Chapter 6

Conclusion and Future Work

We have presented the modeling language ABS and its implementation, that tries to engage in all the three problems of software-engineering, namely performance, security and complexity. We more or less focused on execution performance, because we believe that this is under-represented in the research field of modeling languages, and formal methods in general. However, through our experiments and case studies we did investigate on matters of complexity (e.g. the preferential attachment problem) by manually proving the correctness of a subset of our implementation.

More specifically, in Chapter 3 we have presented a concurrent, object-oriented language (ABS) and its compilation to Haskell using continuations. The runtime utilized Haskell-GHC' lightweight (green) threads to automatically benefit from any multicore parallelism. The compilation to a subset of ABS is formalised in order to establish that the program behaviour and the resource consumption are preserved by the translation. Compared to the only other formalised ABS backend [Johnsen et al., 2010a] (in Maude), our Haskell translation admits the preservation of resource consumption, and as a side benefit, makes uses of an overall faster backend.[1] The performance that the backend promises has been shown through a list of micro-benchmarks and real-world cases of the cache coherence protocol and preferential-attachment implementations.

In Chapter 4, we integrate the timed extension of ABS into HABS with its real-time characterization of the abstract time. The virtualized resources and systems (deployment components) are also present in HABS as first-class citizens of the language. We made use of these new modeling constructs to build an industrial case

[1] `http://abstools.github.io/abs-bench` keeps an up-to-date benchmark of all ABS backends.

study for the human-in-the-loop simulation of virtualized (cloud) services. Our initial experimental results on the use of the presented tool-suite provides clear evidence for the viability of human-in-the-loop (HITL) simulation of Cloud services for training purposes. The training sessions themselves can further be used to provide feedback to the underlying ABS models of the Cloud services and the monitors. Ultimately, the resulting fine-tuning of these models may reach a level of maturity and confidence that allows their deployment in the real-time monitoring and management of the actual service instances. In general, we believe that HITL simulation of Cloud services provides a variety of interesting and challenging research problems, for example mining the log files to calculate an approximation of the "intrinsic" processing time of the individual service requests, cancelling the effect of time sharing.

In Chapter 5, we presented an extension to the ABS language that permits the management of an application's own cloud-deployment inside the language itself. We discussed the realization of such extension (by a Haskell transcompiler) and the execution of an ABS cloud application (based on Cloud-Haskell). Results showed that ABS can benefit from the extra performance that the Cloud offers. Moreover, the extension gives to ABS the expression power it needs to fuse the application logic with the application's own (dynamic) deployment logic. A positive side-effect of the proposed extension is that, ABS being primarily a modeling language, could now be used to model also an application's deployment. Indeed, such cloud-aware software models could be simulated against different and dynamically-varying cloud deployment scenarios. We believe that the cloud extension of ABS leads to new opportunities for furthering the application of formal methods to cloud computing, for example: specifying, verifying, and monitoring Service Level Agreements (SLA) of software systems — with that being the overall goal of ENVISAGE, our current research project. Indeed, we like to envisage software that is aware of its deployment and thus can control it, while its users merely monitor its behaviour via SLAs signed between the interested parties.

6.1 Future Work

There are multiple directions to take for improving in the future the HABS language and runtime system: in the front of the parallel runtime library, the resource-modeling and simulation, and/or the distributed part of the HABS framework.

The parallel version of HABS presented in Chapter 3 has been extensively studied and tested through benchmarks, experiments and case studies. Also, a simplification of the runtime library has been formally shown to preserve the correctness and resources during the translation of a subset of ABS to the Haskell functional language. However, in real execution we rely on the lower-level C-written GHC runtime system, which we have little guarantees about. A solution to this would be to utilize an approach of systematic testing, as done in [Walker and Runciman, 2015], to test if our HABS runtime library maintains certain progress properties and guarantees by

exhaustively testing the threading non-determinism.

Regarding the resource-aware modeling of ABS through the Timed-ABS extension and the simulation of cloud resources, an interesting area to investigate is the support of ABS for discrete-event simulation. We envision a design of such added support where methods correspond to discrete events of the simulation and method calls are merely the "firing" of such events.In this regard, since the "firing" of events happens at specific (as in discrete) time, each ABS method call has to be *annotated* with a timestamp, for example `[Time: t1] o!m1();`. The method's parameters become arguments to such events and as such can be seen as events (instances of events) themselves. Assuming that the events can be ordered (lexicographic ordering of method names) as well as their arguments (data-specific ordering) and since we can provide a fixed ordering (across any successive executions of the same program) of COG identities, we can guarantee the reproducibility of such discrete-event simulations done with the timestamp-augmented extension of the ABS concurrent language. For the implementation side (i.e. the simulation engine), there has been extensive literature for the advancement of algorithms for executing such simulations; two main categories of fast simulation-engine algorithms, which can also benefit from parallel and distributed execution, have been established: that of conservative class algorithms found in [Misra, 1986], and that of optimistic algorithms first proposed in [Jefferson and Sowizral, 1985]. Still, however, there is no clear winner between those categories which besides the common trade-off between execution time& memory, is very much dependent on the specific model that is simulated.

The distributed runtime of HABS detailed in Chapter 5 may receive for the future multiple optimizations that can make the distributed computing in ABS both faster and more robust. As a first optimization, we consider the propagation of futures to their holders. Currently, any resolved future has to be asked for its value directly to its resolver (the method's callee). Since future values are immutable, a first optimization would be instead of contacting the resolver itself is to contact the specific Deployment Component that passed the future onto the holder in a propagation-like fashion. On a similar front, if the deployment component of the resolver of the future fails (because of an exception or hardware error), the value of the future will not be available to its holder anymore, even if its (immutable) value has be computed (resolved). To solve this, we could introduce a way of many-copy replication and caching of the future values among the members (Deployment Components) of the distributed system. In this way, the holder of futures can fetch their values even after the error of their original Deployment Components.

Finally, the distributed version of HABS would greatly benefit from experimenting with a large computing scenario, involving perhaps a larger scale version of the Preferential Attachment case study (with many computing nodes) or some analysis of big-data; the ABS language is a good fit for expressing the concurrent aspects of these analyses and, moreover, the management of the cloud infrastructure can become easier with the cloud extension of ABS. A testing of such a large scale can give us more confidence of the robustness and readiness of the distributed HABS

platform.

Acknowledgements

I would like to thank my fiancée Joëlle who was always there for me. I also like to thank my mother Kaiti, father Antonis, and sister Rina for supporting me from abroad. Thanks goes to my "clean parents" (schoonouders) and Jeske for de gezelligheid.

Finally, I want to thank some friends for being there for me before and during my PhD: Charalampos Kiskinis, Christos (Ntempa) Raptis, Christos Orlis, Stergios Gkatsis, Thomas Taskoudis.