plus a main method. A method M has a name m , a set of parameters, and a body, made of a set of local variables and a statement. Types and terms are standard of active object languages, for instance **new** creates an active object, **get** accesses a future, and $v!m(\overline{v})$ performs a method invocation on an active object and thus systematically creates a future. The type constructor for future is $\text{Fut}(T)$ like ABS or any explicit future construct. Sequence is denoted as $;$ and is associative with a neutral element **skip**. Consequently, each statement that is not **skip** can be rewritten as $s; s'$ with s neither **skip** nor a sequence. \oplus denotes the (standard) operations on integers and booleans. Finally, including an *await* enables cooperative scheduling: it interrupts a thread until a condition is validated. Several design choices had to be made in our language we discuss them briefly below:

- For simplicity, we suppose that local variables and fields have disjoint names.
- We specify a service of requests in FIFO order with a causal ordering of request transmission, like in ASP [5], Rebeca [9] or Encore [10]. Also, FIFO communication is supported by many actor and active object implementations, and it reduces the possible interleaving of messages.
- Adding subtyping is outside the scope of our study.
- With more complex active object models, it is sometimes necessary to have a syntactic distinction between synchronous and asynchronous invocations. For instance, ABS uses $!$ to identify asynchronous method invocations that create futures. Our core language adopts ABS syntax here but does not have synchronous invocation.

$cn ::= \overline{\alpha(a, p, \bar{q})} \overline{f(w)}$	configuration
$p ::= \emptyset \mid q$	current request service
$q ::= \{\ell \mid s\}_f$	request
$w ::= x \mid \alpha \mid f \mid \mathbf{null} \mid \textit{integer-values}$	runtime values
$\ell, a ::= [\bar{x} \mapsto \bar{w}]$	local store and object fields
$e ::= w \mid v \oplus v$	expressions can now have runtime values
$s ::= \mathbf{skip} \mid x = z \mid \mathbf{if} \ e \ \{s\} \ \mathbf{else} \ \{s\}$	statements
$\quad \mid s ; s \mid \mathbf{return} \ e \mid \mathbf{await} \ e$	
$z ::= e \mid v!m(\bar{v}) \mid \mathbf{new} \ \mathbf{Act}(\bar{v}) \mid \mathbf{get} \ v$	expressions with side effects

Fig. 4. Runtime Syntax of the core language .

The operational semantics of our language is shown in Figure 5; it expresses a small-step reduction semantics as a transition between runtime configurations. The syntax of configurations and runtime terms is defined in Figure 4, statements are the same as in the static syntax except that they can contain runtime values like reference to an object or a future (inside assignment or **get** statement). A configuration is an *unordered* set of active objects and futures. Each active object is of the form $\alpha(a, p, \bar{q})$ where α is the active object identifier, a stores the value of object fields, p is the task currently be executed, and \bar{q} a list of pending tasks. The configuration also contains futures that are resolved by a value w (when a future is not yet resolved, it is not in the configuration). A task q is made of a set of local variables ℓ and a statement s to be executed, additionally each task is supposed to fulfil a future f . The currently performed task p is either empty \emptyset or a single task q .

The semantics uses an auxiliary operator – bind – that creates a context for evaluating a method invocation. If the object α is of type **Act**, and m is defined in **Act**, i.e., $\mathbf{Act}\{..T \ m(\bar{T} \ \bar{x}) \ \{\bar{T} \ \bar{y} \ s\}..\}$ is one class of the program P , then⁴: $\mathbf{bind}(\alpha, (f, m, \bar{w})) \triangleq \{[\mathbf{this} \mapsto \alpha, \bar{x} \mapsto \bar{w}] \mid s\}$.

To deal with assignment, we use a dedicated operator for updating the current fields or local variables:

$$(a + \ell)[x \mapsto w] = a' + \ell' \iff \begin{array}{l} a' = a[x \mapsto w] \text{ and } \ell' = \ell, \text{ if } x \in \text{dom}(a), \\ a' = a \text{ and } \ell' = \ell[x \mapsto w], \text{ otherwise} \end{array}$$

We also define a predicate checking whether a thread is enabled, i.e., can progress. A thread is disabled if it starts with an **await** statement on a condition that is **false**.

$$\begin{aligned} \text{disabled}(q) &\iff \exists \ell e s f. (q = \{\ell \mid \mathbf{await} \ e ; s\}_f \wedge \llbracket e \rrbracket_{a+\ell} = \mathbf{false}) \\ \text{enabled}(q) &\iff \neg \text{disabled}(q) \end{aligned}$$

⁴ It is not necessary to initialise the local variables in the local environment because of the way store update is defined.

$$\begin{array}{c}
\frac{w \text{ is not a variable}}{\llbracket w \rrbracket_\ell = w} \qquad \frac{x \in \text{dom}(\ell)}{\llbracket x \rrbracket_\ell = \ell(x)} \qquad \frac{\llbracket v \rrbracket_\ell = k \quad \llbracket v' \rrbracket_\ell = k'}{\llbracket v \oplus v' \rrbracket_\ell = k \oplus k'} \\
\\
\text{CONTEXT} \qquad \frac{cn \rightarrow cn'}{cn \quad cn'' \rightarrow cn' \quad cn''} \qquad \text{ASSIGN} \qquad \frac{\llbracket e \rrbracket_{a+\ell} = w \quad (a+\ell)[x \mapsto w] = a' + \ell'}{\alpha(a, \{\ell \mid x = e ; s\}_f, \bar{q}) \rightarrow \alpha(a', \{\ell' \mid s\}_f, \bar{q}')} \\
\\
\text{NEW} \qquad \frac{\llbracket \bar{v} \rrbracket_{a+\ell} = \bar{w} \quad \beta \text{ fresh} \quad \bar{y} = \text{fields}(\text{Act})}{\alpha(a, \{\ell \mid x = \text{new Act}(\bar{v}) ; s\}_f, \bar{q}') \rightarrow \alpha(a, \{\ell \mid x = \beta ; s\}_f, \bar{q}') \quad \beta(\llbracket \bar{y} \rrbracket \mapsto \bar{w}), \emptyset, \emptyset)} \\
\\
\text{INVK} \qquad \frac{\llbracket v \rrbracket_{a+\ell} = \beta \quad \llbracket \bar{v} \rrbracket_{a+\ell} = \bar{w} \quad \beta \neq \alpha \quad f' \text{ fresh} \quad \text{bind}(\beta, m, \bar{w}) = \{\ell' \mid s\}}{\alpha(a, \{\ell \mid x = v!m(\bar{v}) ; s\}_f, \bar{q}') \beta(a', p, \bar{q}_\beta) \rightarrow \alpha(a, \{\ell \mid x = f' ; s\}_f, \bar{q}') \beta(a', p, \bar{q}_\beta \# \{\ell' \mid s\}_{f'})} \\
\\
\text{INVK-SELF} \qquad \frac{\llbracket v \rrbracket_{a+\ell} = \alpha \quad \llbracket \bar{v} \rrbracket_{a+\ell} = \bar{w} \quad f' \text{ fresh} \quad \text{bind}(\alpha, m, \bar{w}) = \{\ell' \mid s\}}{\alpha(a, \{\ell \mid x = v!m(\bar{v}) ; s\}_f, \bar{q}') \rightarrow \alpha(a, \{\ell \mid x = f' ; s\}_f, \bar{q}' \# \{\ell' \mid s\}_{f'})} \\
\\
\text{RETURN} \qquad \frac{\llbracket v \rrbracket_{a+\ell} = w}{\alpha(a, \{\ell \mid \text{return } v ; s\}_f, \bar{q}) \rightarrow \alpha(a, \emptyset, \bar{q}) \quad f(w)} \qquad \text{GET} \qquad \frac{\llbracket v \rrbracket_{a+\ell} = f'}{\alpha(a, \{\ell \mid y = \text{get } v ; s\}_f, \bar{q}') \quad f'(w) \rightarrow \alpha(a, \{\ell \mid y = w ; s\}_f, \bar{q}') \quad f'(w)} \\
\\
\text{SERVE} \qquad \frac{\forall q' \in \bar{q}_1. \text{disabled}(q') \quad \text{enabled}(q)}{\alpha(a, \emptyset, \bar{q}_1 \# q \# \bar{q}_2) \rightarrow \alpha(a, q, \bar{q}_1 \# \bar{q}_2)} \qquad \text{AWAIT} \qquad \frac{\text{disabled}(q)}{\alpha(a, q, q') \rightarrow \alpha(a, \emptyset, q \# q')}
\end{array}$$

Fig. 5. Semantics of the core language (rules IF-TRUE and IF-FALSE for reducing if omitted).

The semantics of a program features the classical elements of active object programming [8, 11], the stateful aspects of the language are expressed as accesses to either local variables (ℓ) or object fields (a). The first three rules of the semantics define an evaluation operator $\llbracket e \rrbracket_{a+\ell}$ that evaluates an expression. Note that $\llbracket e \rrbracket_{a+\ell} = w$ implies that w can only be an object or future name, **null**, or an integer or boolean value. The semantics in Figure 5 contains the following rules that are standard of active object languages.

ASSIGN deals with assignment to either local variables or object fields.

NEW creates a new active object at a fresh location β .

INVK (method invocation) creates a task and enqueues it in the target active object, and a future identifier f' , a reference to the future can then be used by the invoker α .

INVK-SELF deals with the particular case where the target is the invoking object.

RETURN evaluates a `return` statement and resolves the corresponding futures, finishing a task so that a new task can be performed.

SERVE occurs when there is no current task, it picks the first one that can be activated from the list of pending tasks and starts its execution. This ensures a strict single-threaded execution of each request one after the other.

GET fetches the value associated to a future.

AWAIT suspends a task, waiting for the object to be in a given state before continuing the task. Note that the awaited condition only depends on the internal state of the active object. This scheduling feature is called *cooperative scheduling* because several threads can be executing at the same time but only one progresses and the context switch between a thread and another is triggered by the program itself.

The *initial configuration* for running a program $\text{Act}\{\overline{T\ x\ M}\}\{\overline{T\ x\ s}\}$ consists of a single object performing a single task defined by the main method, the corresponding future f is useless as no other object will fetch the result (it can be any future identifier): $\alpha(\emptyset, \{\emptyset|s\}_f, \emptyset)$. We use \rightarrow^* for the reflexive transitive closure of \rightarrow .

3.2 Type System

We define a simple type system for our language (the syntax of types is defined in Figure 3). The type system is standard for a language with active objects and futures. The type checking rules are presented in Figure 6. Classically, `Act` ranges over class names and types. Γ is used for typing environments. The typing rules have the form $\Gamma \vdash_T s$ for statements where T is the return type of the current method, $\Gamma \vdash e$ for expressions, $\Gamma \vdash M$ for methods, and $\Gamma \vdash P$ for programs. The static type checking is defined in the first twelve rules of the figure. We describe below the most interesting rules.

T-GET removes one future construct.

T-INVK creates a future type. This rule adds a future construct for the result of asynchronous method invocation.

T-PROGRAM Note that the main body return type can be chosen arbitrarily: there is no constraint on the typing of a `return` statement in the main block. The *initial typing environment* Γ , which types the program, associates to each class name a mapping from method names to method signatures. If m is a method of class `Act` defined as follow $T''\ m\ (\overline{T\ x})\{\overline{T'\ x'}\ s\}$, we will have $\Gamma(\text{Act})(m) = \overline{T} \rightarrow T''$.

The type system is extended for typing configurations, this is expressed in the last four rules of Figure 6. A typing environment gives the type of each active object and future. Each element of the configuration is checked individually in a very standard manner. The only complex case happens when checking processes, i.e., statements of requests in the queue or being processed, the complexity only comes from the necessity to build the typing environment for the body of the methods.

Properties of the type system Our type system verifies *subject reduction*.

$$\begin{array}{c}
\text{(T-VAR)} \\
\frac{}{\Gamma \vdash x : \Gamma(x)} \\
\\
\text{(T-NULL)} \\
\frac{}{\Gamma \vdash \mathbf{null} : \mathbf{Act}} \\
\\
\text{(T-ASSIGN)} \\
\frac{\Gamma(x) = T' \quad \Gamma \vdash z : T'}{\Gamma \vdash_T x = z} \\
\\
\text{(T-NEW)} \\
\frac{\Gamma \vdash \bar{v} : \bar{T} \quad \mathit{fields}(\mathbf{Act}) = \bar{T} \bar{x}}{\Gamma \vdash \mathbf{new Act}(\bar{v}) : \mathbf{Act}} \\
\\
\text{(T-EXPRESSION)} \\
\frac{\oplus : T \times T' \rightarrow T'' \quad \Gamma \vdash v : T \quad \Gamma \vdash v' : T'}{\Gamma \vdash v \oplus v' : T''} \\
\\
\text{(T-GET)} \\
\frac{\Gamma \vdash v : \mathbf{Fut}\langle B \rangle}{\Gamma \vdash \mathbf{get } v : B} \\
\\
\text{(T-RETURN)} \\
\frac{\Gamma \vdash e : T}{\Gamma \vdash_T \mathbf{return } e} \\
\\
\text{(T-INVK)} \\
\frac{\Gamma(\mathbf{Act})(\mathbf{m}) = \bar{T} \rightarrow T' \quad \Gamma \vdash v : \mathbf{Act} \quad \Gamma \vdash \bar{v} : \bar{T}}{\Gamma \vdash v \mathbf{!m}(\bar{v}) : \mathbf{Fut}\langle T' \rangle} \\
\\
\text{(T-SEQ)} \\
\frac{\Gamma \vdash_T s \quad \Gamma \vdash_T s'}{\Gamma \vdash_T s ; s'} \\
\\
\text{(T-SKIP)} \\
\frac{}{\Gamma \vdash_T \mathbf{skip}} \\
\\
\text{(T-PROGRAM)} \\
\frac{\forall \mathbf{Act}\{\bar{T} x, \bar{M}\} \in \overline{\mathbf{Act}\{\bar{T} x, \bar{M}\}}, \forall M \in \bar{M}. \Gamma[\bar{x} \mapsto \bar{T}][\mathbf{this} \mapsto \mathbf{Act}] \vdash M}{\Gamma \vdash \mathbf{Act}\{\bar{T} x, \bar{M}\} \{T' x' s\}} \\
\\
\text{(T-METHOD)} \\
\frac{\Gamma[\bar{x} \mapsto \bar{T}][\bar{x}' \mapsto \bar{T}'] \vdash_T s}{\Gamma \vdash T'' \mathbf{m}(\bar{T} x) \{T' x' s\}} \\
\\
\text{(T-CONFIG)} \\
\frac{\forall \alpha(a, p, \bar{q}) \in \overline{\alpha(a, p, \bar{q})}. \Gamma \vdash \alpha(a, p, \bar{q}) \quad \forall f(w) \in \overline{f(w)}. \Gamma \vdash w : \Gamma(f)}{\Gamma \vdash \alpha(a, p, \bar{q}) \overline{f(w)}} \\
\\
\text{(T-OBJ)} \\
\frac{\Gamma(\alpha) = \mathbf{Act} \quad \mathit{fields}(\mathbf{Act}) = \bar{T} \bar{x} \quad \Gamma' = \Gamma[\mathbf{this} \mapsto \mathbf{Act}][\bar{x} \mapsto \bar{T}] \quad \forall x \in \mathit{dom}(a). \Gamma' \vdash a(x) : \Gamma'(x) \quad \forall \{[\bar{y} \mapsto \bar{w}] | s\}_f \in p \cup \bar{q}. \exists \bar{T}'. (\Gamma' \vdash \bar{w} : \bar{T}' \wedge \Gamma'[\bar{y} \mapsto \bar{T}'] \vdash_{\Gamma(f)} s)}{\Gamma \vdash \alpha(a, p, \bar{q})} \\
\\
\text{(T-OBJREF)} \\
\frac{}{\Gamma \vdash \alpha : \Gamma(\alpha)} \\
\\
\text{(T-FUTREF)} \\
\frac{}{\Gamma \vdash f : \Gamma(f)}
\end{array}$$

Fig. 6. Type system (operator \oplus has a predefined signature, rule for if omitted).

Property 1 (Subject reduction). If $\Gamma \vdash cn$ and $cn \rightarrow cn'$ then $\Gamma' \vdash cn'$ with $\Gamma \subseteq \Gamma'$.

Proof (Sketch). The proof is by straightforward induction over the application of transition rules. For example the correct typing of the future value is ensured by the fact that the **return** statement is well-typed in the initial configuration (i.e., it has the return type of the method). This also ensures that the **get** statement is well-typed (accordingly to the future type and the return type of the method), and thus the GET reduction rule obtains the return type without the future construct. Then, it is easy and classical to prove that every bind succeeds (because the target method exists). The proof is standard and thus omitted from the paper. \square

4 Confluence Properties

In the following, we will state under which conditions a program written in our language can behave deterministically. We first identify the configurations modulo renaming of futures and active object identifiers. For this we let σ range over renaming of futures and active object identifiers (mapping names to names), and use $cn\sigma$ to apply the renaming σ to the configuration cn .

Definition 1 (Equivalence). *The configurations cn_1 and cn_2 are equivalent, denoted as $cn_1 \equiv cn_2$, if and only if $\exists \sigma. cn_1 = cn_2\sigma$.*

Note that it is trivial to prove that two equivalent configuration can do the same reduction step (according to the SOS rules) and reach equivalent configurations. Our properties will rely on the topology of active objects. For this we first define the set of active objects referenced by a term of the language as follows.

Definition 2 (References). *We state that active object β is referenced by active object α in configuration cn , written $\beta \in \mathbf{refs}_{cn}(\alpha)$, if inside configuration cn , the content of the active object α holds a reference to active object β . More precisely*

$$\begin{aligned} \mathbf{refs}(\ell) &= \{\beta \mid \beta \in \mathbf{range}(\ell)\} \\ \mathbf{refs}(\{\ell \mid s\}) &= \{\beta \mid \beta \in \mathbf{range}(\ell)\} \\ \mathbf{refs}(\alpha(a, q, \overline{q'})) &= \mathbf{refs}(a) \cup \mathbf{refs}(q) \cup \bigcup_{q' \in \overline{q'}} \mathbf{refs}(q') \\ \mathbf{refs}_{cn}(\alpha) &= \mathbf{refs}(\alpha(a, q, \overline{q'})) \text{ if } \alpha(a, q, \overline{q'}) \in cn \end{aligned}$$

For example, consider the configuration

$$\begin{aligned} cn_1 &= \alpha([x \mapsto \beta], \{[y \mapsto \beta] \mid y := \mathbf{new Act}(\overline{v}); y!m()\}, \emptyset) \quad \gamma(\emptyset, \emptyset, \emptyset) \\ &\quad \beta([z \mapsto f], \{[w \mapsto 1] \mid y := w + 1\}, \{[g \mapsto \gamma] \mid h = g!m()\}) \quad f(3) \end{aligned}$$

We have $\mathbf{refs}_{cn_1}(\alpha) = \{\beta\}$, $\mathbf{refs}_{cn_1}(\beta) = \{\gamma\}$ and $\mathbf{refs}_{cn_1}(\gamma) = \emptyset$

We can now define when a configuration has a tree structure. To be precise, we should call such a configuration a *forest* as there is no requirement on the unicity of the tree root.

Definition 3 (Tree structure). *We say that a configuration has a tree structure when no two objects reference the same third one.*

$$\mathbf{Tree}(cn) = \forall \alpha \beta \in cn. \mathbf{refs}_{cn}(\alpha) \cap \mathbf{refs}_{cn}(\beta) = \emptyset$$

The configuration cn_1 given as example above verifies $\mathbf{Tree}(cn_1)$ because active object α only references active object β , active object β only references γ , and active object γ references nothing. If the object field x of α was mapped to γ instead of β , we would have two active objects referencing γ and the property $\mathbf{Tree}(cn_1)$ would be **false**.

$$\begin{array}{c}
\text{(T-NEW)} \\
\frac{\text{fields}(\mathbf{Act}) = \bar{T} \bar{x} \quad \Gamma \Vdash \bar{v} : \bar{T} \quad \forall v \in \bar{v}. \Gamma \Vdash v : \mathbf{ActB} \implies v = \mathbf{null}}{\Gamma \Vdash \mathbf{new Act}(\bar{v}) : \mathbf{Act}} \\
\\
\text{(T-INVK)} \\
\frac{\Gamma \Vdash v : \mathbf{Act} \quad \Gamma(\mathbf{Act})(\mathbf{m}) = \bar{T} \rightarrow T' \quad \forall v' \in \bar{v}. \Gamma(\exists \mathbf{ActB}. \Vdash v' : \mathbf{ActB}) \implies v' = \mathbf{null} \quad \nexists \mathbf{ActB}. T' = \mathbf{ActB}}{\Gamma \Vdash v!_{\mathbf{m}}(\bar{v}) : \mathbf{Fut}(T')}
\end{array}$$

Fig. 7. Type system modified for no reference passing (each operator \oplus has a predefined signature, rule for if-statement is omitted). $T \neq \mathbf{Act}$ means T is not an object type.

Now, we can state one crucial property of our language; it is a partial confluence property constrained by the structure of the references between active objects. We first prove a local confluence property. It relies on the fact that the only conflicting reductions of the calculus is the concurrent sending of request to a same target active object, from two different active objects. As a consequence, if each object is referenced by a single object, then there is no conflicting reduction and we have local confluence.

Property 2 (Local Confluence). For any configuration cn such that $\mathbf{Tree}(cn)$, if there exists cn_1 and cn_2 such that $cn \rightarrow cn_1$ and $cn \rightarrow cn_2$, then there exists cn'_1 and cn'_2 such that $cn_1 \rightarrow cn'_1 \wedge cn_2 \rightarrow cn'_2 \wedge cn'_1 \equiv cn'_2$.

Proof (Sketch). The proof of local confluence is classically done by case analysis on each pair of reduction rules that can be applied. We start by eliminating the **CONTEXT** rule that is used to extract a sub-configuration and apply it automatically in the proof, which is detailed in an accompanying technical report [12]. \square

Finally, as a consequence of the previous property, we can state the following partial confluence theorem. When at each point of the execution, the graph of dependencies between active objects forms a tree, the program behaves deterministically.

Theorem 1 (Global Confluence). *Let cn be any configuration such that $\forall cn'. cn \rightarrow^* cn' \implies \mathbf{Tree}(cn')$.*

If there exists cn_1 and cn_2 such that $cn \rightarrow^ cn_1$ and $cn \rightarrow^* cn_2$, then there exists cn'_1 and cn'_2 such that $cn_1 \rightarrow^* cn'_1 \wedge cn_2 \rightarrow^* cn'_2 \wedge cn'_1 \equiv cn'_2$.*

5 Static Tree Structure Guarantee

In this section we define a type system that is sufficient to ensure the tree structure of active objects and show that every well typed program according to the type system defined in this section is confluent.

The type system in Figure 6 is modified by revising rules T-NEW and T-INVK, which handles object creations and method invocations, as shown in Figure 7. The modified type system is denoted as \Vdash . The two revised rules ensure that references to an object cannot be passed upon object creation or method invocation, thus only the creator of an object keeps a reference to it.

Note that `this` is useless in a tree-structure setting because an object cannot call itself and it cannot pass its reference to an external object either. Note that, however, we could add a synchronous call on `this` to the calculus (stacking a method invocation), which would not raise any problem (just extending syntax). Alternatively an asynchronous self call that adds the invocation at the head of the queue like `await` would also be safe and maintain confluence property (but with a strange semantics). To keep the typing rules simple, we use `ActA`, `ActB`, \dots , to represents the types of different objects. Alternatively, we could use subtyping relatively to a generic object type.

To show that a well-typed program in our language is confluent, we first show that the type system \Vdash verifies subject reduction and reduction maintains the tree property.

Property 3 (Subject reduction of \Vdash). If $\Gamma \Vdash cn$ and $cn \rightarrow cn'$ then $\Gamma' \Vdash cn'$, where $\Gamma \subseteq \Gamma'$.

Proof (Sketch). The proof is by classical induction over the application of transition rules, and is detailed in an accompanying technical report [12]. The proof also ensures that any return-type and thus any future is not an object, i.e., its type is not an `Act`. More concretely, we never have $\Gamma(f) = \text{Act}$. \square

Property 4 (Reduction maintains tree property). Consider the type-system of our language modified according to Figure 7 and extended to configurations.

$$(\Gamma \Vdash cn \wedge cn \rightarrow cn' \wedge \text{Tree}(cn)) \implies \text{Tree}(cn')$$

Proof. This is due to the fact that the type system prevents the communication of an object reference to a newly created object or as method parameter, or as method result. In fact we prove by induction a stronger property:

$$\begin{aligned} (\Gamma \Vdash cn \wedge cn \rightarrow cn' \wedge \text{Tree}(cn) \wedge \forall f(w) \in cn. w \neq \alpha) \\ \implies \text{Tree}(cn') \wedge \forall f(w) \in cn'. w \neq \alpha \end{aligned}$$

INVK. Let $cn = \alpha_1(a_1, \{\ell_1 \mid x = v!m(\bar{v}) ; s_1\}_f, \bar{q}_1) \alpha_2(a_2, p, \bar{q}_2)$. We are given that $\text{Tree}(cn)$, i.e., $\text{refs}_{cn}(\alpha_1) \cap \text{refs}_{cn}(\alpha_2) = \emptyset$, and $\Gamma \Vdash cn$, which implies $\Gamma_1 \Vdash_{T_1} v!m(\bar{v})$ for some Γ_1 and T_1 . This further gives us by rule T-INVK that (i) $\Gamma_1 \Vdash_{T_1} v : \text{Act}$, (ii) $\Gamma_1 \Vdash_{T_1} \bar{v} : \bar{T}$, (iii) $\Gamma(\text{Act})(m) = \bar{T} \rightarrow T'$, (iv) $\nexists \text{ActB}. T' = \text{ActB}$, and (v) $\forall v' \in \bar{v}. \Gamma'(\exists \text{ActB}. \Vdash v' : \text{ActB}) \implies v' = \text{null}$.

We are further given by rule INVK that $cn \rightarrow cn'$ and $cn' = \alpha_1(a_1, \{\ell_1 \mid x = f_m ; s_1\}_f, \bar{q}_1) \alpha_2(a_2, p, \bar{q}_2 \# \{\ell_m \mid s_m\}_{f_m})$ where $\llbracket v \rrbracket_{a_1+\ell_1} = \alpha_2$ and $\alpha_2 \neq \alpha_1$, $\llbracket \bar{v} \rrbracket_{a_1+\ell_1} = \bar{w}$, $\text{bind}(\alpha_2, m, \bar{w}) = \{\ell_m \mid s_m\}$, and f_m is fresh. Given (v) above, we have $\text{refs}(\ell_m) = \emptyset$; thus $\forall \gamma. \text{refs}(\gamma) \cap \text{refs}(\ell_m) = \emptyset$. This, together with $\text{Tree}(cn)$, implies $\text{Tree}(cn')$ because ℓ_m is the only new term in cn' that can

contain references to active objects. Also $\forall f(w) \in cn'. w \neq \alpha$ because the set of resolved future is the same in cn and cn' .

RETURN. Let $cn = \alpha(a, \{\ell \mid \text{return } e ; s\}_f, \bar{q})$. We are given that $\text{Tree}(cn)$, and $\Gamma \Vdash cn$. We are further given by rule RETURN that $cn \rightarrow cn'$, where $cn' = \alpha(a, \emptyset, \bar{q}) f(w)$ and $\llbracket e \rrbracket_{a+\ell} = w$. Since $\text{Tree}(cn)$, it is easy to see that $\text{Tree}(cn')$. By Property 3, we have $\Gamma' \Vdash cn'$ where $\Gamma \subseteq \Gamma'$ implying that $\Gamma' \Vdash w : \Gamma'(f)$, where $\Gamma'(f) = T$. From the remark on return-types in the proof of Property 3, it is clear a well-typed future can never be of any type Act , i.e., $\nexists \text{Act}. T = \text{Act}$. Since $f(w)$ is the only future that is changed, $\forall f(w) \in cn'. w \neq \alpha$ holds.

The remaining cases are straightforward. \square

Now, we can prove that the type system \Vdash is sufficient to ensure the tree structure required for confluence.

Property 5 (Tree structure). Consider the type-system of our language modified according to Figure 7. If for a program P , $\Gamma \Vdash P$ then the execution of P verifies the conditions of the global confluence theorem, and P has a deterministic behaviour.

Proof. Consider cn_0 is the initial configuration for the program P , we can prove that $\forall cn. cn_0 \rightarrow^* cn \implies \text{Tree}(cn)$. This is a direct consequence of Property 4 and of the fact that cn_0 forms a tree. By application of Property 2 we obtain global confluence. \square

It is easy to see that in the examples of Section 2, the master-worker example in Figure 1 can be typed with our type system. On the other hand, the transmission of object references (Lines 27 and 28) in the map-reduce example in Figure 2 makes it impossible to type with our type system. This reflects the fact that only the first one is deterministic.

Ensuring the confluence property in a more flexible way would require a more dynamic view of the object dependencies, for example by a more powerful static analysis or a linear type system that would allow the creator to forget a reference and send it to another object. These more dynamic systems are not studied in this article and left for future work.

6 Related Work

We review the closest related work and discuss how different actor calculi could be made partially confluent by following the approach advocated in this paper. Table 1 summarises the features of some of the languages we discuss, with respect to the key points that make our approach feasible in practice. FIFO channels are mandatory to ensure determinacy of communication between two given objects. Futures can be safely added to the language to handle responses to messages in a deterministic manner *provided they can only be accessed in a blocking manner*. In the following, when a language appears to us as a meaningful target for our approach, we explain briefly how our result is applicable. We consider that for the

Language	FIFO channels	Blocking future synchronisation	Cooperative scheduling
ProActive and ASP	YES	YES	NO
Rebeca	YES	NO	NO
AmbientTalk	NO	NO	NO
ABS	NO	YES	Non-deterministic
Encore	YES	YES	Non-deterministic
Akka	YES	Discouraged	Non-deterministic
Lustre with futures	YES	YES	NO

Table 1. Deterministic characteristics for a few actor and active object languages.

other languages, the decisions made in the design of the language are somehow contradictory with the principles of our approach.

ProActive [13] uses active objects to implement remotely accessible, asynchronous objects. The ASP calculus [5] formalises the ProActive Java library. This paper also identifies partial confluence properties for active objects, which can be seen as a follow-up to [5], except that our futures are explicit, where ASP features implicit futures. Compared to the original work, the presented core language is more streamlined, making this contribution easier to adapt to many programming languages.

Applying our approach to ProActive. This paper can be seen both as an extension of [5] and as an adaptation to explicit futures. Additionally we partially address cooperative scheduling via a restricted *await* primitive. We also identify a simple type system that allows us to ensure deterministic behaviour of programs.

Rebeca [9] and its variants mostly consist of actors communicating by asynchronous messages over FIFO queues, which makes model-checking for Rebeca programs less prone to state-explosion than most distributed systems [14]. Ensuring a tree structure of Rebeca actors would then be sufficient to guarantee deterministic behaviour; unfortunately the absence of futures in Rebeca forces callbacks to be used to transmit results of computations, and it is very challenging to maintain a tree-structure in the presence of callbacks.

AmbientTalk [15], based on the E Programming Language [16], implements an actor model with a communicating event-loop. It targets embedded systems and uses asynchronous reaction to future resolution, which prevents deadlocks at the price of more non-determinism, creating a race between the reaction to the future resolution and the rest of the computation in the same actor.

Creol [17] and languages inheriting from it, JCoBox [18], ABS [8] and Encore [10], rely on *cooperative scheduling* allowing the single execution thread of the active object to interrupt the service of one request and start (or recover) another at explicitly defined program points. A main difference between ABS and Encore is that the former is built upon Erlang [19] that does not ensure

FIFO ordering of messages, while the latter is built upon Pony [20] that ensures causal ordering of messages. In addition, Encore supports an advanced capability-based type system [21] which enables race-free data sharing between active objects. Confluence properties for cooperative scheduling in ABS have previously been studied, based on controlling the local scheduler [22, 23].

Applying our approach to languages à la ABS. ABS is a good candidate for our approach because of the numerous formal developments it supports. However, ABS features much less determinism than our core language because communications are unordered, and cooperative scheduling entails unpredictable interleaving between the treatment of different messages. For example, Encore is similar to ABS but already ensures FIFO ordering of messages, it would thus be easier to adapt our work to Encore.

Concerning cooperative scheduling in JCoBox, ABS and Encore, we can state that *await* on a future creates a non-blocking future access and should be proscribed if determinism is expected. Other *await* statements (on the internal state of an active object) can be kept in the language, but the cooperative scheduling policy has to be adapted to make it deterministic.

Futures are becoming increasingly mainstream and are now available through libraries in many languages, including Java, Scala, C++, and Rust. Akka [24, 25] is a scalable library for actors on top of Java and Scala. Communication in Akka is FIFO which allows scheduling to be performed deterministically. Concerning return-values, Akka used to favour asynchronous reaction to future resolution which is not deterministic by nature. In the newest release, Akka 2.6.0, callbacks are the preferred strategy for returning values. By nature, callbacks entail a non-tree structure of object dependencies and create race-conditions between the handling of callbacks and of standard requests.

Lohstroh et al. [26] recently proposed a deterministic actor language. The key ingredient for determinism is the *logical timing* of messages based on a protocol which combines physical and logical timing to ensure determinacy. Unfortunately the resulting language is only deterministic when each message reaching the same actor is tagged with a different time, which may not be easy to ensure. Additionally, to the best of our knowledge, there is no proof of correctness of the used scheduling protocol and its adaptation to the context of the paper. We believe our approach could provide the right abstractions to prove correctness of such scheduling approaches for determinacy, adapting the proof of confluence provided in this paper and relating it to the scheduling protocol could prove the confluence property of [26].

Ownership type systems [27] can enforce a given object topology. Their application to active objects [28], especially inside the Encore language [10, 21], ensures the separation between different memory spaces. Ownership types guarantee that each passive (or data) object is referenced by a single active object. Ownership types are in general adapted to enforce a tree topology, and these works could be extended to active objects so that their dependencies form a tree

(and passive objects are still owned by a single active object). This significant extension of type systems is outside the scope of this paper but would allow more programs in our calculus to be accepted by the type checker and proven deterministic. Other modern type system features, especially linearity and borrowing [29], should also be considered for the same reasons. In particular we envisage the use of linear types and borrowing techniques to extend our results to computations where the tree structure of active objects may change over time.

Outside the actor community, the addition of futures in Lustre has been proposed in 2012 [30]. In this work, the authors provide an asynchronous computation primitive based on futures inside a synchronous language. As futures have good properties with respect to parallelism and determinism, they obtain a language that is equivalent to the synchronous language but with more parallelism. Our approach is very close to futures in Lustre for two reasons: firstly, both set up a programming model that ensure deterministic behaviour by using futures and asynchronous invocations, secondly, the way futures are encoded in Lustre corresponds in fact to an actor-like program where the dependency between actors form a tree and communication is over FIFO channels.

Applying our approach to Lustre with futures. We prove here that, in an asynchronous setting, futures in Lustre still have a deterministic behaviour (the same behaviour as synchronous programs). Additionally, our *await* primitive could be used in Lustre with future to enable cooperative scheduling.

7 Conclusion

This paper has given guidelines on how to implement deterministic active objects and ensure that in any given framework a program behaves deterministically if this is desired. We formalised a basic active object calculus where communication between objects is performed by asynchronous method invocations on FIFO channels, replies by means of futures, and synchronisation by a blocking wait on future access. We added a deterministic cooperative scheduling policy, allowing a thread to be suspended and recovered *depending on the internal state of the object*. These conditions are the necessary prerequisites for our approach to be applicable; in such system we identify precisely the possible races. Our first result can be summarised as: *in our calculus the only source of non-determinacy is the concurrent sending of messages from two active objects to the same third one*. Then we showed that with the given semantics we can design a type system that ensure determinacy of results by enforcing a tree structure for objects. For example, if the active objects were using a communication library ensuring FIFO ordering and deterministic scheduling, our type system would ensure that the correctly typed active objects using this library behave deterministically.

The current results are still restrictive in the programs that can be expressed and the rigidity of its properties; however, we believe that we have a minimal and reliable basis for further studies. In the future, we plan to introduce more dynamic trees for example using linearity and borrowing types, but also primitive

to attach and detach tree to the object dependence graph, in order to constantly ensure a tree structure, but allow the structure of the tree to evolve at runtime.

References

1. de Boer, F., Serbanescu, V., Hähnle, R., Henrio, L., Rochas, J., Din, C.C., Johnsen, E.B., Sirjani, M., Khamespanah, E., Fernandez-Reyes, K., Yang, A.M.: A survey of active object languages. *ACM Comput. Surv.* **50**(5) (October 2017) 76:1–76:39
2. Baker Jr., H.G., Hewitt, C.: The incremental garbage collection of processes. In: *Proc. Symp. on Artificial Intelligence and Programming Languages*. ACM (1977) 55–59
3. Agha, G.: *Actors: a model of concurrent computation in distributed systems*. MIT Press (1986)
4. Lopez, C.T., Marr, S., Boix, E.G., Mössenböck, H.: A study of concurrency bugs and advanced development support for actor-based programs. In: *Programming with Actors - State-of-the-Art and Research Perspectives*. Volume 10789 of *Lecture Notes in Computer Science*. Springer (2018) 155–185
5. Caromel, D., Henrio, L., Serpette, B.: Asynchronous and deterministic objects. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press (2004) 123–134
6. Fernandez-Reyes, K., Clarke, D., Henrio, L., Johnsen, E.B., Wrigstad, T.: Godot: All the benefits of implicit and explicit futures. In: *Proc. 33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Volume 134 of *LIPICs.*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019) 2:1–2:28
7. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: *Proc. 16th European Symposium on Programming (ESOP'07)*. Volume 4421 of *Lecture Notes in Computer Science*. Springer (2007) 316–330
8. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: *Proc. 9th Intl. Symp. on Formal Methods for Components and Objects (FMCO)*. Volume 6957 of *Lecture Notes in Computer Science*. Springer (2011) 142–164
9. Sirjani, M., de Boer, F.S., Movaghar-Rahimabadi, A.: Modular verification of a component-based actor language. *Journal of Universal Computer Science* **11**(10) (2005) 1695–1717
10. Brandauer, S., Castegren, E., Clarke, D., Fernandez-Reyes, K., Johnsen, E.B., Pun, K.I., Tapia Tarifa, S.L., Wrigstad, T., Yang, A.M.: Parallel objects for multicores: A glimpse at the parallel language Encore. In: *Formal Methods for Multicore Programming*. Volume 9104 of *Lecture Notes in Computer Science*. Springer (2015) 1–56
11. Caromel, D., Henrio, L.: *A Theory of Distributed Objects*. Springer-Verlag (2004)
12. Henrio, L., Johnsen, E.B., Pun, K.I.: *Active Objects with Deterministic Behaviour (long version)*. Research Report 8, Western Norway University of Applied Sciences (2020)
13. Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., Quilici, R.: Programming, composing, deploying for the grid. In: *Grid Computing: Software Environments and Tools*. Springer, London (2006) 205–229
14. Jaghoori, M.M., Sirjani, M., Mousavi, M.R., Khamespanah, E., Movaghar, A.: Symmetry and partial order reduction techniques in model checking rebecca. *Acta Informatica* **47**(1) (2010) 33–66

15. Dedecker, J., Cutsem, T.V., Mostinckx, S., Meuter, W.D.: Ambient-oriented programming in ambienttalk. In: Proceedings of 20th European Conference on Object-oriented Programming (ECOOP). Springer (2006)
16. Miller, M.S., Tribble, E.D., Shapiro, J.: Concurrency among strangers: Programming in E as plan coordination. In: Trustworthy Global Computing. Volume 3705 of Lecture Notes in Computer Science. Springer (2005) 195–229
17. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling* **6**(1) (March 2007) 35–58
18. Schafer, J., Poetzsch-Heffter, A.: Jcobox: Generalizing active objects to concurrent components. *ECOOP 2010–Object-Oriented Programming* (2010) 275–299
19. Armstrong, J.: *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf Series. Pragmatic Bookshelf (2007)
20. Clebsch, S., Drossopoulou, S., Blessing, S., McNeil, A.: Deny capabilities for safe, fast actors. In: Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control. AGERE 2015, New York, NY, USA, Association for Computing Machinery (2015) 1–12
21. Castegren, E., Wrigstad, T.: Reference capabilities for concurrency control. In: Proc. 30th European Conference on Object-Oriented Programming (ECOOP 2016). Volume 56 of LIPIcs., Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016) 5:1–5:26
22. Bezirgiannis, N., de Boer, F.S., Johnsen, E.B., Pun, K.I., Tapia Tarifa, S.L.: Implementing SOS with active objects: A case study of a multicore memory system. In: Proc. 22nd Intl. Conf. on Fundamental Approaches to Software Engineering (FASE 2019). Volume 11424 of Lecture Notes in Computer Science., Springer (2019) 332–350
23. Tveito, L., Johnsen, E.B., Schlatte, R.: Global reproducibility through local control for distributed active objects. In: Proc. 23rd International Conference on Fundamental Approaches to Software Engineering (FASE 2020). Volume 12076 of Lecture Notes in Computer Science., Springer (2020) 140–160
24. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science* **410**(2-3) (2009) 202–220
25. Wyatt, D.: *Akka Concurrency*. Artima (2013)
26. Lohstroh, M., Lee, E.A.: Deterministic actors. In: 2019 Forum for Specification and Design Languages, FDL 2019, Southampton, United Kingdom, September 2-4, 2019, IEEE (2019) 1–8
27. Clarke, D., Noble, J., Wrigstad, T., eds.: *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Volume 7850 of Lecture Notes in Computer Science. Springer (2013)
28. Clarke, D., Wrigstad, T., Östlund, J., Johnsen, E.B.: Minimal ownership for active objects. In: Proc. 6th Asian Symposium on Programming Languages and Systems (APLAS 2008). Volume 5356 of Lecture Notes in Computer Science., Springer (2008) 139–154
29. Haller, P., Odersky, M.: Capabilities for uniqueness and borrowing. In: Proc. 24th European Conference on Object-Oriented Programming (ECOOP 2010). Volume 6183 of Lecture Notes in Computer Science., Springer (2010) 354–378
30. Cohen, A., Gérard, L., Pouzet, M.: Programming parallelism with futures in Lustre. In: ACM International Conference on Embedded Software (EMSOFT’12), Tampere, Finland, ACM (October 7-12 2012) Best paper award.