Modeling and analyzing resource-sensitive actors: A tutorial introduction*

Rudolf Schlatte, Einar Broch Johnsen, Eduard Kamburjan, and S. Lizeth Tapia Tarifa

University of Oslo, Oslo, Norway,
{rudi,einarj,eduard,sltarifa}@ifi.uio.no

Abstract. Actor languages decouple communication from synchronization, which makes them suitable for distributed and scalable applications with flexible synchronization patterns, but also facilitates analysis. ABS is a timed actor-based modeling language which supports cooperative scheduling and the specification of timing- and resource-sensitive behavior. Cooperative scheduling allows a process which is executing in an actor to be suspended while it is waiting for an event to occur, such that another process in the same actor can execute. Timed semantics allows the specification of the temporal behavior of the modeled system. Resource-sensitive behavior takes a supply and demand perspective of execution, relating cost centers which provision resources to processes which require them. These modeling concepts have been used in ABS to model cloud computing, e.g., data-processing applications running on the Hadoop platform and micro-services running on containers orchestrated by Kubernetes. In this tutorial, we present ABS and its execution environment, and discuss the use of cooperative scheduling and resources in modeling cyber-physical systems and applications deployed on virtualized infrastructure.

Keywords: Resource-sensitive behavior · Cloud computing · Distributed actor systems

1 Introduction

Models of modern distributed applications often require that we not only describe the interactions between endpoints, but also precisely specify how the application uses system resources and how its timed behavior depends on these resources. In contexts such as cloud systems, the resource usage may be the critical property to be analyzed with the model. The resource model, the concurrency model and the performed computations all interact and cannot be analyzed in isolation. To model systems that incorporate all these aspects, we need modeling languages that are rich enough to express (a) timed behavior, (b) resource usage, (c) interactions between distributed components, and (d) complex computations.

* This research is supported by the Research Council of Norway via the SIRIUS research center (Grant 237898).

This tutorial shows how these aspects of distributed applications can be combined in resource-sensitive models using the *Abstract Behavioral Specification* (ABS) language [19], which has been successfully applied in several industrial case studies of such systems (see Sec. 7). ABS is based on object-oriented programming principles. Its basics are easy to learn for anybody with basic programming experience in mainstream programming languages, such as Java. This tutorial gives a brief language overview and mainly focuses on the usage of its advanced features for modeling timed and resource-sensitive behavior. For a full overview of the language, we refer to the user manual [1].

Beyond an overview over the modeling features for timed and resource-sensitive behavior, we also address the practical questions of usage. We describe how to use the compiler and simulator, as well as additional features such as the built-in visualization and debugging capabilities: how to compile ABS models, how to run them, how to visualize their executions as interleaved scheduling decisions, and how to replay a specific scheduling. We suggest useful patterns for distributed resource modeling.

All example files in this paper are available at https://github.com/abstools/absexamples/tree/master/collaboratory/tutorials/discotec-2021. For a detailed tutorial on other aspects of ABS, we refer to Hähnle [17] on the layered semantics of ABS and to Clarke et al. [11] for variability modeling.

Overview. Sec. 2 gives an overview of ABS and shows how to compile and execute a simple example. Sec. 3 goes into more depth on how to specify time and resource behavior, and Sec. 4 shows some typical patterns of larger distributed systems. Sec. 5 discusses visualization and external control of simulations, while Sec. 6 shows how to deterministically record and replay simulations. Finally, Sec. 7 gives pointers to further case studies and tools. Instructions on how to download and use the compiler toolchain of ABS can be found in App. A.

2 The ABS Language

ABS decouples communication and synchronization of processes running on actors via its use of (1) asynchronous method calls, (2) futures and (3) cooperative scheduling. *Asynchronous* method calls do not synchronize between caller and callee. Instead, the caller receives a *future*, which resolves once their callee process terminates later. Futures, thus, allow synchronization between processes on different actors (objects). Futures are first-class constructs in ABS, i.e., they can be explicitly represented and manipulated. Different processes inside one actor synchronize using *cooperative scheduling*: on each actor, only one process runs at a time, with scheduling happening at explicit release points. Schedulability of processes at the release point may depend on the state of a future, or on a Boolean condition over the state of the object the process runs on. ABS also includes a side-effect-free *functional layer* that is used to define functions and algebraic datatypes. This section presents ABS by way of example; for detailed reference, see the manual [1].

2.1 A Simple ABS Model

```
1 module Hello;
2 interface Hello {  Bool printMessage(); }
3
4 class Hello(String name) implements Hello {
5     Bool printMessage() { println(`Hello $name$!`);  return True; }
  }
6 // Main block:
7 {   Hello hello = new Hello("world");
8     Fut<Bool> f = hello!printMessage();
9     await f?;
10    Bool r = f.get;
11    println("Main block finished."); }
```

Fig. 1: A small ABS model

Figure 1 shows a very small ABS model. All ABS models consist of one or more `module` definitions, containing zero or more definitions of interfaces, classes, functions, or datatypes. An optional *main block* defines the behavior of a model.

The example defines an interface (Line 2) followed by a class implementing that interface (Line 4). (Note that since classes are not types in ABS, classes and interfaces may share the same name.) The main block starting at Line 7 first creates an object, then asynchronously sends it a `printMessage` and awaits for (i.e., synchronizes on) the resulting process to terminate, and finally retrieves the value stored in the future before printing a message. When removing Lines 9 and 10, the order of the two lines of output is nondeterministic. This is because the two `print` statements are executed in processes running on different actors (but usually the message from the main block will be printed first).

ABS models can be compiled to execute on Erlang with the command `absc -erlang` *`filename`*`.abs`. After compilation, models can be simulated using the command `gen/erl/run`.

Observe that during the execution of Lines 8–10 the message `printMessage()` is put in the message pool of the object `hello`, then the `await` statement *suspends* the caller's execution process (in this case the main block) until the return value of that method `printMessage()` has been processed and stored in the future `f`. At this point, the calling process becomes schedulable again, and proceeds to retrieve the value stored in the future using a `get` expression. (A `get` expression without a preceding `await` *blocks* the process until the future is resolved, i.e., no other process will be scheduled.)

Suspending a process means that the object can execute other messages from its message pool in the meantime. ABS demands that each process suspends itself explicitly via `await` or termination. I.e., it is *not* possible that two processes

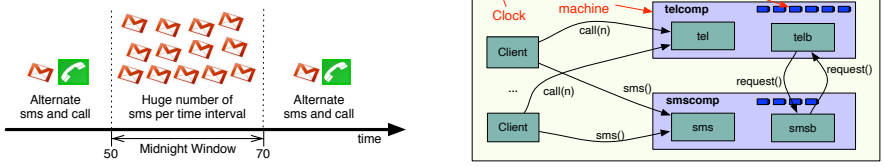
(if the method is called twice) executing `printMessage` overlap. The method `printMessage` contains no `await` statement, so one process starts, prints and terminates before another one does the same.

ABS supports the shorthand `await o!m(args)` (where `o` is an object, `m` a method and `arg` the arguments to the method) for the statements `f = o!m(args); await f?;` and `o.m(args)` for `f = o!m(args); r = f.get;` which *blocks* the caller object (does not release control) until the future `f` has been resolved (i.e., the future has received the return value from the call). The statement `this.m(args)` models a synchronous self-call, which corresponds to a standard call to a subroutine (no blocking mechanism).

3 Specifying Time and Resource Behavior

Timed ABS [9] is an extension to the core ABS language that introduces a notion of abstract time. It allows the modeling of an explicit passage of time, to represent execution time inside methods. This modeling abstraction can be used to model, in addition to the functional behavior, the timing-related behavior of real systems running on real hardware. In contrast to real systems, time in an ABS model does not advance by itself. Instead, it is expressed in terms of a `duration` statement (as in, e.g., UPPAAL [27]). Timed ABS adds a clock to the language semantics that advances in response to `duration` statements, as in Line 7 in Fig. 4. Time is expressed as a rational number, so the clock can advance in infinitesimally small steps.

Deployment is modelled using *deployment components* [22]. A deployment component is a modelling abstraction that captures locations offering (restricted) computing *resources*. ABS also supports cost annotations associated to statements to model resource consumption, as in Line 6 in Fig. 4. The combination of deployment components with computing resources and cost annotations allows modeling implicit passage of time. Here time advances when the available resources per time interval have been consumed.



(a) The New Year's Eve client scenario. (b) The telephone company scenario.

Fig. 2: A scenario capturing client handsets and two telephone services with cooperating load balancers.

When integrating the time model and the resource consumption model, the global clock only advances when all processes are blocked or suspended and no

process is ready to run. This means that for time to advance, all processes are in one of the following states: (1) the process is awaiting for a guard that is not enabled (using `await` statements); (2) the process is blocked on a future that is not available (using `get` expressions); (3) the process is suspended waiting for time to advance (using `duration`); or (4) the process is waiting for some resource (using cost annotations). In practice this means that all processes run as long as there is "work to be done," capturing the so called run-to-completion semantics.

In the rest of this section we present a simple example, adopted from [21], to showcase the behavioral specification of time and resource consumption.

3.1 Example: phone services on New Year's Eve

```
1  def Bool isMidnightWindow(Int interval) =
2    let Rat curr = timeValue(now()) % interval
3    in curr > 50 && curr < 70;
4
5  class Handset (Int interval, Int cyclelength, TelephoneServer ts,
   SMSServer smss) {
6    Bool call = False;
7    Unit run() {
8      while (timeValue(now()) < 250) {
9        if (isMidnightWindow(interval)) {
10         Int i = 0;
11         while (i < 10) { smss!sendSMS(); i = i + 1; }
12         await duration(1);
13       } else { if (call) { await ts!call(1); } else { smss!sendSMS();
   } }
14       call = !call;
15       await duration(cyclelength); } } }
```

Fig. 3: The client handset in Timed ABS.

At midnight on New Year's Eve, people stop making phone calls and instead send text messages with seasonal greetings. Inspired by this observation of human nature, we model a workload, depicted in Fig. 2a, in which people in a "normal" behavior alternate between a phone call and a message, but change their behavior during the so called "midnight window" to only sending a massive amount of messages. This behaviour is modeled by the handset clients implemented in Fig. 3. The handset makes requests to the two services. The normal behaviour of the handset is to alternate between sending an SMS and making a call at each time interval. When it makes a call, the client waits for the call to end before proceeding (synchronous call). The handset's spike occurs between the time window starting at time 50 and ending at time 70 that represents the

```
1 interface TelephoneServer  { Unit call(Int calltime);}
2 class TelephoneServer implements TelephoneServer  {
3   Int callcount = 0;
4   Unit call(Int calltime){
5     while (calltime > 0) {
6       [Cost: 1] calltime = calltime - 1;
7       await duration(1);}
8     callcount = callcount + 1;} }
9
10 interface SMSServer { Unit sendSMS();}
11 class SMSServer implements SMSServer  {
12   Int smscount = 0;
13   Unit sendSMS() { [Cost: 1] smscount = smscount + 1;} }
```

Fig. 4: The mobile phone services modelled in Timed ABS

"midnight window", and checked by the function `isMidnightWindow` in Fig. 3. During the spike, the handset sends 10 SMS requests at each time interval.

Telephone and SMS are the two services handling the method calls from the `Handset`. The services are deployed on dedicated virtual machines, as depicted in Fig. 2b. The abstract implementations of the services in Timed ABS are given in Fig. 4. The telephone service offers a method `call` which is invoked synchronously (i.e., the caller waits for the callee), with the duration of the call as parameter. The SMS service offers a method `sendSMS` which is invoked asynchronously (waiting from the callee is not needed). Cost are accrued for each time interval during a `call` and for each `sendSMS` invocation to model the computing resources that handling calls and messages consume.

The model also considers resources and includes dynamic load balancing, which enables the two virtual machines hosting the telephone and SMS services to exchange resources during load spikes. This is captured by the `Balancer` class in Fig. 5, whose instances run on each service. The `Balancer` class implements an abstract balancing strategy, transfers resources to its partner virtual machine when receiving a `request` message, monitors its own load, and requests assistance when needed. The class has an active process defined by its `run` method, which monitors the local load. The ABS expression `thisDC()` returns a reference to the deployment component on which an object is deployed, and the method call `load(Speed, n)` returns the average usage $(0 - 100\%)$ of processing speed in the previous n time intervals. If the load is above `90`, the balancer requests resources from its partner. If a `Balancer` receives a request for resources, it will consider if it has enough resources and it is not overloaded, and if so, transfer part of its available computing resources to its partner, this guarantees that the capacity is always maintained above a minimum (see Line 16 in Fig. 5).

The configuration of the system, depicted in Fig. 2b, is given in the main block shown in Fig. 6. The two deployment components model the hosting virtual

```
class Balancer(String name, Rat minimum) implements Balancer {
  Balancer partner = null;
  Rat ld = 100;

  Unit run() {
    await partner != null;
    while (timeValue(now()) < 250) {
      await duration(1);
      ld = await thisDC()!load(Speed, 1);
      if (ld > 90) {await partner!requestdc(thisDC());} } }

  Unit requestdc(DC comp) {
    InfRat total = await thisDC()!total(Speed);
    Rat ld = await thisDC()!load(Speed, 1);
    Rat requested = finvalue(total) / 3;
    if (ld < 50 && (finvalue(total)-requested>minimum)) {
      thisDC()!transfer(comp, requested, Speed);} }

  Unit setPartner(Balancer p) {partner = p;} }
```

Fig. 5: The balancer in Timed ABS

```
{ Rat minimum = 15;
  DC smsdc = new DeploymentComponent("smsdc",map[Pair(Speed, 80)]);
  DC teldc = new DeploymentComponent("teldc",map[Pair(Speed, 80)]);
  [DC: smsdc] SMSServer sms = new SMSServer();
  [DC: teldc] TelephoneServer tel = new TelephoneServer();
  [DC: smsdc] Balancer smsb = new Balancer("smsb",minimum);
  [DC: teldc] Balancer telb = new Balancer("telb",minimum);
  await smsb!setPartner(telb);
  await telb!setPartner(smsb);

  new Handset(100,1,tel,sms);  // start many clients
  new ....
}
```

Fig. 6: The main block configuration Timed ABS

machines; objects are created inside deployment components via the `[DC: id]` annotations. After compilation, it is possible to simulate the example with the command `gen/erl/run -l 100 -p 8080`. Here `-l 100` allows to run the model until simulated time `100`, and `-p 8080` allows to observe the load in the deployment components using a browser (via `http://localhost:8080/`, see Sec. 5 for more details about visualization).

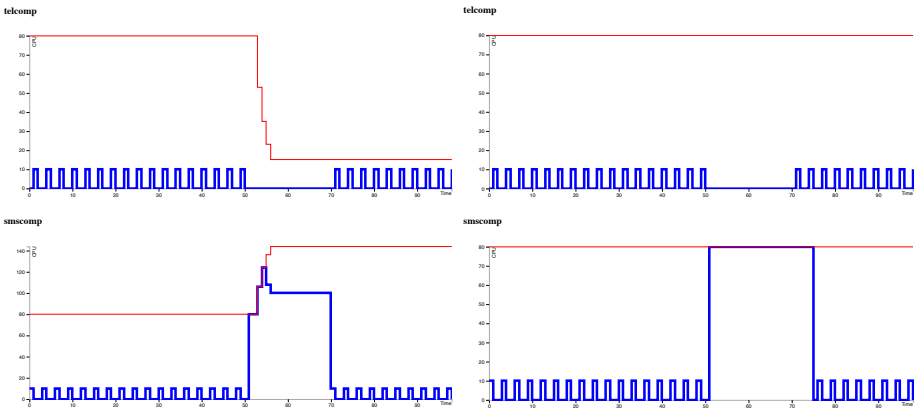


Fig. 7: Running the example with (to the left) and without (to the right) load balancing. Here red lines capture the available resources per time interval in the deployment components and blue lines capture the amount of used resources by the services per time interval. With load balancing the overload of the SMS service is quickly overcome after time 70, without load balancing the SMS service is overloaded until time 75.

Figure 7 shows two simulations of the example, one with and one without active resource balancers (by commenting out Lines 6–9 in Fig. 6). Observe in particular the changing resp. constant total capacity of the two deployment components, and the duration of overload of the SMS component during and after the midnight window.

4 Advanced Synchronization Patterns

So far, we have seen asynchronous method calls that produce futures, and how to read the resulting value from such a future. This section shows some advanced synchronization patterns that an ABS model can employ to coordinate among a number of independent worker objects and their processes. We use as an example a model of MapReduce as originally described in [12]. The ABS source file is at https://github.com/abstools/absexamples/blob/master/collaboratory/tutorials/discotec-2021/MapReduce.abs.

4.1 Coordinating Multiple Processes via Object State

Our model of MapReduce employs a pool of worker machines, each modeled as an ABS object, and one coordinator object that creates `invokeMap` and `invokeReduce` processes on the workers, awaits the results, and calculates intermediate values and the final result.

```
1 Set<Worker> workers = set[];
2 Worker getWorker() {
3   Int maxWorkers = 5;
4   if (emptySet(workers) && nWorkers < maxWorkers) {
5     nWorkers = nWorkers + 1;
6     DeploymentComponent dc = new DeploymentComponent(
7         `worker $nWorkers$`, map[Pair(Speed, 20)]);
8     [DC: dc] Worker w = new Worker(this);
9     workers = insertElement(workers, w);
10  }
11  await size(workers) > 0;
12  Worker w = take(workers);
13  workers = remove(workers, w);
14  return w;
15 }
16 Unit finished(Worker w) {
17  workers = insertElement(workers, w);
18 }
```

Fig. 8: A worker pool in ABS with maximum size 5.

The coordinator implements the worker pool, as shown in Fig. 8, as an object field called `workers`. The two methods `getWorker` and `finished` coordinate via this field. (Remember that multiple processes of `getWorker` and `finished` can run at the same time.)

The method `finished` simply adds a worker back into the pool. The method `getWorker`, on the other hand, can only return workers that are currently in the pool. In case fewer workers than the maximum have been created, the condition in Line 4 evaluates to `True` and a new worker on a fresh deployment component is created and added to the pool.

Multiple processes can hit the `await` condition in Line 11; one of them will be woken up and return a worker object to the caller each time a worker is added to the pool via the `finished` method. Note that the worker itself can call `finished` after completing its task, thereby adding itself back to the pool, so the coordinator does not need to keep track of worker status.

```
1  foreach (item in items) {
2    Worker w = this.getWorker();
3    String key = fst(item); List<String> value = snd(item);
4    Fut<List<Pair<String, Int>>> fMap = w!invokeMap(key, value);
5    // Worker.invokeMap calls `finished' after finishing the task
6    fMapResults = insertElement(fMapResults, fMap);
7  }
8  foreach (fMapResult in elements(fMapResults)) {
9    await fMapResult?;
10   List<Pair<String, Int>> mapResult = fMapResult.get;
11   // process results of this map task ...
12 }
```

Fig. 9: Distributing tasks among a pool of workers and collecting their results.

4.2 Synchronizing on Multiple Processes via Futures

Figure 9 shows how to distribute tasks among workers in such a worker pool and collect results afterwards. The set `fMapResults` holds the futures of the `invokeMap` processes. Tasks are distributed among worker instances obtained via `getWorker`, ensuring that each worker processes only one task at a time. After all tasks have been distributed, Line 8 and onward collect all results for further processing. It is easy to see that this constitutes a *barrier*; the subsequent `reduce` phase of the MapReduce model (not shown) will only start after all results from the `map` phase have been received by the coordinator.

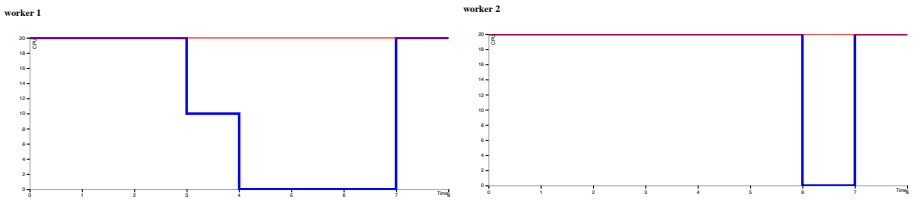


Fig. 10: Machine load during map and reduce phases. Reduce begins at time $t = 7$.

Figure 10 shows the load over time on the deployment components of two of the worker objects (the other three workers process tasks over the whole simulation run). It can be seen that the load on these workers starts dropping before time $t = 7$, where all `map` tasks are finished and the `reduce` phase begins. The graph shown is the default visualization implemented by the Model API (see Sec. 5).

5 Visualizations via the Model API

This section describes the Model API, a way to communicate with a running ABS model via HTTP requests. The Model API can be used to retrieve data from a model and to insert asynchronous method calls. At compile time, HTML and Javascript files can be added to a model to implement bespoke visualizations.

5.1 Exporting Objects and Methods

Objects and methods are made accessible for the model API via annotations. Figure 11 shows the use of the `HTTPCallable` annotation to mark the method `getText` in Line 3 as callable via the Model API, and the use of `HTTPName` in Line 9 to make the object `t` visible via the name `text`.

```
1 module Example;
2 interface Text {
3     [HTTPCallable] String getText();
4 }
5 class CText(String message) implements Text {
6     String getText() { return message; }
7 }
8 {
9     [HTTPName: "text"] Text t = new CText("This is the message");
10 }
```

Fig. 11: A Model API Example.

To use the Model API, the model is started with a parameter `-p xxxx`, where `xxxx` is the port number where the model listens for requests. After starting the above model with `gen/erl/run -p 8080`, calling the `getText` method from the command line results in a JSON-formatted result as follows:

```
$ curl localhost:8080/call/text/getText
{
  "result": "This is the message"
}
```

Full documentation of the Model API can be found in the ABS manual at https://abs-models.org/manual/#-the-model-api.

5.2 Adding a Custom Visualization

When a model is started with the parameter `-p xxxx`, it is possible to connect with a web browser to the URL `http://localhost:xxxx/` (where `xxxx`

is a port number). By default, this URL shows a simple visualization of the load and total availability of the CPU resource of all deployment components in the model. It is possible to replace this visualization with custom, model-specific visualizations. This is done by compiling the model with the parameter `-modelapi-index-file` and giving the name of a HTML file which will then be displayed by the browser. Additionally, the parameter `-modelapi-static-dir` can be used to include a directory of static assets (Javascript libraries, graphics files, CSS stylesheets, etc.) into the model. These files are accessible within the Model API with `/static/`*`filename`*.

For space reasons, this paper does not show an example of this technique. We have prepared a worked example of a water tank controller and a simulated water tank which can be found at https://abs-models.org/documentation/examples/single-watertank/.

6 Record and Replay of Simulations

As already discussed, the semantics of the ABS language are nondeterministic wrt. scheduling of processes. This section discusses a feature of the ABS simulator that makes it possible to *record and replay* simulator runs, such that the scheduling decisions taken during record are faithfully reproduced during replay. This section only shows how to use this feature; a technical discussion can be found in [32].

```
1  module Example;
2  interface Printer {
3      Unit printMessage(String message);
4  }
5  class CPrinter implements Printer {
6      Unit printMessage(String message) {
7          println(`The message is: $message$`);
8      }
9  }
10 {
11     Printer printer = new CPrinter();
12     foreach (m in list["Hello", "Hola", "Hallo", "Hei"]) {
13         printer!printMessage(m);
14     }
15 }
```

Fig. 12: A nondeterministic ABS program.

Consider the ABS code in Fig. 12, which will print four different greetings in arbitrary order when run. Running the example with the command `gen/erl/run`

`-t trace.json` will produce a file `trace.json`. Afterwards, running with the command `gen/erl/run -r trace.json` will print the same interleaving of messages as observed during the recording run.

7 Case Studies and Other Analysis Tools

Fredhopper Cloud Services provides search and targeting facilities for e-commerce retailers as a service (SaaS) [3]. Their software is under constant development and relies on automated configuration [16], continuous monitoring and testing. Using ABS-models allows changes to be analyzed prior to deployment. This includes low-level effects such as consuming the resources of a DC.

For the Fredhopper case study, a general replay tool for logs from the production system was built [6]. The replay tool interacts with the simulator via the Model API and enables simulating real-world scenarios from Fredhopper's system logs. This was used to validate the correlation between the model and the actual system, and for predicting the effects of changes in the system.

ABS-YARN is a case study showing how ABS may be used for modeling applications running on Apache's Hadoop YARN framework with different configurations [30]. Hadoop is a popular cloud infrastructure system and YARN (Yet Another Resource Negotiator) provides a platform for job scheduling and for cluster resource management for Hadoop configurations.

Simulation shows that the ABS framework can accurately reflect the behavior of YARN and can efficiently compare different deployment decisions. This work was extended to Hadoop Spark Streaming [28] and used to study different instance purchasing options provided by Amazon Web Services [20].

The HyVar toolchain is a framework that targets context-dependent software upgrades for car ECUs (Electronic Control Units). The framework collects information from a fleet of cars, analyzes this information to decide how to update the software running on the different cars, and sends software updates back to the cars when needed. The objective of the Timed ABS simulations was to efficiently analyze the *scalability* of the toolchain [29].

Multicore Memory Systems. This case study focussed on how a model of multicore memory systems [8], formalized in structural operational semantics, could be implemented as a simulator of memory operations on a multicore architecture in ABS. The focus of this work was on correctness preserving transformations of operational rules with pattern matching synchronization conditions at the SOS level to a decentralised and asynchronous actor model [7]. The await-statements of ABS were crucial to ensure a granularity of interleaving between processes which corresponded to the SOS model.

Kubernetes. This case study develops a formal model in Timed ABS of a containerized orchestration system for cloud-native microservices, which are loosely-coupled collections of services designed to be able to adapt to traffic in very fine-grained and flexible ways. The model focussed on resource consumption and scaling of microservices deployed and managed by Kubernetes [18].

This work performed experiments on HPC4AI, a cluster for deploying high-performance applications, in order to compare the observed behavior of real systems to corresponding observations of instances of the model [31].

Compugene is a case study that models transcription of mRNA in the context of computational biology [1]. To do so, the effect of different rates of degradation in neighboring cells has been taken into account. The time model of ABS is crucial to model biological processes.

The simulation is used to compare the mathematical model (expressed in ABS) with experimental results, with the aim to reduce the need for the more time-consuming experiments.

FormbaR is a case study that models the railway operations rulebooks of the main German railway infrastructure company [24]. Using an executable model, changes in the rulebooks can be prototyped with quick feedback cycles for their maintenance. The time model is needed to faithfully mirror train driving.

The simulator was leveraged for analyzing the effects of rule changes. A special visualization tool using the Model API is used to interact with the simulation without modifying the ABS code and to summarize the results of the execution [26]. In a more general context, this shows how the model API enables *debugging tools* where the user can inject faults and observe the outcome.

Other Tools.

Several static analyses tools have been developed for ABS and we list them here together with bigger case studies that use them, as they offer further insights into modeling with ABS in practice. For tools and case studies of other Active Object languages we refer to the survey by de Boer et al. [10]. Many static analyses, such as deadlock checkers, are bundled in the SACO [2] suite and its extensions [23].

Systematic Checking. Systematic testing explores all execution paths in a program and is implemented for ABS in the SYCO tool [4]. SYCO has been applied to check properties of Software defined networks [5], a paradigm to dynamically and flexibly adapt the policy of network-switches using a central controller and the C-to-ABS [33] framework, a tool that translates C into ABS code to analyze safety properties of C code with underspecified behavior.

Deductive Verification. Deductive verification checks safety properties, such as object invariants, using logical reasoning and as such can reason about unbounded systems and unbounded inputs. Two deductive verification tools are available for ABS: KeY-ABS [13], which has been used to verify a Network-on-Chip packet switching platform [14] and Crowbar [25], which has been applied to a case study in the C-to-ABS framework described above.

[1] https://www.compugene.tu-darmstadt.de

A Installing the ABS Compiler

The ABS compiler is contained in a file `absfrontend.jar`; the latest released version can be found at https://github.com/abstools/abstools/releases/latest. To run the compiler and the generated models, a Java development kit (JDK) version 11 or greater, and Erlang version 22 or greater must be installed.

Then, to compile an ABS file `model.abs`, use the command `java -jar absfrond.jar -erlang model.abs`. The compiler creates a subdirectory `gen/`, the compiled model can be run with the command `gen/erl/run` (or, on Windows, `gen/erl/run.bat`).

More detailed installation instructions, including links to the necessary software, is at https://abs-models.org/getting_started/local-install/.

Models that do not use the Model API can also be written and run in a web-based IDE based on EasyInterface [15]. To run the collaboratory locally, install docker[2] then run the following command: `docker run -d -rm -p 8080:80 -name collaboratory abslang/collaboratory:latest` and connect a browser to `http://localhost:8080/`. On the start page, you can access ABS documentation and examples, and go to the web-based IDE. To shut down the collaboratory, run the command `docker stop collaboratory`.

References

1. ABS Development Team: ABS Documentation, https://abs-models.org/manual/, Version 1.9.2
2. Albert, E., Arenas, P., Flores-Montoya, A., Genaim, S., Gómez-Zamalloa, M., Martin-Martin, E., Puebla, G., Román-Díez, G.: SACO: static analyzer for concurrent objects. In: TACAS. LNCS, vol. 8413, pp. 562–567. Springer (2014)
3. Albert, E., de Boer, F.S., Hähnle, R., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L., Wong, P.Y.H.: Formal modeling and analysis of resource management for cloud architectures: an industrial case study using Real-Time ABS. Service Oriented Computing and Applications **8**(4), 323–339 (2014)
4. Albert, E., Gómez-Zamalloa, M., Isabel, M.: SYCO: a systematic testing tool for concurrent objects. In: CC. pp. 269–270. ACM (2016)
5. Albert, E., Gómez-Zamalloa, M., Isabel, M., Rubio, A., Sammartino, M., Silva, A.: Actor-based model checking for software-defined networks. J. Log. Algebraic Methods Program. **118**, 100617 (2021)
6. Bezirgiannis, N., de Boer, F.S., de Gouw, S.: Human-in-the-loop simulation of cloud services. In: ESOCC. LNCS, vol. 10465, pp. 143–158. Springer (2017)
7. Bezirgiannis, N., de Boer, F.S., Johnsen, E.B., Pun, K.I., Tapia Tarifa, S.L.: Implementing SOS with active objects: A case study of a multicore memory system. In: FASE. LNCS, vol. 11424, pp. 332–350. Springer (2019)
8. Bijo, S., Johnsen, E.B., Pun, K.I., Tapia Tarifa, S.L.: A formal model of data access for multicore architectures with multilevel caches. Sci. Comput. Program. **179**, 24–53 (2019)

[2] https://www.docker.com/products/docker-desktop

9. Bjørk, J., de Boer, F.S., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: User-defined schedulers for real-time concurrent objects. Innovations in Systems and Software Engineering **9**(1), 29–43 (2013)
10. de Boer, F.S., Serbanescu, V., Hähnle, R., Henrio, L., Rochas, J., Din, C.C., Johnsen, E.B., Sirjani, M., Khamespanah, E., Fernandez-Reyes, K., Yang, A.M.: A survey of active object languages. ACM Comput. Surv. **50**(5), 76:1–76:39 (2017)
11. Clarke, D., Muschevici, R., Proença, J., Schaefer, I., Schlatte, R.: Variability modelling in the ABS language. In: FMCO. LNCS, vol. 6957, pp. 204–224. Springer (2010)
12. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
13. Din, C.C., Bubel, R., Hähnle, R.: KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In: CADE. LNCS, vol. 9195, pp. 517–526. Springer (2015)
14. Din, C.C., Tapia Tarifa, S.L., Hähnle, R., Johnsen, E.B.: History-based specification and verification of scalable concurrent and distributed systems. In: ICFEM. LNCS, vol. 9407, pp. 217–233. Springer (2015)
15. Doménech, J., Genaim, S., Johnsen, E.B., Schlatte, R.: EasyInterface: A toolkit for rapid development of guis for research prototype tools. In: FASE. LNCS, vol. 10202, pp. 379–383. Springer (2017)
16. de Gouw, S., Mauro, J., Nobakht, B., Zavattaro, G.: Declarative elasticity in ABS. In: ESOCC. LNCS, vol. 9846, pp. 118–134. Springer (2016)
17. Hähnle, R.: The abstract behavioral specification language: A tutorial introduction. In: FMCO. LNCS, vol. 7866, pp. 1–37. Springer (2012)
18. Hightower, K., Burns, B., Beda, J.: Kubernetes: Up and Running Dive into the Future of Infrastructure. O'Reilly (2017)
19. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: FMCO. LNCS, vol. 6957, pp. 142–164. Springer (2010)
20. Johnsen, E.B., Lin, J., Yu, I.C.: Comparing AWS deployments using model-based predictions. In: ISoLA (2). LNCS, vol. 9953, pp. 482–496 (2016)
21. Johnsen, E.B., Owe, O., Schlatte, R., Tapia Tarifa, S.L.: Dynamic resource reallocation between deployment components. In: ICFEM. LNCS, vol. 6447, pp. 646–661. Springer (2010)
22. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Integrating deployment architectures and resource consumption in timed object-oriented models. Journal of Logical and Algebraic Methods in Programming **84**(1), 67–91 (2015)
23. Kamburjan, E.: Detecting deadlocks in formal system models with condition synchronization. Electron. Commun. Eur. Assoc. Softw. Sci. Technol. **76** (2018)
24. Kamburjan, E., Hähnle, R., Schön, S.: Formal modeling and analysis of railway operations with active objects. Sci. Comput. Program. **166**, 167–193 (2018)
25. Kamburjan, E., Scaletta, M., Rollshausen, N.: Crowbar: Behavioral symbolic execution for deductive verification of active objects. CoRR **abs/2102.10127** (2021)
26. Kamburjan, E., Stromberg, J.: Tool support for validation of formal system models: Interactive visualization and requirements traceability. In: F-IDE@FM. EPTCS, vol. 310, pp. 70–85 (2019)
27. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. Journal on Software Tools for Technology Transfer **1**(1–2), 134–152 (1997)
28. Lin, J., Lee, M., Yu, I.C., Johnsen, E.B.: A configurable and executable model of Spark Streaming on Apache YARN. Int. J. Grid Util. Comput. **11**(2), 185–195 (2020)

29. Lin, J.C., Mauro, J., Røst, T.B., Yu, I.C.: A Model-Based Scalability Optimization Methodology for Cloud Applications. In: IEEE SC2. pp. 163–170. IEEE Computer Society (2017)
30. Lin, J.C., Yu, I.C., Johnsen, E.B., Lee, M.C.: ABS-YARN: A formal framework for modeling Hadoop YARN clusters. In: FASE. LNCS, vol. 9633, pp. 49–65. Springer (2016)
31. Turin, G., Borgarelli, A., Donetti, S., Johnsen, E.B., Tapia Tarifa, S.L., Damiani, F.: A formal model of the Kubernetes container framework. In: ISoLA. LNCS, vol. 12476, pp. 558–577. Springer (2020)
32. Tveito, L., Johnsen, E.B., Schlatte, R.: Global reproducibility through local control for distributed active objects. In: FASE. LNCS, vol. 12076, pp. 140–160. Springer (2020)
33. Wasser, N., Tabar, A.H., Hähnle, R.: Automated model extraction: From non-deterministic C code to active objects. Sci. Comput. Program. **204**, 102597 (2021)