Software development by abstract behavioural specification

Serbânescu, V.

Chapter 1

Introduction

1.1 Research Context

Software development or software engineering is one of, if not the most sought after skill for anything that is technology based, whether it's business, industry, research or even when this skill is learned simply as a hobby. The importance of writing fast, clean code that is easy to test, debug and works correctly is essential for the development and maintenance of any prototype, product and system. It is so significant that research into the development process and planning has become as important as the coding process itself. Several product development methodologies [PR94] exist that address different issues and achieve different goals.

Software modeling and modeling languages have the purpose of facilitating product development by designing correct and reliable applications. By abstracting from the implementation details, program complexity and inner workings of libraries, these approaches allow for an easier use of formal analysis techniques and proofs to support the reliability and robustness of a product when it is designed. However there is still a gap that exists between modeling languages and programming languages with the process of software development often going though two separate parts with respect to modeling and implementation. The Unifying Model Language [RJB04] is a general modeling language that provides a standard way of visualization of an application or software model. It provides a visual representation of the layout of a program through standard diagrams that represent the the program's structure in terms of classes, objects, methods and functions depending on the code's programming paradigm. This provides a connection between the software model and the program that implements it, however it is still treated as a separate process in relation to the actual coding effort required.

Software Development A product or system has several quality criteria that are defined that range from expected correct behavior to specific performance requirements. Specifically an application development process starts from a set of requirements with the end product expected to meet those requirements. The process itself involves several well-known methodologies that vary in terms of flexibility and structure.

Early engineering strategies focused on very structured planning based on the waterfall model [PWB09]. This followed a very linear approach that started with applications requirements followed by architectural design, implementation, verification and finally product maintenance. These models are very well described, intuitive and easy to follow by all those involved in the development process. However they have little flexibility and changes to the application after a phase is finished can be very expensive and difficult to make.

Most recent methodologies now focus on more agile approaches [BBVB+01, Mar02] that take into account a lot of feedback and flexibility from both a client and a developer perspective. Products are developed much faster in cycles knows as *sprints* [Sch97]. At the end of a sprint, a finalized version of the product or prototype is tested by a set of clients that may change requirements, detect errors or bugs and provide feedback to set up a new sprint. After several sprints, a certain level of system stability is reached and the product is released. These methodologies focus heavily on fast coding of a feature or improvement and quickly testing it until it has the desired functionality. The programs are in general very difficult and complex to formally analyze especially if they involve a certain degree of parallelism.

In general, software methodologies do not include any formal verification and analysis concepts in software development process. The research conducted in this thesis is a step towards integration of formal methods at programming level. It centers around a scientifically mature modeling language called Abstract Behavioural Specification (ABS) [JHS+12] which was developed in the European Project "Highly Adaptable and Trustworthy Software using Formal Models(HATS)" [hat]. This project and the ABS language were continued into the project "Engineering Virtualized Services(ENVISAGE) [AdBH+13] of which this thesis research was part of. The modeling language provides a high-level integration between formal methods and some of the most important concepts of parallel and concurrent programming: asynchronous communication, actors and futures.

Concurrency Models Software development is very closely related to the hardware it runs on. Hardware evolves at a much faster rate with processing power, memory size and, most importantly, the number of cores per chip increasing rapidly and coming at a very affordable price. Most popular mobile devices and laptops now come with eight cores and hyper-threading capabilities allowing, in theory at least, a high-degree of parallelism. Added to this is the transition of computing capability from a purchasable device to a utility to be used on demand through cloud services.

On the other hand developing parallel software is still constrained by Amdahl's Law [HM08], as an application can never be faster that its sequential portion. As such, a lot of care needs to be taken with parts of an application that deal with data aggregation [SPCA14], joining multiple threads or performing reduction operations as these must not make up the significant computation part of the program. In high performance parallel applications, threads are the entities form the basis of concurrency. Every programming language proposes several thread-abstractions with the purpose of making parallel programming easier for the developer with respect to data concurrency, error handling and debugging. Programming languages offer constructs for thread-management that, depending on the language, give the programmer a certain level of control on the parallelism introduced and the performance and speed-up that they can expect. In Java, the constructs are high-level wrappers that give the developer an intuitive means of configuring the creation, management and scheduling of threads. However it is the role of Java Virtual Machine(JVM) to handle thread allocation, context switches and reloads on the system. It is the JVM that decides how these virtual threads are mapped on the native threads of the system. On the other hand languages like C++ offer, through several libraries, the possibility to directly map virtual threads to native threads offering a clear parallelism degree, but more low-level and difficult programming constructs [But97]. In general the higher the level of abstraction for threads, the more memory the representation occupies as a price to a more seamless programming experience.

The Actor-based model of computation [Agh85, KSA09] is particularly tailored for writing concurrent and distributed systems. It abstracts from explicit locking and thread management, making it easier to write correct concurrent and parallel systems. Actors represent processes that execute in parallel and interact via asynchronous communication through entities called messages. In contrast to the object oriented model underlying Java, the Actor model is an inherently concurrent execution model based on asynchronous message passing. Moreover the Actor based model includes many important features such as encapsulation, scheduling policies, location transparency and locality of references which makes the actor model a suitable programming model for distributed, parallel and concurrent programming [LDMA09]. The actor extensions that have been implemented in terms of the multi-threaded model of execution provided by JVM have a general approach of creating an application-level context in which all actors are initialized. The messages passed in between actors have a general super-type which the actors have to cast, pattern-match or otherwise process at runtime in order to execute a particular control sequence. The difficulty of this event-driven passing of data is observed when bugs appear in exchanged messages as handling errors can only be performed at runtime making testing of actor models quite cumbersome.

The ABS language proposes a new actor programming model based on asynchronous method invocation. Actors can only interact with each other via calls to methods exposed by their implemented interfaces. As such, this approach lifts the detection of possible bugs to compile-time. The equivalent of a wrong message sent to an actor is detected by ABS's semantic analyzer as a method call with the wrong argument types, wrong number of arguments or a method that is not exposed by the actor's interface at all. This "programming to interfaces" discipline brings ABS closer to mainstream programming languages with respect to syntax. ABS is however a modeling language with very powerful formal verification tools that have been built around it that allow resource-analysis, deadlock detection and real-time simulation. Its concurrent model is only at a modeling level and still requires a programming language to execute in a real computing environment.

Programming Paradigms Software development today has a very wide spectrum of applications and thus software is written using a variety of programming paradigms and mainstream languages. Specifically, programming languages offer different paradigms that affect both the way the code is written but also how fast it runs. For simple applications, the most straightforward approach is writing code in a sequential single-threaded manner. The coding approach can be typically an object-oriented programming pattern (OOP) with mainstream languages like Java, C++ or Python. These languages are generally based on encapsulating behavior and state into objects make up parts of the application and interact with each other to perform the correct functionality of the application. In general this approach is very well structured and intuitive, making it very easy to learn for developers. Another typical approach to coding is a functional paradigm using several popular languages like Erlang, Haskell or Scheme. This approach uses function definitions, function applications, expressions and declarations to define the flow of a program and meet an application's requirements based on the input given. In general the code that uses this approach is much more compact and the runtime of functional languages is optimized for such a paradigm to ensure a very good performance. A very popular language that offers the possibility to write code both in a functional style and object-oriented is Scala which also makes this language difficult to classify. It plays a very important role in the research conducted in this thesis, being used to provide both a programming API and a backend as part of the contributions of this thesis.

1.2 Research Objectives

The overall objective of this research and this thesis is bridging the gap between modeling and programming in order to provide a smooth integration between formal methods and two of the most well-known and used languages for software development, the Java and Scala languages. The research focuses mainly on sequential and highly parallelizable applications, but part of the research also involves some theoretical proposals for distributed systems. It is a first step towards having a programming language with support for formal models.

Objective 1 The first specific objective is to develop a runtime that implements the proposed concurrency model of the Abstract Behavioural Specification (ABS) Language and its features. This objective mainly focuses on performance and scalability of the implementation. The runtime has the purpose of optimal thread management in the Java language and provides a full integration of actors and futures with asynchronous programming [IS12].

Objective 2 The second objective of this research is to bring the high-level modeling constructs, especially those that model asynchronous programming and concurrent behavior in ABS to the level of the Java language through an API. The base API, called JAAC is exposed Java and presents some constructs which are quite permissive with regards to type checking of the proposed asynchronous call and vulnerable to unwanted code being run on actors. The API is extended to Scala syntax, called ASCOOP, where all (a)synchronous method calls are now type checked by the Scala compiler allowing only calls to methods that are exposed by a class or interface.

Objective 3 The third specific objective is the development of a compiler from the software model to Java and extended to Scala that provides a formally correct translation and behavior. This translation fully supports the semantics of the core modeling language which includes modeling actors and actor groups as software components. The compiler support includes asynchronous communication, coroutine suspension and resume constructs. Finally the compiler translates the ABS language extensions for timed-models and resource consumption.

1.3 Research Challenges

Each of the specific objectives has a series of challenges encountered throughout each objective's roadmap.

For the first objective, the runtime has two big challenges in terms of correctness and performance. The concurrency model of ABS includes asynchronous communication, concurrent object groups and coroutine support which pose a difficult problem in production code, especially in a JVM based language. The runtime also needs a module to support two extensions of ABS: symbolic time and resource simulation. A lot of the implementation requires the careful usage and management of threads, as in JVM these entities are quite expensive and significantly affect performance. The proposed runtime must be transparent to the programmer, but under the hood it must not affect the Quality of Service (QoS) [Kan02] of a system. This runtime is being used by both the proposed API and compiler so it must not slow the program down in any way. Another challenge of runtime that requires a lot of research effort is implementing a model that is used to correctly emulate the coroutine extension of ABS.

As already explained in the second objective's description, the proposed API is difficult to type-check in Java and can result in erroneous behaviour being programmed. Even when extended to Scala, the API must be user-friendly and easy to learn in order to present a viable alternative to existing libraries that support part of the concurrency model of ABS.

The third objective presents a big challenge in translating the coroutine model of ABS and it involves the correct pre-processing of methods that contained the coroutine constructs of ABS. This requires a formal proof of the translation scheme from the ABS coroutines model to the model for emulating coroutines in the proposed runtime.

1.4 Main Contributions and Thesis Outline

Main Thesis Publications and Contributions This thesis is based on the following six publications :

No.	Title	Proceedings / Journal / Book	Year
1.	Towards Type-based Optimizations in Distributed Applications using ABS and Java 8 [SNA+14].	1st Workshop of Adaptive Resource Management and Scheduling for Cloud Computing	2014
2	A design Pattern for Optimizations in Data Intensive Applications using ABS and Java 8. [SAB+15]	Concurrency and Computation: Practice and Experience	2015
3.	A Java-Based Distributed Approach for Generating Large-Scale Social Network Graphs [ŞAdB16]	Resource Management for Big Data Platforms	2016
4.	Actors with Coroutine Support in Java. [SBJ18a]	15th International Conference on Formal Aspects of Component Software	2018
5.	ASCOOP: Actors in Scala with Cooperative Scheduling. [SBJ18b]	21st IEEE International Conference on Computational Science and Engineering	2018
6.	On The Nature of Cooperative Scheduling in Active Objects. [SB]	The 35th ACM/SIGAPP Symposium On Applied Computing	2020

Application Contributions The systems developed in this thesis have also had important contributions in the following manuscripts:

No.	Title	Proceedings / Journal / Book	Year
1.	High Performance Computing Applications using Parallel Data Processing Units. [ASdB15]	6th International Conference on Fundamentals of Software Engineering	2015
2.	Multi-threaded Actors. [AdBS16]	9th Interaction and Concurrency Experience	2016
3.	A Survey of Active Object Languages. [BSH+17]	ACM Computing Surveys	2017

Thesis European Projects

No.	Name	Years
1.	Engineering Virtualized Services - ENVISAGE	2013-2016
2.	From Inherent Concurrency to Massive Parallelism through Type-based Optimizations - UPSCALE	2014-2017

Applications Repositories A significant part of this thesis is the programming effort that was put into developing three applications that are at the core of the main objective. The first application is an implementation in Java of the ABS concurrency model and behaviour with all of its core features including asynchronous communication and cooperative scheduling. The sources for this implementation are found at `https://github.com/JaacRepo/JAAC.git` and can be built into a jar library that provides a base API in Java for these features.

The second application is a compiler that provides a source to source translation between ABS code and Java code. This essentially turns an ABS model into an executable binary that can run directly on a computer. The sources of the compiler can be found at `https://github.com/JaacRepo/absCompiler.git`.

The third application provides an extended API in Scala called ASCOOP, to be used directly in Scala programs to write code using a syntax very similar to ABS that emulates the ABS features directly. The repository for this API is the same as the first one, with the jar library that can now be linked to a Scala project and use the constructs to write the code.

Thesis Outline The research presented in this thesis is organized based on the research questions and specific objectives presented in this chapter.

Chapter 2 The second chapter presents a preliminary overview of the the concurrency elements in the two programming languages that the thesis is focused on: Java and Scala. It focuses on some basic utilities for parallel programming provided by Java, and then dives into some high-level abstractions and wrappers that facilitate concurrent programming like future and actors. It covers the limits that Java has on emulating asynchronous communication, as the language does not provide a direct construct for this. The chapter also discusses the existing relations between actors and futures, as well as existing support for coroutines in the Java language.

Chapter 3 The third chapter describes the main features concerning actors, programming paradigms, asynchronous communication and coroutines that the ABS modeling language supports. This chapter is an overview of the core concepts and two of its extensions that are covered in this thesis.

Chapter 4 The fourth covers the Java library called JAAC[1] which serves as a backend to be used for generation of programs in Java from ABS sources, as well as allowing programmers to write Java programs directly following the ABS concurrency model. This library provides a bridge between modeling and programming: by *reverse engineering* the ABS model underlying such an application of the library API we can apply the formal development and analysis techniques supported by the ABS language, e.g. functional correctness [DBH15], static analysis [AAFM$^+$14] and deadlock detection [GLL16, AdBdV18]. The Chapter is mainly based on papers 1[SNA$^+$14] and 4[SBJ18a] .

Chapter 5 The fifth chapter presents a comparative analysis between using software modeling and programming as two separate entities brought together either through a compiler or direct integration in the programming language. In the second part of this chapter the previous iterations of the Java runtime for the modeling language are presented, focusing on how each contributed to the current solution. The basis for this chapter are the investigative research conducted in papers 2 [SAB$^+$15] and 3 [ŞAdB16].

Chapter 6 The sixth chapter shows the use of the main features of the API through some coding examples in Java, where the API is slightly permissive and vulnerable to programmer mistakes without providing errors or warnings, whereas these issues are mitigated through the API's extension in Scala where such vulnerabilities yield errors at compile time. The chapter is based on paper 5 [SBJ18b]. The entire implementation is based on replacing synchronous calls with asynchronous calls followed by an await (suspension) on the created implicit future. The implementation also includes an underlying scheduler which differentiates priorities between explicit futures created by the program and implicit futures [FRCH$^+$19] created by the replacement mechanism of synchronous calls.

Chapter 7 The seventh chapter covers the challenges brought by the implementation of the compiler. The compiler is a direct translation in Scala, and presents work done on pre-processing continuations to support emulation of co-routines. Also the chapter provides a formal proof as to how these co-routines

[1] `https://github.com/JaacRepo/JAAC`

are correctly compiled and expected behaviour is obtained. The chapter is based on paper 4 [SBJ18a] and 6 [SB].

Chapter 8 The eighth chapter shows the use of both approaches (the API and the compiler) in several case studies and benchmarks that this research contributed to. First the runtime system is evaluated against ABS's most stable and well-known backend (that uses the Erlang language and its lightweight threads) using some typical actor benchmarks. The ASCOOP API is compared to exiting actors libraries like Scala and Akka. Finally, the ABS compiler is used to model several real world case studies that were part of the European Projects like a Map Reduce application, a cache and memory system simulation and a model of a railway system.

Chapter 9 The last chapter draws the conclusions of this thesis and how far this research accomplished its objectives. It summarizes the main challenges encountered in attempting to bridge the gap between modeling and programming in the general software development process and the lessons learned.