From Modelling to Systematic Deployment of Distributed Active Objects

Ludovic Henrio, Justine Rochas

From Modelling to Systematic Deployment of Distributed Active Objects

Ludovic Henrio and Justine Rochas

Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271 06900 Sophia Antipolis, France
ludovic.henrio@cnrs.fr, justine.rochas@unice.fr

Abstract. In the context of the expansion of actors and active objects, we are still facing a gap between the safety guaranteed by modelling and verification languages and the efficiency of distributed middlewares. In this paper, we reconcile two active object-based languages, ABS and ProActive, that respectively target the aforementioned goals. We compile ABS programs into ProActive, making possible to benefit from the strengths of both languages, while requiring no modification on the source code. After introducing the translational semantics, we establish the properties and the correctness of the translation. Overall, this paper presents an approach to running different active object models in distributed environments, and more generally studies the implementation of programming languages based on active objects.

1 Introduction

Writing distributed and concurrent applications is a challenging task. In distributed environments, the absence of shared memory makes information sharing more difficult. In concurrent environments, data sharing is easy but shared data must be manipulated with caution. Several languages and tools have been developed to handle those two programming challenges and make distributed and concurrent systems safe by construction. Among them, the active object programming model [18] helps building safe multi-core applications in object-oriented programming languages. The active object model derives from the actor model [1] that is particularly regaining popularity with Scala [11] and Akka[1]. Such models are natively adapted to distribution because entities do not share memory and behave independently from each other.

There exist now several programming languages implementing and enhancing in various ways the active object and actor models. In particular, emerging active object languages, like the Abstract Behavioral Specification language [15] (hereafter ABS), provide various programming abstractions or static guarantees that help the developer designing and implementing robust distributed systems. Among existing implementations of active objects, ProActive[2] is a Java middleware implementing multi-threaded active objects that provides a holistic support

[1] http://akka.io
[2] http://proactive.inria.fr/

for deployment and execution of active objects on distributed infrastructures. This paper reconciles cooperative active object languages by translating their main concurrent paradigms into ProActive, thus benefiting from its support for deployment. We illustrate our approach on ABS, which has a wide support for modelling and verification. We translate all the concurrent object layer of ABS into ProActive. We also introduce in this paper MultiASP, a formal language that models ProActive, in order to verify the translation.

Beyond the generic high-level approach to cross-translating active object languages, the practical contribution of this paper is a ProActive backend for ABS, that automatically translates an ABS application into a distributed ProActive application. As a result, the programmer can design and verify his program using the powerful toolset of ABS, and then generate efficient distributed Java code that runs with ProActive. The proof of correctness of the translation ensures the equivalence of execution in terms of the operational semantics. Consequently, it guarantees that the verified properties dealing with the program behaviour (e.g. absence of deadlocks, typing properties) will still be valid. Our approach requires no change in the ABS code except the minimal (required) deployment information. Overall, our contribution can be summarised in four points:

- We analyse existing active object programming paradigms in Section 2.
- We provide MultiASP, a class-based semantics of the multi-threaded active objects featured in ProActive in Section 3.
- We present a systematic strategy to translate active objects with cooperative scheduling into ProActive, and present more specifically the ProActive backend for ABS in Section 4. The translation is formalised in Section 5.
- We prove translation equivalence in Section 6 and highlight similarities and differences between active object models. In particular the proof of equivalence reveals intrinsic differences between explicitly typed futures and transparent first-class futures.

2 Background and Related Works

The actor model was one of the first to schematically consider concurrent entities evolving independently and communicating via asynchronous messages. Later on, active objects have been designed as the object-oriented counterpart of the actor model. The principle of active objects is to have a thread associated to them. We call this notion *activity*: a thread together with the objects managed by this thread. Objects from different activities communicate with remote method invocations: when a method is invoked on a remote active object, this creates a *request* in the remote activity; the invoker continues its execution while the invoked active object serves the request asynchronously. Requests wait in a *request queue* until they are executed. In order to allow the invoker to continue execution, a placeholder for the expected result is created, known as *future* [9]: an empty object that will later be filled by the result of the request. When the value of a future is known, we say that it is *resolved*.

2.1 Design choices for active object-based languages

Implementing active objects raises the three following questions:

How are objects associated to activities? In uniform active object models, all objects are active and have their own execution thread (e.g. Creol [16]). This model is distinguished from non uniform active object models which feature active and passive objects (e.g. ASP [6]). Each passive object is a normal object not equipped with any thread nor request queue; there is no race condition on the access to passive object because each of them is accessible by a single active object. In practice, non uniform active object models are more scalable, but they are trickier to formalise than uniform active object models. A trade-off between those two models appeared with JCoBox [20] that introduced the active object group model, where all objects are accessible from any object, but where objects of the same group share the same execution thread.

How are requests scheduled? The way requests are executed in active objects depends on the threading model used. In the original programming model, active objects are mono-threaded. With cooperative scheduling like in Creol, requests in execution can be paused on some condition (e.g. awaiting on the resolution of a future), letting another request progress in the meantime. In all cooperative active object languages, while no data race is possible, interleaving of the different request services (triggered by the different release points) makes the behaviour more difficult to predict than for the mono-threaded model. Still, the previous models are inefficient on multi-cores and can lead to deadlocks due to reentrant calls and/or inadequately placed release points. Newest active object models like multiactive objects [12] and Encore [5] feature controlled multi-threading. Such active object models succeed in maximising local parallelism while avoiding communication overhead, thanks to shared memory between the different threads [12]. Also, controlled multi-threading prevents many deadlocks in active object executions.

Is the programmer aware of distributed aspects? Existing implementations of active objects either choose to hide asynchrony and distribution or, on the contrary to use an explicit syntax for handling asynchronous method calls and to use an explicit type for handling futures. This makes the programmer aware of where synchronisation occurs, but consequently requires more expertise. The choice of transparency also impacts the language possibilities, like future reference transmission: it is easier to transmit futures between active objects when no specific future type is used, and the programmer does not have to know how many future indirections have to be unfolded to get the final value.

2.2 Overview of active object-based languages

Creol [16] is a uniform active object language that features cooperative scheduling based on `await` operations that can release the execution thread. In this language, asynchronous invocations and futures are explicit, and futures are not transmitted between activities. De Boer et al. formalised such futures based on

g::=b | x? | $g \wedge g'$	guard
s::=`skip` | $x = z$ | suspend | await g	statement
| `return` e | `if` e $\{s\}$ `else` $\{s\}$ | s `;` s	
z::=e | e.`m`($\overline{e}$) | e!`m`($\overline{e}$) | `new` $[cog]C(\overline{e})$ | x.get	expression with side effect
e::=v | x | `this` | *arithmetic-bool-exp*	expression
v::=`null` | *primitive-val*	value

Fig. 1: Class-based syntax of the concurrent object layer of ABS. Field access is restricted to current object (`this`).

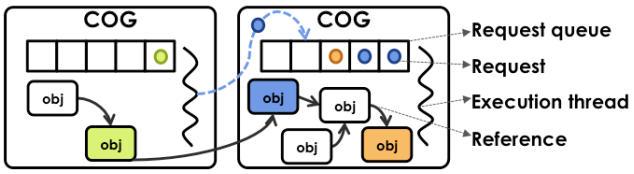

Fig. 2: An example of ABS program execution

Creol in [4]. Overall, explicit future access, explicit release points, and explicit asynchronous calls make Creol rich and precise but also more difficult to program than the languages featuring more transparency.

JCoBox [20] is an active object programming model implemented in a language based on Java. It has an object group model, called CoBox, and also features cooperative scheduling. In each CoBox, a single thread is active at a time; it can be released using `await()`. JCoBox better addresses practical aspects than Creol: it is integrated with Java and the object group model improves thread scalability, however JCoBox does not support distributed execution. Thread interleaving is similar and has the same advantages and drawbacks as in Creol.

AmbientTalk [7] is an object-oriented distributed programming language that can execute on the JVM. One original aspect of AmbientTalk is that a future access is a non-blocking operation: it is an asynchronous call that returns another future; the call will be performed when the invoked future is resolved. The AmbientTalk future model forces two activities to coordinate only through callbacks. This inversion of control has the advantage to avoid deadlocks but also breaks the program into independent procedures where sequences of instructions are difficult to enforce.

ABS [15] is an active object-based language that targets modelling of distributed applications. The fragment of the ABS syntax regarding the concurrent object layer is shown on Figure 1. ABS has an object group model, like JCoBox, based on the notion of concurrent object group (hereafter COG). Asynchronous method calls and futures are explicit:

```
1 Fut<V> future = object!method();
```

Figure 2 pictures an ABS configuration with a request sending between COGs. Requests are scheduled in a cooperative manner thanks to the `await` keyword,

inspired from Creol and JCoBox and used as follows:

```
await future?;  await a > 2 && b < 3;
```

In those examples, the execution thread is released if the future is not resolved or if the condition is not fulfilled. ABS also features a `get` accessor to retrieve a future's value; it blocks the execution thread until the future is resolved:

```
V v = future.get;
```

The ABS tool suite[3] provides a wide variety of static verification engines that help designing safe distributed and concurrent applications. Those engines include a deadlock analyser [10], resource, cost, and deployment analysers for cloud environments [2, 17], and general program properties verification with the ABS-Key tool [8]. The ABS tool suite also includes a frontend compiler and several backend translators into various programming languages. The Java backend for ABS translates ABS programs into concurrent Java code that runs on a single machine. The Haskell backend for ABS [3] performs the translation into distributed Haskell code. The ABS semantics is preserved thanks to the thread continuation support of Haskell, which is not supported on the JVM.

ASP and ProActive. Asynchronous Sequential Processes (ASP) [6] is a mono-threaded active object programming language that has a non-uniform object model. In ASP, active objects are transparent to the programmer and futures are created and manipulated implicitly. A wait-by-necessity is triggered upon access to an unresolved future. Futures are first class: they are transparently passed and updated across activities. ProActive is the Java library that implements ASP. ProActive is a middleware that supports application deployment on distributed infrastructures such as clusters, grids and clouds. The program below creates explicitly an active object using `newActive` instead of `new`. The variable `v` stores an implicit future that is the result of a (transparent) asynchronous call.

```
T t = PAActiveObject.newActive(T.class, parameters, node);
V v = t.bar();
o.foo(v);   // does not block even if v is unresolved (o is any active or passive object)
v.foobar(); // blocks if v is unresolved
```

Recently, ProActive integrated multiactive objects [12] to enable multi-threaded request processing. MultiASP, presented in the next section, is an update of ASP and thus formalises the new version of ProActive. In practice, a programmer declares which requests of an active object can safely be executed in parallel, namely which requests are *compatible*, as shown in the following example:

```
@Group(name="group1", selfCompatible=true)
@Group(name="group2", selfCompatible=false)
@Compatible({"group1", "group2"})
public class MyClass {
  @MemberOf("group1")   public ... method1(...) { ... }
  @MemberOf("group2")   public ... method2(...) { ... }
}
```

In this example, a request for `method1` can be executed at the same time as a request for `method2`, but two requests for `method2` cannot be executed at the

[3] http://abs-models.org/

same time. With similar annotations, it is also possible to set a limit on the number of threads running in parallel [13]. The limit can be applied in two ways: a hard limit restrains the overall number of threads whereas a soft limit only counts threads that are not in wait-by-necessity.

Encore. Encore [5] is an active object-based parallel language currently in development. Encore features active and passive objects but even if passive objects are private by default, they can be shared at different scales depending on qualifying keywords. Asynchronous calls are transparent for active objects (by default) but futures are explicit, using a dedicated type. Finally, an active object has a single thread of execution by default, but parallelism is automatically created by attaching callbacks to future updates and using parallel combinators.

2.3 Positioning of this work

The reason why there are many different implementations of the active object programming model is to better fit particular objectives, from reasoning about programs to optimised program execution. Implementations that focus on the deployment of real-world systems comply to constraints related to existing execution platforms and languages. They are mostly used by programmers interested in the performance of the application. ProActive and Encore typically fit in this category. On the other side, some active object languages target verification and proof of programs, but have not been originally designed for efficient execution, like typically ABS and Creol. They are massively used and developed by academics and less constrained by existing execution platforms.

We give a proven translation of ABS programs into ProActive code in order to reconcile both domains: verified applications also have the right to be run efficiently. We also study the generalisation of our approach to other active object languages. Overall, our objective is to show that generic active object abstractions can be correctly encoded with different active object implementations.

3 Class-based Semantics of MultiASP

We start by introducing the semantics of MultiASP[4], the calculus representing ProActive and multiactive objects. Unlike the preliminary formalisation of multiactive objects in [12], we present here a class-based formalisation and the formalisation of threading policies. MultiASP is an imperative programming language and its syntax is close to the one of ABS.

Syntax of MultiASP. Figure 3 shows the static syntax of MultiASP. A program is made of classes and a main method. $\overline{x}$ denotes local variables in method bodies and object fields in class declarations. There are two ways to create an object: *new* creates a new object in the current activity, and *newActive* creates a new active object. $e.\mathtt{m}(\overline{e})$ is the generic method invocation, there is no syntactic distinction between local and remote (asynchronous) invocations. Similarly, as

[4] Formalised in Isabelle/HOL: `www-sop.inria.fr/members/Ludovic.Henrio/misc.html`

P	::= $\overline{C}\ \{\, \overline{x}\ ;\ \ s \,\}$	program
S	::= $\mathtt{m}(\overline{x})$	method signature
C	::= $\mathtt{class}\ \mathtt{C}(\overline{x})\ \{\, \overline{x}\ \overline{M} \,\}$	class
M	::= $S\{\, \overline{x}\ s \,\}$	method definition
s	::= $\mathtt{skip} \mid x = z \mid \mathtt{return}\ e \mid s\ ;\ s$	statement
z	::= $e \mid e.\mathtt{m}(\overline{e}) \mid \mathtt{new}\ \mathtt{C}(\overline{e}) \mid \mathtt{newActive}\ \mathtt{C}(\overline{e})$	expression with side effects
e	::= $v \mid x \mid \mathtt{this} \mid \textit{arithmetic-bool-exp}$	expression
v	::= $\mathtt{null} \mid \textit{primitive-val}$	value

Fig. 3: Class-based static syntax of MultiASP

v	::= $o \mid \alpha \mid \ldots$	*Storable*	::= $[\overrightarrow{x \mapsto v}] \mid v \mid f$
elem	::= $\textsc{fut}(f, v, \sigma) \mid \textsc{fut}(f, \bot) \mid \textsc{act}(\alpha, o, \sigma, p, Rq)$	σ	::= $\overrightarrow{o \mapsto \textit{Storable}}$
cn	::= $\overline{\textit{elem}}$	q	::= $(f, m, \overline{v})$
E	::= $\{\ell \mid s\}$	*Rq*	::= $\varnothing \mid q :: Rq$
F	::= $E \mid E :: F$	ℓ	::= $\mathtt{this} \mapsto v, \overrightarrow{x \mapsto v}$
p	::= $\overrightarrow{q \mapsto F}$	s	::= $x = \bullet \mid \ldots$

Fig. 4: Runtime syntax of MultiASP

synchronisation on futures is transparent and handled with wait-by-necessity, there is no particular syntax for interacting with a future. A special variable `this` exists for accessing the current object.

Semantics of MultiASP. MultiASP semantics is defined as a transition relation between configurations, noted *cn*, and for which the runtime syntax is displayed in Figure 4. At runtime, the dynamic configuration of a MultiASP program consists of a set of activities and a set of futures. The transition relation uses three infinite sets: *object locations* in the local store, ranged over by o, o', $\cdots$; *active objects names*, ranged over by α, β, $\cdots$; and *future names*, ranged over by f, f', $\cdots$. *Activities* are of the form $\textsc{act}(\alpha, o, \sigma, p, Rq)$ where α is an activity name; o is the location of the active object in σ; σ is a *local store* mapping object locations to storable values; p is a set of *requests currently served* (a mapping from requests to their thread F); and Rq is a FIFO *request queue* of requests awaiting to be served. A thread is a stack of methods being executed, and each *method execution* E consists of *local variables* ℓ and *statement* s to execute. The first method of the stack is the one that is executing, the others have been put in the stack due to local synchronous method calls. ℓ is a mapping from local variables (including `this`) to runtime values. A configuration also contains *future binders*. They are of two forms: $\textsc{fut}(f, \bot)$, meaning that the value for the future has not been computed yet, and $\textsc{fut}(f, v, \sigma)$, when the *reply value* is known; if it is an object (and not a static value), then v will be its location in the store σ.

An object o is fresh if it does not exist in the store in which it is added. Similarly, a future or an activity name is fresh if it does not exist in the current configuration. *Runtime values* ($v, \cdots$) can be either static values, object locations, or active object names. An object is a mapping from field names to their values, denoted $[\overrightarrow{x \mapsto v}]$. We denote mappings by $\overrightarrow{\_ \mapsto \_}$, and use union $\cup$ (resp. disjoint union $\uplus$) over mappings. Mapping updates are of the form $\sigma[x \mapsto v]$. *dom* returns the domain of a mapping. *Storable values* are objects, futures, or runtime values.

$$
\begin{array}{l}
\mathrm{serialise}(o,\sigma) = \\
\quad (o \mapsto \sigma(o)) \cup \mathrm{serialise}(\sigma(o),\sigma) \\
\mathrm{serialise}([\overrightarrow{x \mapsto v}],\sigma) = \\
\quad \bigcup_{v' \in \overline{v}} \mathrm{serialise}(v',\sigma) \\
\mathrm{serialise}(f,\sigma) = \\
\quad \mathrm{serialise}(\alpha,\sigma) = \\
\quad \mathrm{serialise}(\texttt{null},\sigma) = \varnothing \\
\mathrm{serialise}(\textit{primitive-val},\sigma) = \varnothing
\end{array}
$$

Fig. 5: Serialisation

$$
\begin{array}{rcll}
[\![\textit{primitive-val}]\!]_{(\sigma+\ell)} & \triangleq & \textit{primitive-val} & \\
[\![f]\!]_{(\sigma+\ell)} & \triangleq & \bot & \\
[\![\alpha]\!]_{(\sigma+\ell)} & \triangleq & \alpha & \\
[\![\texttt{null}]\!]_{(\sigma+\ell)} & \triangleq & \texttt{null} & \\
[\![x]\!]_{(\sigma+\ell)} & \triangleq & [\![\ell(x)]\!]_{(\sigma+\ell)} & \text{if } x \in \mathrm{dom}(\ell) \\
[\![x]\!]_{(\sigma+\ell)} & \triangleq & [\![\ell(\texttt{this})(x)]\!]_{(\sigma+\ell)} & \text{if } x \notin \mathrm{dom}(\ell) \\
[\![o]\!]_{(\sigma+\ell)} & \triangleq & o & \text{if } \sigma(o) = f \text{ or } \sigma(o) = [\overrightarrow{x \mapsto v}] \\
[\![o]\!]_{(\sigma+\ell)} & \triangleq & [\![\sigma(o)]\!]_{(\sigma+\ell)} & \text{else}
\end{array}
$$

Fig. 6: Evaluation function

The following auxiliary functions are used in the semantic rules: $[\![e]\!]_{(\sigma+\ell)}$ returns the value of e by computing the arithmetic and boolean expressions and by retrieving the values stored in σ or ℓ; the evaluation function is displayed in Figure 6. If the value of e is a reference to a location in the store, it follows references recursively; it only returns a location if the location points to an object or a future. $[\![\overline{e}]\!]_{(\sigma+\ell)}$ returns the tuple of values of $\overline{e}$. $\mathit{fields}(\texttt{C})$ returns fields as defined in the class declaration `C`. *bind* initialises method execution: $\mathrm{bind}(o,\texttt{m},\overline{v'}) = \{y \mapsto v', \overrightarrow{z \mapsto \texttt{null}}, \texttt{this} \mapsto o \mid s\}$, where the arguments of method `m`, typed in the class of o, are $\overline{y}$, and where the method body is $\{\overline{z}; s\}$. *ready* is a predicate deciding whether a request q in the queue Rq is ready to be served: $\mathit{ready}(q,p,Rq)$ is *true* if q is compatible with all requests in p (requests currently served by the activity) and with older requests in Rq. Serialisation reflects the communication style happening in Java RMI; it ensures that each activity has a single entry point: the active object. Consequently, all references to passive objects are serialised when communicated between activities, so that they are always handled locally. $\mathit{serialise}(o,\sigma)$ marks and copies the objects referenced from o to deeply serialise, recursively; it returns a new store made of all the objects that are referenced by o. *serialise* is defined as the mapping verifying the constraints of Figure 5. $\mathit{rename}_\sigma(\overline{v},\sigma')$ renames the object locations appearing in $\overline{v}$ and σ', making them disjoint from the object locations of σ; it returns a renamed set of values $\overline{v'}$ and a store σ''.

Figure 7 shows the part of MultiASP semantics that regards active object execution. Rules involving classical objects, namely object creation, field assignment, passive invocation, and local return of method call have been removed due to space limitation. The full MultiASP semantics can be found in the extended version of this paper in [14]. In all cases, rules only show activities and futures involved in the current reduction. SERVE picks the first request that is ready in the queue (compatible with executing requests and with older requests in the queue) and allocates a new thread to serve it. It fetches the method body and creates the execution context. ASSIGN-LOCAL assigns a value to a local variable. If the statement to be executed is an assignment of an expression that can be reduced to a value, then the mapping of local variables is updated accordingly. NEW-ACTIVE creates a new activity that contains a new active object. It picks a fresh activity name, and assigns serialised object parameters: the initial local store of the activity is the piece of store referenced by the parameters. INVK-ACTIVE

$$\textsc{Assign-Local}\quad \frac{x \in \mathrm{dom}(\ell) \qquad v = [\![e]\!]_{(\sigma+\ell)}}{\textsc{act}(\alpha, o_\alpha, \sigma, \{q \mapsto \{\ell \mid x = e; s\} :: F\} \uplus p, Rq) \to \textsc{act}(\alpha, o_\alpha, \sigma, \{q \mapsto \{\ell[x \mapsto v] \mid s\} :: F\} \uplus p, Rq)}$$

$$\textsc{Serve}\quad \frac{\mathrm{ready}(q, p, Rq) \qquad q = (f, m, \overline{v}) \qquad \mathrm{bind}(o_\alpha, m, \overline{v}) = \{\ell \mid s\}}{\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq :: q :: Rq') \to \textsc{act}(\alpha, o_\alpha, \sigma, \{q \mapsto \{\ell | s\}\} \uplus p, Rq :: Rq')}$$

$$\textsc{New-Active}\quad \frac{\mathit{fields}(\mathtt{C}) = \overline{x} \qquad o, \gamma \ \text{ fresh} \qquad \sigma' = \{o \mapsto [\overrightarrow{x = v}]\} \cup \mathrm{serialise}(\overline{v}, \sigma) \qquad [\![\overline{e}]\!]_{(\sigma+\ell)} = \overline{v}}{\begin{array}{c}\textsc{act}(\alpha, o_\alpha, \sigma, \{q \mapsto \{\ell \mid x = \mathtt{newActive\ C}(\overline{e}); s\} :: F\} \uplus p, Rq) \\ \to \textsc{act}(\alpha, o_\alpha, \sigma, \{q \mapsto \{\ell \mid x = \gamma; s\} :: F\} \uplus p, Rq) \ \ \textsc{act}(\gamma, o, \sigma', \varnothing, \varnothing)\end{array}}$$

$$\textsc{Invk-Active}\quad \frac{\begin{array}{c}[\![e]\!]_{(\sigma+\ell)} = \beta \qquad [\![\overline{e}]\!]_{(\sigma+\ell)} = \overline{v} \\ f, o \ \text{ fresh} \qquad \sigma_1 = \sigma \cup \{o \mapsto f\} \qquad (\overline{v_r}, \sigma_r) = \mathrm{rename}_{\sigma'}(\overline{v}, \mathrm{serialise}(\overline{v}, \sigma)) \qquad \sigma'' = \sigma' \cup \sigma_r\end{array}}{\begin{array}{c}\textsc{act}(\alpha, o_\alpha, \sigma, \{q \mapsto \{\ell \mid x = e.\mathtt{m}(\overline{e}); s\} :: F\} \uplus p, Rq) \ \textsc{act}(\beta, o_\beta, \sigma', p', Rq') \\ \to \textsc{act}(\alpha, o_\alpha, \sigma_1, \{q \mapsto \{\ell \mid x = o; s\} :: F\} \uplus p, Rq) \\ \textsc{act}(\beta, o_\beta, \sigma'', p', Rq' :: (f, m, \overline{v_r}))\textsc{fut}(f, \bot)\end{array}}$$

$$\textsc{Update}\quad \frac{\sigma(o) = f \qquad (v_r, \sigma_r) = \mathrm{rename}_\sigma(v, \sigma') \qquad \sigma'' = \sigma[o \mapsto v_r] \cup \sigma_r}{\textsc{act}(\alpha, o_\alpha, \sigma, p, Rq) \ \textsc{fut}(f, v, \sigma') \to \textsc{act}(\alpha, o_\alpha, \sigma'', p, Rq) \ \ \textsc{fut}(f, v, \sigma')}$$

$$\textsc{Return}\quad \frac{v = [\![e]\!]_{(\sigma+\ell)}}{\begin{array}{c}\textsc{act}(\alpha, o_\alpha, \sigma, \{(f, m, \overline{v}) \mapsto \{\ell \mid \mathtt{return}\ e; s_r\}\} \uplus p, Rq) \ \textsc{fut}(f, \bot) \\ \to \textsc{act}(\alpha, o_\alpha, \sigma, p, Rq) \ \ \textsc{fut}(f, v, \mathrm{serialise}(v, \sigma))\end{array}}$$

Fig. 7: Semantics of MultiASP

performs an asynchronous remote method invocation on an active object. It creates a fresh future with undefined value. The arguments of the invocation are serialised and put in the store of the invoked activity, possibly renaming locations to avoid clashes. The special case $\alpha = \beta$ requires a trivial adaptation of this rule (not shown here). Return is triggered when a request finishes. It stores the value computed by the request as a future value. Serialisation is necessary to pack the objects referenced by the future value. Update updates a future reference with a resolved value. This is performed at any time when a future is referenced and the future value is resolved. Finally, the main effect of the missing rules is to modify the local store (New-Object and Assign-Field) and to affect the execution context (Invk-Passive and Return-Local).

Threading Policies. We extend the above semantics to specify the threading policies featured in multiactive objects (see Section 2.2). First, we extend the syntax of MultiASP so that the threading policy can be programmatically changed from a *soft limit*, i.e. a thread blocked in a wait-by-necessity is not counted in the limit, to a *hard limit*, i.e. all threads are counted in the limit:

$$s ::= \ldots \mid \textbf{setLimitSoft} \mid \textbf{setLimitHard}$$

Each request q belongs to a group $group(q)$. The filter $p\big|_g$ gives, among the active threads p, only requests of group g. There is a thread limit $\mathcal{L}_g$ defined for each group. We tag each of the currently served request as either *active* or *passive*. p contains then two kinds of served requests: active ones, noted $q_A \mapsto F$, and passive ones, noted $q_P \mapsto F$. Active(p) returns the number of active requests in p. Finally, each activity is either in a *soft limit* state written $\textsc{act}(\ldots)_S$ (by default at activity creation), or in a *hard limit* state written $\textsc{act}(\ldots)_H$. *sh* is a variable ranging over S and H. MultiASP semantics is modified as follows:

- Each rule allowing a thread to progress requires now that the thread is active, i.e. q is replaced by q_A in all rules except Serve and Update.
- The rule Serve is only triggered if the thread limit is not reached, i.e. if $\text{Active}(p\big|_{group(q)}) < \mathcal{L}_g$. Simmilarly, a rule for activating a thread is added:

$$\textsc{Activate-Thread}\quad \frac{Group(q) = g \qquad \text{Active}(p\big|_g) < \mathcal{L}_g}{\textsc{act}(\alpha, o_\alpha, \sigma, \{q_P \mapsto F\} \uplus p, Rq)_{sh} \to \textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto F\} \uplus p, Rq)_{sh}}$$

- There are two additional rules for switching the kind of limit, we show one hereafter (Set-Soft-Limit is the reverse):

$$\textsc{Set-Hard-Limit}\quad \textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell \mid \textbf{setLimitHard}; s\} :: F\} \uplus p, Rq)_{sh} \to \textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell \mid s\} :: F\} \uplus p, Rq)_H$$

- If the kind of limit is a *soft limit*, a wait-by-necessity passivates the current thread[5]; a rule for method invocation on a future is added:

$$\textsc{Invk-Future}\quad \frac{[\![e]\!]_{(\sigma+\ell)} = o \qquad \sigma(o) = f}{\textsc{act}(\alpha, o_\alpha, \sigma, \{q_A \mapsto \{\ell \mid x = e.\mathtt{m}(\overline{e}); s\} :: F\} \uplus p, Rq)_S \to \textsc{act}(\alpha, o_\alpha, \sigma, \{q_P \mapsto \{\ell \mid x = e.\mathtt{m}(\overline{e}); s\} :: F\} \uplus p, Rq)_S}$$

4 Example-Driven Translation Principles

In this section, we informally present the ProActive backend for ABS, that translates ABS programs into ProActive code. Basically, this section shows how the formal translation that will be defined in Section 5 is instantiated in practice in ProActive. This backend is based on the existing Java backend for ABS. We keep the translation of the functional layer unchanged and provide a translation of the object and concurrency layers.

Object Addressing and Invocation. To handle the differences between two active object languages, one needs first to define what happens when a new object (active or not) is created. As translating each ABS object into a ProActive active object is not a viable solution (because it is not scalable and because it

[5] Wait-by-necessity occurs only in case of method invocation on a future since field access is only allowed on the current object.

requires a complex synchronisation of processes), we put several objects under the control of one active object, which fits the active object group model of ABS. To this end, in the translation, we introduce a class `COG` for representing ABS COGs; only objects of the `COG` class are active objects in the ProActive translation. We translate the ABS `new` statement that creates a new object in a new COG:

```
1 Server server = new Server();
```

This instruction is translated into ProActive by the ProActive backend:

```
1 Server server = new Server();
2 COG cog = PAActiveObject.newActive(COG.class, new Object[]{Server.class}, node);
3 server.setCog(cog);
4 cog.registerObject(server);
```

Line 1 creates a regular server object. Lines 2 uses the `newActive` ProActive primitive to create a new COG active object. Additionally to the constructor parameters, ProActive allows the specification of the node onto which the active object is deployed. Line 3 makes the local server aware of its COG. Finally in line 4, due to the ProActive by-copy parameter passing, the server object is copied in the local memory space of the newly created remote COG, and is thus locally accessible there. For objects created with `new local` in ABS, the ProActive backend simply registers them locally in the current COG. To enable the same object invocation model as in ABS, we use a two-level reference system in the ProActive translation: each COG is accessible by a global reference and each translated ABS object is accessible inside its COG through a local identifier. The pair (COG, identifier) is a unique reference for each object and allows the runtime to retrieve any object. When objects are transmitted between COG (e.g. as parameter of method invocations), a lightweight copy is transmitted by the ProActive middleware; it can be used to reach the original object by using its COG and identifier. As only the COG and the identifier are needed to reference an ABS object, we tune the object serialisation mechanism so that only those fields are transmitted between active objects, thus saving memory and bandwidth. The same strategy can be applied to translate any language featuring active object groups into non uniform active objects. For uniform active objects, creating one active object per translated object handles straightforwardly the translation but limits scalability; grouping several objects behind a same active object (proxy) would produce a more efficient program.

In order to explain now how we translate ABS asynchronous method calls in ProActive, consider the following ABS asynchronous method call:

```
1 server!start(param1, param2);
```

In ProActive such a call becomes a remote method invocation. In order to handle it with our object translation model, we perform a generic method call (implicitly asynchronous) named `execute`, on the COG of the translated `server` object:

```
1 server.getCog().execute(server.getId(), "start", new Object[]{param1, param2});
```

When run, the `execute` method of the `COG` class retrieves the target object through its identifier and runs the `start` method on it by reflection with the given

parameters. Upon `execute` remote call, objects `param1` and `param2` are copied to the memory space of the retrieved COG. Consequently, two copies of `param1` and `param2` exist in the translation whereas only one of them exists in ABS. However, if method calls occurs on them, the requests for those objects always go to the COG that manages those objects. This callback ensures that only one copy of a translated object is manipulated, like in ABS. Consequently, the behaviour by reference of ABS-like languages can be simulated with the behaviour by copy of ProActive. This mechanism is also applied for future updates.

Cooperative Scheduling. Active object languages often support special threading models and have constructs to impact on the scheduling of requests. Those constructs can be translated into adequate request scheduling of multiactive objects. For demonstration, we consider here the translation that the ProActive backend gives for ABS `await` statements (representative of cooperative scheduling), and for ABS `get` statements (representative of explicit futures).

- `await` statements on futures. An await statement on an unresolved futures releases the execution thread, for example:

```
await startedFut?;
```

In order to have the same behavior in the ProActive translation, we force a wait-by-necessity. We use the `getFutureValue` ProActive primitive to do that:

```
PAFuture.getFutureValue(startedFut);
```

As in ProActive a wait-by-necessity blocks the thread, we need to configure the ProActive COG class with multiactive object annotations (see Section 2.2) in order to qualify the `execute` method and to specify a soft thread limit:

```
@Group(name="scheduling", selfCompatible=true)
@DefineThreadConfig(threadPoolSize=1, hardLimit=false)
public class COG {
  ...
  @MemberOf("scheduling")
  public ABSValue execute(UUID objectID, String methodName, Object[] args) {...}
}
```

This configuration allows a thread to process an `execute` request while a current thread that processes another `execute` request is waiting for a future. Indeed, the `hardLimit=false` parameter ensures that the threads counted in the limit (of 1 thread) are only *active* threads. In the example, the thread can be handed over to another `execute` request if `startedFut` is not resolved, just like in ABS.

- `get` statements. The ABS `get` statement blocks the execution thread to retrieve a future's value, as for example on the previous future variable:

```
Bool started = startedFut.get;
```

The ProActive backend translates this ABS instruction into the following code:

```
getCog().switchHardLimit(true); // the retrieved COG is local: the call is synchronous
PAFuture.getFutureValue(startedFut);
getCog().switchHardLimit(false);
```

This temporarily hardens the threading policy (i.e. all threads are counted in the thread limit) so that no other thread can start while the future is awaited.

- Other synchronisation constructs. We also tackled the translation of ABS `suspend` statements and of `await` statements on conditions. In this paper, we only provide the formal definition of their translation in Section 5. The details of their translation into ProActive code can be found in [19].

Wrap Up and Applicability. In order to finalise the ProActive backend for ABS, we add deployment information in the translation; for that we use the deployment descriptor embedded in ProActive: configuration files binding virtual nodes to physical machines. On the ABS side, `new cog` is followed by the name of a node for deployment. This is the *only* modification that ABS programs must incur to be executed in a distributed way. An experimental evaluation (detailed in the extended version of this paper in [14]) shows that a significant speedup can be achieved by a distributed execution of an ABS program thanks to the ProActive backend. It also shows that the program obtained with the ProActive backend incurs an overhead of less than 10% compared to a native ProActive application.

We have presented in details the ProActive backend for ABS and discussed the translation of common active object constructs. The concepts applied in the case of ABS are generic and can systematically turn various active object languages into deployable active objects. As an example, JCoBox is similar enough to ABS so that the approach presented here is straightforwardly applicable. The most challenging aspect is that JCoBox features a globally accessible and immutable memory, which could be translated into one active object, or which could rely on copies since the immutable property holds. Regarding Creol, in which all objects are active, the best approach is to group several objects behind a same proxy for performance reasons. Then, preserving the semantics of Creol relies on a precise interleaving of local threads. The transposition to AmbientTalk is trickier on the scheduling aspect, due to the existence of callbacks. However, we found that a callback on a future can be translated as a request that is ready to run but that starts by a wait-by-necessity on the adequate future.

5 Translational Semantics

This section formalises the translation given by the ProActive backend by introducing the translational semantics from ABS to MultiASP. We refer to Figure 1 for the concurrent object layer of ABS. Runtime syntax and semantics of ABS can be found in [14]. Most of the translation from ABS to MultiASP impacts statements. The rest of the source structure (classes, interfaces, methods) is unchanged except the two following:

1) We define a new class `COG`. It has methods to store and retrieve local objects, and to execute a method on a local object; `UUID` is the type of object identifiers:

```
Class COG {
  UUID freshID()
  UUID register(Object x, UUID id)
  Object retrieve(UUID id)
  Object execute(UUID id, MethodName m, params) { \\
    w=this.retrieve(id); x=w.m(params); return x}
}
```

2) All translated ABS classes are extended with two parameters: a *cog* parameter, storing the COG to which the object belongs, and an *id* parameter, storing the object's identifier in that COG; methods *cog*() and *myId*() return those two parameters; a dummy method *get*() that returns `null` is added to each object.

The translation of statements and expressions is shown in Figure 8. Each of them is explained below. *Object instantiation* first gets a fresh identifier from the current COG. Then, the new object is created with the current COG and the identifier[6]. It is stored in a reserved temporary local variable *no*. Finally, the object is referenced in the current COG and stored in x. *Object instantiation in a new* COG is similar to object instantiation in the current COG but method invocations on *newcog* variable are asynchronous remote method calls. The new object is thus copied to the memory space of the remote new COG via the *register* invocation, before being assigned to x. *Await future* uses the dummy *get()* method, that all translated objects have, in order to trigger the wait-by-necessity mechanism and potentially block the thread if the future is not resolved. *Get future* sets a hard limit on the current activity, so that no other thread starts, and then restores the soft limit after having waited for the future. *Await on conditions* performs sequential *get()* within an activity in soft limit. Conditional guards are detailed later in this section. *Asynchronous method call* retrieves the COG of the object and relies on the *execute* asynchronous method call as described in Section 4. *Synchronous local method call* distinguishes two cases, like in ABS. Either the call is local and an execution context is pushed in the stack, or the call is remote and, like in ABS, we perform an asynchronous remote method invocation and immediately wait the associated future within an activity in hard limit. Finally, instructions that do not deal with method invocation, future manipulation, or object creation, are kept unchanged.

In the translation, there exist different multiactive object groups and each group has its own thread limit. Group g_1 encapsulates $freshId$ requests; those requests cannot execute in parallel safely, so g_1 is not self compatible and can only use one thread at a time. Group g_2 gathers *execute* requests. It is limited to one thread to comply with the threading model of ABS, and the requests are self compatible to enable interleaving. Group g_3 contains $register$ requests that are self compatible and that have an infinite thread limit. Concerning compatibility between groups, they are all compatible except g_3 and g_2: their compatibility is defined dynamically such that an $execute$ request and a $register$ request are compatible only if they do not affect the same identifier. In summary:

$$group(freshId) = g_1 \qquad group(execute) = g_2 \qquad group(register) = g_3$$
$$\mathcal{L}_{g_1} = 1 \qquad \mathcal{L}_{g_2} = 1 \qquad \mathcal{L}_{g_3} = \infty$$
$$\forall q, q'.\ (q \neq q' \neq freshId() \wedge (\nexists id. q = register(x, id) \wedge q' = execute(id, m, \overline{e}))) \Rightarrow compatible(q, q')$$

In order to support ABS conditional guards, for each guard g, we generate a method *condition_g* that takes as parameters the needed local variables $\overline{x}$. The

[6] The step in which the COG of the new object is set in ProActive is directly encoded in the object constructor in MultiASP.

$[\![x = e!m(\overline{e})]\!] \triangleq t = e.cog(); id = e.myId(); x = t.execute(id, m, \overline{e})$

$[\![await\ x?]\!] \triangleq w = x.get()$

$[\![await\ g \wedge g']\!] \triangleq [\![await\ g]\!]; [\![await\ g']\!]$

$[\![x = y.get]\!] \triangleq$ **setLimitHard**; $w = y.get()$; **setLimitSoft**; $x = y$

$[\![x = e.m(\overline{e})]\!] \triangleq a = e.cog(); b = \texttt{this}.cog();$ `if`$(a == b)\ \{x = e.m(\overline{e})\}$ `else` $\{t = e.cog(); id = e.myId(); x = t.execute(id, m, \overline{e});$ **setLimitHard**; $w = x.get();$ **setLimitSoft**$\}$

$[\![x = e]\!] \triangleq x = e$

$[\![x = \texttt{new local}\ C(\overline{e})]\!] \triangleq t = \texttt{this}.cog(); id = t.freshId(); no = \texttt{new}\ C(\overline{e}, t, id); z = t.register(no, id); x = no$

$[\![x = \texttt{new}\ C(\overline{e})]\!] \triangleq newcog = \texttt{newActive}\ COG(); id = newcog.freshId(); no = \texttt{new}\ C(\overline{e}, newcog, id); z = newcog.register(no, id); x = no$

$[\![await\ g]\!]_{\overline{x}} \triangleq$ `if`$(\neg g)\ \{\ t = \texttt{this}.cog();\ id = \texttt{this}.myId();\ z = t.execute\_condition(id, condition\_g, \overline{x}); w = z.get\ \}$

$[\![suspend]\!] \triangleq t = \texttt{this}.cog();\ id = \texttt{this}.myId();\ z = t.execute\_condition(id, condition\_True, \overline{x}); w = z.get$

Fig. 8: Translational semantics from ABS to MultiASP

method body can normally access the fields of the object `this`. A condition evaluation g is defined as follows: $condition\_g(\overline{x}) = \texttt{while}(\neg g)\ skip; \texttt{return null}$. We encode the suspend statement the same way with a *True* condition. We define an *execute_condition* method in the `COG` class; it executes generated condition methods. The *execute_condition* method has its own group with an infinite thread limit because any number of conditions can evaluate in parallel. More formally, we have: $group(execute\_condition) = g_4 \quad \mathcal{L}_{g_4} = \infty$

6 Translation Equivalence and Active Object Insights

Proving that MultiASP executions exactly simulate ABS semantics is not possible by direct bisimulation of the two semantics. Instead, we prove two different theorems stating under which conditions each semantics simulates the other. We present all technical details on the equivalence and the proof in the research report associated to this paper [14]. We summarise below the highlights of the proof, the principles of the underlying equivalence between MultiASP and ABS terms, the differences between the languages and the restrictions of the proof.

Communication and request serving ordering. The semantics of ABS relies on a completely asynchronous communication scheme while MultiASP ensures causal ordering of requests. The equivalence can only be valid for the ABS reductions that preserve causal ordering of requests. Also, MultiASP serves requests in FIFO order, so similarly we execute a FIFO service of ABS requests, like in the existing Java backend for ABS. Note that those differences are more related to scheduling and communication patterns than to the nature of the two languages.

Shallow translation. ABS requests, COGs and futures respectively match one-to-one MultiASP requests, active objects and futures. Likewise, except for COG

objects, for each ABS object there exist several copies of this object in MultiASP, all with the same COG and the same identifier, but only one of those copies (the one hosted in the right COG) is equivalent to the ABS object.

Futures. Because of the difference between the future update mechanisms of ABS and MultiASP, the equivalence relation can follow as many local future indirections in the store as necessary. A variable holding a pointer to a future object in MultiASP is equivalent to the same variable holding directly the future reference in ABS. But also, the equivalence can follow future references in ABS: a future might have been updated transparently in MultiASP while in ABS, the explicit future read has not been performed yet.

Equating MultiASP *and ABS configurations.* A crucial part of the correctness proof consists in stating whether an ABS and a MultiASP configuration are considered equivalent. The principles of this equivalence are the following:

- Equivalence can "follow futures": A MultiASP value v is equivalent to an ABS future provided the future's value is equivalent to v; indeed in MultiASP a future can be automatically updated earlier than in the ABS case.
- Objects are identified by their identifier and their COG name: the value of the object fields are meaningless except in the COG that initially created the object. It is in this COG that we check that fields are equivalent.
- Equivalence between requests distinguishes two cases. 1) active tasks: there is a single active task per COG in ABS and it must correspond to the single active thread serving an *execute* request in MultiASP. The second element in the call stack corresponds to the invoked request. 2) inactive tasks in ABS correspond either to passive requests being currently interrupted or to not-yet-served requests in MultiASP. For each task, equivalence of executed statements, of local variables, and of corresponding future is checked.

Observational equivalence. The precise formulation of our theorems proves that the ABS behaviour is faithfully simulated by our translation and conversely. This is proven by adequately choosing the *observable* and *not observable* actions in the weak simulation. For example remote method invocation, object creation, and field assignment can be observed and faithfully simulated. The most striking observable reduction in ABS that is not always observable in MultiASP is the future value update. For example, in ABS the configurations (a) $\mathit{fut}(f, f')\ \mathit{fut}(f', \bot)$ and the configuration (b) $\mathit{fut}(f, \bot)$ are observationally different, whereas in MultiASP they are not. Indeed, in MultiASP, there is no process able to detect whether the first future has been updated or not. However, this example is artificial as no information is stored in the first future of configuration (a); any access to the future's value will have to follow indirections and eventually access the value that is not a future. Thus, transparency of futures and of future updates create an intrinsic difference between the two languages. This is why, in the theorem, we exclude the possibility to have a future's value being a future in the configuration. Eliminating syntactically such programs is not possible, thus we reason on reductions for which the value of a future is not a future; this is not a major restriction on expressiveness because it is still possible to have a future value that is an object containing a future (as future wrappers).

In the other direction, namely from MultiASP to ABS, the translation adds several steps in the reduction. However, the added sequences of actions never introduce concurrency so equivalence still holds because we can ignore additional local actions such as assignments and method calls that are not in the ABS program source (e.g. *myId()*).

Theorem 1 (ABS to MultiASP**).** *The translation simulates all ABS executions with FIFO policy and rendez-vous communications provided that no future value is a future reference.*

Theorem 2 (MultiASP **to ABS).** *Any reduction of the* MultiASP *translation corresponds to a valid ABS execution.*

Globally, our translational semantics fully respects the ABS semantics and simulates exactly all executions complying to the aforementioned restrictions, which either are already existing restrictions of the Java backend for ABS, or for which we have given relevant alternatives.

7 Conclusion

This paper tackled the question of providing active object languages, aimed at modelling and verification, with systematic deployment for distributed computing. For that, we have identified the necessary design choices for active object models and languages, involving: object referencing, language transparency, and request scheduling. These design choices have to be considered when implementing any active object language. We have introduced MultiASP, a multi-threaded active object language that has showed to be expressive enough to embody the main paradigms of ABS, featuring in particular cooperative scheduling. We demonstrated how to translate the constructs of an easy to program and verify active object language into the executable code of an efficient and scalable active object middleware. We have instantiated our approach by translating ABS into the ProActive middleware, that implements MultiASP in Java. The immediate outcome of this work is a ProActive backend for ABS. Our approach could be quite easily ported other active object languages since we reason more on active object abstractions than on language specifics. Typically, our work can be straightforwardly adapted to any active object language featuring cooperative scheduling, like Creol and JCoBox. Porting our results on AmbientTalk only requires minor adaptations. A comparison of the ProActive backend against a currently developed Java 8 backend for ABS [21] is ongoing. This analysis focuses on the different implementation approaches for efficiently encoding the ABS semantics. More generally, the provided proof of correctness highlighted the intrinsic differences between active object languages and models. This work will help active object users to choose the language that is the most adapted for their needs, and also help active object designers to identify the implication of specific language constructs and abstractions.

References

1. G. Agha and C. Hewitt. Concurrent programming using actors. In *Foundations of Software Technology and Theoretical Computer Science*. Springer, 1985.
2. E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: Static analyzer for concurrent objects. In *TACAS'14*. Springer, 2014.
3. N. Bezirgiannis and F. Boer. *SOFSEM 2016*, chapter ABS: A High-Level Modeling Language for Cloud-Aware Programming. Springer, Berlin, Heidelberg, 2016.
4. F. S. D. Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *ESOP'07*. Springer, 2007.
5. S. Brandauer, E. Castegren, D. Clarke, K. Fernandez-Reyes, E. Johnsen, K. Pun, S. Tarifa, T. Wrigstad, and A. Yang. Parallel objects for multicores: A glimpse at the parallel language Encore. In *Formal Methods for Multicore Programming*, LNCS. Springer, 2015.
6. D. Caromel and L. Henrio. *A Theory of Distributed Objects*. Springer, 2004.
7. J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-oriented programming in ambienttalk. In *ECOOP'06*. Springer, 2006.
8. C. Din, R. Bubel, and R. Hahnle. KeY-ABS: A deductive verification tool for the concurrent modelling language abs. In A. P. Felty and A. Middeldorp, editors, *Automated Deduction - CADE-25*, LNCS. Springer, 2015.
9. C. Flanagan and M. Felleisen. The semantics of future and its use in program optimization. In *POPL '95*. ACM, 1995.
10. E. Giachino, C. Laneve, and M. Lienhardt. A framework for deadlock detection in ABS. *Journal of Software and Systems Modeling*, 2014.
11. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, Feb. 2009.
12. L. Henrio, F. Huet, and Z. István. Multi-threaded active objects. In *COORDINATION'13*. Springer, 2013.
13. L. Henrio and J. Rochas. Declarative Scheduling for Active Objects. In *SAC'14*. ACM, 2014.
14. L. Henrio and J. Rochas. From Modelling to Systematic Deployment of Distributed Active Objects – Extended Version. Research report, I3S, Apr. 2016.
15. E. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *FMCO'12*. Springer, 2012.
16. E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 2006.
17. E. B. Johnsen, R. Schlatte, and S. L. T. Tarifa. Integrating deployment architectures and resource consumption in timed object-oriented models. *Journal of Logical and Algebraic Methods in Programming*, 2015.
18. R. G. Lavender and D. C. Schmidt. Active object: an object behavioral pattern for concurrent programming. In *Pattern languages of program design 2*. 1996.
19. J. Rochas and L. Henrio. A ProActive Backend for ABS: from Modelling to Deployment. Research Report RR-8596, Sept. 2014.
20. J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *ECOOP'10*. Springer, 2010.
21. V. Serbanescu, K. Azadbakht, F. de Boer, C. Nagarajagowda, and B. Nobakht. A design pattern for optimizations in data intensive applications using ABS and JAVA 8. *Concurrency and Computation: Practice and Experience*, 2016.