Chapter 1

Introduction

The latest advancements in technology and economic progress led to the ubiquity of computers (hardware and software) in our daily lives. Currently, computer systems are present (embedded) in our phones, watches, automobiles, and even coffee machines and lamps; the future seems to be even more intrusive with computers appearing inside our clothes and under our skin. This enormous information-gathering from all these computers (sensors) demands an analogously-large computing power to process this information in a fast or timely manner.

The future does not look to be so bright, however, when it comes to hardware's raw processing power. For long, it has been established that Moore's law is constrained by the speed of light, that is, there is a limit on how fast the information can flow (and thus be processed) inside a computer system. For mainly this reason, hardware manufacturers have been trying to make the sizes of transistors (i.e. raw processing power) and the distances between them smaller and smaller through these years of development. Yet, there are indications that we have reached this time another limit, where manufacturing at atomic (or even subatomic) levels of transistor size is unstable to produce circuits and thus not economically-viable at large scales (production yield).

For the last decade, this sole reason has driven manufacturers to turn this time to parallel computing for keeping up with the Moore's law, by packing more and more multiple (and otherwise independent) computing resources (CPU cores) to form a (single) larger computing system (i.e. the multicore CPU). This revolution has already reached the mainstream consumer hardware where, as of 2017, a common smartphone can contain up to 10 cores and an affordable desktop machine up to 32 cores. The underlying idea behind

multicore computing is that the performance improves when we split up the workload into multiple (equal) parts and process these *in parallel* to produce back the same result. For many common workloads (beside graphics computing like GPUs), the cores have to *communicate* sometimes with each other to perform a single task. This communication (information flow) between the cores is, once again, constrained by the speed of light. There are even certain workloads where the cores' communication is such an overhead that it is faster to run the computation sequentially (i.e. with a single core). Although the industry yet seems optimistic in coming up with increasingly larger counts of CPU cores, there exists the eventual cut-off point, where the limit of light speed will deem CPUs with million or billion cores unsuitable. It has become a recent topic of wide discussion whether the Moore's law will still hold for the latest hardware developments[1].

Another recent advancement that has contributed to the performance of computer processing is the creation of the "Cloud" infrastructure. Although the similar distributed computing paradigm has been investigated long before the Cloud, it has only been the past decade where such online offerings of hardware infrastructure have become economically viable. Furthermore, the nowadays dominance of "Anything"-as-a-service has contributed to the recent popularization of cloud systems, because of the easy scaling (vertical or horizontal) that the Cloud offers. However, distributed (and cloud) computing hardware is restricted from the same constraint of the speed of light. In fact, the importance of this constraint is many-fold magnified since the individual computing processors of the Cloud are often geographically sparsely located: not interconnected through silicon (as in multicores) but through longer networks (e.g. ethernet cables). In many cases the communication overhead in the Cloud is so profound that poses beside the physical limitation (lightspeed), an algorithmic problem: how the computation can be distributed (split-up) without being slowed down by the communication overhead. The challenge arises: how can we utilize these cloud resources optimally?

These limitations in hardware's raw computing power have moved the attention instead to software, so as to "squeeze" (optimize) the last amount of performance gain possible. Coupled with the programming paradigm shift that the multicore revolution brought — where coming up with parallel algorithms and coding them can often be hard and erroneous — led to a huge burden put to the software development for being fast while all the while being increas-

[1]The Economist - The End of Moore's law `http://www.economist.com/blogs/economist-explains/2015/04/economist-explains-17`
ElectronicsWeekly - Is Moore's law still the law? `https://www.electronicsweekly.com/news/moores-law-still-law-2017-09/`

ingly complex. To reach optimal performance on the computing infrastructure where the software is deployed, the software needs to be aware and be able to control (to some extent) the utilizations of the underlying resources.

Software modeling is a relatively-recently introduced concept to tackle mainly the "pillar" of complexity. It achieves this, by allowing the user to abstract from implementation details and instead focus on the functional correctness of the software. Modeling deals with constructing a higher-level abstraction (model) of the software even before it is actually constructed. A model is governed by a set of formal (concrete) rules which makes it difficult by-definition to introduce errors into the model. Furthermore, this rigor can help reason in a high mathematical level about the internals of the software, and as a result, faults in the design and infrastructure of the software can be detected early on. Often, these formal rules are written as computer programs themselves (proof assistants or theorem provers) which allows *automatic* checking of a model against a set of rules, instead of manually proving its correctness. Little has been done, however, to achieve performance in software modeling comparable to a (lower-level) optimized executable program. There have been some previous efforts [Long et al., 2005, Moreira et al., 2010] to generate (efficient) code from a model and later include it as part of a program, but these do not take the available computing resources into account (and thus cannot exploit them optimally). Furthermore, the integration of such generated code in production code has often been omitted or under-specified. Hence, the question arises: how can we optimize performance of software taking aspects of the available computing resources into account, while still remaining at the high-level of abstraction that is crucial for model-based approaches?

Summarizing, the general trend in programming languages is to move away from explicit implementation details and instead focus on abstraction and code portability (e.g. Java) through high-level formalisms. The software technology is trying to "catch-up" with the hardware developments, but this requires the explicit control of the hardware resources and its optimal usage. **The main challenge that arises is how we can abstract away from implementation details, but still manage the hardware resources at a sufficient abstraction level, so that we can benefit from the underlying performance. In this thesis, our main contribution is to address this challenge by constructing a language to write software which can take advantage of recent hardware developments (multicore, cloud) without many compromises in the levels of abstraction.**

The language discussed in this thesis is a modeling language that engages both pillars of software-engineering, namely complexity and more-importantly performance of execution. To achieve this, we aim to provide an interface

of inter-operation between the model, the production code and the hardware infrastructure where the software runs on. Besides multicore hardware, we also investigate in running the modeling language in the modern distributed (cloud) computing systems.

1.1 Why ABS

We base our modeling language upon the Abstract Behavioral Specification language (ABS), with its development starting in to 2006 [Johnsen et al., 2006]. Even before that, the ABS language is the continuation of the high-level, concurrent Creol modeling language which is in-turn born out of of the well-known first-ever object-oriented programming language SIMULA, that goes back as early as 1965.

ABS is generally regarded as a modeling language. A modeling language differs from a programming language in that its *primary* goal is not to (easily) construct a software product; a modeling language's purpose is merely to help the user lay down information and structure it at one's will. This structured information (model) may or may not later act as a "vehicle" for constructing software. It can still be the case that a model is solely used for the purpose of brainstorming, idea exploration, experimentation, simulation, or even (human) communication. In this respect, models are usually left abstract or even incomplete; this is aided from the fact that a modeling language is usually governed by a small set of well-defined rules to express the information in a high-level as possible. Moving on, ABS is executable — compared for example to the widely-known modeling language UML — since there is a "mechanized" way to interpret its semantics as transition rules (i.e. an operational semantics) and thus attach a "meaning" to every (well-constructed) model. The question arises again as to how then an *executable* modeling language differs from a programming language which also attaches meanings (semantics) to a program (instead of a model). The answer lies in the separation of their purposes: a programming language aims to generate (fast) production code, an executable modeling language only generates code for the purpose of model reduction, visualization and interactive feedback of information. Although performance of execution is not a primary goal, it can become important if the modeler wants to execute larger or more complicated models and interact with them in a timely manner.

Users of a modeling language (modellers) are generally not expert programmers. ABS aims to stay familiar to the average user by supporting a "friendly" object-oriented programming layer which resembles that of Java. ABS offers a

functional layer but unlike other, fully-featured functional programming languages, the language has arguably a smaller learning curve; this is because on one hand its functional features are minimal, and on the other hand, the connection with the object-oriented, imperative world is simpler, compared to monads, type & effect systems, or uniqueness types of other languages.

To further accommodate the average modeler, the ABS ecosystem provides a plethora of development tools: an interactive development environment in the Emacs text editor, a developer plugin for eclipse, interactive debugger and method-call visualizer.

The grammar (syntax) and operational semantics (meaning) of the ABS language are well defined using formal method techniques: in this way the documentation of the language becomes more clear and precise, and more, importantly it enables the rigorous analysis of the language. In fact, many analysis and verification tools have been developed over the course of the years for the ABS language, ranging from termination analysis [Albert et al., 2013], resource analysis [Albert et al., 2015a], deadlock analysis [Giachino et al., 2014], to monitoring [Boer et al., 2013, Wong et al., 2015] theorem proving and full-blown verification [Din et al., 2015, Din et al., 2017].

Commonly in software, and in engineering in general, concurrency and parallelism are two concepts which are both difficult to grasp as well as implement. A major challenge in the design of modeling languages is an appropriate development of a concurrency model. ABS adds support for concurrency and inherent parallelism to the object-oriented paradigm. More specifically, the ABS language combines the Actor model formalism with the notion of the *object* to create the *active object*: the communication to an active object can be as well asynchronous and is encapsulated behind the usual method calls. The language's concurrency model goes a step further and introduces its main, and characteristic, feature of *cooperative scheduling*, also known as (semi-)coroutines. In such a setting, active objects form groups (the so-called Concurrent Object Groups); all active objects inside a group share their computing resources (i.e. thread of execution). A running object can programmatically decide to deliberately yield its control so as another object of the same group can execute, i.e. explicit cooperation which is in contrast to the usual preemption of thread mechanisms.

ABS' concurrency model avoids dangerous programming idioms such as threads and lock mechanisms. The immutability of datastructures in the purely-functional layer together with the notion of future values (write-once placeholders which will be computed in the "future") leads to less race conditions. The fields of an object can only be private, which avoids incidents of pointer aliasing. Lastly, the "yielding of control" of cooperative scheduling

happens in explicit places in the program, which makes it more clear on which are the concurrent interleavings of that program. You can find out more about the concurrency model offered by ABS at Section 2.7.

The challenges that we faced during the development of our modeling language include finding the right programming constructs to translate the model to, executing the model through a fast runtime, and showing that the resulting executed model conforms still to the set of rules laid out by the modeling language (i.e. proving correctness). Besides having a generally efficient ABS implementation, we were faced with the implementation of the "cooperation" feature of ABS which is arguably difficult to implement. We try to address this difficulty by developing an efficient runtime environment for ABS.

1.2 Targetting Haskell

To execute the proposed ABS modeling language we translate it to lower-level Haskell program code. Haskell ([Peyton Jones, 2003]) is a general-purpose programming language that first appeared in 1987; its name derives from the mathematician Haskell Curry. Unlike most existing programming languages, designed by a single person or company, Haskell was designed by a committee of academicians for the purpose of "agreeing on a common (lazy functional) language" (from the talk of Simon Peyton-Jones: Escape from the ivory tower: the Haskell journey). Haskell differs from other functional languages since it is *purely* functional: functions play a key role, but they cannot contain any side-effects. This permits the user to "make better sense" of the program's code through equational reasoning and referential transparency. Still, programming completely without side-effects can be a burden or in certain cases impossible — e.g. interacting with the real-world has side-effects — and for this reason Haskell introduces the concepts of Monads (borrowed from Category Theory) and monadic programming to allow side-effects in the language but without breaking purity: there is a clear distinction at the type-level between purely functional and monadic (side-effectful) code. For this reason the type-system of Haskell has been regarded as a very strong static type-system, with other reasons being the support for parametric polymorphism, class-constrained (ad-hoc) polymorphism, type-level programming, datatype-generic programming [Gibbons, 2007], and a limited form of dependently-typed programming [McBride, 2000]. The semantics of Haskell is *by default* call-by-need (also known as lazy). Compared to the commonly-found strict semantics (call-by-value and call-by-reference), Haskell expressions and their sub-expressions will only be evaluated at the specific part that is required by the computation.

Furthermore, unlike the similar call-by-name semantics, lazy semantics will avoid re-computing already evaluated (sub)expressions, which leads to better sharing. Last, lazy semantics admits more expressive power for the language (e.g. when dealing with infinite data structures). Still, the language allows for partially (in places) introducing strictness which may improve the program's performance — most functional languages are strict by-default and optionally lazy.

The choice of Haskell was made since it provides language features that closely match those of the functional layer of ABS, and also certain runtime facilities that make the translation of ABS more straightforward. First of all, both languages offer a purely-functional layer: whereas ABS restricts the mixing of pure and impure code at the syntactic level, Haskell achieves this instead on the type-level. Furthermore, their type-systems share certain commonalities, that is algebraic datatypes with support for parametric polymorphism, ad-hoc polymorphism through ABS interfaces - Haskell's typeclasses. Finally, the module system of both language is quite similar; in fact, the ABS module system was inspired from that of Haskell.

The Haskell type system has been formalized in [Sulzmann et al., 2007, Eisenberg, 2015]. However, the operational semantics of Haskell, and specifically that of the GHC Haskell compiler is hypothetical (not been proven correct yet) as the author say: "It is hypothetical [the semantics] because GHC does not strictly implement a concrete operational semantics anywhere in its code. While all the typing rules can be traced back to lines of real code, the operational semantics do not, in general, have as clear a provenance." Still, since both languages are very similar and stay on the same (high) level of abstraction, it enabled us to prove the correctness and resource preservation of the translation of a subset of ABS to a subset of Haskell (with continuations) which is detailed in Section 3.7.

At the runtime side, the canonical Glasgow Haskell Compiler (GHC) provides a fast and well-tested runtime system where we base the concurrency mechanisms of ABS upon. GHC's features support such as first-class continuations, lightweight (green) threads, load-balancing of threads to multicores for automatic parallelism gain (also known as the M:N hybrid thread model), parallel garbage collection, STM-based datastructures (software transactional memory) among others allowed us to straightforwardly express and thus implement the ABS concurrency abstractions, and more importantly the cooperative scheduling of ABS, in terms of Haskell constructs.

Finally, albeit not directly related to Haskell as the target language, Haskell was chosen as the host language to write the ABS-to-Haskell transcompilation phase, since Haskell is arguably regarded as one of the best languages to

write compilers, reasons ranging from the support for brevity through algebraic datatypes, pattern matching, recursion to compilation's safety and correctness provided by the language's elaborate & strong type system.

It is worth noting that we opted against using Haskell directly, but only through a translation. Although Haskell can be very expressive and safe, e.g. monads, its user learning curve is steep with many concepts rooted in the category theory of mathematics, e.g. again monads. Furthemore, these concepts are yet to reach a status of mainstream, so the average user that writes software programs is most likely impervious to them.

Through the translation of ABS to Haskell, we manage to contribute also to the ecosystem of Haskell:

- a Haskell runtime library to express cooperative scheduling.
- a methodology of providing the object-oriented paradigm for Haskell, which Haskell normally lacks, as the consequence of implementing it in our ABS translation.

1.3 Validation

This work has been carried out in the context of the Envisage Project. The ENVISAGE project is a EU-funded project for:

> The development of a semantic foundation for virtualization and service-level agreements (SLA) that goes beyond todays cloud technologies. This foundation makes it possible to efficiently develop SLA-aware and scalable services, supported by highly automated analysis tools using formal methods. SLA-aware services are able to control their own resource management and renegotiate SLA across the heterogeneous virtualized computing landscape.

Our work was validated on two case studies: an industrial case study of the cloud services offered by the SDL-Fredhopper company `https://www.fredhopper.com/` and a case study on the Preferential Attachment problem of dynamics, which is concerned with the efficient generation of social-network-like graphs.

1.4 Outline

Chapter 2 *Abstract Behavioral Specification* (ABS) [Johnsen et al., 2010a] is a formally-defined language for modeling actor-based programs. An ac-

tor program consists of computing entities called *actors*, each with a private state, and thread of control. Actors can communicate by exchanging messages asynchronously, i.e. without waiting for message delivery/reply. In ABS, the notion of actor corresponds to the *active object*, where objects are the concurrency units, i.e. each object conceptually has a dedicated thread of execution. Communication is based on asynchronous method calls where the caller object does not wait for the callee to reply with the method's return value. Instead, the object can later use a *future* variable [Flanagan and Felleisen, 1995, Boer et al., 2007] to extract the result of the asynchronous method. Each asynchronous method call adds a new *process* to the callee object's process queue. ABS supports *cooperative scheduling*, which means that inside an object, the active process can decide to explicitly suspend its execution so as to allow another process from the queue to execute. This way, the interleaving of processes inside an active object is textually controlled by the programmer, similar to coroutines [Knuth, 1973]. However, flexible and state-dependent interleaving is still supported: in particular, a process may suspend its execution waiting for a reply to a method call.

Chapter 3 Whereas ABS has successfully been used to model [Wong et al., 2012], analyze [Albert et al., 2014a], and verify [Johnsen et al., 2010a] actor programs, the "real" execution of such programs has been a struggle, attributed to the fact that implementing cooperative scheduling efficiently can be hard (common languages as Java and C++ have to resort to instrumentation techniques, e.g. fibers [Srinivasan and Mycroft, 2008]). This led to the creation of numerous ABS backends with different cooperative scheduling implementations:[2] ABS→Maude using an interpreter and term rewriting, ABS→Java using heavyweight threads and manual stack management, ABS→Erlang using lightweight threads and thread parking, ABS→Haskell using lightweight threads and continuations.

Implementing cooperative scheduling can be non-trivial, even for modern high-level programming languages (e.g. Java, C++) because of their stack-based nature. A recent relevant technology is to use *fibers* [Srinivasan and Mycroft, 2008], which adds support for cooperative threads by instrumenting low-level code (commonly via bytecode manipulation) to save and restore parts of the stack. We instead opted for source-to-source translating ABS programs to Haskell, a functional language with language-level

[2] See `http://abs-models.org/documentation/manual/#-abs-backends` for more information about ABS backends.

support for coroutines, based on the hypothesis that a high-level translation serves as a better middleground between execution performance and most importantly semantic correctness. Our transcompiler translates ABS programs to equivalent Haskell-code, which is then compiled to native code by a Haskell compiler and executed. Prior alternative approaches for executing ABS have been an Erlang translator, that utilizes Erlang's preemptive lightweight processes to simulate cooperative threads, and a Java translator, that manages a global dynamic pool of heavyweight threads.

Furthermore, we present and discuss a formal translation of a actor-based language with *cooperative scheduling* (a subset of ABS) to the functional language Haskell. Here we make use of a different, more high-level translation of ABS to Haskell than the translation implemented in the ABS→Haskell backend. This formal translation is proven correct with respect to a formal semantics of the source language and a high-level operational semantics of the target, i.e. a subset of Haskell. The main correctness theorem is expressed in terms of a simulation relation between the operational semantics of actor programs and their translation. This allows us to then prove that the resource consumption is preserved over this translation, as we establish an equivalence of the cost of the original and Haskell-translated execution traces. Finally, the method that was developed is general but applied only to a subset of ABS; for future work we consider to apply this method for all ABS constructs for formally verifying the complete ABS language.

Chapter 4 In this chapter we discuss an extension of ABS to write software that can programmatically take control of its computing (hardware-virtualized) resources. This type of programming which we name "resource-aware" programming differs from the usual emulation or hardware description languages, such as Verilog, because it does not focus on the design of (new) hardware but on how software can take advantage and be "aware" of the underlying hardware. We construct an integrated tool-suite for the simulation of software services which are offered on the Cloud hardware. The tool -suite uses the Abstract Behavioral Specification (ABS) language for modeling the software services and their Cloud deployment. For the real-time execution of the ABS models we use a Haskell backend which is based on a source-to-source translation of ABS into Haskell. The tool-suite then allows Cloud engineers to interact in real-time with the execution of the model by deploying and managing service instances. The resulting human-in-the-loop simulation of Cloud services can be used both for training purposes and for the (semi-)automated support for the real-time monitoring and management of the actual service

instances and their computing resources.

Chapter 5 Cloud technology has become an invaluable tool to the IT business, because of its attractive economic model. Yet, from the programmers' perspective, the development of cloud applications remains a major challenge. In this paper we introduce a programming language that allows Cloud applications to monitor and control their own deployment. Our language originates from the Abstract Behavioral Specification (ABS) language: a high-level object-oriented language for modeling concurrent systems. We extend the ABS language with Deployment Components which abstract over Virtual Machines of the Cloud and which enable any ABS application to distribute itself among multiple Cloud-machines. ABS models are executed by transforming them to distributed-object Haskell code. As a result, we obtain a Cloud-aware programming language which supports a full development cycle including modeling, resource analysis and code generation.

This thesis is derived work from the publications:

- Bezirgiannis, N. and Boer, F. d. ABS: A High-Level Modeling Language for Cloud-Aware Programming In SOFSEM 2016
- Albert, E., Bezirgiannis, N., Boer, F. d., and Martin-Martin, E. A Formal, Resource Consumption-Preserving Translation of Actors to Haskell. In LOPSTR2016
- Azadbakht, K., Bezirgiannis, N., Boer, F. d., and Aliakbary, S. A High-level and Scalable Approach for Generating Scale-free Graphs Using Active Objects. In SAC2016
- Azadbakht, K., Bezirgiannis, N., and Boer, F. d. (2017a). Distributed Network Generation Based on Preferential Attachment in ABS. In SOFSEM2017
- Azadbakht, K., Bezirgiannis, N., and Boer, F. d. (2017b). On Futures for Streaming Data in ABS. In FORTE2017
- Bezirgiannis, N., Boer, F. d., and Gouw, S. d. Human-in-the-Loop Simulation of Cloud Services. In ESOCC2017.

The paper "Human-in-the-Loop Simulation of Cloud Services" was awarded the *Best Paper* of the Conference: (ESOCC) 6th European Conference on Service-Oriented and Cloud Computing.

Finally, all code developed during this thesis can be found at the git repository:

Chapter 3	[Bezirgiannis and Boer, 2016] [Albert et al., 2016] [Azadbakht et al., 2016]
Chapter 4	[Bezirgiannis et al., 2017]
Chapter 5	[Bezirgiannis and Boer, 2016] [Azadbakht et al., 2017a] [Azadbakht et al., 2017b]

Table 1.1: Contribution of publications to chapters of the thesis

`https://github.com/abstools/habs`

Note that the code was still in active development during the writing of this thesis; therefore, the latest implementation code might not reflect the hereby-included code snippets.