# Inside Every Multithreaded Program There Are Active Objects Struggling To Get Out

Frank de Boer[1], Einar Broch Johnsen[2], Rudolf Schlatte[2], and S. Lizeth Tapia Tarifa[2]

[1] CWI, Amsterdam, the Netherlands
F.S.de.Boer@cwi.nl
[2] Department of Informatics, University of Oslo, Oslo, Norway
{einarj,rudi,sltarifa}@ifi.uio.no

**Abstract.** Multithreading and actors offer different models of concurrency to the programmer. With multithreading, the programmer needs to deal with shared-state and data races, which make programs complex to understand, error-prone and challenging to verify, but potentially very efficient if these issues are mastered to perfection. On the other hand, actors — and their object-oriented incarnation as active objects, — which are inherently concurrent and protect their internal state against races, seem easy to understand and intuitive, but programs may be exposed to deadlocks due to callbacks. Is it possible to simply transition programs from the one concurrency model to the other at will, and thereby get the best of both worlds? We believe such a seamless transition between these concurrency models opens an interesting direction of research that remains to be investigated. As a step in this direction, this paper provides a high-level, informal outline of the translations between multithreading and active object concurrency, highlighting how intuitive or non-intuitive it is to move from one concurrency model to the other.

**Keywords:** Active objects · Actors · Multithreading · Asynchronous programming · Object-oriented programming

## 1 Introduction

The title of this paper is inspired by a quote, attributed to Tony Hoare [44]:

*"Inside every large program, there is a small program trying to get out."*

Reflecting on this quote in the context of concurrency, Marjan Sirjani's work has often championed the simple program trying to get out: the abstract actor-based model that is amenable to automated verification. In particular, her persistent efforts in actor-based modeling and verification in Rebeca [40–42] explores a number of analysis techniques based on formal semantics [35, 36], including simulation [35], model checking [27, 32, 43], statistical model checking [26], partial order reduction [8], and rewriting logic [36]. At the root of this effort lies (we believe) an intuition that actor-based programs [4] are in fact simpler and more intuitive than the competition, namely multithreaded programs (e.g., [7]). In particular, active objects [12], which combine actor-based concurrency with object-based structuring mechanisms seem particularly attractive. In this paper, we expand on this intuition by studying the relation between these two concurrency paradigms. We scope our study to multithreaded programs in Java [33] and active objects in ABS [28]; the further translation of active objects into actors can be achieved via an encoding.

Our study shows that a multithreaded program can be transformed into active objects without affecting the main class structure. This transformation basically consists of a change of the global perspective of the multithreaded flow of control to the local perspective of active objects. That is, instead of the parallel execution of the different threads of method calls (implemented by stacks) the focus is on the parallel execution of the called methods within an object. This paradigm shift is enabled by the basic synchronization mechanisms of the cooperative scheduling of the method execution within active objects, as featured in ABS.

Conversely, modeling a program based on active objects by a multithreaded program requires an invasive and disruptive transformation which involves the introduction of complex synchronization mechanisms, notably mechanisms for suspending and resuming threads which are notoriously complex and error-prone in a multithreaded setting.

*Outline.* In the following section, we introduce the main programming concepts underlying multithreaded programs (MT, for short) and active objects (AO, for short), respectively. In Sections 3 and 4 we discuss the relation between the two paradigms.

We abstract from the technical details and provide a high-level, informal outline of the translations between the two concurrency models. This allows us to focus on the main, basic ideas which otherwise would be obscured by the many technical details of the formal syntax and semantics. However, we think this high-level outline provides a clear guideline for a further formalization.

## 2 Preliminaries

Object-oriented programming abstractions were first introduced in Simula [17, 18]. The basic idea is that of an object as an instance of an *abstract data type* which is represented by a *class*. A class specifies the data structures (referenced by 'fields') and the *methods* which describe the data operations. Objects are dynamically created instances of classes. Abstracting from the method implementations, calling a method of an object thus forms the basic operation of an object-oriented program.

One can distinguish two different ways of calling a method of an object. The first one follows the *rendez-vous* pattern of procedure calls in languages like PASCAL (e.g., [46]). This gives rise to *synchronous* method calls: the caller *suspends* when the method is executed and *resumes* its execution after the method has returned (a value). A sequence of such method calls is called a *thread* (and is usually implemented by a *stack* [19]). This naturally gives rise to *sequential* execution. On the other hand, a method can be called *asynchronously*; that is, the caller continues its execution after the call, without waiting for the callee to respond. This naturally gives rise to the *parallel* execution of the caller and the execution of the called method by the callee, which in turn gives rise to a *co-routine* (used for simulation of parallelism in Simula [17]), a mode of execution of the methods of an object by means of explicit local suspend/resume operations. In the AO language ABS, asynchronous method calls further involve the dynamic creation of *futures* [9]; In ABS, futures are first-class values that act as references to the return value uniquely associated with each method call [13].

In this paper, we consider the AO model of ABS[3] because of its high-level programming abstractions of cooperative scheduling and futures. However, it is worthwhile to mention that these abstractions themselves can be translated into the pure asynchronous AO model of the Rebeca language [43], where methods are executed in a *run-to-completion* mode. A translation of Boolean await statements in ABS [28], for example, can be given by introducing for each occurrence of such a statement a new method. The body of this method consists of a conditional statement statement such that the *syntactic* continuation of the await statement (as given by the method containing this await statement) is executed if the condition of the await statement holds, otherwise a self-call to this new method is executed. Subsequently, all occurrences of await statements (including those appearing in the new methods) are translated into a self-call of the associated method, followed by the return statement (so that the method containing the await statement immediately returns after this self-call). Futures can be modeled by a special class and passing as an additional parameter of an asynchronous method call a newly created instance of this class.

In contrast, in the setting of synchronous method calls, parallel execution of code is achieved by creating *threads*, e.g., via the C function pthread_create or by calling the method start on a Java Thread object. The new thread is started with a programmer-specified "run method", e.g., in C by passing in a function pointer and in Java by either subclassing the Thread class or passing in an object instance of type Runnable. The calling thread continues its execution (run methods do not return a value). Execution of a run method thus spawns a new thread of (synchronous) method calls. In Java, threads further synchronize on so-called *synchronized* methods or blocks, which guarantee *mutual exclusion*, that is, such wrapped code cannot be executed in parallel on a single object.

## 2.1 Executing Multithreaded Programs

A characteristic feature of multithreaded programs in an object-oriented setting is the presence of one or more threads of execution that *keep their identity* as they execute in the context of different objects. Observe that this thread identifier is different from the self-reference **this** typically used by the programmer. When

[3] ABS webpage: abs-models.org

```
class C { private int x = 0; private int y = 0; private int z = 0;
    public void m_1(){this.x = this.x+this.z; this.y = this.y+this.z; };
    public void m_2(){this.z = 5; };
}
final C c = new C();
new Thread ( () −> { c.m_1(); } ).start();
new Thread ( () −> { c.m_2(); } ).start();
```

Fig. 1: Example of a Java MT system accessing shared state

```
class C { /*private fields declaration*/
    public void m_1(/*parameter declaration*/){ s_1m_1; s_2m_1; ···; s_n_1m_1;}
    public void m_2(/*parameter declaration*/){ s_1m_2; s_2m_2; ···; s_n_2m_2;}
    ...
 }
final C c = new C();
new Thread ( () −> {c.m_1(/*parameter instantiation*/);} ).start();
new Thread ( () −> {c.m_2(/*parameter instantiation*/);} ).start();
...
```

Fig. 2: Pattern of a Java MT system accessing shared state

executing a method call in a thread in Java, the self-reference **this** evaluates to the callee object in which the thread executes the method.

In multithreaded programs, it is possible for **this** to evaluate to the same object *at the same time* in multiple threads. If multiple threads change "their" object's state at the same time, data races occur. To illustrate this issue, Figure 1 shows an example of a Java MT system where both methods $\mathsf{m}_1$ and $\mathsf{m}_2$ start execution concurrently in an interleaved manner via lambda expressions, and as a result, it is unclear if the final state of the fields will be {x=0,y=0,z=5}, {x=0,y=5,z=5}, or {x=5,y=5,z=5}. Such a pattern of interleaved execution is depicted in Figure 2.

To avoid data races in this setting, all threads have to cooperate and employ locks to define critical sections. Some syntactic sugar can be used to ease this process; for example, Java's **synchronized** methods and blocks guarantee that only one thread executing synchronized code can execute at a time, thus protecting the shared state from races from other synchronized threads. For the example in Figure 1, by synchronizing the execution of the method bodies, we restrict the final state to be either {x=0,y=0,z=5} or {x=5,y=5,z=5}. Such a pattern of atomic execution with synchronized (atomic) blocks is depicted in Figure 3, where a method can contain multiple atomic blocks. Note that multiple threads can still execute "on" such an object if they execute code outside the protected code parts. It is worth noting that Java supports *reentrance* [1]: a thread that holds a lock by executing a synchronized method may take the lock again, thus Java supports recursion for synchronized methods. Also note that a thread may

```
class C { /*private fields declaration*/
  public void m1(/*parameters' declaration*/){ synchronized(this){s1m1; s2m1; ···; sn1m1;}}
  public void m2(/*parameters' declaration*/){ synchronized(this){s1m2; s2m2; ···; sn2m2;}}
    ...
 }
final C c = new C();
new Thread ( () −> {c.m1(/*parameters' instantiation*/);} ).start();
new Thread ( () −> {c.m2(/*parameters' instantiation*/);} ).start();
...
```

Fig. 3: Pattern of a Java MT system accessing shared state with methods that are executed atomically via *synchronized* methods.

```
interface I { Unit method(Int i); }

class C(I o) {
  Int x = 0;
  Unit run() {
    println("Before Boolean await");
    await x > 0;
    println("After Boolean await");
    Fut<Unit> f = o!m(x);
    println("Before future await");
    await f?;
    println("After future await");
  }

  Unit incX() { x = x + 1; }
}
```

Fig. 4: Example of await statements with Boolean and future conditions

acquire locks to multiple objects by nested synchronized method calls, which may lead to deadlocks if, e.g., threads try to acquire locks in different order.

Multithreaded program execution can be seen as if, after each atomic statement, execution switches to another thread. This makes correctness proofs for multithreaded code very challenging.

## 2.2 Executing Active Objects

A characteristic feature of active object programs is that communication and synchronization are decoupled [29]: there is no transfer of control associated with method calls. Each object “owns” its threads. Calling a method creates a fresh thread on the callee object. The caller receives a *future* that represents the fresh thread and can be used to obtain the result of the method call.

Since the object owns its threads, it is not possible for multiple threads to execute at the same time on the same object. This means that all code is implicitly protected as if surrounded by *synchronous* blocks, and that the execution inside an active object is race free by design. Instead, with *cooperative concurrency* [28, 29, 37], explicit **await**-statements allow the object to switch from executing one thread to executing another thread, leaving the first thread suspended. These **await**-statements can carry a condition that expresses when the current thread becomes eligible to run again. In the simplest case, the condition can be simply *true*. Other conditions include waiting for a future to obtain the result of the method call associated with that future, and waiting for a Boolean condition over the object state to hold (e.g., a thread can wait until a counter reaches zero). Figure 4 shows a code example with these different **await**-statements. ABS finally supports unconditional cooperative scheduling with the statement **suspend**, which always releases control when executing and is always schedulable when suspended. A formal account of cooperative scheduling in ABS have been given in terms of an operational semantics [28] as well as a recent denotational semantics [21].

From the perspective of verification, since **await**-statements in ABS models are explicit in the code, it becomes feasible to prove object invariants and other correctness conditions of active object code. Thus, active objects can be seen as the maintainers of local invariants [13, 20, 22]. Another interesting aspect of the active object concurrency model is that a transition system can be translated to active objects with cooperative concurrency in a way that *preserves global properties* [10, 15]; this translation enables the use of a transition system model with very expressive synchronization to inductively establish global properties, then obtaining a decentralized active object program with the same global properties.

Of further note is that ABS has a notion of *blocking*: a thread can wait for the result of a future without relinquishing control by having a **get**-statement to a future right after an asynchronous call associated with such future; the **get**-statement will block the active object until the result of the associated method call is available in that future. This behavior can be used to express synchronization similar to synchronous method calls in Java, but does not directly support reentrance. The drawback of this synchronization is that the object cannot switch to another thread, including the one created from the process the original thread is waiting for, in the case of recursion. Such callbacks lead to deadlock; we will explore this area further in the sequel.

## 3 From Multithreading to Active Objects

A faithful translation of code from the multithreaded (MT) programs to the active object (AO) programs should preserve the interleaving structure of code segments of the multithreaded execution model. The basic idea underlying the translation of a multithreaded (MT) program is to model a synchronous method call directly by an asynchronous method call followed by a so-called **await**-statement which suspends the calling method instance until the called method returns. This suspension allows for the execution of other methods by the object, while the calling thread must wait for the return of the call. We can further model that the entire calling object is blocked for execution during the call by

```
interface C {Unit m1(); Unit m2();};
class C() implements C {Int x = 0; Int y = 0; Int z = 0;
      Unit m1(){x = x+z; suspend; y = y+z; }
      Unit m2(){z = 5; } }
{ C c = new C(); c!m1(); c!m2();}
```

Fig. 5: The AO version of the example in Figure 1.

```
interface C {Unit m1(/*parameters' declaration*/); Unit m2(/*parameters' declaration*/); ...};
class C() implements C {/*private fields declaration*/
      Unit m1(/*parameters' declaration*/){s1m1; suspend; s2m1; suspend; ···; suspend; sn1m1;}
      Unit m2(/*parameters' declaration*/){s1m2; suspend; s2m2; suspend; ···; suspend; sn2m2;}
      }...
{ C c = new C(); c!m1(/*parameters' instantiation*/); c!m2(/*parameters' instantiation*/); ...}
```

Fig. 6: The AO version of the pattern in Figure 2

waiting on a **get**-statement instead of the **await**-statement; this models a synchronous call from a synchronized method [29]. However, this approach does not directly support recursive calls; in fact, the semantics of Creol [29] (that was inherited in ABS [28]), treated this case separately — but the proposed solution did not address the general case of reentrance (as we would encounter with mutual recursion between synchronized methods in different objects).

To model the mechanism of *reentrance* for synchronized method calls from the multithreaded setting in the active object setting, we introduce for each object a *lock* which specifies the thread id and the number of times that this id has acquired the lock. The thread id corresponds to the unique reference to the initial instance of the corresponding thread class. To allow each method invocation of a thread to access its thread id, the thread id is simply passed on as an additional parameter to synchronous method calls. A method instance holds a lock of an object if the lock stores its thread id in the lock with a non-zero counter. A lock is free if its counter is zero. We then extend every method (body) with an initial **await**-statement which checks whether the method instance holds the lock or whether the lock is free. In both cases the counter of the lock is incremented (by one). In case the lock is free the stored thread id is updated, that is, set to the thread id of the executing method instance. Upon return of a method, the lock is simply decremented.

In ABS, the granularity of the interleaving of the execution of different method instances by an object, is controlled by explicit suspend/resume statements (similar to **await**-statements without condition guards). To allow for the arbitrary interleaving needed to model the behavior of multithreaded execution, we need to inject interleaving points in code that would otherwise be "too synchronized" when executing in the active object. In ABS, one simply adds to

```
interface C {Unit m_1(/*parameters' declaration*/); Unit m_2(/*parameters' declaration*/); ...};
class C() implements C {/*private variable declaration*/
      Unit m_1(/*parameters' declaration*/){ s_{1m_1}; s_{2m_1}; ···; s_{n_1m_1}; }
      Unit m_2(/*parameters' declaration*/){ s_{1m_2}; s_{2m_2}; ···; s_{n_2m_2};} }
      ...
{ C c = new C(); c!m_1(/*parameters' instantiation*/); c!m_2(/*parameters' instantiation*/); ...}
```

Fig. 7: The AO version of the pattern with *synchronized* methods in Figure 3.

the different control points a **suspend**-statement which unconditionally releases and resumes control. Figure 5 shows the AO version of the example in Figure 1, where to capture the interleaved execution inside implicit atomic execution of threads in an AO setting, we introduce a **suspend**-statement after each statement, giving the possibility to another suspended thread to become active and start execution. The AO pattern version of the MT pattern in Figure 2 is depicted in Figure 6. Furthermore, synchronized MT methods or blocks are similar to the implicit atomic execution of threads in an AO setting, as can be seen in the depicted AO pattern version in Figure 7 of the MT pattern in Figure 3, where capturing various atomic blocks inside one method can be done by adding a **suspend**-statement between the blocks.

The code of an arbitrary Java class may mix synchronized and unsynchronized blocks of execution in different methods. This effect can be translated into ABS by combining the two concerns we have discussed, namely the counting lock for synchronized code over explicit thread identifiers that are passed in synchronous calls, and the injected suspension points to allow arbitrary interleaving on unsynchronized code. Observe that in a *pure actor* setting, these effects need to be modeled by decoupling the translated code into smaller blocks corresponding to atomic statements in Java, that are successively triggered by separate messages, and by renaming the actor to force synchronization for call-backs by providing unique return-addresses to calls that need to be synchronized. This line of encoding was studied in early work by Agha *et al.* (e.g., [2]). This kind of encoding breaks the class method structure of the Java program that we are able to preserve in ABS because of the supported cooperative scheduling and futures.

## 4 From Active Objects to Multithreading

A faithful translation of code from the active object model (AO) to the multithreaded model (MT) preserves the following properties:

– Only one process at a time executes on an object.
– When a method is called, the calling process continues execution.
– Processes on an object can interleave execution via suspension points.
– Processes can obtain method call results via future variables.

Note that the first two properties are enough to implement pure actor behavior, the remaining points are necessary to realize the active object semantics. We consider two approaches to this encoding problem, by means of thread suspension (Section 4.1) and by means of continuations (Section 4.2). Both approaches, as presented, create potentially large numbers of threads, so they are most suited to languages like Erlang or Java $\geq 21$ (which adds so-called virtual threads) that do not place restrictions on thread count. If thread creation is an expensive operation in the chosen language, for example because each language thread is implemented as an operating-system thread, the approaches in this chapter can be adapted to use thread pools and/or task queues.

## 4.1 From AO to MT with Thread Suspension

AO classes can be directly translated into MT classes, with no special provisions except for synchronization points; this section presents a simplified version of the JCoBox model [37], which was used in the original Java interpreter for ABS. AO method calls, which are asynchronous, are naturally modeled in the MT model by executing each call on a separate thread, but care must be taken to preserve the AO properties. In Java and similar languages, this can be done by wrapping each method call together with its argument values either in a closure (a lambda expression) or in an object implementing the Runnable interface, that is then executed on a MT thread. Figure 8 shows a worked example using the ExecutorService library class that also produces a Java Future object that the caller can use to obtain the method's return value.

AO suspension points are points in code where the MT translation must give up its lock on the object to let other threads execute. Figure 8 shows the encoding of **await** statements both for futures and for Boolean conditions. A future is monotonic in the sense that, once completed, it will always remain available. Boolean conditions, on the other hand, can change between every suspension point, and must be always re-checked before continuing execution. Note that the transformation also inserts a call to **this.notifyAll**() before each suspension point. This is necessary so that threads waiting on a Boolean await can check the await condition. Since in ABS all members are private and can only be changed via method calls, these calls are also sufficient.

## 4.2 From AO to MT with Continuations

An alternative translation of AO code into the MT model introduces a new method for each suspension point, following the pattern of the translation of ABS **await**-statements into Rebeca discussed in Section 2. This section presents a simplified version of this approach, studied by Serbanescu [39]. Figure 9 shows the example of Figure 4 translated in this style. This translation method introduces one synchronized "continuation method" for each occurrence of an **await**-statement in the AO method such that each such occurrence is modeled by the creation of a fresh thread continuing after the suspension point, passing along local variables in scope at the point of the await. Then, the original thread

```
import java.util.concurrent.*;

interface I { void method(int i); }

class C {
  I o;
  int x = 0;
  public C(I o) { this.o = o; }

  void run() {
    final Future f;
    synchronized(this) {
      System.out.println("Before Boolean await");
      while (!(x > 0)) {
        this.notifyAll();
        this.wait(); // Releases lock
      }
      System.out.println("After Boolean await");

      ExecutorService executor = Executors.newSingleThreadExecutor();
      f = executor.submit(() −> o.method(x));
      System.out.println("Before future await");
      this.notifyAll();
    }
    f.get(); // other methods can run in the meantime
    synchronized(this) {
      System.out.println("After future await");
      this.notifyAll();
    }
  }

  synchronized void incX() { x = x + 1; this.notifyAll(); }
}
```

Fig. 8: A Java translation of cooperative scheduling in ABS, using thread suspension (exception handling elided for brevity)

ends and releases the object lock, letting other methods execute on the object. The continuation of the method can start executing when the await condition is fulfilled.

The body of the continuation of the run method itself starts with a prelude consisting of a transformation of the **await**-statement and its *syntactic* continuation into a *waiting loop* which checks the condition of the **await**-statement. If this condition holds, the loop is exited and the method executes the rest of the AO method. Note that differently from the translation of **await**-statements in the Rebeca language, it is not necessary to spawn a new process each time the condition of the **await**-statement does not hold; instead, the thread checks its await condition multiple times and starts running when the condition holds. As with the other translation, any threads waiting on the object lock are notified before exiting a synchronized method, so they can check their await condition.

```
import java.util.concurrent.*;
interface I { void method(int i); }

class C {
  I o;
  int x = 0;
  public C(I o) { this.o = o; }

  synchronized void run() {
    System.out.println("Before Boolean await");
    new Thread(() −> this.run2()).start();
    this.notifyAll();
  }
  synchronized void run2() {
    while (!(x > 0)) { this.notifyAll(); this.wait(); }
    System.out.println("After Boolean await");
    ExecutorService executor = Executors.newSingleThreadExecutor();
    final Future f = executor.submit(() −> o.method(x));
    new Thread(() −> this.run3(f)).start();
    System.out.println("Before future await");
    this.notifyAll();
  }
  void run3(Future f) {
    while (!f.isDone()) { f.get(); }
    synchronized(this) {
      System.out.println("After future await");
      this.notifyAll();
    }
  }

  synchronized void incX() { x = x + 1; this.notifyAll(); }
}
```

Fig. 9: Java translation of cooperative scheduling in ABS, using continuations (exception handling elided for brevity)

## 5 Conclusion

This paper has discussed differences between MT and AO, and in particular the nature of their basic underlying communication mechanisms. In AO, methods are called asynchronously and as such the generated method invocations are executed in parallel. In other words, the paradigm of AO programming is *intrinsically parallel*. In MT, methods are called synchronously, which imposes a last-in-first-out (LIFO) ordering of the execution of the method invocations, and as such gives rise to an *intrinsically sequential* thread-based execution. Parallel execution of threads in MT is obtained by calling asynchronously the run method (or start method) of an instance of a predefined class Thread.

We have shown that it is straightforward to model a synchronous method call in AO, using the **await**-statement of ABS to suspend the caller. To model the arbitrary interleaving of threads in AO simply requires the introduction of **await**-statements at the interleaving points. This is cumbersome, but only syntactically: It does not complicate the object structures. Thread synchronization is modeled in a straightforward manner by a basic semaphore. This requires the introduction of an additional parameter for a (synchronized) method which denotes the executing thread (i.e., the initial object executing the run method). It should be observed that our outlined encoding from MT to AO relies heavily on the synchronization features of ABS, with asynchronous method calls and cooperative scheduling. The further translation into a pure actor language would require carefully encoding synchronization and call-backs that are naturally supported in ABS through asynchronous message passing and through control restrictions on th e message-interface(e.g, by means of the become-statement in pure actors). On the other hand, modeling arbitrary asynchronous method calls in MT requires quite complex explicit handling of concurrency mechanisms, which gives rise to complicated object structures to handle synchronization points in the AO programs (e.g., wrapping each method call into a proxy of the called object).

We believe a more formal study of these encodings and how they preserve the semantics of the two concurrency paradigms would be an interesting line of research. Research on the relationship between MT on the one side and AO and actors on the other side, is surprisingly scarce. Among interesting work pointing in this general direction, Agha and Palmskog [3] studied how to learn actor structure from the execution traces of MT programs, and de Boer and Hiep [14] the synthesis of actor programs from traces. Haller and Odersky [24] studied how MT execution can be unified with event-based communication. Several papers compare different mechanisms for asynchronous communication with futures (e.g., [23]). Other papers study the extension of AO models in ABS with resource-sensitive behavior [6,11,31,38], these resources act as global synchronizers that affect the intrinsic compositionality of the underlying actor model (for example to model virtualized systems [5,30]). Varela and Agha [45] discuss drawbacks of MT and argue for actors, and Lee [34] highlight challenges with MT and discuss different alternatives, including actors, suggesting that a solution might lie in more abstract coordination languages. Brandauer *et al.* [16] and Henrio and Rochas [25] discuss different ways to overcome the intra-actor sequentialization in the AO model.

## References

1. Ábrahám-Mumm, E., de Boer, F.S., de Roever, W.P., Steffen, M.: Verification for Java's reentrant multithreading concept. In: Nielsen, M., Engberg, U. (eds.) Proc. 5th International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2002). Lecture Notes in Computer Science, vol. 2303, pp. 5–20. Springer (2002), https://doi.org/10.1007/3-540-45931-6_2
2. Agha, G.: Concurrent object-oriented programming. Commun. ACM **33**(9), 125–141 (1990), https://doi.org/10.1145/83880.84528
3. Agha, G., Palmskog, K.: Transforming threads into actors: Learning concurrency structure from execution traces. In: Lohstroh, M., Derler, P., Sirjani, M. (eds.) Principles of Modeling - Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 10760, pp. 16–37. Springer (2018), https://doi.org/10.1007/978-3-319-95246-8_2

4. Agha, G.A.: ACTORS - a model of concurrent computation in distributed systems. MIT Press series in artificial intelligence, MIT Press (1990)
5. Albert, E., de Boer, F.S., Hähnle, R., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L., Wong, P.Y.H.: Formal modeling and analysis of resource management for cloud architectures: an industrial case study using real-time ABS. Serv. Oriented Comput. Appl. **8**(4), 323–339 (2014). https://doi.org/10.1007/S11761-013-0148-0
6. Albert, E., Genaim, S., Gómez-Zamalloa, M., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Simulating concurrent behaviors with worst-case cost bounds. In: Butler, M.J., Schulte, W. (eds.) Proc. 17th International Symposium on Formal Methods (FM 2011). Lecture Notes in Computer Science, vol. 6664, pp. 353–368. Springer (2011). https://doi.org/10.1007/978-3-642-21437-0_27
7. Andrews, G.R.: Concurrent programming - principles and practice. Benjamin/Cummings (1991)
8. Bagheri, M., Sirjani, M., Khamespanah, E., Hojjat, H., Movaghar, A.: Partial order reduction for timed actors. In: Bloem, R., Dimitrova, R., Fan, C., Sharygina, N. (eds.) Proc. 13th International Conference on Software Verification (VSTTE 2021). Lecture Notes in Computer Science, vol. 13124, pp. 43–60. Springer (2021), https://doi.org/10.1007/978-3-030-95561-8_4
9. Baker, H.G., Hewitt, C.: The incremental garbage collection of processes. In: Low, J. (ed.) Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages. pp. 55–59. ACM (1977), https://doi.org/10.1145/800228.806932
10. 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: Hähnle, R., van der Aalst, W.M.P. (eds.) Proc. 22nd Intl. Conf. on Fundamental Approaches to Software Engineering (FASE 2019). Lecture Notes in Computer Science, vol. 11424, pp. 332–350. Springer (2019), https://doi.org/10.1007/978-3-030-16722-6_20
11. Bjørk, J., de Boer, F.S., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: User-defined schedulers for real-time concurrent objects. Innov. Syst. Softw. Eng. **9**(1), 29–43 (2013), https://doi.org/10.1007/s11334-012-0184-5
12. de Boer, F., Din, C.C., Fernandez-Reyes, K., Hähnle, R., Henrio, L., Johnsen, E.B., Khamespanah, E., Rochas, J., Serbanescu, V., Sirjani, M., Yang, A.M.: A survey of active object languages. ACM Computing Surveys **50**(5), 76:1–76:39 (Oct 2017), https://doi.org/10.1145/3122848
13. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: Nicola, R.D. (ed.) Proc. 16th European Symposium on Programming (ESOP 2007). Lecture Notes in Computer Science, vol. 4421, pp. 316–330. Springer (2007), https://doi.org/10.1007/978-3-540-71316-6_22
14. de Boer, F.S., Hiep, H.A.: Axiomatic characterization of trace reachability for concurrent objects. In: Ahrendt, W., Tapia Tarifa, S.L. (eds.) Proc. 15th International Conference on Integrated Formal Methods (IFM 2019). Lecture Notes in Computer Science, vol. 11918, pp. 157–174. Springer (2019), https://doi.org/10.1007/978-3-030-34968-4_9
15. de Boer, F.S., Johnsen, E.B., Pun, V.K.I., Tapia Tarifa, S.L.: Proving correctness of parallel implementations of transition system models. ACM Trans. Program. Lang. Syst. **46**(3) (Sep 2024), https://doi.org/10.1145/3660630
16. 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: Bernardo, M., Johnsen, E.B. (eds.) Formal Methods for Multicore Programming (SFM 2015). Lecture Notes in

Computer Science, vol. 9104, pp. 1–56. Springer (2015), https://doi.org/10.1007/978-3-319-18941-3_1
17. Dahl, O.J., Myhrhaug, B., Nygaard, K.: SIMULA 67 common base language. Tech. Rep. S-2, Norwegian Computing Center (1968)
18. Dahl, O.J., Nygaard, K.: SIMULA - an ALGOL-based simulation language. Commun. ACM **9**(9), 671–678 (1966), https://doi.org/10.1145/365813.365819
19. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976), https://www.worldcat.org/oclc/01958445
20. Din, C.C., Bubel, R., Hähnle, R.: KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In: Felty, A.P., Middeldorp, A. (eds.) Proc. 25th International Conference on Automated Deduction (CADE-25). Lecture Notes in Computer Science, vol. 9195, pp. 517–526. Springer (2015), https://doi.org/10.1007/978-3-319-21401-6_35
21. Din, C.C., Hähnle, R., Henrio, L., Johnsen, E.B., Pun, V.K.I., Tapia Tarifa, S.L.: Locally abstract, globally concrete semantics of concurrent programming languages. ACM Trans. Program. Lang. Syst. **46**(1), 3:1–3:58 (2024). https://doi.org/10.1145/3648439, https://doi.org/10.1145/3648439
22. Din, C.C., Tapia Tarifa, S.L., Hähnle, R., Johnsen, E.B.: History-based specification and verification of scalable concurrent and distributed systems. In: Butler, M.J., Conchon, S., Zaïdi, F. (eds.) Proc. 17th International Conference on Formal Engineering Methods (ICFEM 2015). Lecture Notes in Computer Science, vol. 9407, pp. 217–233. Springer (2015). https://doi.org/10.1007/978-3-319-25423-4_14
23. Fernandez-Reyes, K., Clarke, D., Henrio, L., Johnsen, E.B., Wrigstad, T.: Godot: All the benefits of implicit and explicit futures. In: Donaldson, A.F. (ed.) Proc. 33rd European Conference on Object-Oriented Programming (ECOOP 2019). LIPIcs, vol. 134, pp. 2:1–2:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019), https://doi.org/10.4230/LIPIcs.ECOOP.2019.2
24. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theor. Comput. Sci. **410**(2-3), 202–220 (2009), https://doi.org/10.1016/j.tcs.2008.09.019
25. Henrio, L., Rochas, J.: Multiactive objects and their applications. Log. Methods Comput. Sci. **13**(4) (2017), https://doi.org/10.23638/LMCS-13(4:12)2017
26. Jafari, A., Khamespanah, E., Kristinsson, H., Sirjani, M., Magnusson, B.: Statistical model checking of timed rebeca models. Comput. Lang. Syst. Struct. **45**, 53–79 (2016), https://doi.org/10.1016/j.cl.2016.01.004
27. Jaghoori, M.M., Movaghar, A., Sirjani, M.: Modere: the model-checking engine of Rebeca. In: Haddad, H. (ed.) Proc. Symposium on Applied Computing (SAC 2006). pp. 1810–1815. ACM (2006), https://doi.org/10.1145/1141277.1141704
28. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010). Lecture Notes in Computer Science, vol. 6957, pp. 142–164. Springer (2010), https://doi.org/10.1007/978-3-642-25271-6_8
29. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Softw. Syst. Model. **6**(1), 39–58 (2007), https://doi.org/10.1007/s10270-006-0011-2
30. Johnsen, E.B., Owe, O., Schlatte, R., Tapia Tarifa, S.L.: Dynamic resource reallocation between deployment components. In: Dong, J.S., Zhu, H. (eds.) Proc.

12th International Conference on Formal Engineering Methods, ICFEM 2010. Lecture Notes in Computer Science, vol. 6447, pp. 646–661. Springer (2010). https://doi.org/10.1007/978-3-642-16901-4_42

31. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Integrating deployment architectures and resource consumption in timed object-oriented models. J. Log. Algebraic Methods Program. **84**(1), 67–91 (2015), https://doi.org/10.1016/j.jlamp.2014.07.001
32. Khamespanah, E., Mechitov, K., Sirjani, M., Agha, G.A.: Schedulability analysis of distributed real-time sensor network applications using actor-based model checking. In: Bosnacki, D., Wijs, A. (eds.) Proc. 23rd International Symposium on Model Checking Software (SPIN 2016). Lecture Notes in Computer Science, vol. 9641, pp. 165–181. Springer (2016), https://doi.org/10.1007/978-3-319-32582-8_11
33. Lea, D.: Concurrent Programming in Java. Addison Wesley (1996)
34. Lee, E.A.: The problem with threads. Computer **39**(5), 33–42 (2006), https://doi.org/10.1109/MC.2006.180
35. Reynisson, A.H., Sirjani, M., Aceto, L., Cimini, M., Jafari, A., Ingólfsdóttir, A., Sigurdarson, S.H.: Modelling and simulation of asynchronous real-time systems using Timed Rebeca. Science of Computer Programming **89**, 41–68 (2014), https://doi.org/10.1016/J.SCICO.2014.01.008
36. Sabahi-Kaviani, Z., Khosravi, R., Ölveczky, P.C., Khamespanah, E., Sirjani, M.: Formal semantics and efficient analysis of Timed Rebeca in Real-Time Maude. Science of Computer Programming **113**, 85–118 (2015), https://doi.org/10.1016/J.SCICO.2015.07.003
37. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing active objects to concurrent components. In: D'Hondt, T. (ed.) Proc. 24th European Conference, on Object-Oriented Programming (ECOOP 2010). Lecture Notes in Computer Science, vol. 6183, pp. 275–299. Springer (2010), https://doi.org/10.1007/978-3-642-14107-2_13
38. Schlatte, R., Johnsen, E.B., Mauro, J., Tapia Tarifa, S.L., Yu, I.C.: Release the beasts: When formal methods meet real world data. In: de Boer, F.S., Bonsangue, M.M., Rutten, J. (eds.) It's All About Coordination — Essays to Celebrate the Lifelong Scientific Achievements of Farhad Arbab. Lecture Notes in Computer Science, vol. 10865, pp. 107–121. Springer (2018), https://doi.org/10.1007/978-3-319-90089-6_8
39. Serbanescu, V., de Boer, F.S., Jaghoori, M.M.: Actors with coroutine support in Java. In: Bae, K., Ölveczky, P.C. (eds.) Proc. 15th International Conference, FACS 2018. Lecture Notes in Computer Science, vol. 11222, pp. 237–255. Springer (2018), https://doi.org/10.1007/978-3-030-02146-7_12
40. Sirjani, M.: Rebeca: Theory, applications, and tools. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Proc. 5th International Symposium on Formal Methods for Components and Objects (FMCO 2006). Lecture Notes in Computer Science, vol. 4709, pp. 102–126. Springer (2006), https://doi.org/10.1007/978-3-540-74792-5_5
41. Sirjani, M., Jaghoori, M.M.: Ten years of analyzing actors: Rebeca experience. In: Agha, G., Danvy, O., Meseguer, J. (eds.) Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday. Lecture Notes in Computer Science, vol. 7000, pp. 20–56. Springer (2011), https://doi.org/10.1007/978-3-642-24933-4_3
42. Sirjani, M., Khamespanah, E.: On time actors. In: Ábrahám, E., Bonsangue, M.M., Johnsen, E.B. (eds.) Theory and Practice of Formal Methods - Essays Dedicated

to Frank de Boer on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 9660, pp. 373–392. Springer (2016), https://doi.org/10.1007/978-3-319-30734-3_25

43. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using Rebeca. Fundam. Informaticae **63**(4), 385–410 (2004), http://content.iospress.com/articles/fundamenta-informaticae/fi63-4-05
44. Turski, W. (ed.): Efficient Production of Large Programs. Computation Centre of the Polish Aacdemy of Science (Aug 10–14 1970), https://www.cs.ox.ac.uk/publications/publication8100-abstract.html
45. Varela, C.A., Agha, G.: What after Java? From objects to actors. Comput. Networks **30**(1-7), 573–577 (1998), https://doi.org/10.1016/S0169-7552(98)00079-8
46. Wirth, N.: The programming language Pascal. Acta Informatica **1**, 35–63 (1971), https://doi.org/10.1007/BF00264291