1 Introduction

Object-oriented programming [23, 120] has been of one the dominant paradigms for software engineering. Object orientation provides principles for abstraction and encapsulation. Abstraction is accomplished by high-level concepts of interfaces, classes, and objects (class instances). Encapsulation provides means to hide implementation details of objects such as their internal state. SIMULA 67 [44] introduced notions of classes, subclasses, and virtual procedures. In the main block of a program, objects are created, then their procedures are called. One object interacts with another object using the notion of a method. Method invocations are *blocking*; i.e. the caller object waits until it receives the result of the method call from the callee object. This model of interaction was not the intention of the pioneers of the paradigm: object interactions were meant to be *messages* among objects and objects behaved as autonomous entities possibly on remote locations on a network; Alan Kay clarified later [95, 96]. On the contrary, almost all the object-oriented languages at hand have followed the blocking and synchronous model of messaging. Object-oriented programming has inspired another paradigm: the actor model.

One of the fundamental elements of the actor model [4, 3] is *asynchronous* message passing. In that approach, interactions between objects are modeled as non-blocking messages. One object, the sender, communicates a message to the other object, the receiver. In contrast with abstractions provided by object orientation, a message is not bound to any interface in the actor model. At the receiver side, a message is matched with possible patterns and when a match is found, the message is processed by the receiver. Actor model features location transparency; i.e. the physical location of objects is not visible to other objects. A system is composed of objects that communicate through messages.

A considerable amount of research has been carried out to combine object-oriented programming with the actor model [131]. Language extensions, libraries, and even new programming languages are the outcomes of such research. For example, [94] presents a comparative analysis of such research for Java and JVM languages.

Multicore and distributed computing raised a new challenge: combining object orientation, the actor model, and *concurrency*. Concurrency motivates utilizing computational power to make programs run faster. One goal has been to combine concurrency with object-oriented programming. However, due to an exponential

number of interleavings, concurrency makes it harder to verify programs in terms of correctness of runtime behavior [72, 85, 5]. Concurrency has different forms in different paradigms.

Existing Concurrency Models In a concurrent setting, coroutines [41, 102] enable interactions with collaborative pre-emption: a coroutine has no *return* statement, but it may explicitly *yield* to another coroutine (transfer control of the execution). The yield relation is symmetric. Creating an instance of a coroutine spawns a new process with its own state. A coroutine allows multiple entry points for suspending and resuming execution at explicitly specified locations. The explicit interleaving points supports a compositional understanding of the code [75]. Coroutines were originally features in an object-oriented programming language in SIMULA 67 [44]. Object orientation is based on caller-callee interaction, which is asymmetric: the caller invokes a method in the callee and *blocks* until the corresponding *return* occurs in the method of the callee. Object orientation focuses on an object-view with caller-callee interaction and stack-based execution whereas coroutines focus on a process-view with flexible transfer of control.

In a concurrent setting with objects, multi-threading is a common approach to provide an object-oriented concurrency model. A thread holds a single stack of synchronous method calls. Execution in a thread is sequential. The stack has a starting location (i.e. start of invocation) and a finish location (i.e. the return point). Future values can be used to hold the eventual return value of a method at the call site [105, 46]. In a caller-callee setting, an object calls methods from other objects. If multiple threads execute simultaneously in an object, their instructions are interleaved in an uncontrolled manner. In coroutines, the points of interleaving are explicit (i.e. *yield*) whereas multi-threading is based on implicit scheduling of method invocations. Furthermore, interactions in both coroutines and multi-threading are blocking and synchronous. In contrast, the actor model relies on asynchronous communication (which is non-blocking) as one of its core elements.

In the actor model, all communication is asynchronous. The unit of communication is a *message*. A queue of messages (the inbox) is utilized among actors in the system. The notion of a message is not bound to a specific definition or object interface. When an actor receives a message, it may use pattern matching techniques to extract the content of the message. When the actor completes the processing of a message, it may decide to reply to the message by sending another message. The actor model works based on run-to-completion style of execution. While a message is processed, an actor cannot be pre-empted or intentionally yield to allow other actors in the system to make progress. Integration of actor model and object orientation leads to the use of asynchronous method calls.

Problem Statement and Approach The main challenge is to generate production code from an actor-based language which supports asynchronous method calls and coroutine-style execution. We take Java [63] as the target programming language because it is one of the mainstream languages [150] and it offers a wide range of mature and production-ready libraries and frameworks to use.

There exist actor-based executable modeling languages that support asynchronous method calls and coroutine style execution; e.g. Rebeca and ABS. Modeling languages are used to *model* software systems for understanding, analysis, and verification. Execution of models is restricted to simulation. As such, they do not generate production-ready code intended to run industrial environments.

Rebeca [145, 143, 144] is a modeling language for reactive and concurrent systems. In Rebeca, a number of reactive objects (*rebecs*) interact at runtime. Each rebec has its own unique thread of control and an unbounded queue of messages. Rebecs interactions are based on asynchronous message passing. Each message is put in the unbounded queue of the receiver rebec and specifies a unique method to be invoked when the message is processed. Rebeca uses run-to-completion execution and does not support future values.

ABS [87, 67] is a modeling language for concurrent objects and distributed systems. ABS uses asynchronous communication among objects. A message in ABS is generated from a method enclosed by an interface. this defines the interface of the sent messages. In addition, ABS supports future values in its asynchronous communication. ABS introduces `release` semantics that is based on co-operative scheduling of objects; i.e. similar to that of *yield* in coroutines [88]. The ABS semantics is completely formalized by a structural operational semantics [133, 86]. This allows ABS models to take advantage of a wide range of static and dynamic analysis techniques [157, 47, 57, 7]. Co-operative scheduling in ABS has been additionally extended for real-time scheduling with priorities and time constraints [18, 89]. All the above characteristics make the ABS language an attractive choice if it can be used as a programming language at industrial and business scale. ABS has mainly developed as a modeling language; for example, ABS does not support a common I/O API for files or networking, and it does not provide a standard library of data structures.

JSR 166 [91] introduced a new concurrency model [104] in Java 1.5. The concurrency model lays the grounds to support for the actor model with asynchronous method calls. This gave rise to various research to support actor model on Java [94] and further extend Java as a functional programming language [129, 71, 126, 24]. From an industrial perspective, the development of the Java language from version 6 to version 8 experienced a slow pace that influenced the growth of businesses

and justified alternatives. Scala [128], a dynamic, functional, and object-oriented language on JVM, rose to the challenge to fill the gap during the period between Java 7 and Java 8. Java 8 [64] alleviated this gap by releasing fundamental features such as lambda expressions that are essential for concurrency and functional programming.

One core challenge is how to create a mapping of coroutines to multi-threading in Java. Supporting coroutines in Java can be mostly classified in two major categories. One category relies on a *modified* JVM platform (e.g. Da Vinci JVM) in the implementation of thread stack and state such as [148], [149] and [111]. The other category involves libraries such as Commons Javaflow[1] or Coroutines[2] that utilize byte-code modification [45] at runtime in JVM. In this research, we did not intend to use any of the two above approaches. Custom or modified JVM platform implementations are not mainstream and not officially supported by the Java team which jeopardizes portability. To keep up with new versions of the Java language, research and development of a modified JVM requires explicit and periodic maintenance and upgrade effort. Moreover, as byte-code modification changes the byte-code of a running program or inserts new byte-code into JVM at runtime, this complicates reasoning and correctness analysis/verification [110, 109]. We aim to rely only on the mainstream JVM platform released by the Java team at Oracle. Moreover, we do not intend to use byte-code engineering used for instance in Encore [52].

Another straightforward way to support coroutines in Java multi-threading is that since a thread owns a single stack, we can translate every invocation (entry point) in a coroutine to a new thread in Java [140, 139]. This naive approach unsurprisingly leads to a poor performance since threads are resource-intensive in Java and the number of threads is not scalable at runtime due to resource limitations in JVM when the number of objects increase (cf. Chapter 2). Therefore, it is reasonable to use a pool of threads (JSR-166 [91]) to direct the execution of all method invocations (messages). We utilize Java 8 features (JSR 335 [60] and JSR 292 [138]) to model a message as a data structure expressed in a lambda expression (cf. Chapter 4).

1.1 Objectives and Architecture

In this section, we present a high-level overview of design goals of our framework and its implementation.

Polyglot Programming With the rise of distributed computing challenges, software engineering practice has turned to methods that combine multiple programming

[1] `http://commons.apache.org/sandbox/commons-javaflow/`
[2] `https://github.com/offbynull/coroutines`

languages and models to complete a task. In this approach, different languages with different focus and abstractions contribute to the same problem statement in different layers. Polyglot programming essentially enables software practice to apply the *right* language in the appropriate layer of a system. Layers of a system require points of integration. ABS is an attractive choice for the concurrency and distributed layers. Therefore, ABS should be able to provide integration points to other layers. The programmer develops models with ABS that *partially* take advantage of features of another language e.g. Java. This approach is also referred to as *Foreign Function Interface*.

Listing 1: Using Java in ABS

```
1 java.util.List<String> params = new java.util.ArrayList<>();    // Java
2 myObj ! doSomething(params);                                     // ABS
```

Listing 1 shows a snippet of ABS code that uses `java.util.ArrayList` as a data structure. Ideally in ABS, the programmer is able to directly use the libraries and API from Java. This removes the necessity to redundantly repeat definition of common data structures and API at ABS. Furthermore, this allows to take advantage of the already rich and mature library API of Java.

Scalability In distributed systems, the number of messages delivered among objects in the environment is not predictable at runtime. The goal is to ensure the actor system scales in performance with least influence from the number of asynchronous messages delivered in the system. Due to support of co-operative scheduling for asynchronous messages, ABS is a fit for distributed systems.

Separation of Concerns We approach ABS modeling and development with a component-based and modular software engineering practices. The scope of the research spans a number of layers around ABS language:

- *Compiling ABS to a target programming language* One first objective is to compile an ABS model to a target programming language. Target languages potentially include mainstream programming languages such as Java, Erlang, Haskell, and Scala. We propose a new architecture for the ABS tool-set and engineering that enables different programming languages to utilize the same architecture.

- *Using ABS concurrency as an API in an existing programming language* The ABS language syntax and semantics are formalized precisely and rigorously by a structural operational semantics [87]. If a mapping from ABS to a programming API is provided, a programmer is able to take advantage of ABS semantics without directly programming in ABS. This enables industry users

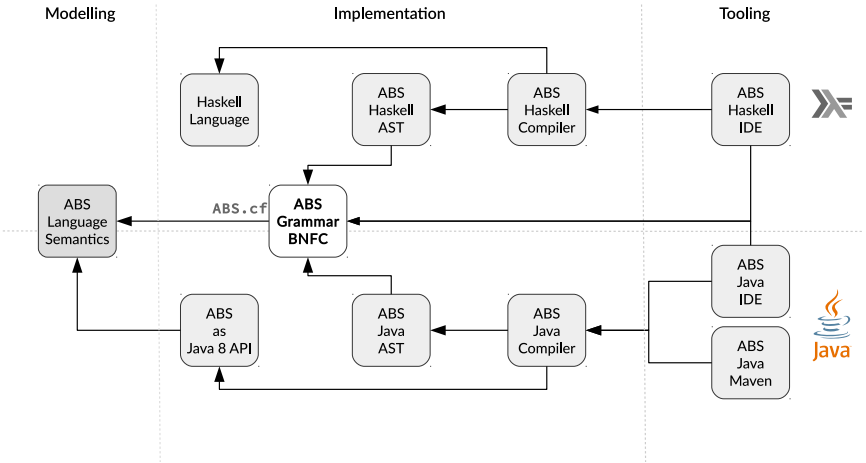

Figure 1.1: General Architecture of ABS API and Java Language Backend

of mainstream languages to model their systems in ABS semantics using the programming languages and platforms they are already familiar with.

- *Modular Architecture of ABS Tools* ABS language provides rigorous semantics to model concurrent and distributed systems. For practical reasons, it is important that the user (that can be a programmer, an analyst, or a researcher) has access to a tool-set and IDE that allows working with ABS models in a user-friendly way. The ABS IDE and tool-set should be easy to reuse and extend.

The above objectives and design principles are realized by the modular architecture presented in Figure 1.1.

1.2 Literature Overview

We briefly discuss related work in the context of programming languages, actor model, and concurrency. In the overview, we distinguish two levels; one is at the level of the programming languages and the other is for the external (third-party) libraries developed for programming languages.

1.2.1 Programming Languages

In this section, we briefly provide an overview of the programming languages that have targeted similar problem statements. Various programming languages, in the past decade, have emerged to provide an actor-based model of asynchronous message passing [131]. Table 1.1 presents different classes of actor model and concurrent model of programming.

Language	Abstraction	Type
Erlang[13, 51]	Process	Implicit By Design
Elixir[152, 50]	Agent	Implicit By Design
Haskell[39]	forkIO & MVars	Implicit By Design
Go[59]	Goroutine	Implicit By Design
Rust[117, 40]	Send & Sync	Implicit By Design
Scala[69]	Akka Actors [3]	External Library
Pony[134, 35]	actor	First-Class Citizen

Table 1.1: Actor Model Support in Programming Languages

Library	Technique	JVM Language
Killim[147, 146]	Byte-Code Modification	Java
Quasar[36]	Byte-Code Modification, Java 8	Clojure, Java
Akka[151, 69]	Scala Byte-Code on JVM	Scala, Java

Table 1.2: Actor programming libraries in Java

First-Class Citizen Languages in which the actor model is by-design part of the syntax and semantics of the language. Pony [134, 35] targets high-performance computing using actor models and shared memory. Having the actor model as part of a language design simplifies formal verification.

Implicit By Design Refers to languages that have no explicit notion of actors in their syntax or semantics, but do provide fundamental constructs for concurrency and asynchronous message passing. Thus, it becomes an easy task in this kind of programming language to create an abstraction to support the actor model by coding.

1.2.2 Frameworks and Libraries

Since programming languages faced challenges to provide the necessary syntax and semantics for actor model and concurrency at the level of the language, many libraries and frameworks aim to fill this gap Table 1.2 presents a summary. We observe that the more the language itself is close to the actor model semantics, the less external libraries and frameworks target this gap. In the following, we briefly enumerate frameworks and libraries for JVM [4].

[3] Scala 2.11.0 adopts Akka as default actor model implementation: `http://docs.scala-lang.org/overviews/core/actors-migration-guide.html`

[4] A more comprehensive list can be obtained at [94] and `https://en.wikipedia.org/wiki/Actor_model#Programming_with_Actors`

Topic	Part	Chapter/Section
Formalization of the mapping from ABS to Java including the operational semantics and ABS co-operative scheduling in Java	Programming Model (Part II)	Chapter 2 and 3
Design and implementation of ABS concurrency layer in Java	Implementation (Part III)	Chapter 4
Monitoring method call sequences using annotations	Application (Part IV)	Chapter 5
Design and implementation of a massive-scale monitoring system based on ABS API in Java	Application (Part IV)	Chapter 6

Table 1.3: Actors at Work – Thesis Organization

One of the main techniques used in libraries to deliver actor programming in JVM is byte-code engineering [45, 26, 127]. Byte-code engineering modifies the generated byte-code for compiled classes in Java either during compilation or at runtime. Although, this technique is commonly used and argued to provide better performance optimization [153], it introduces challenges regarding the verification of the running byte-code [110, 109].

1.3 Outline and Contributions

The core contributions of this thesis target the intersection of object orientation, actor model, and concurrency. We choose Java as the main target programming language and as one of the mainstream object-oriented languages. We formalize a subset of Java and its concurrency API [91] to facilitate formal verification and reasoning about it (cf. Chapter 3). We create an abstract mapping from a concurrent-object modeling language, ABS [87], to the programming semantics of concurrent Java (cf. Chapter 3). We provide the formal semantics of the mapping and runtime properties of the concurrency layer including deadlines and scheduling policies (cf. Chapter 2). We provide an implementation of the ABS concurrency layer as a Java API library and framework utilizing the latest language additions in Java 8 [62] (cf. Chapter 4). We design and implement a runtime monitoring framework, JMSeq, to verify the correct ordering of execution of methods through code annotations in JVM (cf. Chapter 5). In addition, we design a large-scale monitoring system as a real-world application; the monitoring system is built with ABS concurrent objects and formal semantics that leverages schedulability analysis to verify correctness of the monitors [53] (cf. Chapter 6). Table 1.3 summarizes the structure of this text.

In addition, Table 1.4 summarizes the conference and journal publications as a result of this research:

Topic	Proceedings / Journal	Year
Chapter 2	ACM SAC 2012, pp. 1883–1888	2012
Chapter 3	COORD 2013, pp. 181–195	2013
Chapter 4	ISoLA 2014, pp. 37–53	2014
Chapter 5	FACS 2010, pp. 53–70 *and* Journal of Science of Computer Programming, vol. 94, part 3, pp. 362–378	2010 *and* 2014
Chapter 6	ESOCC 2015, pp. 125–140	2015

Table 1.4: Actors at Work – Conference and Journal Publications

All implementations of this thesis can be found at

`https://github.com/CrispOSS/jabs`

and the source of thesis can be found at

`https://github.com/nobeh/thesis`