A Survey of Active Object Languages

FRANK DE BOER and VLAD SERBANESCU, Centrum Wiskunde and Informatica, The Nederlands
REINER HÄHNLE, TU Darmstadt , Germany
LUDOVIC HENRIO and JUSTINE ROCHAS, Université Côte d'Azur, CNRS, I3S, France
CRYSTAL CHANG DIN and EINAR BROCH JOHNSEN, University of Oslo , Norway
MARJAN SIRJANI, Reykjavik University, Iceland
EHSAN KHAMESPANAH, University of Tehran, Iran
KIKO FERNANDEZ-REYES and ALBERT MINGKUN YANG, Uppsala University, Sweden

To program parallel systems efficiently and easily, a wide range of programming models have been proposed, each with different choices concerning synchronization and communication between parallel entities. Among them, the actor model is based on loosely coupled parallel entities that communicate by means of asynchronous messages and mailboxes. Some actor languages provide a strong integration with object-oriented concepts; these are often called active object languages. This paper reviews four major actor and active object languages and compares them according to carefully chosen dimensions that cover central aspects of the programming paradigms and their implementation.

CCS Concepts: • **Software and its engineering → Parallel programming languages**; **Concurrent programming structures**;

Additional Key Words and Phrases: Programming languages, active objects, actors, concurrency, distributed systems

ACM Reference Format:
Frank De Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. 2017. A Survey of Active Object Languages. *ACM Comput. Surv.* 50, 5, Article 76 (October 2017), 38 pages.
https://doi.org/10.1145/3122848

1 INTRODUCTION

Designing a programming language for concurrent systems is a difficult task. The programming abstractions provided in concurrent programming libraries are often rather low-level and difficult to reason about, both from the point of view of the programmer and for verification tools. This article focuses on one family of concurrent programming languages that tries to mitigate this problem: active object languages. The first and most important characteristic of this family of languages is inherited from the actor paradigm. It relies on organizing the program in terms of well-defined entities called actors. Each actor is an independent entity that behaves autonomously and that communicates with other actors through asynchronous messages. Actors execute concurrently and communicate asynchronously, without transfer of control. This makes the actor model attractive [3, 8]: Using the actor model, information sharing is simplified, programming is less error prone,

0360-0300/2017/10-ART76 $15.00
https://doi.org/10.1145/3122848

analysis becomes much easier, and programs scale better to parallel and distributed architectures because control threads are implicit rather than explicit in the programs.

Within actor-based languages we focus in this survey on one specific category that uses asynchronous method calls as its communication paradigm. Languages in this category are often called *active object* languages. Whereas messages sent to an actor are generally selected by pattern matching over the message queue, asynchronous method calls restrict the communication between active objects to messages which trigger method activations. Consequently, messages to an actor can be indefinitely ignored if the actor never switches to an appropriate state, whereas it can be statically guaranteed that asynchronous method calls to an active object are understood. The basic unit of concurrency is the same in all actor languages, and the active object languages have the same attractive features as actor languages in general, but the way methods are called seems to pose a rather subtle distinction. Although this merely causes minor differences regarding the expressiveness of the different languages, it causes considerable differences in terms of the abstractions offered to the programmer. By supporting *communication by asynchronous method calls*, the programming model of active object languages is tightly integrated with the *object-oriented* programming paradigm and its programming abstractions, which are familiar to most programmers.

We start this survey by giving the historical context that led to the existence of active object languages and that justifies their importance. We then focus on four languages that are representative of this language family. We start with Rebeca, which is closest to the classical actor model [3], but nevertheless provides communication by a form of asynchronous method invocation. The three other languages—ABS, ASP, and Encore—are precise representatives of active object frameworks. Additionally, each of them implements replies to asynchronous method calls in the form of *futures* (see Section 2.1), which enables advanced synchronization between the active objects.

The languages presented here come with strong support for formal methods and verification techniques; Rebeca and ABS even make this their main focus, having been conceived as modeling languages: actors do not merely offer a convenient programming abstraction but also greatly simplify static program analysis by naturally structuring programs in entities that can serve as the basis for compositional analyses. ASP and Encore instead put the focus on highly optimized, efficient implementations. Whereas these languages all fit inside the category of active object languages, they propose different solutions for how they support synchronization and for how much internal concurrency they allow inside active objects. The languages are surveyed with respect to their objectives, level of formalization, and their implementation and tooling support. Furthermore, they are compared with respect to a number of *dimensions* in the design space of active object languages: the degree of synchronization, the degree of transparency, and the degree of data sharing.

This paper is organized as follows. Section 2 presents an overview of the actor and active object languages; it first adopts a historical perspective, and then focuses on the four chosen languages, justifying their choice, presenting their design principles, the approach they advocate, and the tools they offer. Section 3 focuses on the implementation and runtime support for the presented languages. Section 4 discusses design aspects of active object and related languages, and Section 5 concludes the paper.

2 ACTIVE OBJECT LANGUAGES

2.1 A Brief History of Actor and Active Object Languages

The development of high-level programming languages is generally driven by the quest for suitable abstractions of the low-level implementation details of specific underlying hardware architectures. A major challenge in the development of such languages is the high-level specification of concurrent

and parallel threads of control and their synchronization. In process calculi like CSP [57] and CCS [75] independent threads of control communicate data synchronously, e.g., via channels. The Ada programming language [10] extends this model of synchronous communication by a rendez-vous interpretation of the execution of procedure calls and returns. In the shared-variable model of concurrency, threads of control communicate and synchronize directly via shared data. For example, the popular object-oriented programming language Java extends the sequential thread of control of Simula and Smalltalk, in which a thread consists of a stack of procedure (method) calls, from one to multiple threads in a manner resembling operating systems [44].

The shared-variable model of concurrency underlying the Java language requires that the developer ensures so-called *thread safety*; insufficient synchronization may cause data races between threads, rendering program behavior "unpredictable and sometimes surprising" [44]. To avoid data races, complex low-level synchronization mechanisms are used, which in turn may give rise to deadlocks and thread starvation. The result of executing multi-threaded programs depends on an intricate and fine-grained granularity of interleaving. This makes their behavior difficult to understand and to verify. In addition, the multi-threaded concurrency model decouples data (represented statically by classes) and control (represented at runtime by threads), which is unintuitive and difficult to use in a distributed setting.

The Actor model of computation [3, 55] departs from the synchronous communication mechanisms described above and is based on loosely coupled processes which interact via asynchronous message passing. Actors integrate and encapsulate both data and a single thread of control, and communicate without transfer of control. Message delivery is guaranteed but messages may arrive out of order. This loose coupling makes Actor languages conceptually attractive for parallel and distributed programming; in fact, the Actor model originated from the need "to efficiently run the coming generation of PLANNER-like artificial intelligence languages including those requiring a high degree of parallelism" [55]. Erlang [8] is one of the most successful actor languages, especially because of its adoption by programmers in the industry, and because of its massively parallel execution model supporting a huge number of independent actors.

In the original Actor models mentioned above, the interpretation and processing of messages is confined to and part of the overall internal behavior of an actor. However, this does not allow a clear separation of concerns between communication and computation, and as such does not support a "programming to interfaces" discipline [72]. A major advance in the development of high-level programming languages is the compositional construction of programs in terms of modules and their interfaces, which gave rise to the object-oriented programming paradigm. Modules are used to model abstract data types and as such provide a powerful abstraction which fully integrates data and control. In the object-oriented paradigm, modules are generalized to classes. Classes can be characterized as modules from which objects can be dynamically instantiated and referred to by unique identifiers (generated at runtime). The concepts of classes and objects were introduced in Simula 67 [34], the first object-oriented language. Whereas Simula was a sequential language with a single thread of control, it was developed to simulate real-world (concurrent) systems and already featured the concept of *coroutines* [33].

Active (or concurrent) object languages integrate the basic Actor model with object-oriented concepts and as such do support interface abstractions of implementation details. This integration was first proposed in ABCL [93]. It can be realized by modeling a message as an asynchronous method call, thus fixing the interpretation of a message by a corresponding method definition. Returning values from a method call can be modeled by another asynchronous message, as done in the actor-based language Rebeca [86, 88],

The notion of method calls can be further integrated by directly supporting return values while maintaining a loose coupling between actors; this leads to the notion of future variables. *Futures*

were originally discovered by Baker and Hewitt in the late 1970s [9], later rediscovered by Liskov and Shrira as promises in Argus [70], by Halstead in MultiLisp [47], before finding their way into object-oriented languages such as ConcurrentSmalltalk [92], ABCL [93], Eiffel// [23], and CJava [32].

A future can be pictured as a mailbox that will eventually contain the return value from a method call. Hence, the calling method may proceed with its computation and pick up the reply later. This, however, requires additional synchronization by means of a blocking get-operation on the future. In Creol [62, 63] such a synchronization mechanism is combined with cooperative scheduling of the method activations of the active object, using an explicit statement to release control. This allows coroutine-like method call execution inside an active object which encapsulates a single thread of control. Cooperative scheduling permits a compositional proof theory [35].

Java's multi-threaded concurrency model can be integrated with active objects on the basis of cooperative scheduling. In the Abstract Behavioral Specification (ABS) language [61], Creol's cooperative scheduling is combined with the structuring mechanism of concurrent object groups [82], originally developed for Java. A group of active objects has a thread pool with threads generated by (external) asynchronous method calls and extended as a stack of (internal) synchronous method calls. Within a pool, at most one thread is executing at any time and the granularity of interleaving is explicitly controlled in the code by means of the cooperative scheduling statements.

Caromel et al. [2004] designed ASP, an imperative, asynchronous object calculus with transparent futures. Their active objects may have internal passive objects which are passed between active objects by deep copying the entire (passive) object graph. Unnecessary copying can be avoided with ownership type systems [27]. To limit the complexity of reasoning in a distributed and concurrent setting, ASP is restricted to ensure that reduction is confluent and deterministic. ASP constitutes the theoretical foundation for the ProActive language [21].

Like ASP, the Encore programming language [16] integrates active objects into an object-oriented model and distinguishes among active and passive objects (such that passive objects are owned by active ones). Unlike ASP, passive objects are data race-free and may be passed by reference, due to a capability type system [24, 25].

Selection of representative languages. In the remaining article we discuss these four active object languages in detail: Rebeca, ABS, ASP, Encore. Our selection intends to cover different aspects: modeling languages designed for analysis (ABS, Rebeca) vs. languages optimized for efficient execution (Encore, ProActive/ASP); close to the classical actor model and distributed systems (Rebeca, partially ABS, ProActive/ASP) vs. targeted at multicore platforms (Encore); code generation (ABS) vs. runtime environment (Encore, ProActive/ASP). We have also made sure that the four languages discussed in detail have left the experimental stage and are available in stable, public distributions. All four language have a formally defined semantics allowing us to precisely compare the behavior of the programs written in them.

Other high-level concurrent languages. A detailed discussion of other high-level concurrent languages can be found in Section 4.2.

2.2 Dimensions of Comparison between Languages

Before we discuss details of the different active object languages, we define the key points that we consider as important when comparing their design.

Objective of the language. Identifying the objective of the language, for which purpose it was created, is crucial to understand the design of a programming language and the trade-offs that have been made between different aspects. For example, the performance of the language can be a

crucial factor for which another aspect can be given less priority, such as the accessibility of the language to non-experts.

Degree of synchronization. There is a close relation between the design of a concurrent programming language and the degree of synchronization that can be expressed in it. Each language has a different set of synchronization primitives, even though for active object languages the choices are limited. Some languages, inspired by pure actors, have no synchronization primitive: concurrent entities evolve in a completely asynchronous manner and there is no instruction to wait for some event to happen. Synchronization between processes is due to the causal dependency created by the flow of messages. Many active object languages use futures as a synchronization mechanism. A future represents a result that has not been computed yet. Once the result has been computed, it can be made available automatically or be explicitly retrieved, depending on the language design; in each case we say the future is *resolved*. A process can also block while waiting for a future to be resolved. In active object languages, futures represent the result of asynchronous invocations. Usually a future access is a synchronization point. This kind of synchronization can make concurrent programming easier, as it ensures a sequential workflow upon which the programmer can rely.

Some active object languages support cooperative scheduling: a thread can be suspended to handle another message[1] and resumed later. Suspension can be triggered when checking whether a future was resolved. This breaks the sequential processing of a message, but can make program execution more efficient and less deadlock-prone.

Another synchronization constraint is related to message ordering [26]. Ensuring an order of message reception, like point-to-point FIFO, adds synchronization which can lead to a loss of efficiency and a gain in program properties. The programmer can rely on some order of message delivery, which simplifies programming. Often some order of message delivery is necessary for the execution of the application, especially when the messages reach a stateful object. Ensuring that the operations are done in the correct order by explicit synchronization is more costly and more constraining than relying on a message ordering property.

Degree of transparency. This aspect concerns the number and complexity of the programming abstractions we need to understand. Some abstractions are made explicitly visible in the program and some are transparent (i.e., hidden to the programmer). For example, if futures are transparent, then variables pointing to futures are not explicitly distinguished from other variables by a static type and no explicit instruction is provided to access a future: the access is blocking if the future is not resolved yet.

In general, the more transparency, the easier it is to write simple programs, because the programmer does not need to know the specifics of the concurrency model and parallelism is provided automatically. However, for complex programs, the benefits of transparency are weaker, because exposing the programming abstractions can make programming, debugging, or optimization easier.

Degree of data sharing. Data sharing is a crucial aspect of concurrent and distributed programming. Depending on the target application domain and the potential for distributed execution, the constraints regarding data sharing can vary a lot. The most efficient way to communicate data between different cores on the same machine is to share the data, whereas in a distributed setting, copying data to different consumers is often more efficient, as it avoids communication overhead.

Efficiency aside, the complexity of the language implementation varies with the degree of data sharing. Copying data raises the problem of data consistency, as data modifications may not be

[1]The terms *message* and *method call* are used interchangeably in the literature on active object languages, depending on the tradition that influenced language development. Here we adopt the terminology that is standard for the language under discussion.

reflected in all copies. Shared data access makes data consistency easier to obtain, but creates additional communication about the shared memory, additional synchronization, and delays. In practice, active object and actor languages with distributed runtime systems often use a "no shared memory" approach favoring data copies with very few globally known and shared entities.

Data sharing occurs between threads and relates to the question of which objects are active. The first active object models were *uniform*: all objects were active with their own thread and communicating by asynchronous method invocations only. Later, active object models were designed with more focus on efficient access to objects. Two additional models were proposed. First, *non-uniform* active object models, where some objects are full-fledged active objects with their own threads, and others are passive objects (i.e., standard objects). Second, concurrent object groups provide a different alternative where all objects are accessible from anywhere but the objects in the same group share the same thread and their access is synchronized.

Formal semantics. To establish formal properties of a language as well as for formal verification, a formal semantics is required. Most active object languages have a formal semantics, probably due to the fact that active object and actor models were created to make concurrent entities run more safely. A formal semantics can be used to prove generic language properties that help programming (e.g., data race-freedom), to prove the soundness of an implementation or of optimizations, and to implement tools for the formal analysis of programs. All these aspects have the potential to increase the trustworthiness of a language and the programs written in it.

Implementation and tooling support. An important dimension of language comparison concerns the provided tool suite. Some active object languages have been designed with certain tool support in mind, which can explain some of the design decisions that were taken. The tool support around a programming language ranges from utilities to help the programmer in designing, writing, analyzing, optimizing, and verifying their programs to utilities to support the deployment and execution of these programs.

Futures play a particularly important role in the design of an active object language, both concerning transparency and synchronisation. Futures can be either explicit, with a specific type and dedicated operations like in ABS or Encore, or implicit with automatic synchronisation like in ProActive. The implicit synchronisation is easier to program but explicit futures makes the programmer aware of synchronisation points and makes it easier to spot deadlocks. Also, with implicit futures code fragments can remain oblivious of whether they operate on regular references or on future references. Future synchronisation can be blocking like in ProActive, blocking with thread release like in Creol, or asynchronous via callbacks like in AmbientTalk. The blocking access has the advantage to guarantee the sequential processing of a message but can lead to deadlocks. The asynchronous callbacks are deadlock free but enforce less sequential execution and creates additional race conditions. Blocking with potential thread release provides a compromise, where messages are treated sequentially but can be interrupted to handle another message. ABS and Encore provides both strict sequentiality (get) and potential thread release (await); when accessing a future, the programmer must choose between a potential race condition and a potential deadlock.

2.3 Representative Examples of Active Object Languages

We give detailed presentations of the four languages identified at the end of Section 2.1. To ease comparison, each presentation follows the dimensions introduced in the previous section.

2.3.1 Rebeca.

General presentation and objective of the language. Rebeca (*Reactive Objects Language*) is an actor-based modeling language created in 2001 [88, 89] as an imperative interpretation of Agha's actor model [3].

Rebeca is designed to model and verify concurrent and distributed systems with the goal to bridge the gap between software engineers and the formal methods community, by being a usable and at the same time analyzable modeling language. It has a formal semantics and allows efficient compositional verification based on model checking [86, 87] with state space reduction techniques [60]. Efficient analysis relies on a number of restrictions: no shared variables, no blocking send or receive, single-threaded actors, and non-preemptive message execution (the execution of different messages does not interfere).

With usability as a primary design goal, Rebeca's concrete syntax is Java-like, its computation and communication model is kept simple, and analysis support is in the form of model checking. Rebeca is an actor-based language without shared variables between actors and with asynchronous message passing; there is no blocking send or receive statement. Therefore, learning Rebeca is easy and using model checking tools requires far less expertise than deduction-based analyses. The semantics of Rebeca helps analyzability as follows: Rebeca actors are isolated and hence various abstractions, as well as modular and compositional verification become more feasible. Rebeca offers natural modeling capabilities and efficient model checking tools for concurrent distributed, event-based applications with pure asynchronous message passing. However, sometimes synchronization among components or different communication models are vital, and the modeling of some applications becomes cumbersome. Following the strategy of "pay for what you need", core Rebeca can be extended in different dimensions, creating the Rebeca family, to support modeling and analysis of a wider variety of application domains (see Table 1).

Extended Rebeca [85, 86] groups actors in components, with globally asynchronous communication between components and locally synchronous communication inside a component. RebecaSys is developed to support hardware/software co-design (or system-level design) [77] and adds a wait statement to be faithful to the target design language, SystemC. Global variables are added, but their use is controlled. In Variable Rebeca, the modeler can use annotation to model variable behaviors and hence define a product line of actors with different computations [81]. Variable Rebeca preserves the semantics of Rebeca and all members of Rebeca family can be extended by the annotation mechanism used in Variable Rebeca. Broadcasting Rebeca [94] and Wireless Rebeca [95] focus on modeling and verifying of network protocols, and provide broadcasting and multi-casting message passing, respectively. Timed Rebeca [2, 78] addresses real-time behavior and is supported by customized efficient analysis tools. Probabilistic Timed Rebeca [59] extends Timed Rebeca to capture probabilistic behavior and can only be analyzed using back-end tools. These extensions are orthogonal to other language extensions which preserve the semantics of core Rebeca, these extensions can be added on top of Variable Rebeca, and Broadcasting and Wireless Rebeca.

Language description. We describe Rebeca using a Media Service example, shown in Fig. 1. Clients send requests for watching movies to a dispatcher and the dispatcher non-deterministically redirects requests to media servers. The Rebeca model of Media Service consists of a number of *reactive classes*, which are `Server`, `Dispatcher`, and `Client`, each describing the type of a certain number of *actors* (called *rebecs* in Rebeca). A reactive class specifies the size of its message queue (Line 1) and may declare *state variables* (Line 14). Each actor has a set of known actors to which it can send messages. For example, an actor of type `Dispatcher` knows three actors of type `Server` (Line 2), to which it can send a `reqForMovie` message (Lines 6–8). Each reactive class in Rebeca may have a constructor. Constructors have the same name as the declaring reactive class and do not return

Table 1. Summary of dimensions of comparison for members of Rebeca family

	Objective	Synchro-nization	Transparency	Data Sharing	Formal Semantics	Tool Support
Rebeca	Modeling and verification of distributed, concurrent systems	None	(Full) message queues and interleaving of execution	None	SOS, ACP	Afra integrated tool, various backends
Extended Rebeca	Globally asynchronous-locally synchronous systems	Locally synchro-nous messages	As in Rebeca	None	SOS	None
RebecaSys	Hardware/ software co-design	Wait statement	As in Rebeca + synchronization over global variables	Global variables	LTS	Model Checking, Simulation
Variable Rebeca	Product lines of actors	None	As in Rebeca + feature inclusion for selected product	None	SOS	Rebeca tool (after manual mapping to Rebeca)
Broadcasting Rebeca	Actors with broadcasting abilities	None	As in Rebeca + broadcasting to all actors mechanism	None	SOS	Model Checking
Wireless Rebeca	Ad-hoc mobile networks	None	As in BR + handling connectivity of nodes + multi-casting	None	SOS	Model Checking
Timed Rebeca	Realtime actors	Delay statement	As in Rebeca + Progress of time	None	SOS, RT Maude, Timed Automata, FTTS	Afra integrated tool, various backends
Probabilistic Timed Rebeca	Probabilistic real-time actors	As in TRebeca	As in TR	None	SOS	Backends IMCA and Prism

a value. Their task is to initialize the actor's state variables (Line 16) and to put initially needed messages in the queue of that actor (Line 22).

Message servers are implemented in reactive classes similar to methods in Java. In contrast to Java methods, message servers do not return values and their declaration starts with the keyword `msgsrv`. A message server is executed upon receiving its corresponding message. Its body may include assignments, conditionals (Lines 5–9), loops, and message sending statements (Lines 18, 24). Periodic behavior is modeled by sending messages to oneself (Line 26). Since communication is asynchronous, each actor has a *message queue* from which it takes the next message to serve. The ordering of messages in these queues is FIFO. An actor takes the first message from its queue, executes the corresponding message server non-preemptively, and then takes the next message. The actor stays idle if there is no message in the queue. A *non-deterministic assignment* statement models non-determinism in a message server (Line 4). The `main` block creates the actors of the model. In the example, seven actors are created and receive their known actors and the parameter values of their constructors (Lines 29–31).

Degree of synchronization. Following Agha's [1986] actor model, Rebeca's actors only communicate by asynchronous message passing, but Rebeca ensures a FIFO point-to-point message ordering, guaranteeing some execution order. Extended Rebeca [85, 86] adds tightly coupled components; actors inside a component can be synchronized by a handshake communication mechanism guaranteeing a causal message ordering. RebecaSys adds *wait* statements to the syntax and a block of shared *global variables* to the model. All actors can read from and write to global variables. Waiting on global variables introduces a form of synchronization between actors reminiscent of

```
1  reactiveclass Dispatcher (5) {
2    knownrebecs { Server m1, m2, m3; }
3    msgsrv reqForMovie(Client client) {
4      byte selectedServer = ?(1, 2, 3);
5      switch (selectedServer) {
6        case 1: m1.reqForMovie(client);
7        case 2: m2.reqForMovie(client);
8        case 3: m3.reqForMovie(client);
9      }
10   }
11 }
12
13 reactiveclass Server (5) {
14   statevars { byte maxClient; }
15   Server(byte maxC)
16     { maxClient = maxC; }
17   msgsrv reqForMovie(Client client)
18     { client.receiveMovie(); }
19 }
20 reactiveclass Client (2) {
21   knownrebecs { Dispatcher disp; }
22   Client() { self.watchMovie(); }
23   msgsrv watchMovie()
24     { disp.reqForMovie(self); }
25   msgsrv receiveMovie()
26     { self.watchMovie(); }
27 }
28 main {
29   Server s1():(2),s2():(3),s3():(2);
30   Dispatcher di(s1,s2,s3):();
31   Client c1(di):(),c2(di):(),c3(di):();
32 }
```

Fig. 1. A simple Rebeca model

cooperative scheduling over futures, as an actor's processor is released while waiting for a given Boolean expression to become true.

Broadcasting Rebeca adds a broadcast statement to Rebeca, but there are no known actors, so sending a message directly to an actor is impossible. Wireless Rebeca adds support for multi-cast to neighbors and uni-cast to the sender to Broadcasting Rebeca. Neighbors of an actor are those actors that sent a message to it. There is no additional synchronization between two given actors in Broadcasting and Wireless Rebeca.

Degree of transparency. In Rebeca, programmers are not exposed to asynchronous communication among actors. The syntax is the same as for sequential programming. There is no way to access the content of message queues, and acknowledgments are not sent upon starting or finishing the execution of a message server. Programmers have no control over the interleaved execution of actors; i.e., the internal thread of an actor takes a message from the actor's message queue and serves it in an isolated environment, regardless of the states of other actors. This transparency holds for all Rebeca extensions. Timed Rebeca and Probabilistic Timed Rebeca change the message queue into a bag with time-stamped messages, nevertheless, the transparency is unchanged.

Degree of data sharing. Rebeca is a typical *uniform active object* language which guarantees the absence of data sharing. There are no shared variables among actors in Rebeca. Parameters in messages sent among actors are passed by value. This is even the case if a reference to an actor is passed as a parameter; the sent reference is created as a shallow copy of the original reference, e.g., in Fig. 1, Line 8 `Dispatcher` sends a request to the server `m3` and passes the reference to the client to `m3` as a parameter instead of passing a fresh copy of the client actor. In RebecaSys, global variables can be defined and used in a controlled manner for wait statements as explained above.

Formal semantics. To support analyzability and develop formal verification tools, Rebeca and its extensions all have formal semantics. The model checking tool set of Rebeca is developed based on Rebeca's semantics [89]. An ACP (Algebra of Communicating Processes) semantics is

defined in [58]. The semantics of Timed Rebeca is defined in terms of timed automata [67], timed transition systems [68], Real-time Maude [80], and floating time transition systems [68]. Floating time transition systems constitute an innovative semantics for real-time actors, resulting in a significantly reduced time and memory consumption when verifying timed actor models. Formal semantics of Probabilistic Timed Rebeca is defined in terms of Timed Markov Decision Processes in [59].

Implementation and tooling support. There is a wide range of analysis and mapping tool sets for Rebeca family models:[2]

- Eclipse-based editor with syntax highlighting for writing models and properties
- Compiler and build system integrated with the error location mechanism of Eclipse
- Model checking tool for LTL and CTL model checking of Rebeca, as well as Floating Time Transition System generator and analyzer for Timed Rebeca
- State space generators for CADP and μCRL analysis and visualization tool sets
- Facilities for traversing counter examples in case of property violation

Most of these tools and libraries are part of Afra, the modeling and model checking IDE for Rebeca family models, but there are some stand-alone tools as well:

- Simulator backend for Rebeca and Timed Rebeca in Erlang [78]
- Analysis backend for Timed Rebeca in Real-Time Maude [80]
- Model checking tool chain for Probabilistic Timed Rebeca [59] using PRISM and IMCA
- SysFier: model checking tool for SystemC designs [77]
- Sarir: μCRL2 model checker for Rebeca [58]
- Distributed model checking tool for Rebeca models [66]
- Model checking tool for Broadcasting Rebeca [94]
- Bounded Rational LTL model checker of Rebeca [12]
- Guided search engine for deadlock detection [84]

2.3.2 *ABS.*

General presentation and objective of the language. The *A*bstract *B*ehavioral *S*pecification language (ABS) [61] is an object-oriented, concurrent modeling language developed since 2009. Its ancestors include Creol [64] and JCoBox [82].

In contrast to design-oriented or architectural languages, ABS code is fully executable. There is a simulator as well as several code generation backends (at the moment, for Java, Haskell, and Erlang). At the same time, ABS abstracts away from features that make automatic analysis difficult in mainstream programming languages. For example, ABS has an object model, but it does not support code inheritance and it enforces programming to interfaces as well as strong encapsulation. It retains, however, modeling features that are essential for real-world applications, for example, aliasing and unbounded object creation. The main design goal of ABS was to create a language that permits the complex behavior of concurrent objects to be specified in a concise, natural manner, while keeping automated analysis of that behavior feasible and scalable.

There are extensions of ABS to model product variability [45] as well as time and other resources [65] which have been used to, e.g., model and analyze industrial software deployed on the cloud [6], but these extensions are considered to be out of scope of the present article.

Language description. The language layers of ABS are displayed in Fig. 2. Based on parametric (first-order) abstract data types, a pure, first-order functional language with pattern matching and strict evaluation is defined. On top of this functional layer, there are objects and a standard

[2]These are accessible from the Rebeca home page, see http://rebeca-lang.org

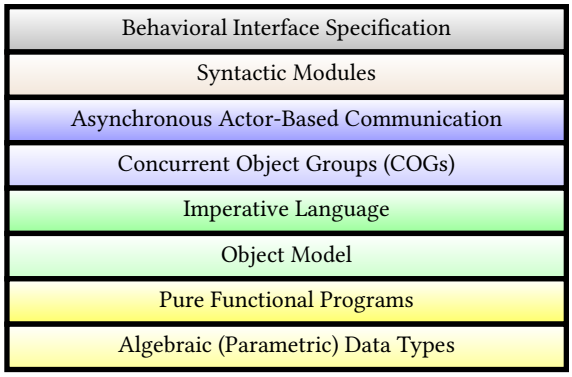


Fig. 2. Conceptual layers of the ABS language

```
1 module Services;
2 import Data, init, modify
3   from CustomerData;
4
5 interface Service {
6   Unit process(Fut<Data> fd);
7 }
8 class Server implements Service{
9   Unit process(Fut<Data> fd) {
10    await fd?;
11    Data rd = fd.get;
12    rd.modify();
13  }
14 }
15 { // main block
16 Service s = new local Server();
17 Data d = new local Data();
18 Fut<Data> fd = d!init();
19 s!process(fd);
20 }
```

Fig. 3. A simple ABS model

imperative layer. So far, this is very much like Scala-light. The central issue for achieving automated, scalable analysis is the design of the concurrency model.

The concurrency model of ABS combines asynchronous communication from the Actor model with cooperative concurrency. We explain this concurrency model by means of the code in Fig. 3. The unit of distribution in ABS is a *concurrent object group* (COG), which can be thought of as a set of tasks that share a common heap and a single processor. Tasks in different COGs can execute in parallel, but at most one task is active in a given COG at any time. Each task executes code owned by an object in the COG. New tasks are created by asynchronous method calls to objects in the COG as well as, initially, by selecting the main block of a module. An example of the latter is the code in Lines 16–19.

Line 16 declares and creates a new object of interface type `Service`, using the implementation in class `Server`. The directive `local` places the object in the current COG. Without `local`, a new COG would be created together with the object. The next line declares and creates a data object (note that interface `Data` must be imported) in the same COG. Hence, `s` and `d` share the same heap. Line 18 calls an initialization method (implementation not shown) on the data. The notation "`!`" denotes an asynchronous call; its effect is to create a new task in the COG of `d` that executes the code of `init()`.

ABS enforces the "programming-to-interfaces" discipline [72] and has no other hiding mechanism than interfaces: an object may implement several interfaces, so each pointer to the object is typed by an interface controlling the methods available on that pointer. Fields are not accessible via interfaces. Asynchronous calls do not interrupt the caller, so the statement following the call is executed immediately. Therefore, we need a handle that we can use to retrieve the result of an asynchronous call once its result has been computed. In ABS, the type of such handles has a future

annotation. As can be seen in Line 19, futures can be passed as parameters. This makes it possible to use the result of an asynchronous call in different places without copying it (for example, for barrier synchronization or delegation).

After the execution of the main block has finished, two tasks in the current COG are waiting, corresponding to the calls to `init()` and `process()`, respectively. None of them could have started while the main block was still executing, because there was no synchronization expression in the latter. ABS does not determine which of `init()` and `process()` is started first. In fact, ABS can be parameterized with different scheduling strategies, which can be programmed at the functional layer [14]. The semantics leaves scheduling open, so the static analyzers for ABS take all possible scheduling sequences into account.

ABS introduces *cooperative concurrency* internally in COGs. This means that no task is preempted (interrupted) unless its modeler explicitly allows this to happen. Two expressions in ABS explicitly release control: `release` and `await`. The former is unconditional while the latter has a Boolean argument and can be used to synchronize with another task. In particular, synchronization can depend on the resolution of futures; i.e., cooperative scheduling can depend on the presence of return values from asynchronous method calls. In the example, a synchronization point is reached at Line 10 in the body of `process()`. It ensures that the value of the future `fd` is available. If `init()` had not been already scheduled, it will be scheduled now. Once the value of `fd` is available, it is retrieved with a `get` expression. Note that `rd` and `d` might well be aliased. The standard ABS idiom for asynchronous method calls in ABS is as follows:

```
Fut<T> fx = o!m(); ... ; await fx?; T x = fx.get;
```

In many cases of simple control flow, `await` and `get` follow an asynchronous call directly, without intervening code. For this common situation the abbreviation

```
T x = await o!m();
```

is provided, which avoids the declaration of an explicit future.

Synchronous calls are also supported in ABS. Synchronous calls result in a stack-like behavior, i.e., the call yields the processor to the callee and blocks the caller until it returns. In Line 12, we are only interested in the side effect of the `modify()` method. Synchronous calls are only permitted inside the COG of the caller object, and the caller may decide whether to call any local method synchronously or asynchronously. Note that the entire stack will be suspended in the case of a processor release.

The concurrency model of ABS is designed to make formal analysis feasible. While formal analysis of multi-threaded languages with interleaving semantics, such as Java, is possible in principle [1, 15], such analysis is highly complex and currently out of scope for relaxed memory consistency models. The concurrency model of ABS carefully restricts the possible interactions between concurrent tasks by means of cooperative concurrency, while still allowing the complex, realistic behavior of asynchronous systems to be described. Analysis in this setting has been shown to be compositional [4, 35] and scalable [40]. Section 3.2 discusses a possible implementation of cooperative scheduling for ABS.

Degree of synchronization. If one attempts to retrieve a future value that is not yet ready, this results in the blockage of its COG until the value becomes available. Obviously, this can easily lead to deadlocks. Many deadlocks can be avoided by guarding `get` expressions with an `await` (Line 10), however, not all deadlocks can be prevented in this manner. In addition, ABS comes with automated deadlock detection tools [7, 43]. An important point is that no data races can occur between explicit synchronization points (`release`, `await`), which makes computations deterministic. Communication is asynchronous in ABS, and no particular ordering has to be ensured on request communication

and service. Obviously, execution in general is non-deterministic unless the scheduling of service requests is also controlled.

Degree of transparency. ABS is an abstract language and implementation-specific aspects including scheduling, message queuing, and object representation are hidden from the modeler. Abstract data types and interfaces can be used to postpone detailed design decisions while still permitting analysis of those aspects of a model that are fully specified. In ABS the user may allow task interleaving by explicitly introducing task release points using `release` and `await`. Using these features, the user is exposed to the notion of asynchronous vs. synchronous calls, futures, and thread interleaving, but the language still retains a compositional proof theory [39].

Degree of data sharing. There is no designated active object in a COG, all objects may be accessed remotely. Thus, values from the functional language of ABS are passed by copy with a method call whereas all objects are passed by reference (i.e., the pointer is copied), independent of whether the called object is local or remote. All objects in ABS have strictly private visibility, that is, they can only access their own fields directly. The fields of any other object must be accessed via explicit method calls. ABS features a *concurrent object group* model where objects in the same group can safely be invoked directly because they are manipulated by a single thread, invocations between different COGs are by nature asynchronous.

Formal semantics. ABS has a small step operational semantics [56, 61] that permits to prove soundness theorems for the various analyses that are available for ABS. This semantics is directly expressed in terms of rewrite rules in the Maude format [28], which yields a sound interpreter for ABS.

In addition there is an axiomatic semantics in the form of a program logic [38]. The behavior of interfaces and classes can be specified by invariants over the histories of symbolic states as contracts between processes. Because preemption is excluded in ABS it is sufficient to establish invariants upon object creation, at explicit synchronization points and upon the termination of methods. A composition theorem for ABS about the relation between local and global invariants has been established [39], which makes it possible to prove global behavioral properties of an ABS model by (method-)local reasoning.

Implementation and tooling support. As ABS has been developed with the goal of being analyzable, there is a wide range of tools available.[3] Most of them support the full ABS language and are fully automatic, see the overviews [17, 90]. There is an Eclipse plug-in that provides a development environment for ABS and integrates most tools. An alternative is the web-based ABS collaboratory [41] that runs in a browser and permits to try out most ABS tools. The collaboratory can be used either as a service or installed locally. Here is a list of currently supported tools for ABS:

- An editor with syntax highlighting and integrated build system, including compiler error location
- An ABS simulator/interpreter with interactive debugger
- Visualization of ABS model execution as a sequence diagram
- Code generator backends for Erlang, Haskell, ProActive [54] (see also Section 3.4), Java 8 [83].
- A glass box test case generator for *concurrent* models [5]
- Sound deadlock analysis tools [7, 43]
- A worst-case resource analysis tool that can be parameterized with a cost model for execution time, memory consumption, transmission size, peak cost and various other cost categories [4]

[3]These are accessible from the ABS home page, see http://abs-models.org

- A deductive verification tool to prove expressive, history-based properties [38, 40] for models with an *unbounded* number of objects and tasks

2.3.3 *ProActive and ASP.*

General presentation and objective of the language. ASP [21] is an active object programming language specifically designed for programming and deploying distributed systems. ProActive is a Java library implementing the semantics of the ASP calculus. The language is designed taking the constraints of distributed programming into account, and relies on Remote Method Invocation (RMI) as the communication layer even though other communication mechanisms can be used. ProActive is intended for distributed execution; it is a middleware that supports application deployment on distributed infrastructures such as clusters, grids and clouds. Several choices for the design of the language can be explained by these practical concerns.

One design choice for ASP and ProActive is to ensure maximal transparency for the programmer: active objects and futures are used like usual Java objects. The ProActive middleware automatically triggers asynchronous remote invocations and performs blocking synchronization when needed.

A further design choice is that active objects are coarse-grained entities. We create a dedicated thread for each of them and they come with a quite heavy machinery for managing each object and communicating between them. This machinery fully ensures the distributed nature of the computation. As a consequence, using the ProActive library, it is not possible to instantiate thousands of active objects on the same core. In ASP not all the objects are active and in ProActive passive objects are standard (Java) objects, consequently their number is not particularly limited.

Since 2010, ASP features *multi-active objects* [49] meaning that in each active object, several threads can run in parallel and process several requests of this active object, but each thread is still isolated inside a single activity. Such multi-active objects feature at the same time local concurrency and global parallelism.

Language description. In ASP, active objects coexist with so-called *passive* objects. An active object together with its service thread(s), its passive objects, and its request queue is called an activity. Each passive object is placed under the responsibility of an active object. Only active objects are accessible between activities. The objects that are not active are only accessible within an activity; if those objects need to be used by several activities, they are copied in each activity. Based on this clear separation, the activity is the unit of distribution, which matches the usage of one memory space per activity. When using multi-active objects in ASP, several threads can execute in the same activity; thus, several threads can potentially access the objects of an activity.

The language is transparent: method calls are automatically turned into asynchronous requests if the targeted object is a remote active object, otherwise it is a synchronous, local method call. Similarly, futures are implicitly created for asynchronous calls. Futures are also transparently manipulated: wait-by-necessity synchronization is automatically triggered upon an access to an unresolved future. In ASP, futures are first-class: they can be passed between activities. In this case, when the future is resolved, the result is automatically updated at all locations.

ProActive offers an API to create active objects, and a runtime for handling ASP features. The following is an example of a ProActive program:

```
O o = PAActiveObject.newActive(O.class, parameters, node);
T t = PAActiveObject.newActive(T.class, parameters, node);
V v = t.bar(); // implicit asynchronous method call
o.foo(v);      // v can be passed without blocking
v.foobar();    // potential wait-by-necessity on v
```

An active object is created using `newActive`, instead of the `new` command of Java. The `newActive` primitive takes as parameters the class to instantiate, the parameters of the constructor, and the node on which the active object will be deployed. The variable `v` is the result of an asynchronous call; it is an implicit future. When the future value is needed to continue execution, such as in `v.foobar()`, wait-by-necessity synchronization automatically applies if the future is not resolved. Proxies in proActive handle active objects and futures transparently. The transparent creation of proxies has some practical restrictions: the active objects and futures cannot be of a primitive or generic type. If a future cannot be created, this creates an immediate synchronisation on the result.

The main principle of the multi-active object programming model is to execute multiple requests of an active object in parallel, while controlling the concurrency. In practice, the programmer can declare which requests (i.e., which methods of the active object) can be safely executed in parallel. Such requests are called *compatible* requests. The internal scheduler of an active object will allocate by default as many threads as necessary to run compatible requests in parallel. In ProActive, multi-active object features can be used through a meta language, based on Java annotations. The following is an example of multi-active object annotations in ProActive:

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

  @MemberOf("group2")
  public ... method2(...) { ... }
}
```

This example defines two request groups, with one method each. The two groups are declared to be compatible (and so are their methods, by extension). The `selfCompatible` parameter defines whether two different requests of the same group are allowed to run in parallel. At runtime, a ready request is automatically executed if it is compatible (i) with requests that are already executing and (ii) with older requests in the queue. The first condition prevents data races. The second condition preserves the ordering of incompatible requests and prevents starvation, which would arise when requests of a group G that are merely compatible with currently executing requests are continuously overtaking requests of another group G' with which they are incompatible.

Without annotations, a multi-active object is a mono-threaded active object without any local parallelism nor any possible race condition. Programming a mono-threaded active object-based application with ProActive is thus extremely simple. If some local parallelism is desired, a compatibility should be declared between requests that can be safely interleaved and for which execution order does not matter.

To define compatibility between two requests, the programmer can use runtime information such as the request parameters or the object's state. Programming a multi-active object-based application with ProActive is thus slightly more difficult than programming mono-threaded ProActive active objects, but far less complex than programming with raw threads and low-level synchronization mechanisms while, at the same time, providing a considerable degree of parallelism. If even more parallelism is required, beyond what is possible with request compatibility, then the programmer can define further requests as compatible and prevent undesired behavior with traditional low-level

Java synchronization primitives. Such a mixed approach goes beyond the traditional active object model.

Further high-level specifications are available in multi-active objects, such as request priority [53]. To avoid thread explosion, a limit can be set on the number of threads running in parallel. That can either be a hard limit restraining the overall number of threads or a soft limit that only counts threads not involved in wait-by-necessity synchronization. Additionally, threads can be limited per group.

To summarize, ASP and ProActive are based on the multi-active object programming model. This model is suitable for non-experts, because it provides high-level features for distribution and safe concurrency.

Degree of synchronization. The only blocking synchronization in ASP is wait-by-necessity on a future. As requests run to completion, potential deadlocks can arise for re-entrant calls, especially if no compatibility annotation is specified. However, synchronization only occurs when the future value is actually needed and future references can be safely transmitted between activities without requiring additional synchronization, which limits the blocking synchronization leading to deadlocks. The ProActive middleware transparently handles future value transmission [51] (a ProActive future has its own serialization procedure). Deadlocks can also be removed by using multi-active objects with no limit or a soft limit on the number of threads. Specifically, when a thread enters wait-by-necessity, it is not counted in the soft thread limit of the active object anymore, so the wait-by-necessity event potentially causes the start of another request.

Communication in ASP is causally ordered and requests are served in FIFO order. This restricts the communication ordering more than a simple FIFO guarantee, providing more properties to the programmer at the expense of additional delay during request emission.

Degree of transparency. As pointed out above, transparency is a central design goal of ASP. The programmer is not exposed to the notion of a future and merely in a limited manner to that of an active object (at object creation time only). The syntax is as for sequential programming, there is no specific construct for awaiting a future value or for performing an asynchronous call. Frequently, sequential code can be reused unchanged in a distributed setting. When dealing with multi-active objects, the programmer is exposed to the parallel treatment of requests and the programming model becomes more explicit concerning concurrency aspects.

Degree of data sharing. ASP is a typical example of a *non-uniform active object model*, where some objects are active and the others are passive and copied when transmitted between activities. ASP follows a strict policy of absence of sharing between active objects. Objects that are not active are passed by copy between activities (as request parameters or request results). This also applies to objects that are referenced by passed objects: when objects are transmitted between activities, a deep-copy mechanism ensures that they are copied together with all their dependencies to the destination side. This mechanism, also used by RMI, slows down request invocation, because of the time spent to transmit data, but it accelerates request treatment, because there is no need to contact another activity to obtain the value of the request parameters. Note that active object and future references are passed by reference.

Coherence between different copies of a passive object is not guaranteed. If a user wants to ensure that an object has a unique coherent state, he should not transmit it by copy: transmitting it as an active object reference would then be the best choice.

Formal semantics. Caromel et al. [2004] formalize the mono-threaded part of ASP and prove determinacy properties. In particular, they prove that the order of future updates has no influence on the execution and that the only source of non-determinacy in mono-threaded ASP is when two

activities can send a request to the same destination. A functional fragment of the calculus has been formalized in Isabelle/HOL [50]. A specific semantics has been designed to evaluate a functional ASP program without risk of deadlock; the absence of deadlocks is proved in Isabelle/HOL. The full semantics of ASP with multi-active objects is formalized by Henrio et al [2013].

Implementation and tooling support. ProActive is the Java library implementing the ASP semantics. To transparently handle active objects and futures in ProActive, a proxy is created for each of them. Proxies encapsulate all code required to perform asynchronous, remote method invocations and wait-by-necessity synchronization.

Whenever an active object or a future is given as a call parameter to an active object, it is in fact their proxy that is copied. Hence, all copies of a proxy of an active object/future point to the same active object/future. A further aspect of ProActive deals with the deployment of ASP active objects on distributed infrastructures. The design choices of the programming language typically target high performance for distributed ProActive applications. To deploy active objects on distributed infrastructures, ProActive has a deployment specification mechanism which makes the physical deployment independent from the deployment logic. This is realized by binding virtual node names, used in the source code, to machine addresses or names. In practice, this binding is implemented in XML configuration files. Since binding happens at *deployment time*, changes of the infrastructure for a ProActive application are localized in few files and do not require recompilation. Several machines can be aggregated under a single virtual node name in the deployment logic, for example, to provide the virtual node with certain properties or non-functional deployment options (e.g., fault tolerance or logging).

Active objects provide a convenient programming model for component-based composition of distributed applications [11]. The ProActive library implements the GCM distributed component model. In this context, the Vercors platform [52] provides verification capacities for ProActive components; Vercors consists of an Eclipse plugin for designing component systems and can both verify the correct behavior of the application using the CADP model-checker and generate executable ProActive/GCM code corresponding to the designed system.

2.3.4 *Encore.*

General presentation. Encore [16] is a general purpose parallel programming language based on active objects developed since 2014. The language has been designed to excel at scalability and rests on four pillar concepts: *parallel by default* using the active object paradigm for coarse-grained parallelism, *independent local heaps* to promote data locality, *type-based synchronization directives* provided by a novel capability system and *coordination of parallel computations and low-level data parallelism* via parallel combinators. On top of these key ingredients, Encore stays within the object-oriented paradigm where active objects and futures can be seen as normal objects and, instead of interfaces, has a trait system à la Scala.

Language description. In Encore, active objects have their own thread of control and communicate with each other via asynchronous messages passing. Messages are appended to the receiver's message queue and processed one at a time in FIFO order. Because of their internal thread of control, active objects have an asynchronous interface, meaning method calls return immediately with a future. In contrast, passive objects do not have an internal thread of control and expose a synchronous interface. Each active object owns a local heap on which passive objects are allocated. Ownership in this context means that the actor is responsible for keeping track of foreign dependencies on passive objects on its local heap, and eventually deallocate them. In Encore, passive objects may be *shared* between active objects and concurrent read/write access to passive objects is supported, by using the aforementioned capability system.

```
1 read class Mtx
2   ...
3   def computeFastestDiagonal(): Par[Mtx]
4     val mtx = this.getMtx()
5     val fLU = async luFact(mtx)
6     val fChlsk = async choleskyFact(mtx)
7     val par = (lift fLU) ||| (lift fChlsk)
8     getDiagonalMtx << bind(mtxInv, par)
9   end
10 end
```

Fig. 4. Data pipeline and speculative parallelism in matrix factorization The types of the functions are `getMtx :: Mtx`, `luFact :: Mtx -> [Factor]`, `choleskyFact :: Mtx -> [Factor]`, `mtxInv :: [Factor] -> Par[Mtx]`, and `getDiagonalMtx :: Fut[Maybe[Mtx]] -> Par[Mtx]`.

Encore is *parallel by default*. On coarse-grained parallelism, different active objects could run concurrently, while within the same active object, the execution is sequential. Fine-grained parallel computations can be created via the notion of tasks, for example, the code (`async { e }`) contains a body `e` that is asynchronously executed and immediately returns with a future type value. Tasks are more lightweight than actors (memory-wise) and increase the degree of asynchronicity in a system.

Unlike ProActive, values resulting from method calls on active objects and spawning of tasks have the explicit future type `Fut[t]`. Futures support future chaining as well as `get` and `await` operations similar to ABS. This is explained in detail below.

Cooperative scheduling of active objects. Encore supports ABS' operators for cross-message control flow, i.e. `suspend` (`suspend :: void -> void`), `get` (`get :: Fut[t] -> t`), and `await` (`await :: Fut[t] -> void`). Encore's `await` statement only supports waiting for the resolution of a future, unlike ABS which supports waiting on general Boolean conditions (Section 2.3.2). Encore's support for future chaining however makes `await` less useful as chaining allows triggering arbitrary operations on the resolved futures. Encore's scheduling of requests is explained in detail in Section 3.5.

Parallelism inside active objects. Active objects provide coarse-grained parallelism via futures but do not offer high-level language constructs for low-level coordination of parallel computations. To express complex coordination workflows, such as pipelines and speculative parallelism, Encore incorporates `ParT` collections and associated parallel combinators [16, 42]. A `ParT` collection is an abstraction for asynchronous computations and values; it is controlled via parallel combinators. `ParT` computations have type `Par[t]` (`t` is a polymorphic type).

We explain the `ParT` combinators with the example from Fig. 4. It first computes the LU and Cholesky factorizations in parallel, spawning tasks (Lines 5, 6). The resulting futures are lifted (`lift :: Fut[t] -> Par[t]`) and grouped into the same `ParT` collection (`||| :: Par[t] -> Par[t] -> Par[t]`), Line 7. Then an inversion of the matrix is performed asynchronously, using the `bind` combinator (`bind :: (t -> Par[t'])-> Par[t] -> Par[t']`), Line 8, which receives a function (first argument) that is applied asynchronously to the items in the `ParT` collection, creating pipeline parallelism. The << combinator (`<< :: (Fut[Maybe[t]] -> Par[t'])-> Par[t] -> Par[t']`), applies the `getDiagonalMtx` function to the first computation that returns the inverted matrix, stopping the remaining computations, i.e., safely stopping speculative work.

The ParT collection and its combinators have been designed to perform operations asynchronously, without stopping the current thread of execution. ParT integrates well with the active object model, as it provides a uniform interface, via parallel combinators, to manipulate a collection of asynchronous values and can express complex coordination workflows. Additional combinators are discussed in Brandauer et al. [2015].

Degree of synchronization. In Encore, the synchronization constructs `get` and `await` provide similar control over futures as in ABS: the former blocks the active object until the future is resolved and the latter releases the current thread of execution if the future is not resolved, so the active object can continue processing other messages. Furthermore, Encore provides a *future chaining* operator which registers a callback (as an anonymous function or lambda) to the future and continues processing the rest of the message. The callback is executed when the future is resolved, using the result of the future as the argument to the callback. This construct allows chaining operations on futures without creating synchronization, similar to the default behavior of futures in AmbientTalk [37], but it is more explicit and easier to control for the programmer. It is also a way to mimic transparent first-class futures of ASP, except that the callback request is only created when the future is resolved. Communication in Encore is asynchronous but requests are served in a FIFO order.

Degree of transparency. In Encore, regular and future variables are distinguished statically (e.g., `int` vs. `Fut[int]`). The `get` operation explicitly extracts the content from a future. Both in future management and with parallel combinators, the programmer is exposed to the ongoing concurrent computations. However, especially with future chaining and parallel combinators, the scheduling and ordering of operations is automatic and the programmer expresses concurrency from a high-level point of view.

Data sharing. Encore combines a *concurrent object group* model with a *non-uniform active object* model where some objects are active and other objects are passive.

In Encore, active objects are protected by their own thread of control while passive objects are protected by a *capability* type. Encore's type system sees passive objects as resources protected by a capability governing the permitted kind of access to the object [16]. A capability is built from a trait and a *kind*. The trait provides the interface-like feeling of statically-typed, Java-like OOP languages while the *kind* expresses the "protection" provided by the interface. By changing the *kind*, the interface changes its protection level. For instance, by changing the *kind* from *exclusive* to *lock-free*, the interface changes the protection from an actor-like to a lock-free implementation. Like in RebecaSys, this creates controlled data sharing.

Encore's capabilities form the hierarchy in Fig. 5. *Exclusive capabilities* are local to one thread and *linear capabilities* may be transferred between concurrent computations without data races by dynamic means of concurrency control. *Shared capabilities* may be shared across concurrent computations, because they encapsulate a means of protection against concurrent access: *pessimistic capabilities* like locks and actors serialize computation; *optimistic capabilities* like STM and lock-free capabilities allow parallel operations, but use a roll-back schema when conflicts arise. *Immutable* and *read-only capabilities* are always safe to access concurrently, because of the absence of writes. Finally, *subordinate capabilities* allow constructing aggregates of objects whose data race-freedom stems from their proper encapsulation inside another capability.

Encore's capabilities are created from traits, and importantly, different traits in the same class can construct different capabilities. This allows a substructural treatment of objects, e.g., taking a pair and splitting it into two disjoint halves, which can be pointed to and modified separately. A more experimental feature of Encore is the combination of certain capabilities to express, e.g.,

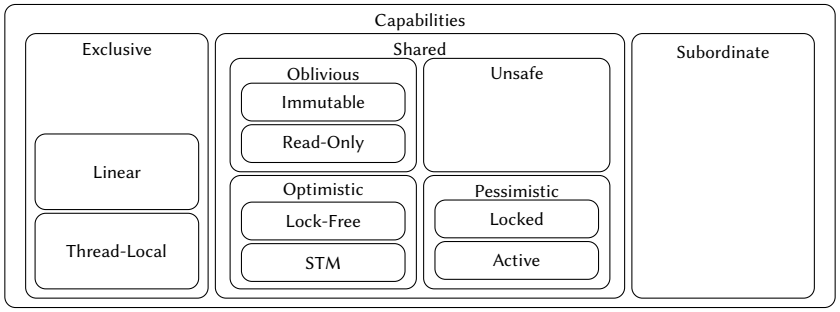


Fig. 5. Encore capability hierarchy

an active object with a partially synchronous interface (like a priority channel), or active objects which are encapsulated inside other active objects to create constrained active object topologies.

Formal semantics. The Encore concurrency model is formalized [16] using a small step operational semantics. Parallel combinators are formalized as well, including a soundness proof with the implicit task parallelism model [42, 71]. The capability type system is formalized with proofs of soundness and data race-freedom [24, 25].

Implementation and tooling support. Encore is a relatively new programming language and there is limited tooling support. However, the Encore compiler can emit readable C code which can be analyzed and debugged using any available tool that works with the C11 standard. The currently supported Encore tools are:

- Emacs and Atom editors with syntax highlighting and compiler error support
- Support for the GDB/LLDB interactive debugger
- Vagrant support for rapid installation of Encore in a virtual machine

3 IMPLEMENTATION OF ACTIVE OBJECTS

This section describes the implementation of the active object programming models introduced above. Active object languages claim to support more intuitive, easier-to-use concurrent programming models than traditional programming languages, while retaining the latter's efficiency (or even improve upon it). Hence, implementation aspects of active object languages are as important as their design. Their discussion exhibits a number of difficult, partially unsolved, research questions. They constitute an essential part of this survey.

The implementation of a programming language can be done in a standalone manner with a full compiler tool chain or it can be implemented as an API inside an established language. A third way to implement an active object language is to cross-translate it to another language with a code generation backend, i.e., a translator that captures the semantics of the source language. The advantage of the latter solution is that several backends can be supported, targeting different execution platforms, optimized for different needs.

Similar to Section 2.2, the following subsection establishes dimensions of comparison regarding the runtime system and efficient implementation of the language semantics. In subsequent sections we use these dimensions to compare different active object implementations. Whenever meaningful, we show experiments illustrating under which conditions an implementation outperforms existing solutions.

3.1 Dimensions of Comparison between Implementations

Whether an implementation has support for physical distribution or not has a strong influence on the different dimensions, for example concerning the possibility to share data between active objects, but also concerning garbage collection. Since an active object is an independent entity from the point of view of thread existence and scheduling, it is in several language considered as the unit of distribution. Among the languages presented here, ASP is clearly targeting distributed implementations, and some distributed implementations of ABS exist. Rebeca and Encore implementations do not support distribution for the moment.

Thread creation and scheduling. Active object languages implemented on top of an existing programming language (or runtime system) must comply with the constraints given by the underlying platform. In the case of multi-threading, some underlying platforms feature light threads, whereas in some frameworks (e.g., Java) each thread is a physical one (i.e. a thread of the operating system). In that case, one can consider implementing light threads on top of physical threads. Having light threads is crucial when the goal of an active object language is to scale to a large number of active objects located on the same machine, or to cope with cooperative multi-threading when tasks can be interrupted. It is a challenge to implement this behavior in programming languages that do not support thread serialization, like Java. Concerning this dimension, the following questions are raised in general: how are active objects and requests mapped to threads? Should the thread be a physical or virtual?

Data sharing and object referencing. This aspect addresses the question whether objects are shared between active objects and, if so, how. Active objects encapsulate their state such that they are independent from each other. Active objects prevent race conditions by allowing a single entity (an ABS COG or an active object) access to each object. This principle partitions the memory, which limits or prevents data races. In practice, a distinction is needed between objects that can be remotely accessed (by asynchronous invocations), objects that must be copied when exchanged between active objects, and objects that can be shared safely (e.g., immutable objects). Accessing objects in another entity might cause communication overhead and delay the treatment of requests. Copying objects might lengthen the initial request invocation, because of the serialization time, but accelerates request treatment. Handling copies also involves coherency issues. To communicate, objects must be able to reference each other. The ability to reference and interact with remote objects raises further questions in a distributed setting, where a global address space is generally too costly. How to efficiently and safely share information between active objects is an open challenge and different data sharing strategies have been proposed in active object implementations.

Error handling. The handling of errors is easier in a sequential program where the context of an operation is known. In a concurrent setting, if a task raises an error, this needs to be reported to another task. When the second task is informed of the error it is difficult to react properly as the conditions that raised the error are not fully known. In distributed implementations exception handling is even more complex, because an exception might reveal node or communication failures. Additionally, active object implementations can offer recovery mechanisms to allow systems to recover after one or more active objects have failed.

Garbage collection. Garbage collection is important for active object applications to scale and to be perennial. In most cases, the garbage collecting strategy must be implemented to fit a particular active object language. Even for active object languages built upon a host language with garbage collection, that language can only provide partial garbage collection. The central question for active object implementations is when *active* objects are no longer needed, i.e., when they cease

having to serve new requests. Again, the question is even more complex in the case of distributed implementations, because reference counting is scattered.

3.2 Java 8 Backend for ABS

ABS (see Section 2.3.2) allows several programming paradigms and design patterns offering both a functional and an object-oriented model. Annotations support custom schedulers to be defined to ease the development of batch systems and workflows. The main goal of the Java 8 backend[4] for ABS is to translate these models into production code to be executed in a parallel or distributed environment. The challenge is to generate real memory structures and execution instances, while being aware of possible resource limitations, communication bottlenecks, latencies and performance issues which are not easy to observe at a modeling level. The version of the Java 8 backend for ABS presented below does not support distributed execution.

Thread creation and scheduling. ABS contains constructs for the two finest levels of granularity in parallel computing, scheduling method calls within an object and scheduling object execution within a task. Cooperative scheduling used to be a major implementation challenge before Java 8, because a straightforward implementation would match a Java thread to each method call and a thread pool to each object. Hence, an asynchronous method call caused the creation of a new thread inside a thread pool along with the start of this thread executing the method. There is a mismatch because a Java object is a thread pool containing the threads originating from the methods invoked by that object. In ABS, however, at most one method executes on an object at any time, so creating a new thread for each fine-grained ABS method call is wasteful. It scaled badly due to the huge number of threads that occupy a large portion of the heap and seriously compromised the performance of previous Java backends for ABS.

Cooperative scheduling of active objects using Java 8. New features of Java 8 allow method calls to be wrapped in lightweight lambda expressions which can be put on the scheduling queue of an `ExecutorService` to which the running objects are mapped, significantly reducing the number of idle threads at runtime. Fig. 6 shows how active objects in this implementation are demand-driven and only create a physical thread once they have a method to execute inside their queue and the `ExecutorService` has a thread available in its pool. Finally, when all messages are blocked or the active object's queue is empty the thread is ended and released back to the pool of the `ExecutorService`. The impact of this approach is evaluated on an ABS benchmark comparing the performance of the Java 8 backend with the ProActive backend for ABS discussed in Section 3.4. The only drawback of the current implementation approach is that if a method call contains a recursive stack of synchronous calls, this stack needs to be saved when encountering an `await` statement, which cannot be realized with lambda expressions. To solve this problem, such a call stack can be modeled by continuation functions and then saved as lambda expressions. These lambda expressions are ordered such that, once released, the continuations can execute and maintain the correct logic of the program.

Data sharing and object referencing. In the Java 8 backend for ABS objects are organized into COGs, each running on one thread. When objects are created, they are assigned to a new or existing COG. All invocations on the object are executed on the thread of the COG to which the object is assigned. Data shared among distributed objects is passed through lambda expressions that are sent as serialized messages between the objects, so all data passed in a distributed system must be serializable. With COGs residing on the same machine, for example applications that run on a multi-core machine, all data is passed as arguments to lambda expressions or synchronous

[4] Available at https://github.com/vlad-serbanescu/abs-api-cwi/tree/LocalOnly

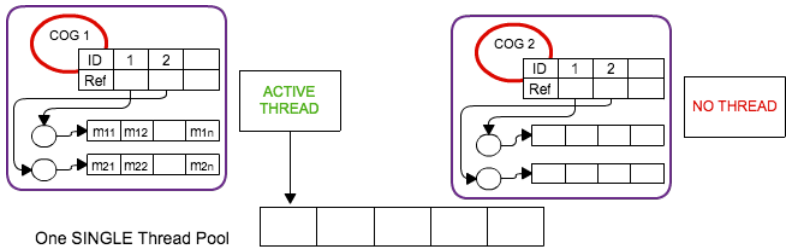


Fig. 6. Thread Creation and Scheduling

method calls in Java 8. Objects can keep references to any other object in any COG as inner fields. Similar to data sharing, objects must be serializable to be transferred between distributed objects. Furthermore, the generated classes will automatically be loaded on all machines that require an object of a particular type, such that remote objects can invoke methods on serialized references they receive.

Error handling. In the Java 8 backend, software errors are handled by Java's exception handling mechanism. All exceptions defined in ABS translate directly into extensions of the Exception class in Java. Furthermore ABS syntax for pattern matching exceptions translates directly to Java's try-catch mechanisms at compile time.

Garbage Collection. Garbage collection in the Java 8 backend only requires bookkeeping futures that lock messages on actors from separate COGs, so an actor responsible for completing a future can notify the awaiting actors to resume execution. Once this notification is completed, these references are deleted and the process of freeing memory is handled completely by Java's garbage collector.

3.3 ProActive

The ProActive library[5] is completely written in standard Java and provides an implementation for the ASP programming model targeting distributed applications. By default, it uses Java's RMI package to implement the communication layer between active objects, although other communication protocols are possible. This fact accounts for most of the implementation specifics mentioned below.

Thread creation and scheduling. In ProActive, an active object with no defined compatibility rule is mapped to a single Java thread to process the requests. Furthermore, there is a Java thread to handle request reception in each active object. Java threads are mapped to operating system threads, thus ProActive uses at least two threads per active object. But as ProActive features multi-active objects, this number can be higher, as the active object scheduler can create Java threads on the fly to process a compatible request. Since Java threads are rather heavy, the ProActive scheduler implementation makes a particular effort in optimizing thread usage. First, a thread pool is instantiated at active object startup to ensure a basic thread reuse policy. Second, when the number of threads is limited at the application level (through multi-active object annotations), that limit literally maps to Java threads, such that fine performance tuning can be achieved at the application level. In addition, threads waiting for a future can be temporarily reused to process the request that will indirectly resolve the awaited future. This can be done during wait-by-necessity. To summarize, in ProActive thread creation and thread scheduling are almost completely exposed to the programmer, allowing him to have far ranging control over the performance of ProActive applications.

[5]Available at https://github.com/scale-proactive

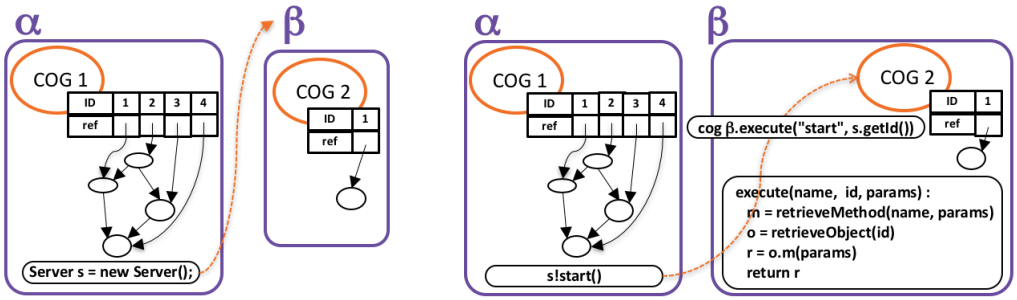
Data sharing and object referencing. Like ASP, ProActive differentiates between active and passive objects in a way that is mostly transparent to the programmer, except that passive objects are passed by copy when communicated between active objects, while references between active objects can be shared and accessed from anywhere. The first reason for this behavior is that ProActive is based on Java RMI, which in turn is based on parameter copy. The second reason is distribution: provision of a consistent distributed memory is too costly in High Performance Computing (HPC), is the primary target of ProActive. Of course, data is shared between the several threads of a multi-active objects. This sharing pattern is implemented using RMI and Java serialization performing a deep copy of the parameters transmitted between objects. Future and active objects are implemented by a proxy that can be passed by reference and is serialized as a reference (without any copy of other referenced objects). The serialization mechanism is also used to track multiple references to the same future and to implement efficient future update strategies [51].

Remote objects in RMI are globally referenced in the RMI registry, which maps remote object names to remote object stubs that can be copied and used anywhere to access the remote object. Networking communication is ensured by the RMI-JRMP protocol. Consequently, all active objects in ProActive are referenced in the RMI registry and can be accessed in this way from any other object through the adapted protocol. For non-active objects, traditional Java object references are used as they are only referenced within the same active object. To summarize, two types of object references exist in ProActive: global references, retrievable from the RMI registry, and local references, acting like standard reference types.

Error handling. Since ProActive targets distributed environments which are prone to failures, a particular effort was made to produce robust ProActive applications by including two error handling mechanisms in the ProActive library. First, an exception chaining mechanism developed on top of the basic RMI exception `RemoteException`, produces readable feedback when ProActive applications crash. Second, a specific API compensates for the difficulty of dealing with asynchronous exceptions and the lack of control over Java's exception mechanism [19].

A further aspect of the ProActive library concerns continuing to execute an application in presence of failed active objects, for example, when a machine hosting an active object of the application crashes. For this purpose ProActive implements a fault tolerant protocol specific to the active object semantics. It enables a set of failed active objects to restart from the latest checkpoint. Checkpoints are recorded per active object, based on the communications between them. This strategy is coupled with the logging of events received by the active object to ensure a deterministic re-execution. The fault-tolerant protocol is ready for applications that feature mono-threaded active objects. It is under development for applications that feature multi-active objects.

Garbage collection. Garbage collection in ProActive is tightly coupled to how objects are referenced. No particular mechanism is needed for the garbage collection of regular objects, which is handled by Java's garbage collector. Since a regular object cannot be referenced by more than one active object, no reference to this object exists in another JVM and standard garbage collection suffices. However, for an active object, we cannot directly know whether a reference to it still exists, because many proxies can be disseminated throughout the network. An adapted algorithm to garbage collect active objects was developed in ProActive which detects useless active objects (those that are idle and only referenced by idle active objects). This is handled by a form of common agreement based on the reference graph between active objects [20].



(a) ABS new in ProActive (b) ABS asynchronous method call in ProActive

Fig. 7. Representation of ABS objects and calls in ProActive

3.4 ProActive Backend for ABS

The ProActive backend for ABS[6] generates ProActive code for an ABS source program [54]. The goal of the ProActive backend is to provide a fully working execution of ABS models in distributed environments, with less optimization than necessary for the Java 8 backend. Since ABS and ASP are based on different active object models, the main challenges in the translation are (i) how to efficiently support object groups in ProActive where only active and passive objects are available, (ii) how to handle objects in the translation since objects are passed by reference in ABS and by copy in ProActive, and (iii) how to simulate cooperative scheduling with multi-threading controlled through annotations. We first present object referencing aspects, because the scheduling depends on the set of objects classified as active.

Data sharing and object referencing. All objects in ABS are accessible from all others (via suitable getter/setter methods); thus, one could implement each ABS object with a ProActive active object. In practice, this is infeasible: a ProActive active object has an associated plain Java thread. So this solution would inevitably lead to substantial memory consumption and context switch overhead. Instead, in the code generated by the ProActive backend only COGs are active objects and serve as the entry point to all objects they contain. This hierarchy implies that other objects than COGs are passive, preserving the performance of the ProActive backend. Thus, on the first level of the index hierarchy we have network-wide accessible COG objects. That mechanism is integrated in ProActive, as it is based on RMI. At the second index level we have locally accessible objects. That mechanism is implemented in the COG class with a map from object identifiers to object references. The translation introduces indirection through COGs that are accessible by remote reference. This configuration of objects is illustrated in Fig. 7a, where a COG 2 is created through a new `Server` object.

An asynchronous call in ABS is translated into a generic asynchronous method call on the COG serving as the index to the targeted object. Then a generic caller method of the COG retrieves the targeted object via a unique identifier and runs the desired method on it using reflection, see Fig. 7b. In ProActive passive objects are not shared between active objects. Therefore, a passive object is copied when it is a parameter of a remote method call, whereas parameters of method calls in ABS are passed by reference. When the translation performs a remote method call on a COG, all parameters get copied. This is not a problem for the identifier of the targeted object, the

[6]Available at: https://bitbucket.org/justinerochas/absfrontend-to-proactive

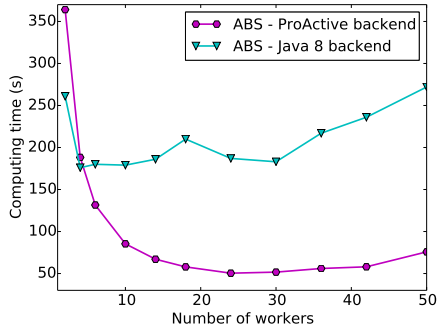
name of the method to run, and the primitive values, because they are immutable. For the reference type parameters the differing parameter passing semantics is still safe, but for a different reason.

While any access to an object is via an asynchronous method call on the copy of method parameters, those invocations end up calling the unique original version of the created object and their hosting COG. Thanks to this mechanism, data sharing in ABS is correctly simulated under the no-sharing philosophy of ProActive, because the copied data is only used as a reference to the original object. This means that, when we copy an object from one node to another, we only need the object's identifier plus a reference to its COG to retrieve it in the right memory space. Its other attributes can be omitted. This observation allows to optimize object copying and saves bandwidth. Consequently, the ProActive backend generates programs that copy less data, but incur more communication than native ProActive applications. Compared to the Java 8 backend for ABS, the ProActive backend does not distinguish whether COGs are located on the same machine to optimize communication, however, this is not a problem since only references are copied. Also, in the ProActive backend class loading is handled by RMI, whereas it is implemented from scratch in the Java 8 backend.

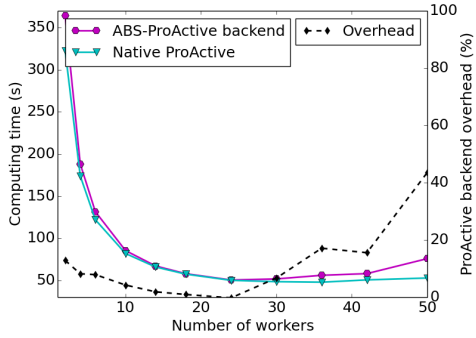
Thread creation and scheduling. Multi-active objects provide several mechanisms to control the scheduling of requests. Their proper tuning allows the ProActive backend to simulate the behavior of ABS cooperative scheduling. When comparing the ProActive backend and the Java 8 backend (Section 3.2), we can state that both deliver one message per asynchronous invocation, but in ProActive a single request queue exists for all objects in the same COG, whereas in the Java 8 backend, there is one request queue per object. The Java 8 backend ensures that all objects in the same COG compete for one single thread, whereas the ProActive backend creates one thread per executing (and paused) request: it is the scheduling derived from compatibility annotations that ensures at most one request is progressing at a time. This strategy avoids continuations which are difficult to implement in Java, because of JVM restrictions. Finally, the Java 8 backend shares a thread pool for many COGs whereas in the ProActive backend each COG has its own thread pool. To summarize, the ProActive backend has to handle more idle threads than the Java 8 backend.

The central idea to implement ABS preemption in ProActive is to exploit thread limits: a hard limit restricts the total number of threads for a multi-active object instance, a soft limit merely restricts the number of active threads. Now we first declare the generic caller method of a COG that executes all other methods to be compatible with itself, i.e., many activations of such methods are allowed to run in parallel. Then we impose on this set of methods a soft thread limit of one. Hence, there is at most one active thread at any time that serves requests in a COG multi-active object. Requests do not run in parallel but interleave using wait-by-necessity: `await` on a future is translated into access to a future, that frees the current active thread (soft limit). Blocking `get` statements are also translated to future accesses, but the current active thread is not released (a hard limit is ensured).

The ABS to ProActive translation illustrates the differences among these languages, especially regarding futures. In ProActive the synchronization is *data-flow oriented*, as futures are transparently accessed and updated, and wait-by-necessity can only be released upon data access. In ABS synchronization on futures is *control-flow oriented*: a future access succeeds upon request termination. However, in most programs a future access is used to retrieve information and the control-flow synchronization of ABS corresponds to a data-flow synchronization. Nevertheless, it is possible to write programs that do not perform data-flow synchronization and behave differently. We have formally proven [54] that all ABS programs are faithfully simulated by the ProActive backend.



(a) ProActive backend vs. Java 8 backend



(b) ProActive translation vs. hand-coded

Fig. 8. Execution time of DNA-matching ABS application

Example and experiments. We use a representative HPC application: pattern matching of a DNA sequence programmed in the MapReduce model [36]. The case study is computation-intensive and involves not a lot of active object communication. As ProActive relies on physical threads for active objects and data copying occurs in each communication, ProActive is better suited for computation-intensive scenarios than for communication-intensive ones. We compared performance of the code generated with the ProActive backend [79] for ABS run in distributed mode, the code generated with the Java 8 backend of ABS run on a single machine, and native code written manually in ProActive for the same algorithm. Map instances (workers) are created in their own COG. We search a pattern of 250 bytes inside a database of 5 MB. Each map searches for the maximum matching sequence of a chunk and a reducer outputs the global maximum matching sequence. We report the global execution time. When deploying with ProActive, we instantiate two workers on each machine (each machine has two dual-core CPUs). In the generated program, we manually replaced the translation of functional ABS types (integers, booleans, lists, maps) with standard Java types, similar to the code the Java 8 backend produces, because the translation does not handle ABS datatypes.

Fig. 8a shows the execution times of both ABS backends for the application ranging from 2 to 50 workers, and 1 to 25 physical machines with ProActive. The execution times of the ProActive backend are sharply decreasing for the first few added machines and then decrease at a slower rate. The first instance added in the Java 8 backend also improves significantly the performance of the

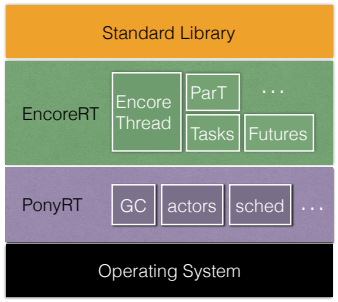


Fig. 9. Encore runtime stack

program and the Java 8 backend performs better with one or two workers. Thanks to the efficient thread management of the Java 8 backend, the performance stays stable until 30 workers. With a high number of instances, the degree of parallelism becomes harmful. In contrast, increasing the degree of parallelism for the ProActive backend results in a linear speedup, because it balances the load between machines and benefits from distribution.

Fig. 8b compares the performance of the generated ProActive code to a hand-written version. The overhead introduced by the translation performed by the ProActive backend is very low (< 10%), except when many machines are involved and the communication rate is high.

3.5 Encore

The Encore implementation[7] consists of two major components, a source-to-source compiler (from Encore to C) implemented in Haskell, and the runtime system, implemented in C. The runtime provides efficient parallel execution but has no support for distribution. The Encore runtime stack (Fig. 9) is built around the runtime used by the Pony language (PonyRT) [31], which includes an actor library and garbage collection for both active objects (actors) [30] and passive objects [29]. The Encore runtime (EncoreRT) extends the PonyRT with new features such as futures, the `ParT` abstraction, a task library, and the notion of *encore threads* (explained below). These features are part of the runtime and thus written in C. On top of these libraries rests the Encore Standard Library, which is written in Encore itself.

Thread creation and scheduling. In Encore, the runtime has schedulers whose responsibility is to schedule active objects. Each scheduler owns a queue of active objects and each active object owns its own mailbox, which contains messages to process.

On system start-up, the runtime maps physical cores to schedulers, saving the overhead of creating and context switching over a large number of threads, although this can be overridden if desired. Each scheduler runs in a loop, scheduling active objects until the whole program terminates. In each iteration of the loop, the scheduler performs three procedures. First, the scheduler pops an active object from the beginning of the queue. Second, it hands over control to the active object so that it can process messages in its mailbox. The active object can only process one message at a time to ensure fairness. Third, if a message is indeed processed in the previous step, the active object is pushed to the end of the scheduler's queue; otherwise, the active object is unscheduled due to having an empty mailbox. Unscheduled active objects are rescheduled when other active objects

[7] Available at https://github.com/parapluu/encore

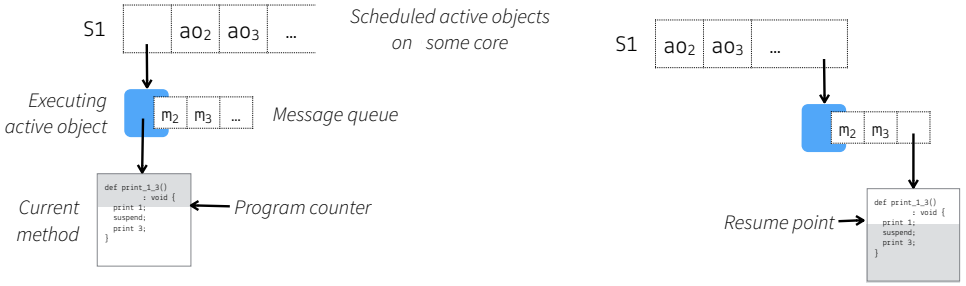


Fig. 10. An Encore scheduler, local to one core. (*Left*) A scheduler has a queue of active objects with non-empty message queues. (*Right*) The state after execution of `suspend` in the left configuration.

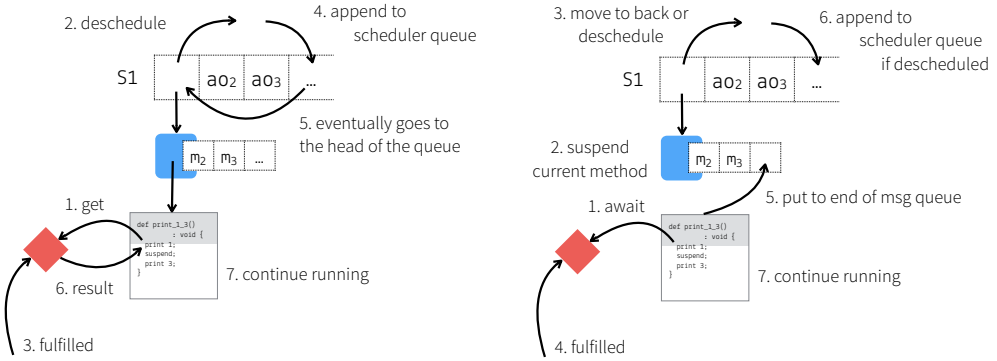


Fig. 11. (Left) An Encore program executing a `get` on an unresolved future. This causes the currently executing active object to be unscheduled, only to be rescheduled after the future has been resolved. Once the object gets to the head of the scheduling queue after this point, the `get` operation returns and the program continues from this point. with non-empty message queues. (Right) An Encore program executing an `await` on an unresolved future. In contrast to `get`, the active object is only unscheduled if it does not have any messages in its message queue, and once the future is resolved, the resuming method is placed at the end of the active object's queue.

send messages to them. Furthermore, in the second step, new active objects could be created, and they would be pushed to the end of queue of the current scheduler. Active objects can be migrated from one scheduler to another via work stealing, which realizes a load balancer; this only happens when a scheduler runs out of active objects.

Active objects in Encore are implemented by assigning them to a lightweight abstraction called *encore thread*, which resembles green threads. Encore threads provide a thread-of-control to the active object, so that it can process messages in its message queue. This implementation detail makes Encore capable of running thousands of active objects on the same machine.

The semantics of `suspend`, `await`, and `get` is illustrated by Figs. 10 and 11, which show how these operations involve *descheduling* (removed from round-robin scheduler queue) and *blocking* (no processing of other messages in the active object's queue) of an active object, and whether a resuming message is prepended or appended to the active object's message queue.

Unlike the ProActive backend for ABS that relies on creating new threads, Encore opts for handling stacks, which can be attached to encore threads. Based on this idea, the current stack is

put aside and a new stack is given to the encore thread, so that it can continue running some other or the same active object using the new stack. To make this process more efficient, Encore has a stack pool that allows reusing unused stacks. Upon resolution of the awaited future, the encore thread can continue processing the suspended execution and, when it has finished, the stack is collected and returned to the stack pool for future reuse.

Data sharing and object referencing. Active and passive objects in Encore are shared by reference, as opposed to ProActive which performs deep copying of the object (Section 3.3). In practice this means that sharing large objects in Encore poses no performance issues. Active objects run in an Encore thread which provides its own thread of control. In terms of the capability system, immutable passive objects are immune to this problem for obvious reasons. Following this line, a locked mutable passive object provides locking guarantees on the object, which prevents data races. In a slightly more advanced usage scenario, we might share part of the mutable passive object so that multiple active objects could work on different parts of the same passive object, concurrently. This would be safe as well. The integration of such shared passive objects with capability types is work in progress.

Error handling. Currently, Encore offers limited support for exception handling. Errors can be expressed using option types, where `Nothing` could represent the error, and programmers can pattern match on it.

Garbage collection. Encore borrows the garbage collector from PonyRT with the necessary extension to accommodate future values. Garbage collection consists of two parts, collecting active objects, which is covered by Clebsch and Drossopoulou [2013], and collecting passive objects [29].

3.6 Erlang Backend for Rebeca

Execution of Rebeca models is performed by translating Rebeca models to Erlang code. Erlang is a dynamically-typed general-purpose programming language for distributed, real-time and fault-tolerant applications with an actor-based concurrency model. Having the same concurrency model, translating Rebeca models to Erlang is realized by a direct mapping of language constructs [78].

Thread creation and scheduling. Actors, as the only concurrent elements of Rebeca models, are mapped to processes in Erlang, which are much lighter than OS-level threads (more than 100,000 of them can be run on a single computer [8]). The programming facilities of Erlang for developing concurrent applications (i.e., spawn, "!", and receive) allow processes to create new processes and to communicate through asynchronous message passing. These capabilities are used to translate Rebeca models to Erlang code without the need for modification. The generated processes are scheduled by the default, reduction-based Erlang scheduler.

Data sharing and object referencing. Reserving a dedicated memory space for each process in Erlang avoids sharing objects between processes. When object communication happens, the sent message and its parameters are stored in the receiver actor's message bag, located in its dedicated memory space. References to actors are permitted for the purpose of sending messages (although not for sharing data among actors).

Error handling. In the current version of Rebeca, there is no mechanism for exception handling; the programmer must rely on traditional messages to deal with errors.

Garbage collection. Rebeca borrows the default garbage collector of Erlang without any modification, which runs one garbage collector for each process, handling garbage collection internally within each actor.

	Rebeca	ABS	ASP/ProActive	Encore
Objective, Design features	• Actor-based modeling language • Simple computation and communication model • No dynamic object creation	• Active object modeling language • COGs • Combines functional and imperative features • Advanced communication model	• Active object programming language • Multi-active objects • Designed for distribution	• Active object programming language • Designed for scalability
Degree of synchronisation	• Asynchronous message passing only • No synchronization • Single-threaded run-to-completion of requests per actor	• Explicit futures • Advanced synchronization • Cooperative scheduling interleaves requests per COG	• Implicit futures • Wait-by-necessity synchronization • Multi-active objects allow parallel treatment of requests	• ABS-like cooperative scheduling • Future chaining • Multi-active objects with parallel combinators for data parallelism
Degree of transparency	• Hides asynchronous message passing • Hides scheduling and message queues	• Hides object representation, scheduling, and message queues • Exposes futures and release points	• Hides futures and distribution • Exposes parallel treatment of requests	• Hides scheduling and low-level parallelism • Exposes futures, release points, and parallel data flow
Degree of data sharing	• No shared values • All arguments passed by copy, even actor references	• No shared values • Objects passed by reference • Functional values passed by copy • Private fields	• No shared values • Active objects passed by reference • Passive objects passed by copy	• Active objects passed by reference • Capability types control data sharing for passive objects
Implementation and tooling support	• Model checking	• Simulation • Wide range of analysis tools • Code generation	• Implemented as a Java library • Supports distributed architectures	• Runtime system written in C • Supports multicore architectures

Fig. 12. The surveyed active object languages according to various design dimensions

4 DISCUSSION

4.1 Design Trade-Offs in Active Object Languages

The four languages reviewed in this paper start from the same programming model, the active object paradigm, but address different challenges and are motivated by different design goals. We review the different design trade-offs, summarized in Fig. 12.

Degree of synchronization. The degree of synchronization is a design choice that is central to the design of parallel programming languages. The original actor model has a semantics where communications are purely asynchronous and no strict synchronization exists. This resembles the principles of Rebeca. However, Rebeca ensures FIFO message communication to provide some guarantees on the ordering of operations. From a global perspective, synchronization hinders efficiency and parallelism, but makes programming easier, and sometimes more efficient. The other programming languages we discussed allow programs to synchronize on the result of a method invocation through the use of futures. However, different synchronization patterns exist on futures: from strict synchronization like the `get` operator of ABS to asynchronous reaction to future fulfillment enabled by future chaining in Encore.

Two synchronization patterns on futures presented in this article feature an interesting compromise: the `await` operation of ABS and Encore enables *cooperative multi-threading* and lets the program serve another request while the future is awaited; and the *wait-by-necessity synchronization* of ProActive that automatically waits upon the availability of useful data. Wait-by-necessity allows the program to pass futures around and never worry about imbricated futures. Different *communication paradigms* have been investigated in the proposed languages, the extreme cases being the causally ordered communications ensured by ASP which makes programming easier, because of the ordering guaranteed to the programmer, and the fully asynchronous communication of the ABS semantics, which allows more parallelism and better overlap between communication

and computation. Interestingly, many implementations of ABS ensure FIFO ordering of messages, even with asynchronous communication as the default semantics of ABS.

Degree of transparency. The degree of transparency is very much related to the synchronization primitives. Right now, in practice there are mostly two extreme approaches: either everything is very explicit and the user syntactically differentiates asynchronous invocations from synchronous ones (e.g., $o!m()$ vs. $o.m()$ in ABS) or everything is hidden to the programmer who programs active objects, as if they were sequential objects and where synchronization on method results is hidden to the programmer. In Rebeca, explicitness is even stronger as the remotely invocable methods are declared by a specific syntax and do not return a result.

Data sharing and data access. Active objects do not share memory according to the Actor programming paradigm; however, we see that at least some objects can be accessed by other objects. Data sharing plays an especially important role in relation to implementation. ASP, designed in close relation to Java RMI, because of the ProActive library implementation, features a non-uniform active object model where active objects can be invoked from anywhere and other objects are transmitted by copy. ABS was designed as a specification language where each object can be invoked by any other object; the first implementations of ABS did not support distribution, and the distributed implementations provide a hierarchical way of addressing objects in order to scale. Encore has a particular role here as it is focused on efficient local parallelism, where it is possible to have efficient addressing of many objects; this allows an implementation of passive objects where no data is copied and a better efficiency is provided.

It is interesting to notice that both ASP and Encore allow the programmer to depart from the pure actor model by providing local multi-threading to an actor, either inside the treatment of a request (the parallel combinators of Encore) or by running several requests in parallel (the multi-threaded active objects of ASP). In each case, the extension of the language is provided in such a way that it relies on the same programming abstractions as classical active objects and for this reason, it is still very easy to write safe and efficient programs. In both cases, multi-threading allows the programs to gain efficiency by better using local concurrency, it also allows the programmer to express some parallelism patterns that are difficult to write with classical active objects. It is important to notice that, even if the strict mono-threading policy of actors can be broken in those language, the programming language still strictly guarantees that no data race can exist between two actors.

Formal semantics. It is noteworthy that all four languages are equipped with a *formal semantics*, which is the basis for unambiguous implementations and for far reaching tool support in terms of static analysis tools, test generators, optimizers, etc. For several of the languages discussed here meta properties of the formal semantics were proven, such as type safety or data race-freedom.

Tooling and execution support. Both Rebeca and ABS started with a strong focus on tool-based formal analysis and verification, but recently added fairly efficient implementations which rely on cross translation. ProActive and Encore focused on efficient implementation on distributed and many-core computer architectures, respectively, and come with "native" implementations based on extensions of existing frameworks.

Rebeca is probably the language that features the largest variety in the different language semantics that can be verified. ABS in contrast is equipped with a large variety of different analyses based on the same semantics. Both languages show the advantage of adopting an active object programming paradigm compared to standard models of distribution and concurrency, for the verifiability of programs. ASP and Encore mainly focused on execution support. While powerful verification tools also exist for those languages, significant efforts have been put into the efficiency of local parallelism in Encore, and to support effective large-scale distribution in ProActive. ProActive

programs target HPC applications: it makes it easy to deploy an application on a large set of machines, but it reaches maximal efficiency when instantiating only a few active objects per physical core. The Java 8 backend for ABS provides light threads and enables the instantiation of many active objects on the same machine. It also permits active objects to achieve faster local interaction than in ProActive. At a finer granularity, Encore provides optimized constructs for safe and efficient programming at a lower level of abstraction.

The different execution environments discussed in this paper cover a broad range of ways to implement an active object language. A language can in principle be implemented from scratch with a whole compiler tool-chain but to offer a more efficient runtime support and to minimize implementation effort all the languages presented here rely on an existing, widely adopted language. Encore is probably the closest to a full implementation of a language and runtime support, as it compiles to C and requires a dedicated runtime environment. It is also probably the most efficient local implementation of active objects. Rebeca and ABS mostly rely on compilation to high-level languages, sometimes augmented with a dedicated API. ProActive takes a different approach and is implemented as a library for an existing language, allowing the programmer to use almost all the functionalities of the host language, albeit a few natural restrictions. The ProActive backend for ABS illustrates the possibility to cross-translate an active object language into another one.

4.2 Other Actor Languages

There are several other programming languages that introduce high-level abstractions for concurrent programming deviating from the active object approach. We focused in this survey on languages based on an active object model, however, numerous programming models with similar concepts that do not follow exactly the actor paradigm exist, for example, Seuss [76], SCOOP [73].

We give an overview of the main languages that adopt, at least partially, the concept of active objects and were not discussed so far. Java Annotations for Concurrency (JAC) [48] supports annotations to specify the kind of parallelism that can occur inside a Java object from a relatively high-level perspective. It is possible to encode versions of actors in JAC. The concurrency model of AmbientTalk [37] is based on the E programming language [74] which implements a communicating event-loop actor model with fully asynchronous futures (called promises): calls on futures trigger an asynchronous invocation that will be executed when the future is available and objects are passed by eventual reference between actors rather than by copy. This facilitates the creation of many small, object-level interfaces (each eventual reference acts as a new entry point to the actor), rather than a single large actor-level interface. AmbientTalk adds new primitives to handle disconnecting and reconnecting nodes in a network to support ambient-oriented programming. In Panini [69] concurrent behavior is specified by composing modules (called “capsules”) that by themselves behave sequentially. The granularity of concurrency is more coarse-grained than in active object languages and there are no explicit synchronization statements.

Akka [46, 91] is a scalable library for implementing actors on top of Java and Scala. Akka Typed Actors is an implementation of the Active Objects pattern. Interactions between actors in Akka only use message passing and all communication is asynchronous. Actors interact in the same way, regardless whether they are on the same or separate hosts, communicate directly or through routing facilities, run on a few threads or many threads, etc. Such details may be altered at deployment time through a configuration mechanism, allowing a program to make use of more (powerful) servers without modification. The scalability of Akka also results from an efficient hierarchical error handling mechanism where actors also play the role of supervisors. Akka is well suited for hybrid cloud architectures and the elastic scaling of cloud platforms. In one experiment it only took four minutes to start a 1000-node Akka cluster on Google Compute Engine (GCE), including the time to start the GCE instances.

Orleans [13, 18] is an actor framework developed at Microsoft research and used in several Microsoft products, including online games relying on a cloud infrastructure. Its strength is the runtime support for creation, execution, and state management of actors. Orleans relies on a non-uniform active object model with copies of transmitted passive objects, like ASP. The semantics of futures is based on continuations and relies on an `await` statement similar to that of ABS and Encore, however, there is no primitive for cooperative scheduling. Consequently, the programmer has to take care of possible modifications of the object state between the `await` and the future access. This semantics for future access is similar to the way futures are handled in general in Akka. There is extensive runtime support in the Orleans framework; in particular, it supports the efficient creation, activation, and deactivation of active objects governed by the requirements of an application. This is why Orleans is called a *virtual actor model*. Orleans also focuses on efficient execution of active objects, including optimized serialization for inter-actor data transmission. Both object management and data transmission scale in terms of distribution and in terms of the number of active objects that can interact within a single application.

5 LESSONS LEARNED AND CONCLUSION

With the advent of many-core computers and large-scale cloud computing, the active object paradigm evolved out of actor-based concurrency as one of the most promising candidates to model asynchronously parallel and distributed computations in a safe manner. We retraced the unfolding of the active object paradigm during the past fifteen years by focusing on four representative active object languages. We are convinced that active object concepts will eventually make their way into mainstream programming languages (the adoption of an actor system in Scala is a first indicator). Therefore, it is essential to understand the trade-offs involved in their design and implementation.

In this paper, we discuss and compare design decisions for four active object languages and show how the semantics of each language influenced its implementation, how different design decisions create new challenges, how these have been addressed, and which limitations remain. One important dimension along which the languages can be distinguished is how they handle efficient data sharing, and how each language finds a balance between copying and safe shared memory. We hope that the trade-offs and options discussed in this survey can serve as a future reference for designers and implementers of programming languages that make use of the active object paradigm.

REFERENCES

[1] Erika Ábrahám, Frank S. de Boer, Willem P. de Roever, and Martin Steffen. 2005. An assertion-based proof system for multithreaded Java. *Theoretical Computer Science* 331, 2-3 (2005), 251–290.

[2] Luca Aceto, Matteo Cimini, Anna Ingólfsdóttir, Arni Hermann Reynisson, Steinar Hugi Sigurdarson, and Marjan Sirjani. 2011. Modelling and Simulation of Asynchronous Real-Time Systems using Timed Rebeca. In *Proc. Workshop on Foundations of Coordination Languages and Software Architectures*, Mohammad Reza Mousavi and António Ravara (Eds.). Electronic Proceedings in Theoretical Computer Science, Vol. 58. 1–19.

[3] Gul Agha. 1986. *Actors: a model of concurrent computation in distributed systems.* MIT Press.

[4] Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, Germán Puebla, and Guillermo Román-Díez. 2014. SACO: Static Analyzer for Concurrent Objects. In *Proc. 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Erika Ábrahám and Klaus Havelund (Eds.). LNCS, Vol. 8413. Springer, 562–567.

[5] Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa. 2015. Test Case Generation of Actor Systems. In *Proc. 13th Intl. Symp. on Automated Technology for Verification and Analysis (ATVA)*, Bernd Finkbeiner, Geguang Pu, and Lijun Zhang (Eds.). LNCS, Vol. 9364. Springer, 259–275.

[6] Elvira Albert, Frank S. de Boer, Reiner Hähnle, Einar Broch Johnsen, Rudolf Schlatte, Silvia Lizeth Tapia Tarifa, and Peter Y. H. Wong. 2014. Formal Modeling of Resource Management for Cloud Architectures: An Industrial Case Study using Real-Time ABS. *Journal of Service-Oriented Computing and Applications* 8, 4 (2014), 323–339.

[7] Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel. 2016. Combining Static Analysis and Testing for Deadlock Detection. In *Proc. 12th Intl. Conf. on Integrated Formal Methods (iFM 2016)*, Erika Ábrahám and Marieke Huisman (Eds.). LNCS, Vol. 9681. Springer, 409–424.

[8] Joe Armstrong. 2007. *Programming Erlang, Software for Concurrent World.* Pragmatic Bookshelf.

[9] Henry. G. Baker Jr. and Carl Hewitt. 1977. The Incremental Garbage Collection of Processes. In *Proc. Symp. on Artificial Intelligence and Programming Languages.* ACM Press, New York, NY, USA, 55–59.

[10] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. 1989. Programming Languages for Distributed Computing Systems. *Comput. Surveys* 21, 3 (1989), 261–322.

[11] Françoise Baude, Ludovic Henrio, and Cristian Ruz. 2015. Programming distributed and adaptable autonomous components—the GCM/ProActive framework. *Software Prac. Experience* 45, 9 (2015), 1189–1227.

[12] Razieh Behjati, Marjan Sirjani, and Majid Nili Ahmadabadi. 2009. Bounded Rational Search for On-the-Fly Model Checking of LTL Properties. In *Proc. Third IPM Intl. Conf. on Fundamentals of Software Engineering (FSEN 2009)*, Farhad Arbab and Marjan Sirjani (Eds.). LNCS, Vol. 5961. Springer, 292–307.

[13] Philip A. Bernstein and Sergey Bykov. 2016. Developing Cloud Services Using the Orleans Virtual Actor Model. *IEEE Internet Computing* 20, 5 (2016), 71–75.

[14] Joakim Bjørk, Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. 2013. User-defined Schedulers for Real-Time Concurrent Objects. *Innovations in Systems and Software Engineering* 9, 1 (2013), 29–43.

[15] Stefan Blom and Marieke Huisman. 2014. The VerCors Tool for Verification of Concurrent Programs. In *Proc. 19th Intl. Symposium on Formal Methods (FM 2014)*, Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun (Eds.). LNCS, Vol. 8442. Springer, 127–131.

[16] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I. Pun, S. Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. 2015. Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore. In *Formal Methods for Multicore Programming: 15th Intl. School on Formal Methods for the Design of Computer, Communication, and Software Systems*, Marco Bernardo and Einar Broch Johnsen (Eds.). LNCS, Vol. 9104. Springer, 1–56.

[17] Richard Bubel, Antonio Flores Montoya, and Reiner Hähnle. 2014. Analysis of Executable Software Models. In *Executable Software Models: 14th Intl. School on Formal Methods for the Design of Computer, Communication, and Software Systems*, Marco Bernardo, Ferruccio Damiani, Reiner Hähnle, Einar Broch Johnsen, and Ina Schaefer (Eds.). LNCS, Vol. 8483. Springer, 1–27.

[18] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: cloud computing for everyone. In *Proc. Symposium on Cloud Computing in conjunction with SOSP, SOCC*, Jeffrey S. Chase and Amr El Abbadi (Eds.). ACM Press, 16.

[19] Denis Caromel and Guillaume Chazarain. 2005. Robust exception handling in an asynchronous environment. ECOOP Workshop on Exception Handling in Object-Oriented Systems. (2005).

[20] Denis Caromel, Guillaume Chazarain, and Ludovic Henrio. 2007. Garbage Collecting the Grid: a Complete DGC for Activities. In *Proc. 8th ACM/IFIP/USENIX Intl. Middleware Conference*, Renato Cerqueira and Roy H. Campbell (Eds.). LNCS, Vol. 4834. Springer, 164–183.

[21] Denis Caromel and Ludovic Henrio. 2005. *A Theory of Distributed Object.* Springer.

[22] Denis Caromel, Ludovic Henrio, and Bernard Serpette. 2004. Asynchronous and deterministic objects. In *Proc. Symp. on Principles of Programming Languages (POPL'04)*, Neil D. Jones and Xavier Leroy (Eds.). ACM Press, 123–134.

[23] Denis Caromel and Yves Roudier. 1996. Reactive Programming in Eiffel//. In *Proc. Conf. on Object-Based Parallel and Distributed Computation (OBPDC '95)*, Jean-Pierre Briot, Jean-Marc Geib, and Akinori Yonezawa (Eds.). LNCS, Vol. 1107. Springer, 125–147.

[24] Elias Castegren and Tobias Wrigstad. 2016. Reference Capabilities for Concurrency Control. In *European Conference on Object-Oriented Programming (ECOOP'16)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). LIPIcs, Vol. 56. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 5:1–5:26.

[25] Elias Castegren and Tobias Wrigstad. 2017. Relaxed Linear References for Lock-free Data Structures. In *European Conference on Object-Oriented Programming (ECOOP'17)*, Peter Müller (Ed.). LIPIcs, Vol. 74. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 6:1–6:32. https://doi.org/10.4230/LIPIcs.ECOOP.2017.6

[26] Charron-Bost, Bernadette, Mattern, Friedemann, and Gerard Tel. 1996. Synchronous, asynchronous, and causally ordered communication. *Distributed Computing* 9 (1996), 173–191. Issue 4.

[27] Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. 2008. Minimal Ownership for Active Objects. In *Proc. 6th Asian Symposium on Programming Languages and Systems (APLAS'08)*, Ganesan Ramalingam (Ed.). LNCS, Vol. 5356. Springer, 139–154.

[28] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. 2007. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic.* LNCS, Vol. 4350. Springer.

[29] Sylvan Clebsch, Sebastian Blessing, Juliana Franco, and Sophia Drossopoulou. 2015. Ownership and reference counting based garbage collection in the actor world. (2015). In *10th Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems Workshop (ICOOOLPS)*.

[30] Sylvan Clebsch and Sophia Drossopoulou. 2013. Fully concurrent garbage collection of actors on many-core machines. In *Proc. ACM SIGPLAN Intl. Conf. on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM Press, 553–570.

[31] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE!)*, Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos Varela (Eds.). ACM Press, 1–12.

[32] Gianpaolo Cugola and Carlo Ghezzi. 1997. CJava: Introducing Concurrent Objects in Java. In *4th Intl. Conf. on Object Oriented Information Systems (OOIS'97)*, Maria E. Orlowska and Roberto Zicari (Eds.). Springer, 504–514.

[33] Ole-Johan Dahl, Bjørn Myrhaug, and Kristen Nygaard. 1968. *(Simula 67) Common Base Language.* Technical Report S-2. Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway.

[34] Ole-Johan Dahl and Kristen Nygaard. 1966. SIMULA - an ALGOL-based simulation language. *Commun. ACM* 9, 9 (1966), 671–678.

[35] Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. 2007. A complete guide to the future. In *Proc. 16th European Symposium on Programming (ESOP'07)*, Rocco De Nicola (Ed.). LNCS, Vol. 4421. Springer, 316–330.

[36] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113.

[37] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. 2006. Ambient-Oriented Programming in AmbientTalk. In *European Conference on Object-Oriented Programming (ECOOP'06)*, Dave Thomas (Ed.). LNCS, Vol. 4067. Springer, 230–254.

[38] Crystal Chang Din, Richard Bubel, and Reiner Hähnle. 2015. KeY-ABS: A Deductive Verification Tool for the Concurrent Modelling Language ABS. In *Automated Deduction (CADE-25)*, Amy P. Felty and Aart Middeldorp (Eds.). LNCS, Vol. 9195. Springer, 517–526.

[39] Crystal Chang Din and Olaf Owe. 2015. Compositional reasoning about active objects with shared futures. *Formal Aspects of Computing* 27, 3 (2015), 551–572.

[40] Crystal Chang Din, S. Lizeth Tapia Tarifa, Reiner Hähnle, and Einar Broch Johnsen. 2015. History-Based Specification and Verification of Scalable Concurrent and Distributed Systems. In *Proc. 17th Intl. Conf. on Formal Engineering Methods (ICFEM)*, Michael Butler, Sylvain Conchon, and Fatiha Zaïdi (Eds.). LNCS, Vol. 9407. Springer.

[41] Jesús Doménech, Samir Genaim, Einar Broch Johnsen, and Rudolf Schlatte. 2017. EasyInterface: A Toolkit for Rapid Development of GUIs for Research Prototype Tools. In *Proc. 20th Intl. Conf. on Fundamental Approaches to Software Engineering (FASE 2017)*, Marieke Huisman and Julia Rubin (Eds.). LNCS, Vol. 10202. Springer, 379–383.

[42] Kiko Fernandez-Reyes, Dave Clarke, and Daniel S. McCain. 2016. ParT: an asynchronous parallel abstraction. In *Proc. 18th Intl. Conf. on Coordination Models and Languages (COORDINATION)*, Alberto Lluch Lafuente and José Proença (Eds.). LNCS, Vol. 9686. Springer, 101–120.

[43] Elena Giachino, Cosimo Laneve, and Michael Lienhardt. 2016. A framework for deadlock detection in core ABS. *Software and System Modeling* 15, 4 (2016), 1013–1048. https://doi.org/10.1007/s10270-014-0444-y

[44] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. 2006. *Java Concurrency in Practice.* Addison-Wesley.

[45] Reiner Hähnle, Michiel Helvensteijn, Einar Broch Johnsen, Michael Lienhardt, Davide Sangiorgi, Ina Schaefer, and Peter Y. H. Wong. 2013. HATS Abstract Behavioral Specification: the Architectural View. In *Formal Methods for Components and Objects (FMCO'12)*, Bernhard Beckert, Ferruccio Damiani, Frank de Boer, and Marcello M. Bonsangue (Eds.). LNCS, Vol. 7542. Springer, 109–132.

[46] Philipp Haller and Martin Odersky. 2009. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science* 410, 2-3 (2009), 202–220.

[47] Robert H. Halstead, Jr. 1985. MULTILISP: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems* 7, 4 (Oct. 1985), 501–538.

[48] Max Haustein and Klaus-Peter Löhr. 2006. JAC: declarative Java concurrency. *Concurrency and Computation: Practice and Experience* 18, 5 (2006), 519–546.

[49] Ludovic Henrio, Fabrice Huet, and Zsolt István. 2013. Multi-threaded Active Objects. In *15th Intl. Conf. on Coordination Models and Languages, Florence, Italy (COORDINATION)*, Christine Julien and Rocco De Nicola (Eds.). LNCS, Vol. 7890. Springer, 90–104.

[50] Ludovic Henrio, Florian Kammüller, and Bianca Lutz. 2012. ASPfun : A typed functional active object calculus. *Science of Computer Programming* 77, 7-8 (July 2012), 823–847.

[51] Ludovic Henrio, Muhammad Uzair Khan, Nadia Ranaldo, and Eugenio Zimeo. 2010. First Class Futures: Specification and implementation of Update Strategies. In *Selected Papers Coregrid Workshop On Grids, Clouds and P2P Computing*,

Mario R. Guarracino, Frédéric Vivien, Jesper Larsson Träff, Mario Cannataro, Marco Danelutto, Anders Hast, Francesca Perla, Andreas Knüpfer, Beniamino Di Martino, and Michael Alexander (Eds.). LNCS, Vol. 6586. Springer, 295–303.

[52] Ludovic Henrio, Oleksandra Kulankhina, Siqi Li, and Eric Madelaine. 2016. Integrated environment for verifying and running distributed components. In *Proc. Intl. Conf. on Fundamental Approaches to Software Engineering (FASE)*, Perdita Stevens and Andrzej Wasowski (Eds.). LNCS, Vol. 9633. Springer, 66–83.

[53] Ludovic Henrio and Justine Rochas. 2014. Declarative Scheduling for Active Objects. In *Proc. Symp. on Applied Computing (SAC)*, Sung Y. Shin (Ed.). ACM Press, 1339–1344.

[54] Ludovic Henrio and Justine Rochas. 2016. From Modelling to Systematic Deployment of Distributed Active Objects. In *Proc. Coordination Models and Languages: 18th Intl. Conf. (COORDINATION)*, Alberto Lluch Lafuente and José ProenÃğa (Eds.). LNCS, Vol. 9686. Springer.

[55] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proc. 3rd Intl. Joint Conference on Artificial Intelligence (IJCAI)*, Nils J. Nilsson (Ed.). W. Kaufmann, 235–245.

[56] Highly Adaptable and Trustworthy Software using Formal Models 2011. Full ABS Modeling Framework. (March 2011). Deliverable 1.2 of project FP7-231620 (HATS), available at http://www.hats-project.eu.

[57] Tony Hoare. 1985. *Communicating Sequential Processes.* Prentice Hall.

[58] Hossein Hojjat, Marjan Sirjani, Mohammad Reza Mousavi, and Jan Friso Groote. 2007. Sarir: A Rebeca to mCRL2 Translator. In *7th Intl. Conf. on Application of Concurrency to System Design (ACSD)*, Twan Basten, Gabriel Juhás, and Sandeep K. Shukla (Eds.). IEEE Press, 216–222.

[59] Ali Jafari, Ehsan Khamespanah, Marjan Sirjani, and Holger Hermanns. 2014. Performance Analysis of Distributed and Asynchronous Systems using Probabilistic Timed Actors. *Electronic Communication of the European Association of Software Science and Technology* 70 (2014).

[60] Mohammad Mahdi Jaghoori, Marjan Sirjani, Mohammad Reza Mousavi, Ehsan Khamespanah, and Ali Movaghar. 2010. Symmetry and partial order reduction techniques in model checking Rebeca. *Acta Informatica Journal* 47, 1 (2010), 33–66.

[61] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. 2011. ABS: A Core Language for Abstract Behavioral Specification. In *Formal Methods for Components and Objects (FMCO'10)*, Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue (Eds.). LNCS, Vol. 6957. Springer, 142–164.

[62] Einar Broch Johnsen and Olaf Owe. 2007. An Asynchronous Communication Model for Distributed Concurrent Objects. *Software and System Modeling* 6, 1 (March 2007), 35–58.

[63] Einar Broch Johnsen, Olaf Owe, and Marte Arnestad. 2003. Combining Active and Reactive Behavior in Concurrent Objects. In *Proc. of the Norwegian Informatics Conference (NIK'03)*, Dag Langmyhr (Ed.). Tapir Academic Publisher, 193–204.

[64] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. 2006. Creol: a type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science* 365, 1 (2006), 23–66.

[65] Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. 2015. Integrating deployment architectures and resource consumption in timed object-oriented models. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 67–91.

[66] Ehsan Khamespanah, Marjan Sirjani, Mohammad Reza Mousavi, Zeynab Sabahi-Kaviani, and Mohamadreza Razzazi. 2015. State Distribution Policy for Distributed Model Checking of Actor Models. *Electronic Communication of the European Association of Software Science and Technology* 72 (2015).

[67] Ehsan Khamespanah, Marjan Sirjani, Zeynab Sabahi-Kaviani, Ramtin Khosravi, and Mohammad-Javad Izadi. 2015. Timed Rebeca schedulability and deadlock freedom analysis using bounded floating time transition system. *Science of Computer Programming* 98 (2015), 184–204.

[68] Ehsan Khamespanah, Marjan Sirjani, Mahesh Viswanathan, and Ramtin Khosravi. 2015. Floating Time Transition System: More Efficient Analysis of Timed Actors. In *Proc. 12th Intl. Symp. on Formal Aspects of Component Software (FACS)*, Christiano Braga and Peter Csaba Ölveczky (Eds.). LNCS, Vol. 9539. Springer, 237–255.

[69] Eric Lin and Hridesh Rajan. 2013. Panini: a capsule-oriented programming language for implicitly concurrent program design. In *Conf. on Systems, Programming, and Applications: Software for Humanity (SPLASH)*, Antony L. Hosking and Patrick Eugster (Eds.). ACM Press, 19–20.

[70] Barbara Liskov and Ljuba Shrira. 1988. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proc. Conf. of Programming Language design and Implementation (PLDI)*, Richard L. Wexelblat (Ed.). ACM Press, New York, NY, USA, 260–267.

[71] Daniel McCain. 2015. *Parallel Combinators for the Encore Programming Language.* Master's thesis. Uppsala University.

[72] Bertrand Meyer. 1992. Applying "Design by Contract". *IEEE Computer* 25, 10 (Oct. 1992), 40–51.

[73] Bertrand Meyer. 1993. Systematic Concurrent Object-oriented Programming. *Commun. ACM* 36, 9 (Sept. 1993), 56–80. https://doi.org/10.1145/162685.162705

[74] Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. 2005. Concurrency among strangers: Programming in E as plan coordination. In *Proc. Intl. Symp. on Trustworthy Global Computing (TGC'05)*, Rocco De Nicola and Davide Sangiorgi (Eds.). LNCS, Vol. 3705. Springer, 195–229.
[75] Robin Milner. 1989. *Communication and Concurrency.* Prentice Hall. ISBN 0-13-114984-9.
[76] Jayadev Misra. 2001. *A Discipline of Multiprogramming: Programming Theory for Distributed Applications.* Springer, Secaucus, NJ, USA.
[77] Niloofar Razavi, Razieh Behjati, Hamideh Sabouri, Ehsan Khamespanah, Amin Shali, and Marjan Sirjani. 2010. Sysfier: Actor-based formal verification of SystemC. *ACM Transactions on Embedded Computing Systems* 10, 2 (2010), 19.
[78] Arni Hermann Reynisson, Marjan Sirjani, Luca Aceto, Matteo Cimini, Ali Jafari, Anna Ingólfsdóttir, and Steinar Hugi Sigurdarson. 2014. Modelling and simulation of asynchronous real-time systems using Timed Rebeca. *Science of Computer Programming* 89 (2014), 41–68.
[79] Justine Rochas and Ludovic Henrio. 2014. *A ProActive Backend for ABS: from Modelling to Deployment.* Research Report RR-8596. INRIA. https://hal.inria.fr/hal-01065072
[80] Zeynab Sabahi-Kaviani, Ramtin Khosravi, Peter Csaba Ölveczky, Ehsan Khamespanah, and Marjan Sirjani. 2015. Formal semantics and efficient analysis of Timed Rebeca in Real-Time Maude. *Science of Computer Programming* 113 (2015), 85–118.
[81] Hamideh Sabouri and Ramtin Khosravi. 2013. Modeling and Verification of Reconfigurable Actor Families. *Journal of Universal Computer Science* 19, 2 (2013), 207–232.
[82] Jan Schäfer and Arnd Poetzsch-Heffter. 2010. JCoBox: Generalizing Active Objects to Concurrent Components. In *European Conference on Object-Oriented Programming (ECOOP'10)*, Theo D'Hondt (Ed.). LNCS, Vol. 6183. Springer, 275–299.
[83] Vlad Serbanescu, Chetan Nagarajagowda, Keyvan Azadbakht, Frank de Boer, and Behrooz Nobakht. 2014. Towards Type-Based Optimizations in Distributed Applications Using ABS and JAVA 8. In *First International Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, Florin Pop and Maria Potop-Butucaru (Eds.). LNCS, Vol. 8907. Springer, 103–112.
[84] Steinar Hugi Sigurdarson, Marjan Sirjani, Yngvi Björnsson, and Arni Hermann Reynisson. 2012. Guided Search for Deadlocks in Actor-Based Models. In *Proc. 9th Intl. Symp. on Formal Aspects of Component Software (FACS)*, Corina S. Pasareanu and Gwen Salaün (Eds.). LNCS, Vol. 7684. Springer, 242–259.
[85] Marjan Sirjani, Frank S. de Boer, Ali Movaghar, and Amin Shali. 2005. Extended Rebeca: A Component-Based Actor Language with Synchronous Message Passing. In *Proc. Fifth Intl. Conf. on Application of Concurrency to System Design (ACSD)*, Jörg Desel and Yosinori Watanabe (Eds.). IEEE Press, 212–221.
[86] Marjan Sirjani, Frank S. de Boer, and Ali Movaghar-Rahimabadi. 2005. Modular Verification of a Component-Based Actor Language. *Journal of Universal Computer Science* 11, 10 (2005), 1695–1717.
[87] Marjan Sirjani and Mohammad Mahdi Jaghoori. 2011. Ten Years of Analyzing Actors: Rebeca Experience. In *Formal Modeling: Actors, Open Systems, Biological Systems*, Gul Agha, Olivier Danvy, and José Meseguer (Eds.). LNCS, Vol. 7000. Springer, 20–56.
[88] Marjan Sirjani, Movaghar, and Mohammad Reza Mousavi. 2001. Compositional Verification of an Object-Based Model for Reactive Systems. In *Proc. 11th Computer Science of Iran Computer Conference (CSICC)*.
[89] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. 2004. Modeling and Verification of Reactive Systems using Rebeca. *Fundamenta Informaticae* 63, 4 (2004), 385–410.
[90] Peter Y. H. Wong, Elvira Albert, Radu Muschevici, José Proença, Jan Schäfer, and Rudolf Schlatte. 2012. The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *Journal on Software Tools for Technology Transfer* 14, 5 (2012), 567–588.
[91] Derek Wyatt. 2013. *Akka Concurrency.* Artima.
[92] Yasuhiko Yokote and Mario Tokoro. 1987. Concurrent Programming in ConcurrentSmalltalk. In *Object-Oriented Concurrent Programming*, Akinori Yonezawa and Mario Tokoro (Eds.). MIT Press, 129–158.
[93] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. 1986. Object-Oriented Concurrent Programming in ABCL/1. In *ACM International Conference on Object-Oriented Programming, Systems, Languages and Applications(OOPSLA'86)*, Norman K. Meyrowitz (Ed.). ACM Press, 258–268.
[94] Behnaz Yousefi, Fatemeh Ghassemi, and Ramtin Khosravi. 2015. Modeling and Efficient Verification of Broadcasting Actors. In *Proc. 6th Intl. Conf. on Fundamentals of Software Engineering (FSEN 2009)*, Mehdi Dastani and Marjan Sirjani (Eds.). LNCS, Vol. 9392. Springer, 69–83.
[95] Behnaz Yousefi, Fatemeh Ghassemi, and Ramtin Khosravi. 2016. Modeling and Efficient Verification of Wireless Ad hoc Networks. *Formal Aspects of Computing* (2016). http://arxiv.org/abs/1604.07179 To appear.

Received November 2016; revised May 2017; accepted July 2017