




Release the Beasts: When Formal Methods Meet Real World Data

Rudolf Schlatte, Einar Broch Johnsen^(✉) , Jacopo Mauro,
S. Lizeth Tapia Tarifa, and Ingrid Chieh Yu

Department of Informatics, University of Oslo, Oslo, Norway
{rudi,einarj,jacopom,sltarifa,ingridcy}@ifi.uio.no

Abstract. It is well-known that the difference between theory and practice seems smaller in theory than in practice. From the perspective of the coordinator, the coordinated components play the role of wild beasts, fortunately imprisoned in boxes. From the perspective of the care-free semanticist, the development of tools is merely a minor step away (possibly hidden in promises of future work). This paper draws parallels between beasts and tool building by describing challenges we have encountered and sharing experiences and lesson learned when going from a compositional semantics to a well-functioning tool interacting with industrial use cases. Concretely, we discuss the development of the simulation backend for Real-Time ABS.

In addition to his scientific contributions, Farhad Arbab has always been an outstanding speaker with a flair for inspiring talks and memorable punchlines. This paper is written for a highly appreciated colleague.

1 Introduction

Inside every box, there is a beast¹ just waiting to be discovered. Look closely and you will find it, lurking in the shadow of some interface. Even if you decide not to look, the beasts will still be there; their behavior an unsolvable mystery to the exogenous spectator. Each beast has its own particularities, its own irregularities, and its own side effects. Every beast is potentially a new friend, some of them can be worth knowing.

Many researchers rely on abstraction for their formalizations and reasoning systems. It is our secret weapon; apply it and a lot of problems simply vanish in a “puff” [1]. Programming languages are also getting increasingly more abstract, allowing us to express programs in more generic ways, relying on some low-level machinery to ensure that they are well-behaved. High-level languages should make it easier to express and prove the correctness of the behavior we want in

This research was supported by the SIRIUS Centre for Scalable Data Access (237898) and by the EU project HyVar: Scalable Hybrid Variability for Distributed Evolving Software Systems (H2020-644298).

¹ The metaphor of the ‘beast in the box’ was invented by Farhad Arbab around 2005.

our systems. We continue to strive for more abstraction [2], for more semantics [3], for more compositionality [4]. As we climb the ladder of abstraction, we leave the operational behind in favor of the denotational, we ignore the “how” in favor of the “what”. Let us consider Reo [5] as a case in point; happily unconcerned with the behavior it coordinates, it is pure compositionality with an endless flow of semantics [6]. Reo’s different semantics describe the flow of data through connectors coinductively [6], using constraint automata [7] abstracting from the distinction between input and output, or as an artist’s palette of colors [8]. Each semantics highlights a particular aspect of Reo’s exogenously coordinated flow. The different semantics also enable the implementation of tools (e.g., [8–10]). The construction of tools reveals another kind of beasts: the implementation details and the interface to the real world. In fact, it is on the path from semantics to tools that we encounter the beasts that are the focus of this paper, where the ideal compositional world of semantics comes with many afterthoughts.

In the rest of this paper, we discuss these afterthoughts and open the boxes to get a closer look inside. Section 2 gives a brief overview of ABS and its semantics, the language that we are using to illustrate our findings when looking into the boxes. Section 3 proposes a starting point when venturing towards implementation and tools. Section 4 discusses implementation issues, Sect. 5 describes interfacing with models, and Sect. 6 describes issues of community and development. Sections 7 and 8 describe two case studies which illustrate the interest of venturing down this road.

2 A Short Overview of Real-Time ABS

Real-Time ABS is a formally defined, actor-based, object-oriented modeling language targeting distributed systems with early deployment decisions and timing requirements. Real-Time ABS extends ABS, a language with a formal syntax and semantics defined in operational semantics (SOS) [13] as well as trace semantics [12]. Compared to other actor or active object languages [26], two distinguishing features of Real-Time ABS are its support for *cooperative concurrency* and the explicit modeling of *deployment decisions* in a real-time setting.

The language is layered and combines a simple, functional language to express local computation; an object-oriented, imperative language for asynchronous communication and synchronization; and real-time and deployment layers which allows object requiring resources for their computations to be placed at locations with restricted resource capacity, and to model the time-sensitive behavior of these objects. The combination of functional and imperative layers makes it easy to model an object-oriented *design*, yet retain a high level of abstraction for internal computations and data modeling. The real-time and deployment layers make it possible to express timing properties and compare deployment decisions early in the software development process.

Real-Time ABS includes a Cloud API, used to model how software applications interact with a cloud provider [27]. The model offer services to client applications to dynamically acquire and release virtual machines on demand.

$$\begin{array}{c}
 \text{(SUSPEND)} \\
 \frac{o(a, \{l \mid \mathbf{suspend}; s\}, q)}{\rightarrow o(a, \mathit{idle}, \{l \mid s\} \circ q)}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{(RELEASE-COG)} \\
 \frac{o(a, \mathit{idle}, q) \ c(o)}{\rightarrow o(a, \mathit{idle}, q) \ c(\epsilon)}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{(ACTIVATE)} \\
 \frac{p = \mathit{select}(q, a, cn)}{\rightarrow \{o(a, p, (q \setminus p)) \ c(o) \ cn \ cl(t)\}}
 \end{array}$$

$$\begin{array}{c}
 \text{(RUN-INSIDE-INTERVAL)} \\
 \frac{0 < d \leq \mathit{mte}(cn', t) \quad [t] = [t + d]}{\rightarrow_t \{cn \ cl(t)\}}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{(RUN-TO-NEW-INTERVAL)} \\
 \frac{0 < d \leq \mathit{mte}(cn', t) \quad [t] = t + d}{\rightarrow_t \{\mathit{timeAdv}(rsr\mathit{Refill}(cn'), d) \ cl(t + d)\}}
 \end{array}$$

Fig. 1. Some rules from the operational semantics of Real-Time ABS.

The model of the cloud provider is based on deployment components, which are computation locations with limited resource capacities and which are used to represent created virtual machines of given processing capacities. The communication interface of the cloud provider allows a model of a client application to create machines with a desired execution capacity, acquire machines to start task executions, release machines, and finally get the accumulated usage cost. This API extension has been used in several case studies. In particular, Sect. 7 reports on experiences with Real-Time ABS using this Cloud API.

Figure 1 illustrates the SOS semantics of Real-Time ABS (for details of the full semantics, see [15, 19]). In these rules, a *configuration* cn is a multiset of terms, including objects, concurrent object groups (cogs), which share a thread of execution, and execution locations with restricted amount of resources. The timed configuration includes a global clock $cl(t)$. The use of brackets encapsulating timed configurations allows the left hand sides of rules to match the *whole* configuration and not just some of its terms. An *object* is a term $o(\sigma, p, q)$ where o is the object’s identifier, σ is a substitution representing the binding of the object’s fields, p is an (active) process, and q a *pool of processes*. For the process pools q , concatenation is denoted by $q_1 \circ q_2$. A *process* $\{\sigma \mid s\}$ consists of a substitution σ of local variable bindings (including the variable *deadline* which denotes the remaining execution time of the process until a soft deadline is passed) and a list s of statements, or it is *idle*. A cog $c(act)$ contains an identifier c and the currently active object act or ϵ if no object of the cog is currently active.

Rule SUSPEND enables cooperative scheduling and suspends the active process to the process pool, leaving the active process *idle*, and RELEASE-COG makes the cog idle if the object holding its lock is idle. Given an idle object with an idle cog, rule ACTIVATE schedules a process from the process queue and grabs the lock of the cog. Here, the function *select* chooses a process which is ready to execute from the process queue. If there is no such process, the premise is false.

Time advance in the semantics is specified by a transition relation \rightarrow_t and the rules RUN-INSIDE-INTERVAL and RUN-TO-NEW-INTERVAL. The model of time is based on maximal progress, so time will only advance when execution is otherwise

blocked (i.e., $\xrightarrow{!}$ denotes the reduction to normal form in the premises of the rules). The rule `RUN-INSIDE-INTERVAL` captures time advance which does not influence the resource availability in the execution locations of the system, and the rule `RUN-TO-NEW-INTERVAL` captures the case when the resources in the execution locations should be “refilled” for the next time interval. The function *mte* calculates the maximal time advance, which is the largest amount by which time can advance such that no “interesting” occurrence will be missed in any object or execution location. The function *timeAdv* updates the active and suspended processes of all objects, decrementing the values of all deadlines and duration statements. The function *rscRefill* captures the effect of time advance on the execution locations, causing the refilling of resources in each of them.

3 Leaving the World of Semantics and Compositionality

Compositionality is often regarded as the key to address real systems using formal methods. In semantics, compositionality gives us maintainability by minimizing the interference between different mathematical objects such that new objects will not violate existing semantic rules and such that different objects can be composed or coordinated via their interfaces. In reasoning, compositionality allows us to reason about each of these objects separately, and later put together the derived local behaviors by means of composition rules. In an ideal, mathematical setting, composition rules such as logical conjunction come naturally [11]. In practice, they are often complex and need to, e.g., resolve interference between processes [4] or match shared events in local traces [12]. Remark that compositionality also often leads to incompleteness in analysis by introducing *abstractions* in local reasoning in terms of interfaces, communication traces, scheduling traces, and other assumptions which generalize the environment.

A major challenge in formal methods is what we may call “*leaking abstractions*”. Leaking abstractions typically occur when our reasoning about a high-level model or a program requires more low-level information than we have available at the surface level: We have lost too much information in our abstractions. For example, the abstractions are leaking when knowledge about the runtime system’s locks, its partitioning of data into blocks, or its (often unspecified or non-deterministic) scheduling decisions are required to reason about the behavior of programs which do not mention any locks, memory blocks, or schedulers in the surface language.

An interesting example of leaking abstractions is deployment. In high-level languages we want to abstract from knowledge of, e.g., memory layout, which processor gets to run a task, or how tasks distribute over nodes in a grid. With virtualization, hardware becomes data in our software programs. In our work on virtualized services for the cloud in the ABS modeling language, described in Sect. 2, we were confronted with how to give a high-level representation of low-level deployment details; we needed to explicitly represent time as well as dynamic deployment decisions allowing the program to change its own deployment. Our solutions were also confronted with real industrial case studies [14].

To capture uniform time advance and their effect on computing resources operationally, we suddenly needed global rules, as the one shown in Fig. 1. To apply the modeling language to industrial case studies, we needed efficient tools which integrated our models with real world operational data. To be useful to practitioners, these tools should not derive theorems from our models, but rather produce easily accessible information; exit Greek variables, enter the world of visual analytics. As all things flow [5], we have focussed on timed data streams depicting the runtime behavior of models.

4 From Operational Semantics to Simulation

If denotational semantics captures the “what” and operational semantics the “how”, a simulator captures the “really how”. This section discusses some details from the experiences gained in moving the perspective from the operational semantics of Real-Time ABS to the realm of execution (see Sect. 2 for a brief introduction to the language). The process of implementing a language’s operational semantics into the tool domain includes many conventional software development tasks: fixing a concrete syntax that is expressible in ASCII, choosing an implementation platform, implementing a parser and type-checker, code generation, etc. In the case of Real-Time ABS, the tool chain runs on top of the Java Virtual Machine, translating ABS models into runnable code using one of several “backends”. The first backend, initially developed for a precursor language called Creol [16], was implemented on top of the rewriting logic system Maude [17]. Later this backend was joined by a language implementation in Java and one in Erlang [18].

In addition to standard software engineering issues, translating a formal semantics such as Real-Time ABS into code presents some unique challenges. In this particular case even though the starting point of this translation was an operational semantics [15, 19] detailing the “how”, some of its rules have a denotational flavour hiding the “really how”. Rules that are straightforward to understand in terms of “what” they are doing, devolve into convoluted code; e.g., the humble negation operator morphs into a global actor resulting in performance bottlenecks, etc. In the remainder of this section, we present a selection of interesting implementation challenges, encountered during the implementation of the Real-Time ABS simulator. The chosen challenges relate to the rules of the operational semantics shown in Fig. 1.

4.1 Clock Advance

A straightforward expression of a logical clock rule is: *If no process can execute, advance the clock by the maximum amount that makes no process miss a deadline.* This can be expressed in a rule such as RUN-INSIDE-INTERVAL of Sect. 2, Fig. 1, where the symbol $\overset{!}{\rightarrow}$ denotes the maximum application of other rules. In the more concrete world of Maude’s rewriting logic, expressing that no process can execute entails checking the status of each process, slowing down simulation.

Lesson 1. *Semantic rules which contain global (whole-program) state are expensive and easily lead to problems of scaling during implementation.*

Rules with global state are not compositional by nature. This makes a direct implementation of semantic rules with global state badly suited for simulating systems with large state. This problem can be circumvented by introducing a centralized coordinator or a distributed protocol.

Note that the property “cannot execute” is non-monotonous since a process waiting for a computation result can become runnable again as a consequence of a process in another cog terminating. On the even less abstract distributed Erlang platform using explicit actors, this can easily lead to temporary “glitches” as a completion message travels from source cog to target cog. As a consequence, it was necessary to implement a dedicated singleton actor tracking the status of each cog. Entertaining months were spent chasing ever more improbable protocol errors that resulted in spurious clock advances.

Lesson 2. *Negations (“it is not the case that...”) translate into universal quantifiers whose implementation requires knowledge of global state.*

4.2 Scheduling Processes

In contrast to the semantics of a logical clock representing dense time, which is specific to ABS, the semantics of scheduling of cooperative processes is well-understood and standard. In ABS, scheduling entails an idle cog picking an enabled process and executing it, thereby becoming busy. The semantics of scheduling is shown in SUSPEND, RELEASE-COG and ACTIVATE of Sect. 2, Fig. 1; the *select* function here returns a runnable process p from the process pool q .

This behavior was implemented in Erlang by choosing one element out of a set of ready process identifiers, sending it an activation signal and removing it from the ready process set. We were satisfied that this simple implementation was trivially correct and according to the desired semantics, until we received a bug report about a deadlock in the simulation engine.

The user, as it turns out, was running two processes communicating via a shared object field: process A was spinning on a field (`while(!field) suspend;`), while process B’s task was to set the field (`field = True;`). Note that both of these processes are enabled and ready to run. Due to the implementation of Erlang’s standard set datatype `gb_sets`, process A happened to be chosen every time, thereby starving process B that would, in turn, have enabled process A to make progress. The solution was to (i) gently mock the offending user’s program, which should have used the ABS construct `await field;` instead of busy-looping, and (ii) implement a randomizing scheduler to cater for a potentially infinite sequence of naïve models in the future.

Lesson 3. *The simplest possible, obviously correct implementation of a semantic rule might not be suitable in practice.*

5 Getting the Real World into the Models

An implementation of a language semantics gives us a “compute kernel” of sorts that can be used to execute programs written in the language. However, pure computation, even when correctly implemented, is not always useful by itself. End users tend to expect facilities for input and output, which are often abstracted away in language semantics.

This section describes some useful patterns and extensions of Real-Time ABS in the areas of input, output, and visualization. Most of these are implemented in terms of a “Model API”. The very first implementations of ABS (and Creol, its precursor) consisted of equations and rewrite rules over terms representing objects, processes and other runtime semantics entities that executed on the rewriting logic system Maude [17]. The result of a computation was represented as a dump² of the final state of the global configuration, rendered as ASCII. The results of simulation (both computation results and model state) were accessed by one-off scripts, often using regular expressions to extract relevant parts of this dumped output.

While working on various case studies with industrial partners, the need to both access model state and influence the model from the outside became apparent. The creation of other backends (and later versions of Maude) enabled the addition of printed character output to the language, but accessing richer, structured data remained elusive.

Lesson 4. *Simulation engines need visualization and text output as a minimum, ideally also a way to access structured state.*

In response to these end user needs, a *Model API* was established for the Erlang backend. The API is based on web technologies: communication via the HTTP protocol, with data returned in JSON format. This choice was made to maximize the ease of implementation of tools to interoperate with a running model; most programming languages come with standard libraries to emit HTTP requests and to parse JSON.

Figure 2 illustrates interaction with a running model from the command line: the user first obtains a list of entry points (entry points are added to the model by the modeler), then a list of callable methods and finally the result of a method call. A representation of an ABS object’s internal state can be obtained in a similar way. This API was used in an industrial case study [20] to drive an ABS model according to traces obtained from the system logs of a real system.

Lesson 5. *Any aspect of a tool that is not core to its functionality (e.g., communication protocols, structured data storage) should be implemented using established industrial standards and existing libraries. This makes it easier for both implementor and end user.*

² The reviewers of the EU project Credo (<https://projects.cwi.nl/credo/>), tasked with implementing Creol [16], correctly pointed out that screenfuls of text (or, for larger model states, hundreds of kilobytes) were not an effective way of communicating and understanding model behavior.

```

~$ curl localhost:8080/call
["helloobj"]
~$ curl localhost:8080/call/helloobj
{"name": "greeting",
  "parameters": [{"name": "name",
                  "type": "ABS.StdLib.String"}],
  "return": "ABS.StdLib.String"}
~$ curl localhost:8080/call/helloobj/greeting?name=Joe
{"result": "Hello_Joe!"}

```

Fig. 2. Interacting with the Model API.

One consequence of adding a Model API, i.e., a way of communicating with a model from outside, is that we move from a closed to an open world where the full behavior of the model can no longer be analyzed statically. This can impact proof theories and other analysis approaches, especially when relying on the whole-program analysis. Modular, compositional analysis methods are less affected as only a few selected modules are opened to the outside world.

Lesson 6. *Tools have different, sometimes conflicting requirements. Making a language implementation more useful for simulation (“programming”) can result in proofs of correctness becoming more difficult, and vice versa.*

6 Getting the Models into the Real World

In the early days of ABS and its extensions, knowledge about the language was transmitted orally. All users were part of the same institution, or at least of the same project, so education and discovery of best practices happened face-to-face. Similarly, bugs and problems were discovered, reported, discussed, and fixed via personal interaction. However, this does not scale for a language with users that are not personally known to the language implementors and designers.

The aim for a widely-used tool must be to make it “self-supporting”; i.e., the users should be able to find the answers to common problems by themselves. Updating the documentation in response to user questions must be an ongoing process.

Lesson 7. *When a user asks a question that is covered by the documentation, ask where they looked for the answer, then update the documentation to put it there.*

Additionally, a lot of programs are developed in a process of “coding-by-imitation”. Good examples and tutorials help in the process of picking up an unfamiliar language.

Lesson 8. *Provide both small and large examples that show best practices and “proper” ways to use a language to its fullest potential.*

Another aspect of language uptake is visibility of ongoing development. For users, access to the source code provides a measure of safety—but maybe more important is a visible and accessible development process. Multi-year commit activity and prompt responses to bug reports assure prospective customers that any problems they might uncover will likely be solved as well.

Lesson 9. *Make development activity visible to interested end users.*

On the other hand, development can lead to “churn” in that introducing and adapting features can break old code. Once a language is used more broadly, care must be taken not to invalidate the users’ work. This can be done in multiple ways: by keeping deprecated features around if they do not conflict with new features; by documenting changes and update paths for outdated code; and by providing means of identifying the tool version used for a specific model and obtaining that version later, should the need arise.

Lesson 10. *Do not break user code unnecessarily, and provide ways forward (adapting code for new tool versions) and backward (obtaining previous tool versions) in case of necessary changes.*

7 Use Case: Scaling with Traffic Data

This section describes a use case modeling a microservice architecture for dispatching car software updates [21]. The use case describes an innovative business model which combines cloud computing and microservices to allow on-demand delivery of scalable and modular applications with pay-as-you-go pricing.

The starting point for the model creation was the existing microservice architecture. Part of the challenge was to create an appropriate abstraction, i.e., a simple executable model which exposes scaling decisions as configurable parameters. ABS helped to cope with this challenge because (i) it natively supports CPU, memory resources and the notion of deployment components [15], (ii) being a full-fledged language it is more flexible than ad-hoc cloud simulators, (iii) it has parallel run-time support in Erlang, (iv) tools for worst-case performance analysis [22] and visualization are available.

Figure 3 shows the chosen methodology. First, we used worst case analysis (e.g., queuing theory) and profiling techniques to understand which parts of the system could be simplified and abstracted. This allowed in a second step to create a simple model with fewer parameters to tune. Finally, to reduce the cost of the cloud resources used by the microservice system and find good scaling parameters, we used automatic parameter configurators [23,24], i.e., tools that rely on machine learning techniques to explore the possible configurations in a smarter and more systematic way and come up with good parameter settings.

The creation of the model³ was quite straightforward. A microservice instance and the load balancer for the redirection of requests was represented with objects.

³ https://github.com/HyVar/abs_optimizer.

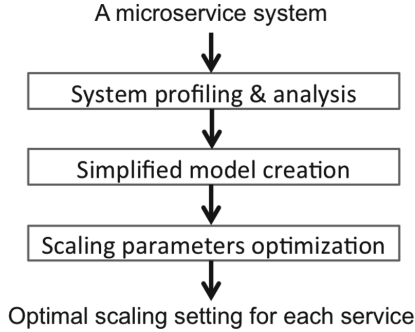


Fig. 3. The scaling optimization methodology.

The internal computation performed by a microservice was abstracted to a skip statement taking a given computation cost c as follows.

```
[Cost: c] skip;
```

The objects were instantiated on deployment components, a native construct of ABS used to represent the virtual machines on which the real microservice instances are deployed. In this way, the acquisition/dismissal of a virtual machine for scaling up/down was modeled by the creation/removal of a deployment component exploiting the ABS native Cloud API (for details, see Sect. 2).

Based on this model, we can now search for good scaling settings using the Sequential Model-Based Optimization for General Algorithm Configuration (SMAC) tool [24], an automatic parameter configurator, to explore possible configurations. This computationally heavy task was done using 64 nodes in a Numascale cluster, a scalable cluster with shared memory⁴. We run 64 instances of SMAC in parallel for 12 h. Every execution of SMAC was performing in sequence the simulations running the generated Erlang processes on 6 dedicated cores. The input request pattern used 24-h of car traffic based on the number of cars registered on the A414 highway, UK, on Monday, March 2, 2015.

Calibrating the microservice system with good scaling settings was in this case vital. In theory a microservice system is a simple thing, in practice it was not. In the beginning, our abstraction was completely disconnected from the actual performances of the system. For instance, instead of having a uniform latency distribution (predicted by our model), we obtained a distribution of latencies like the one presented in Fig. 4. We could not understand why this was happening since, based on our simulations, this was not explainable by considering network problems or the variability of the performance of the cloud. At the end, we discovered that the Amazon default “round-robin” load balancers used in the real system implementation were not adopting a strictly round robin policy. This official response on the AWS blog⁵ highlights the issue:

⁴ <https://www.numascale.com/>.

⁵ <https://forums.aws.amazon.com/message.jspa?messageID=316829>.

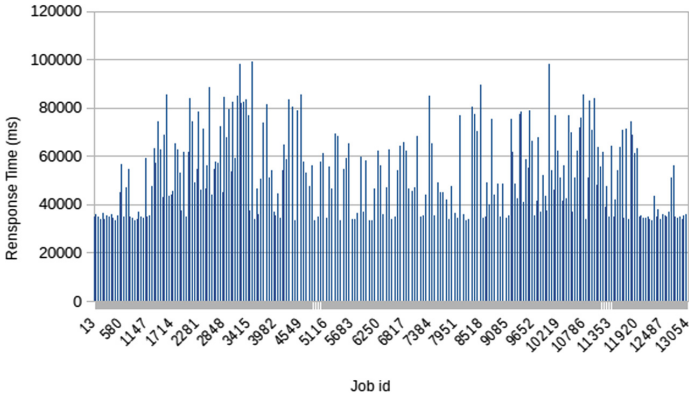


Fig. 4. Uneven request processing times.

“Round-robin does come into play but the client sessions do not always honour TTL’s or DNS caches so you can get skewed results and uneven distributions of requests. The ELB does not take into effect what traffic/requests instances have received to-date in there traffic routing decisions.”

When using Amazon’s load balancers, this problem rendered our abstraction useless. We tried multiple approaches to mitigate this while still running the default Amazon load balancers without achieving a satisfactory level of precision, due to the unknown real policy of Amazon load balancers.

To improve predictability, the original microservice system was changed by replacing the Amazon load balancers with HAProxy⁶, an open source and more controllable solution. This way, the original system was improved and the abstraction was able to predict its behavior, thus allowing good scaling strategies to be found [21]. Figure 5 shows that the simulation was robust enough to mimic the real system and offer a performance estimation usable to set good scaling parameters, even considering the random performance fluctuation of the cloud instances. We used the Model API described in Sect. 5 to visualize the simulations. This visualization greatly simplified the discovery of discrepancies and, later, the gain of confidence in the robustness of the model. In this particular case, it was possible to change the original application to make the model accurate. However, this is not always the case. Developing accurate models which faithfully represent commercial black box components still remains a challenge.

Lesson 11. *Without having a faithful representation of the behavior of the system, an optimization step in the model is useless.*

⁶ <https://www.haproxy.org>.

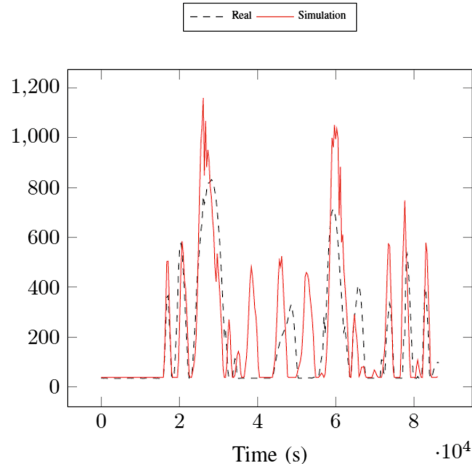


Fig. 5. Comparison latency as predicted by ABS to the latency of the real system.

8 Use Case: Vessel Planning

Whereas Sect. 7 showcased a case study of a distributed software system with virtualized deployment, ABS has been increasingly used to model other kinds of systems (e.g., railways [25]). In this section, we consider an industrial case study from the domain of operational planning. This case study addresses vessel movements and cargo transport in the North Sea. The stakeholders want to improve their workflow to have a better overview of the (potential) bottlenecks delaying overall progress, the general load on different vessels, and the quality of their logistics operation both in terms of the exploitation of vessel capacity and on the timely delivery of material. We use Real-Time ABS as a modeling language to simulate and visualize the actual logistics operations. Compared to the tools currently used, Real-Time ABS simulations provide a different level of overview which helps to gain precision in the decision making phase.

The case study illustrates the usefulness of ABS modeling beyond the realm of computing systems, and makes use of both the input and output-facilities of the Model API to drive the simulation of the model and for visualization of output. Currently Real-Time ABS is here used for simulations. In a longer term perspective, we intend to combine these simulations with stronger analyses to generate solutions and verify their correctness with respect to requirements such as resource restrictions, safety regulations, and space limitations.

We are working with industrial data from different parts of a complex supply chain, and integrate these into a uniform ABS model. The data covers transport plans for a large number of vessels moving between processing plants, with logs for bulk and cargo delivery covering a twelve month period. In this use case, ABS is used to define a general framework for modeling transport plans by means of abstractions for, e.g., vessels, containers, bulk cargo, route segments, and delivery deadlines in a generic way.

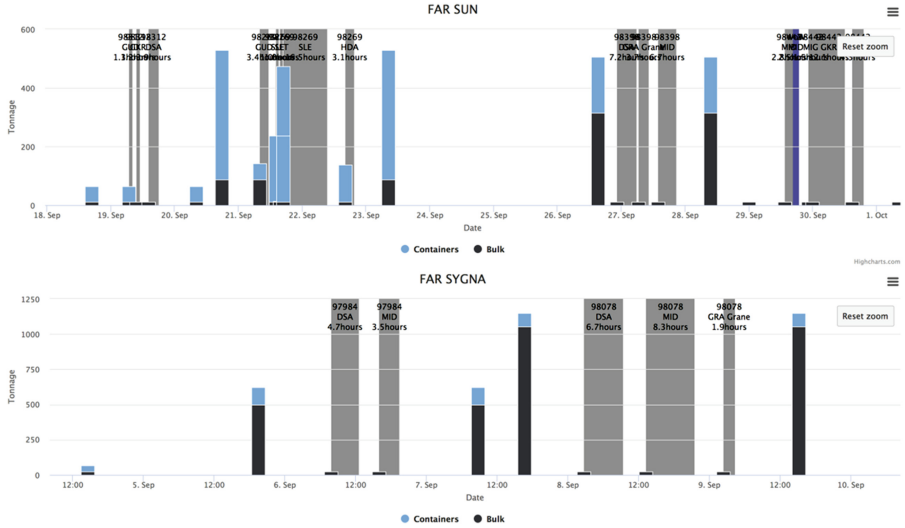


Fig. 6. Visualization of time series data depicting vessel movements.

The model is populated by specific data representing a concrete plan. This is currently done by moving the data from Excel into a SQL database, then generating ABS data structures corresponding to the industrial data set. Thus, the industrial data set acts as the driver for the ABS model. The modeler specifies a time window, and data for this time interval is converted from the SQL database and turned into the model of a concrete plan. This allows the ABS model of the concrete plan for the given time window to be simulated. The planner is presented with a graphical view of the simulated plan, see Fig. 6. This graphical view is dynamically generated in-browser from JSON data fetched via the Model API (described in Sect. 5) and can be easily adapted by a frontend developer; no knowledge of ABS is needed to create different views over the simulation data.

A practical challenge with this case study, in addition to the data cleaning required to convert operational data to fit with the modeling framework in Real-Time ABS and the interaction of the simulation backend and the SQL database, was the conversion of calendar data to model time. Real-Time ABS represents time using rational numbers. We calibrated the model with time 0 representing midnight on the first day simulated. Subsequent dates were numbered 1, 2 . . . , with the fractional part representing time of day. This approach gave us sufficient resolution to model real time using abstract time units.

9 Conclusion

This paper has discussed challenges in moving from formal, compositional language semantics to industrially applicable tools. These challenges span from

pattern matching in reduction rules necessitating protocols in a distributed implementation to documentation and input/output interfaces for real world data. We have compared these challenges to the beasts hidden in the boxes of the exogenous coordinator.

References

1. Arbab, F.: Puff, the magic protocol. In: Agha, G., Danvy, O., Meseguer, J. (eds.) *Formal Modeling: Actors, Open Systems, Biological Systems*. LNCS, vol. 7000, pp. 169–206. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24933-4_9
2. Kramer, J.: Is abstraction the key to computing? *Commun. ACM* **50**(4), 36–42 (2007)
3. Jongmans, S.T.Q., Arbab, F.: Overview of thirty semantic formalisms for Reo. *Sci. Ann. Comput. Sci.* **22**(1), 201–251 (2012)
4. de Roeper, W.P., de Boer, F.S., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: *Concurrency Verification: Introduction to Compositional and Non-compositional Methods*. Cambridge Tracts in Theoretical Computer Science, vol. 54. Cambridge University Press, Cambridge (2001)
5. Arbab, F.: Reo: a channel-based coordination model for component composition. *Math. Struct. Comput. Sci.* **14**(3), 329–366 (2004)
6. Arbab, F., Rutten, J.J.M.M.: A coinductive calculus of component connectors. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) *WADT 2002*. LNCS, vol. 2755, pp. 34–55. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40020-2_2
7. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.M.M.: Modeling component connectors in Reo by constraint automata. *Sci. Comput. Program.* **61**(2), 75–113 (2006)
8. Clarke, D., Costa, D., Arbab, F.: Connector colouring I: synchronisation and context dependency. *Sci. Comput. Program.* **66**(3), 205–225 (2007)
9. Arbab, F., Meng, S., Moon, Y., Kwiatkowska, M.Z., Qu, H.: Reo2MC: a tool chain for performance analysis of coordination models. In: van Vliet, H., Issarny, V. (eds.) *Proceedings of 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 287–288. ACM (2009)
10. Clarke, D., Proença, J., Lazovik, A., Arbab, F.: Channel-based coordination via constraint satisfaction. *Sci. Comput. Program.* **76**(8), 681–710 (2011)
11. Abadi, M., Lamport, L.: Composing specifications. *ACM Trans. Program. Lang. Syst.* **15**(1), 73–132 (1993)
12. Din, C.C., Hähnle, R., Johnsen, E.B., Pun, K.I., Tapia Tarifa, S.L.: Locally abstract, globally concrete semantics of concurrent programming languages. In: Schmidt, R.A., Nalon, C. (eds.) *TABLEAUX 2017*. LNCS (LNAI), vol. 10501, pp. 22–43. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66902-1_2
13. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) *FMCO 2010*. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_8
14. 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. *J. Serv.-Oriented Comput. Appl.* **8**(4), 323–339 (2014)

15. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Integrating deployment architectures and resource consumption in timed object-oriented models. *J. Logical Algebraic Methods Program.* **84**(1), 67–91 (2015)
16. Johnsen, E.B., Owe, O., Yu, I.C.: Creol: a type-safe object-oriented model for distributed concurrent systems. *Theor. Comput. Sci.* **365**(1–2), 23–66 (2006)
17. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. LNCS, vol. 4350. Springer, Heidelberg (2007). <https://doi.org/10.1007/978-3-540-71999-1>
18. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf, Dallas (2007)
19. Bjørk, J., de Boer, F.S., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: User-defined schedulers for real-time concurrent objects. *ISSE* **9**(1), 29–43 (2013)
20. Bezirgiannis, N., de Boer, F., de Gouw, S.: Human-in-the-loop simulation of cloud services. In: De Paoli, F., Schulte, S., Broch Johnsen, E. (eds.) *ESOCC 2017*. LNCS, vol. 10465, pp. 143–158. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67262-5_11
21. Lin, J.C., Mauro, J., Røst, T.B., Yu, I.C.: A model-based scalability optimization methodology for cloud applications. In: *Proceedings of 7th IEEE International Symposium on Cloud and Service Computing (IEEE SC2)*. IEEE CS Press (2017)
22. Albert, E., Flores-Montoya, A., Genaim, S., Martin-Martin, E.: May-happen-in-parallel analysis for actor-based concurrency. *ACM Trans. Comput. Log.* **17**(2), 11:1–11:39 (2016)
23. Kadioglu, S., Malitsky, Y., Sellmann, M., Tierney, K.: ISAC - instance-specific algorithm configuration. In: *Proceedings of 19th European Conference on Artificial Intelligence (ECAI 2010)*. *Frontiers in Artificial Intelligence and Applications*, vol. 215, pp. 751–756. IOS Press (2010)
24. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Coello, C.A.C. (ed.) *LION 2011*. LNCS, vol. 6683, pp. 507–523. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25566-3_40
25. Kamburjan, E., Hähnle, R.: Uniform modeling of railway operations. In: Artho, C., Ölveczky, P.C. (eds.) *FTSCS 2016*. *CCIS*, vol. 694, pp. 55–71. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-53946-1_4
26. Boer, F.D., 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)
27. Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Modeling resource-aware virtualized applications for the cloud in real-time ABS. In: Aoki, T., Taguchi, K. (eds.) *ICFEM 2012*. LNCS, vol. 7635, pp. 71–86. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34281-3_8