Cover Page

Universiteit Leiden

The handle http://hdl.handle.net/1887/97598 holds various files of this Leiden University dissertation.

Author: Serbânescu, V.
Title: Software development by abstract behavioural specification
Issue Date: 2020-06-10

Leiden Institute of Advanced Computer Science

Software Development by Abstract Behavioural Specification

Vlad-Nicolae Şerbănescu

2020

Software Development by Abstract Behavioural Specification

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Leiden
op gezag van Rector Magnificus Prof. mr. C.J.J.M. Stolker
volgens besluit van het College voor Promoties
te verdedigen op woensdag 10 juni 2020
klokke 10:00 uur

door

Vlad-Nicolae Şerbănescu

geboren te Boekarest, Roemenië
in 1989

PhD Committee

Promotor:		
	Prof. dr. F.S. de Boer	
Co-promotor:		
	Dr. M.M. Jaghoori	
other members:		
	Prof. dr. F. Arbab	
	Prof. dr. M.M. Bonsangue	
	Prof. dr. A. Plaat	
	Prof. dr. F. Pop	Politehnica University of Bucharest, Romania
	Dr. R. Schlatte	University of Oslo, Norway

The work in this thesis has been carried out at the Center for Mathematics and Computer Science (CWI) in Amsterdam and Leiden Institute of Advanced Computer Science (LIACS) at Leiden University. This research was supported by the European projects FP7-610582 ENVISAGE (Engineering Virtualized Services) and FP7-612985 UPSCALE (From Inherent Concurrency to Massive Parallelism through Type-based Optimizations).

Cover Design: Cristian-Ştefan Neacşu

Contents

List of Figures

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.

Chapter 2

Asynchronous Programming in the Java Virtual Machine (JVM)

The Java language is one of the mainstream object oriented programming languages that supports a programming to interfaces discipline [Lea00]. It has evolved into a platform to design and implement applications in several domains of both research and industry [VLFGL01, PS12, NAB+11], along with supporting its community with new language constructs and features. With applications reaching exascale dimensions in terms of data volumes and requiring a lot of computing power, focus has increased towards the development of numerous libraries and frameworks with an attempt to provide distribution and concurrency at the level of Java language [SPCA14, SPCA15]. Java is an object-oriented language, that is based on high-level concepts of interfaces and classes. These classes contain fields and methods to encapsulate state and behavior. At runtime, dynamically generated class instances (objects) are created and interact with each other via method calls. The general behavior of these calls is synchronous, as a caller object will block its execution until it receives a result from the execution of the method's block inside the callee object. A Java program's entry point is the `main` method which creates a single thread with its own stack and context. The basic sequential execution model generates a stack of method calls between objects that run on a single thread.

The Java Virtual Machine(JVM) allows an application to have multiple user and daemon threads of execution running concurrently. All threads have an assigned priority in the JVM that orders the scheduling of threads. When it is created initially, a thread receives the same priority as its parent, so in general application-level threads have the same priority. A thread may also be marked as a daemon before it is started, allowing the JVM to stop when only daemon threads are left running in the application. This daemon property is also inherited from the parent thread, if not explicitly set. When a JVM is started, a single user thread is created and its execution begins with the block in the `main` method defined in a class. From that point the JVM continues to create, start, suspend and stop threads by following the control flow of the program. The JVM is then stopped either when the `System.exit( )` method is explicitly called by the user or all user threads have completed execution, either normally or exceptionally.

Multi-threaded execution is an essential feature of the Java platform. Parallel execution takes place by spawning a thread that runs code asynchronously on the same JVM. The basic approach of asynchronous execution in Java is to extend the `Thread` class and specify the behavior by overriding its `run` method. This method's body is empty by default and will run on a separate thread. Listing 2.1 contains an example of creating a new Thread object. Once the thread is instantiated, the user needs to explicitly call the `start` method to execute the body of `run` in parallel.

In Java, the most straightforward way of communication between threads is via the state of shared variables in the program memory, as there is no direct manner in which a thread can invoke an instruction

Listing 2.1: Explicit Thread Creation

```
1  class MyThread extends Thread{
2
3    @Override
4    public void run( ) {
5      //thread control flow
6    }
7  }
8
9  //main block or another class
10 MyThread t = new MyThread( );
11 t.start( );
```

on another thread. All objects that are instantiated in a program may be accessed and modified by the running threads which hold a reference to them and it is up to the programmer to ensure that concurrent access is synchronized. Accesses and modifications to objects by more than one thread at a time will lead to corrupting data and race conditions.

In Java, class fields can be defined using the `static` keyword making their scope global. Static fields are loaded and instantiated only once when the class is loaded without requiring an object instance of that class. As such they may be accessed from anywhere in the program by existing threads.

When a new thread is spawned it has a separate call stack and context of execution. The call stack is used to save the frames generated by sequences of nested synchronous method calls between objects. The stack size in general limits the depth of these sequences to the point where a `StackOverFlowError` is generated when there is not enough space to allocate a new frame. The size of each stack depends on the JVM and Operating System(OS) and can also be specified by the user through the program's environment variables and virtual machine (VM) arguments. A typical default value for 64bit Operating Systems is 1MB, and can easily reach as large as 2MB as object references now require 8 bytes of space compared to the 4 bytes required on 32bit systems. With a 2MB stack size, a program that runs 500 threads can allocate 1GB of main memory for the threads alone, even though just a small percentage of those threads would require the full 2MB stack to execute its control flow. This makes the basic mechanism for a block of code to be run asynchronously very expensive memory-wise. The problem further increases when an application requires multiple context switches between several threads keeping the suspended threads living in the system. As this number becomes very large, the thread explosion affects the main memory thus the application's performance.

The first part of this chapters covers the concurrency utilities extending the above basic multi-threaded model to facilitate asynchronous programming and communication in Java. The second part of the chapter presents the motivation for extending the research towards the Scala language by illustrating its lightweight and easy-to-use syntax for concurrent programming through two abstractions: actors and futures. The last part of this chapter covers related work on introducing coroutines in JVM based languages and the challenges encountered when to ensure correct functionality together with actors and futures.

2.1 Java Concurrency Utilities

Java, by design, does not have a straightforward way of specifying asynchronous method calls in programming code. As mentioned before a method call implicitly blocks the caller, and unless specified through the basic thread model (see Listing 2.1), the flow of a program is sequential and on a single thread. However, Java does provide interfaces and classes together with constructs that offer more flexible and intuitive ways of coding asynchronous communication, spawning and managing threads. There are also many Java packages that offer the programmer utilities for specifying synchronization points between threads based on state or method return values. This section summarizes the main classes that facilitate thread management and synchronization from the programmer's perspective.

Spawning Threads Java offers two interfaces that define tasks that are to be run asynchronously and allow the user to specify the piece of code to be run on a separate thread. These are the `Runnable` and `Callable` interfaces that offer the `run` and `call` methods respectively, that contain the control flow to be executed in parallel. The difference between the two interfaces is that objects that implement `Callable` execute methods that return a value, while `Runnable` objects have methods that do not a return value. Listing 2.2 shows how an anonymous class that extends the `Runnable` interface can be passed as a parameter to a `Thread` constructor. This offers a much more compact way of specifying a block of code to run asynchronously, but still requires an explicit start of the thread.

Listing 2.2: Thread Creation using Anonymous Classes

```
1 Thread t = new Thread(new Runnable( ) {
2   @Override
3     public void run( ) {
4       // thread control flow
5     }
6   });
7
8 t.start( );
```

An anonymous class allows instantiation of an object that implements an interface (in this case `Runnable`) without specifying a class name and the behavior of the overridden method (in this case `run`) will only apply to this object. As such, the user bypasses the need to explicitly create a separate class that either implements `Runnable` or extends `Thread`, being especially useful when needing to create small asynchronous tasks. The behavior of the threads is now defined as part of the arguments passed to the `new` construct rather than in a separate class prior to running the task in parallel. This provides programatically more flexibility, the possible parallel behaviors do not need to be defined separately for small functions. Starting with Java 8, instances of `Runnable` and `Callable` can also be created using lambda expressions [Lam]. An example of creating a `Runnable` using a lambda expression is given in Listing 2.3. The code in this listing allows the user to express an instance of a class (like `MyThread`) containing a single method even more compactly. This instance can then be passed as well to a `Thread` constructor or other concurrency utilities to be discussed further in this section.

Listing 2.3: Declaring a Runnable using a lambda expression

```
1 Runnable task = ( )->{
2   //thread control flow
3 }
```

Managing Threads Already since Java version 5, there has been support for concurrent programming through its `java.util.concurrent` package and throughout its development the Java platform has also

introduced new high-level concurrency Application Programming Interfaces (APIs) to facilitate user experience when developing parallel applications. This package introduces data structures with high-level wrappers to allow creation and management of thread pools to facilitate creating and starting threads. Thread pools allow dynamic control of the number of threads created in the application as well as control over the tasks that are accepted. They also expose methods to specify ordering inside thread pool queues and how tasks are scheduled, suspended and completed. The basic interface that supports this is the `Executor` interface which runs submitted tasks. This decouples task submission from the mechanics of how the submitted task will be scheduled, assigned a thread object and run. An executor simplifies the explicit instantiation and start of a new thread in Listing 2.2 using a single call to the `execute` method as shown in Listing2.4 and since Java 8 this can be simplified further by lambda expressions.

Listing 2.4: Executor Usage

```
1  Executor e = //initialization of Executor and its properties
2  e.execute(new Runnable( ) {
3
4    @Override
5    public void run( ) {
6      // thread control flow
7    }
8  });
9
10 //using a lambda expression
11 e.execute( ( )->{
12   //thread control flow
13 });
```

The `execute` method has no default implementation and needs to be overridden in order to specify how to run the tasks submitted to it. The method allows the programmer to define certain scheduling policies and approaches. For example, in Listing 2.5 the `Executor` is extended to call the `run` method synchronously inside the body of `execute`. Therefore all tasks submitted to such an executor by one thread will be run sequentially.

Listing 2.5: Synchronous Executor Implementation

```
1  class SynchronousExecutor implements Executor {
2
3    public void execute(Runnable r) {
4      r.run( );
5    }
6  }
7
```

Another basic approach of extending an executor is such that each task is run in parallel on a separate `Thread` instance as in Listing 2.6. We will see the importance of these two approaches throughout the rest of the thesis.

Listing 2.6: Asynchronous Executor Implementation

```
1  class AsynchronousExecutor implements Executor {
2
3    public void execute(Runnable r) {
4      new Thread(r).start( );
5    }
6  }
```

```
7
```

Java offers several default implementations of the `Executor` interface that have predefined `execute` methods. The `Executors` class provides factory methods to obtain instances of these implementations. This class is different `Executor` class, as it only provides static factory and utility methods to create predefined configurations for various types of executors, without any specific methods that control thread creation, scheduling and execution. One such implementation is the thread pool which allows the programmer to control the number of threads in the system as well as dynamically adjust them during program execution. In this thesis there are three default implementations that were used throughout the development of the solutions presented in the next chapters:

- `Executors.newCachedThreadPool( )`
- `Executors.newFixedThreadPool(int)`
- `Executors.newForkJoinPool()`

The first one creates a thread pool that instantiates new threads when they are required, but will continue to reuse available idle threads that have been constructed previously. The main advantage of this type of thread pool is that it improves the performance of programs that invoke a lot of small asynchronous tasks. One of the case studies researched in this thesis is specifically tailored to test performance of such a program. If a sequence of short tasks are submitted via the `execute` method, the first few tasks will be assigned newly created threads, while subsequent tasks reuse the same threads once they become idle. The idle threads also have a timeout within which they can be reused by subsequent tasks, otherwise they are terminated and garbage collected. Thus, the thread pool will not consume any resources if it is idle for a long time.

The second construct creates a thread pool with a fixed number of threads passed as a parameter and reuses them for execution. The integer number passed as a parameter represents the maximum number of threads that may be active in the application. All subsequent tasks that are submitted will be placed in a shared unbounded queue until a thread is available. A new thread is created in the pool only if an existing thread is terminated (either successfully or not) and tasks are available for execution in the queue. The main advantage of this thread pool is that it will not starve the system out of resources regardless of how many tasks the program attempts to execute in parallel.

The third construct offers a thread pool which employs a work stealing strategy. The existing threads in the pool have a proactive behaviour attempting to execute existing tasks in the queue if they are idle. This type of pool is especially powerful when executing tasks which in turn spawn subtasks, as the work-stealing mechanism bypasses the process of scheduling tasks and assigning them a thread. Thus it is also useful for processing tasks that perform small computations.

The return type of these three factory methods (and the type of the aforementioned thread pools) is an `ExecutorService` which extends `Executor`. This is a more versatile interface that provides more methods to execute different types of tasks such as instances of `Runnable`, `Callable` or a collection of tasks. This is done through the overloaded `submit` method that can take either one of the three arguments and has a similar functionality to the `execute` method of the parent interface. This allows for more fine-grained control of asynchronous tasks. An example of creating a thread pool and then submitting a task instantiated using lambda expressions is given in Listing 2.7. The task does not necessarily need to be declared separately as in Listing 2.3, especially if its a small computation that is only required once.

Listing 2.7: Using a predefined thread pool to run a task asynchrnously

```
1 ExecutorService pool = Executors.newCachedThreadPool( );
2 pool.submit( ( ) -> {
3  return 10;
4 });
```

The `Executor Service` also offers methods to manage termination of the thread pool. An unused Executor Service should be shut down to allow reclamation of its resources. The user can safely shut it down, which will cause it to reject new tasks This can be done using two different methods.

- The `shutdown( )` method will continue to run previously submitted tasks to execute before terminating.
- The `shutdownNow( )` method prevents queued tasks from starting and attempts to stop currently executing tasks.

Synchronizing on Return Values. Java provides a large collection of interfaces and classes that support synchronization of tasks running in parallel including mutexes, semaphores and barriers. Our focus, however, is on the `Future` interface which is a general concurrency abstraction, that encapsulates a method's result to be returned upon completion of a method call that is associated with that future. The interface provides methods to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. A simple usage of a future is like the one shown in Listing2.8. Here a lambda expression is used to create a `Callable` instance since the method returns a value. The return type of `submit` is a `Future` which is a dynamically generated unique reference to the task to be executed asynchronously. The current thread can then synchronize and obtain the result by calling the `get( )` method. This is a very simple and intuitive synchronization mechanism, yet it has some important implications. The current thread will be blocked until the future completes. This means that the thread will remain live in the main memory occupying the space that was allocated for its context.

Listing 2.8: Using a future to synchronize on the result

```
1 ExecutorService pool = Executors.newCachedThreadPool( );
2 Future<Double> f = pool.submit( ( )−> {
3             Double x = ... //calculate x
4             return x;
5           });
6 // perform other computations in parallel before using the result
7 Double result = f.get( );
8 //subsequent code that will only execute when the thread is released
```

The result can only be retrieved when the computation has completed, thus releasing the calling thread, loading its context and executing subsequent code. Cancellation of a task represented by a future may be performed by the `cancel( )` method. The `get( )` can also be used to determine if the task completed normally, was cancelled or has thrown an `Exception`. Once execution has finished, the computation cannot be cancelled. A particular extension of the `Future` interface is the `CompletableFuture`. This implementation provides methods for the developer to create, process and complete a future manually. We want to observe three important methods:

- `complete(T value)` - This method set the future's result to the `value` passed as a parameter, unless the future has already completed.
- `completeExceptionally(Throwable)` - If not already completed, causes invocations of get( ) and related methods to throw the given exception.
- `CompletableFuture.completedFuture(T value)` - This static factory method creates object instances that behave like futures that have completes with the result `value`.

2.2 Scala Abstractions for Concurrent Programming

Scala is an object-oriented programming language that also has support for functional programming. It is intended to compile to Java bytecode such that it can provide language interoperability with Java and allow the usage and integration of libraries designed in either language. With respect to concurrent programming in Scala, this section covers the inner workings of futures and their intended usage, as well as two libraries that support the actor programming model.

Scala Futures In Scala, the package called `scala.concurrent` contains the trait `Future` to provide a way to reason about performing many operations in parallel without blocking until a synchronization point. Traits in Scala are similar to Java interfaces, except they may contain methods which have a default implementation. Unlike Java futures, the entire computation code that is to be done asynchronously can be declared in a `Future` trait like in Listing 2.9.

Listing 2.9: Declaration of a Future in Scala

```
1 val f: Future[Double] = future {
2   val x = ...
3   //calculate value of x
4   return x;
5 }
```

To bypass the explicit creation of threads and overwriting of the `run( )` method, Scala futures require an `Execution Context`. Similar to the Java `Executor`, it is responsible for thread creation, execution and termination. The package also provides a global static thread pool which is a default `ExecutionContext` implementation that is backed by a `ForkJoinPool` object suitable for execution of a large number of small asynchronous tasks. To reduce performance penalties caused by a large number of threads, the `ForkJoinPool` object sets a default limit to the number of threads it contains, known as level, to the number of available processors (value yielded by the `Runtime.availableProcessors` function). This parallelism level can be overridden by setting one (or more) of the following VM attributes:

- `scala.concurrent.context.minThreads`
- `scala.concurrent.context.numThreads`
- `scala.concurrent.context.maxThreads`

The parallelism level will be set to `numThreads` as long as it remains within the interval [`minThreads; maxThreads`]. An `ExecutionContext` can also be created from a Java `Executor` using `ExecutionContext.fromExecutor` method. Optionally, developers are allowed to extend the `ExecutionContext` trait to customize their own execution contexts.

In Scala futures make use of callbacks by default, to minimize as much as possible the blocking effect of `get()` operations and therefore the number of blocked threads in the system. Scala provides a general callback method `onComplete` to execute upon completion of a future. The body of the callback has to pattern match on the state of the completed future as shown in Listing 2.10. This state is of abstract type `Try`, which represents a computation whose final result may be a completed value or an exception and therefore can be one of two case classes: `Success` or `Failure` which contain as an argument the result or exception respectively.

Listing 2.10: Callback definition in Scala

```
1 f onComplete {
2   case Success(result)=>process(result);
3   case Failure(error)=> println("Computation_yielded_error" + error);
4 }
```

Scala futures also provide more specific methods to define the control sequence to be executed once a future completes, as well as a recovery block if the asynchronous task completed with an exception. These two methods are `onSuccess` (Listing 2.11) and `onFailure` (Listing 2.12) and avoid the need to pattern match on the state of the future.

Listing 2.11: Callback to execute when the future completes sucessfully

```
1 f onSuccess {
2   case result => {
3     process(result)
4   }
5 }
```

Listing 2.12: Callback to execute when the future yields an error

```
1 f onFailure {
2   case error => {
3     println("Computation_yielded_error" + error);
4   }
5 }
```

This is the mechanism specific to Scala through which a result can be used or an error can be handled from a completed future. It is a much more efficient way compared to the approach where the thread that is using the future result has to block using the `get( )` method until it is available. It is important to note that callback function is executed on a separate thread and as such needs to explicitly handle synchronization issues if any of the data structures and operations in the function are not thread safe.

Scala Actors The `scala.actors` package offers an API for concurrent programming with actors. The package provides both asynchronous and synchronous message transmission (the latter are implemented by exchanging several asynchronous messages). Moreover, actors may communicate using futures where requests are handled asynchronously, but return a representation (the future) that allows to await the reply. The typical approach in Scala is that of messages to have any type and thus are only checked at run-time whether the receiver can handle them. All actors contain a mailbox which stores messages until they are processed.

Scala provides a trait hierarchy starting with the basic implementation of actors that can send and receive messages, to supporting more complex capabilities such as obtaining the reference of the sender, replying to a message or grouping actors for monitoring and correct terminations purposes. The API also provides control structures and traits for managing actors in terms of scheduling, error handling and remote communication.

Actor-based programs use the `Actor` interface to define classes like in Listing 2.13. The example shows two actors running a ping-pong benchmark. This is a simple example where two actors (Ping and Pong) exchange a fixed number (count) of empty requests (pings) and replies (pongs) and then the program stops.

Listing 2.13: Creating a Scala Actor

```
1 class PingActor(count: int, pong: Actor) extends Actor {
2
3 }
4
5 class PongActor extends Actor {
6
7 }
```

To define the message exchange between the two actors, one first needs to define the objects that represent the messages like in Listing 2.14. These are defined as case objects to use Scala pattern matching when defining the behavior for receiving each message.

Listing 2.14: Messages as Case Objects

```
1 case object PingMessage
2 case object PongMessage
3 case object StopMessage
```

The actual running behavior is then defined by overriding the `act` method and using `receive` to define the control flow for each received message. This is shown in Listing 2.15 for the `PingActor` class.

Listing 2.15: Defining Scala Actor Behavior

```
1  def act( ) {
2    var pingsLeft = count - 1
3    pong ! PingMessage
4
5    while (true) {
6      receive {
7        case PongMessage =>
8
9          if (pingsLeft > 0) {
10           pong ! PingMessage
11           pingsLeft -= 1
12         } else {
13           pong ! StopMessage
14           exit( )
15         }
16     }
17   }
18 }
```

The Actor model in Scala [HO09] does provide a suspension mechanism, but its use is *not* recommended because it actually blocks the whole thread and causes degradation of performance. It is possible to register a *continuation* piece of code to run upon completion of a future, but that will run in a separate thread which breaks the actor semantics and may cause race conditions inside the actor.

Akka Actor Library Akka actors [Hal12, Gup12] are similar to Scala actors and can be used in both Java and Scala, but are more intuitive and less cumbersome to use in the latter. To define the behavior of an actor in Akka, one needs to extend the `Actor` trait and override the `receive` method. A simple example of defining an Akka Actor is shown in Listing 2.16. Java libraries for programming actors like Akka [SM01] mainly provide pure asynchronous message passing which does not support the use of application programming interfaces (API) because, a message is only typed as a Java Object, so there is no static typing of messages, nor are they part of the actor interface.

An important challenge in both Scala and Akka actors is that messages can only be checked at runtime (they need to be pattern matched in order for the actor to execute a particular control flow). This means that the programmer needs to define for all the possible messages a specific type(see Listing 2.14). Then an actor must pattern match on a given message and execute the matching block of code. This block is often a call to an internal method defined in the actor class. These libraries however do not provide a way to bypass this pattern flow that executes at runtime in favor of a straightforward call to the internal method that is meant to execute asynchronously. The Akka Typed module attempts to solve this challenge

Listing 2.16: Akka Actor

```
1 import akka.actor.Actor
2
3 class MyActor extends Actor {
4
5 def receive = {
6   case "Hello" => println("HelloWorld")
7   case _ => println("received_unknown_message")
8   }
9 }
```

by defining messages as case classes like in Listing 2.17. This provides a close resemblance to methods with arguments, but still requires defining an internal block or method corresponding to the flow to be executed asynchronously.

Listing 2.17: Akka Typed Example

```
1 object PingActor {
2   final case class PingMessage(replyTo: ActorRef[PongMessage])
3   final case class PongMessage( from: ActorRef[PingMesssage])
4   }
```

2.3 Coroutine Support

Coroutines are programming abstractions of state and control flow in a program that allow for multiple entry points for suspending and resuming execution at certain locations. It is a very powerful concept especially in object-oriented programming for organizing control flow of a large number of small tasks. The current existing support for coroutines in JVM can be categorized by two main approaches: one which operates on source code level and one which operates on the bytecode level.

Main examples of bytecode manipulation are Apache Commons Javaflow [Com] and Kilim [SM08]. Even though bytecode manipulation allows for more flexibility, it has several disadvantages regarding maintainability and portability. Further, the application of debugging techniques becomes more involved and source-code based static analysis tools become unusable.

A straightforward way to support coroutines at source-code level in Java (see [Sch11]) is by allocating a thread to every "routine" that can be suspended, since a thread naturally contains already all the information about the call stack and local variables. However that does not scale because threads are well known to be heavyweight in Java. In Scala, macros are also a viable approach to implementing coroutines. Macros are an experimental feature in Scala that allow a programmer to write code at the level of abstract syntax trees and thus to instruct the compiler to generate code differently.

Scala-coroutines project [Sto] uses this feature to implement low-level coroutine support for Scala with explicit suspension and resume points which however in general are prone to errors. Kotlin [Jan17]supports coroutines natively but actors are not first class citizens. Kotlin actors are implemented as coroutines, which by definition ensures a single thread of execution within the actor. However one cannot process the messages inside actors in a coroutine manner. In other words, it is not possible to process other messages if one message is suspended.

2.4 Summary

Asynchronous calls can be emulated in Java using the basic thread programming pattern or constructs that simplify this pattern. However there is not a straightforward way to issue these calls similar to synchronous calls. A big research challenge in this thesis is to have a mechanism for issuing a call to method (using its signature), executing it on a separate thread and capturing its result in a future. The goal is to have a method call similar to the model of the method call issued in the researched modelling language, ABS. Emulation of the asynchronous call is supported, but it involves declaration of a few classes and instances, possible duplicate code if more than one call is required, or unwanted code that may result from misuse of the generic classes. There is not standalone construct for a method call that type-checks the call to be of the correct signature. In Java, Scala or Akka, the notion of an asynchronous method call is not present as an integrated part of the language. Bridging this gap between the asynchronous call model and its current complex syntax in JVM-based languages and libraries is a significant part of this thesis.

Chapter 3

Abstract Behavioural Specification Language

The starting point for the actor programming model assumed in this thesis is the Abstract Behavioral Specification language (ABS) introduced in [JHS+12]. ABS is a modeling language that was part of the HATS European Project [hat, WDS11] which was continued by the ENVISAGE European Project [AdBH+13, JST12a]. An ABS model describes a dynamic system of actors which interact only by means of asynchronous method calls. ABS combines the actor programming model with a"programming to interfaces" paradigm that enables static type checking of message passing at compile time.

Actor-based models of computation in general assume a run-to-completion mode of execution of the messages [SM01, BSH+17, ASdB15]. ABS extends the actor-based model with `coroutines` [HFW84] by introducing explicit suspend statements. Suspending the execution of a method allows the actor to execute another method invocation. This suspension and resumption mechanism thus gives rise to multiple control flows in a single actor. The suspension of a method invocation in ABS can have an enabling condition that controls when it can be resumed. Typical enabling conditions are awaiting completion of a future or awaiting until the internal state of the actor satisfies a given boolean condition. This feature combination results in a concurrent object-oriented model which is inherently compositional.

Actors in ABS serve as a major building block for constructing software components. Actors in ABS can model a distributed environment as they interact via asynchronous communication. It can be viewed as a Java-like correspondent of MPI [CW10, SOHL+98] where ABS objects are corresponding to MPI processes. Internally with support for cooperative scheduling they allow for a fine-grained and powerful internal synchronization between the different method invocations of an actor. ABS has support for user-defined schedulers [BdBJ+13] to use algorithms for specific task management [SP15] or distributed environments [VPT+15]. Therefore they are a natural and intuitive basis for component-based software engineering and service-oriented computing and related Internet- and web-based programming models [CJO10]. ABS has been applied to several major case studies like in the domain of cloud computing [FMAG13, AdBH+14], simulation of railway models [HM16, KH17] and modeling software product lines [KH16].

ABS has a syntax similar to Java with classes and interfaces for encapsulation, but includes several extra features such as a functional programming model, the possibility to issue asynchronous method calls with compile-time type checking and also integrates actor-based models with futures and coroutines. Specifically any object created in ABS represents an actor with encapsulated data. Similar to Java, their behavior and state is defined by implementing interfaces with their corresponding methods. Thus they interact by making asynchronous calls to these methods which generate messages that are pushed into a queue specific to each actor. An actor progresses by taking a message out of its queue and processing it

by executing its corresponding method.

3.1 Programming to Interfaces in ABS

The ABS language is a class-based object-oriented language that features algebraic data types and side effect-free functions. Actors implement a shared-nothing model for concurrency. Each actor can be viewed as a separate node who sends messages to other actors with a syntax that is very close to Java method calls. The actors are identified as objects with references like in Java that may be passed as method arguments. Instantiating an object in ABS creates an Actor with its member fields and methods which define its behavior and interactions with other actors. Interfaces in ABS describe the functionality of objects. Thus, interfaces in ABS are similar to interfaces in Java. Unlike Java, objects are only typed by interfaces and not by their class. They have a name, which defines a nominal type, and they can extend zero or more other interfaces. The interface body consists of a list of method signature declarations.

To best illustrate the programming to interfaces features of ABS, we look at a simple example, the Prime Sieve of Eratosthenes, written in Listing 3.1. This example was chosen as it was the very first example researched in this thesis to identify the performance problems that an ABS model had once it was translated into Java using the original tools developed prior to this research. It is a parallelized implementation of the Sieve of Eratosthenes [Bok87, O'N09]. This example aims to illustrate how a general parallelized model can be implemented in an asynchronous manner. It shows the benefit of observing certain behaviours in the programming phase, as well as showing that the actor-based model still performs well when compared to implementations that apply low-level optimizations. At the same time this first case study is perfect for modeling partitions as actors as well as making it easy to simulate an application that can work on a multi-core platform using a shared memory or a distributed platform where communication between actors is key. The Sieve of Eratosthenes also allows us to illustrate several optimizations that result from the actor based model, as well as how certain well known optimizations are easy to apply in this model without significantly increasing the code size and therefore the design phase of a distributed application.

The program first creates the interface `ISieve` (line 1) which contains the method `sieve()` for sieving all numbers present in a partition that are divisible by the argument *n* by setting in the `currentMap` data structure the value of a key to `false`, where the keys represent the candidate numbers found in each partition. A second interface, `IGen` (line 5)contains the method, `run_par( )`, for retrieving the next prime number in the first partition and sending an asynchronous method call to all partitions to sieve using that number. The method implementations are in the classes `Sieve` and `Generator` respectively. The `Generator` class also contains an initialization phase in which the candidate numbers are split into partitions and stored in a `processors_Map`. As can be seen in lines 17 and 51 asynchronous method calls are modeled by the statement `actor!method( )`. These statements return a dynamically generated reference that can be assigned to a Future variable (`Fut<Bool> f` or`Fut<Unit> f`). The `lookupUnsafe` method is a function defined in the ABS standard library for retrieving values associated to keys that are known to exist in the map apriori.

3.2 Algebraic Data Types (ADT)

Algebraic Data Types in ABS are used to implement user-defined, immutable data types. Because values of algebraic data types are immutable, they can be safely passed on to other objects making it easy to reason about program correctness. Data type constructors enumerate the possible values of a data type. Constructors can have zero or more arguments. These arguments can optionally have a name, which needs to be a valid identifier. This implicitly defines an accessor function for that argument. This is a function that, when passed a value expressed with the given constructor, returns the argument. The

Listing 3.1: Prime Sieve Case Study in ABS

```
1  interface ISieve {
2   Bool sieve(Int n);
3  }
4
5  interface IGen{
6   Unit runPar( );
7  }
8
9  class Generator (Int limit, Int elem) implements IGen{
10
11  / * initialization block for all partitions and nodes: currentMap, processorsMap * /
12  Unit runPar( ){
13
14    while ((currentMap!=EmptyMap)&&(n*n<limit)){
15     Int k=1;
16     while(k<elem){
17      Fut<Bool> f = lookupUnsafe(processorsMap, k)!sieve(n);
18      k=k+1;
19     }
20     if(lookupUnsafe(currentMap,n)==True){
21      primes = Cons(n,primes);
22      currentMap = removeKey(currentMap, n);
23      Int first = n*n;
24      Int j= first;
25      while (j<last){
26       currentMap=put(currentMap,j, False);
27       j = j+(2*n);
28     } }
29     n=n+2;
30 } } }
31
32 class Sieve(Int p, Int size, Int pe) implements ISieve {
33
34  / * Initialization block for each partition * /
35  Bool sieve(Int thePrime){
36
37    Int n = thePrime;
38    Int first = n*n;
39    Int j;
40
41    / * compute the first candidate and store it in variable j * /
42    while (j<=last){
43     currentMap=put(currentMap,j, False);
44     j = j+(2*n);
45    }
46    return True;
47 }   }
48
49 { // Main block:
50  IGen g=new Generator(50000,1);
51  Fut<Unit> f = g!runPar( );
52 }
```

name of an accessor function must be unique in the module it is defined in. It is an error to have multiple accessor functions with the same name, or to have a function definition with the same name as an accessor function.

Algebraic data types can carry type parameters. Data types with type parameters are called parametric data types. Parametric data types are declared like normal data types but have an additional type parameter section inside broken brackets (<>) after the data type name. The definition of the `Maybe` data type in the ABS standard library shown in Listing 3.2.

Listing 3.2: Data Type Maybe in ABS

```
1 data Maybe<A> = Nothing | Just(A fromJust);
```

The example shows a parametric data type `Maybe` that has two constructors (`Nothing` without arguments and `Just` with one argument). The `Just` constructor also contains a named argument `fromJust` that defines an accessor function that extract the argument from and ADT. An example of declaring this data type and using the accessor function is given in Listing 3.3.

Listing 3.3: Declaring an ADT in ABS

```
1 Maybe<Int> x = Just(2);
2 Int two = fromJust(x);
```

Parametric Data Types are useful to define container data types, such as lists, sets or maps. These data structures have a default implementation in the ABS Standard Library defined as follows.

- Lists are implemented as recursive single linked lists.
- Sets are implemented as sorted lists with unique elements.
- Maps are implemented as lists of associative pairs.

Literal values of recursive data types can be arbitrarily long, and nested constructor expressions can become unwieldy. ABS provides a special syntax for n-ary constructors, which are transformed into constructor expressions via a user-supplied function. The implementation of a `List` in the ABS standard library together with its constructor is shown in Listing 3.4

Listing 3.4: Constructing a recursive type in ABS

```
1 data List<A> = Nil | Cons(A head, List<A> tail);
2
3 def List<A> list<A>(List<A> l) = l;
4
5 List<Int> = list [1,2,4];
```

ABS supports pattern matching via the `case` expression. This expression consists of an input expression and a series of branches, each consisting of a pattern and a right hand side expression. The case expression evaluates its input expression and attempts to match the resulting value against the branches until a matching pattern is found. The value of the case expression itself is the value of the expression on the right-hand side of the first matching pattern. A simple example of pattern matching is illustrated in Listing 3.5.

Listing 3.5: Pattern Matching in ABS

```
1 Maybe<Int> x = Just(2);
2 case x{
3   Nothing => 0;
4   Just(x) => x;
5 }
```

3.3 Asynchronous Model

The asynchronous model of ABS is defined by combining the “objects-as-actors" model with several language constructs. ABS features one construct for the asynchronous communication, another construct for blocking an actor’s execution on a future and a third and very powerful construct for suspending and scheduling methods within one single actor, a construct that introduces the notion of *cooperative scheduling*. An actor processes messages from its queue in a FIFO order and calls its scheduler whenever a running method suspends. Before discussing these constructs, we need to discuss how ABS has a high-level hierarchy for grouping objects and restricting how they communicate with each other. This is the notion of Concurrent Object Groups.

3.3.1 Concurrent Object Groups (COGs)

In the ABS, the core language semantics imposes that all objects created in the program are actors with an independent behavior with a possibility to communicate with other actors and use futures to synchronize at certain points. The language offers a feature that allows the programmer to group actors into Concurrent Object Groups (COG) which allow objects to communicate with each other synchronously, but apart from the actors in the same COG, all other actors are considered remote and may only invoke each other asynchronously. The physical location of an actor in ABS is completely transparent as there are no virtual machines or IP addresses inserted in this modeling language.

Each COG runs one process at a time, while processes on different COGs run in parallel. This means that each COG is a unit of concurrency and is in charge of scheduling the processes running on its objects. In the next section we will observe two constructs that control the execution and suspension of processes within a COG. These constructs can be preceded by annotations that allow custom schedulers to be defined in order to satisfy an application’s specific requirements. Further annotations can be associated to method calls to specify costs and deadlines in order to create a very powerful scheduler. All these constructs are written in a very simple and concise way in ABS, in order to allow system designers a simple view of their application which can be even large enough to be deployed in a cloud environment [JST12b].

3.3.2 Await and Get Construct

In this section we informally describe the main features of the flow of control underlying the semantics of the coroutine abstraction as proposed by the ABS language. We describe the main concepts of (a)synchronous method invocation and their coroutine manner of execution through the example in Listing 3.6 which presents a general behaviour of a pool of workers. For a detailed description of the syntax of the ABS language we refer to [JHS+12].

The example sketches the behaviour of two kinds of actors: a `WorkerPool` and a `Worker`. A `WorkerPool` actor maintains a set of `Worker` actors (line 2). An asynchronous invocation of the method `sendWork` (line 4) suspends until the set of workers is non-empty (line 5). It then selects a worker from the set and asynchronously calls its `doWork` method (line 8). The future uniquely associated with this call is used to store the return value of this call. Note that thus asynchronous method calls in ABS by default return futures (see [DBCJ07] for details about the type system for ABS which covers futures). An asynchronous invocation of the method `finished` simply adds the `Worker` parameter back to the set of workers (lines 14 and 15). Each worker actor stores a reference to its worker pool `p` which is passed as a parameter upon instantiation (line 19). Before returning the result (line 26) the method `doWork` asynchronously calls the method `finished` of its associated `WorkerPool` reference (line 25). The suspension mechanism underlying the `await` statement in line 5 pauses the current method that is executing and saves the current local context if the set of worker actors is empty. However, the actor itself is free to schedule other asynchronous calls present in its queue. As such the actor can schedule any

asynchronous update of the set of worker actors by a call of the method `finished`. Such an update then will allow the resumption of the execution of the method `sendWork`.

Listing 3.6: Example of a Pool of Workers

```
1  class WorkerPool( ){
2    Set<Worker> workers; // initialization omitted for brevity
3
4    Result sendWork( ) {
5      await !(emptySet(workers));
6      Worker w = take(workers);
7      workers = remove(workers, w);
8      Fut<Result> f = w ! doWork( );
9      await f?;
10     Result result = f.get;
11     return result;
12   }
13
14   Unit finished(Worker w) {
15     workers = insertElement(workers, w);
16   }
17 }
18
19 class Worker(WorkerPool p) implements Worker{
20   Result doWork( ){
21     Result r;
22
23     // computation
24
25     p ! finished(this);
26     return r;
27   }
28 }
```

The await statement also has a second form, **await** `f?` (line 9) which suspends the executing method invocation and resumes it based on the completion status of the future `f`. The method can then be rescheduled when the method invocation corresponding to `f` has computed the return value. In contrast to that, futures in ABS can also part of an expression `f.get` (line 11 in Listing 3.6) that blocks all the method invocations of an actor until the return value has been computed. In particular, the statement on line 11 can never be blocking as the preceding await always ensures `f` is complete. Line 1 of Listing 3.7 depicts a sugared syntax in ABS of the `await` construct. This construct is used to suspend execution of an asynchronous invocation and retrieve its result once the implicitly generated future holds the computed return value. It is a shortened version of lines 4-6.

Listing 3.7: ABS Await sugared syntax

```
1 Result result = await w ! doWork( );
2
3 //can be expanded to
4 Fut<Result> f = w ! doWork( );
5 await f?;
6 Result result = f.get;
```

3.4 Synchronous Calls

In ABS, actors which form concurrent object groups also may invoke their own methods synchronously. For example, we may want to move the functionality of obtaining a worker in the `doWork` method to a separate method `getWorker` such as in Listing 3.8. We can then call this method synchronously like in line 7. It is important to observe that suspension of a synchronous method call gives rise to suspension of an entire call stack. In our example, suspension of the await statement in line 2 gives rise to a stack which consists of a top frame that holds the suspended synchronous call and as bottom frame the continuation of the asynchronous method invocation of `sendWork` upon return of the synchronous call in line 7. These call stacks cannot be interleaved arbitrarily, only one call stack in an actor is executing until it is either terminated or suspended.

Listing 3.8: Synchronous Call in ABS

```
1  Worker getWorker( ){
2    await !(emptySet(workers));
3    Worker w = take(workers);
4  }
5
6  Result sendWork( ) {
7    Worker w = this.getWorker( );
8    workers = remove(workers, w);
9    Fut<Result> f = w ! doWork( );
10   return f.get;
11 }
```

3.5 Timed ABS

According to the ABS manual[absb], Timed ABS is an extension to the core ABS language that introduces a notion of abstract time. With this extension ABS can model and estimate execution time of real computing resources. In contrast to real systems, time in an ABS model only advances through explicit constructs, it is actually symbolic time controlled by the programmer. As such the semantics of ABS introduce a symbolic clock and provides the user with constructs to advance the clock. This symbolic clock needs a global synchronization mechanism with respect to all the processes that exist in the system. Any ABS program that does not use this extension will not advance time and the whole execution of the model will be considered to occur at time zero. Logical time is expressed as a rational number and time can be advanced by the smallest margin. The logical clock can only advance when all processes are blocked or suspended on conditions that cannot be fulfilled. This means that for time to advance, all processes are in one of the following states:

- The process is awaiting for a guard that is not enabled.
- The process is blocked on a future for which the corresponding call has not completed.
- The process is waiting for some resources (these are covered in Section 3.6)
- The process is suspended on a time construct waiting for time to advance.

The function `now( )` always returns the current time is shown in Listing 3.9. Unlike in Java where two calls to `System.currentTimeMillis()` always return different values, two calls to `now` can return the same value(line 4) if no explicit construct to advance time is in between the calls.

Listing 3.9: Current Time Function in ABS[absb]

```
1 Time t1 = now( );
2 Int i = pow(2, 50);
3 Time t2 = now( );
4 assert (t1 == t2);
```

Listing 3.10 shows the two statements for modeling symbolic time in ABS. The first statement on line 1 suspends the COG until the time is able to advance at least 1 and at most 5 time units. The second statement on line 2 blocks the COG until the clock is able to advance between 6 and 7 units. The difference between suspension and blocking is the same as explained in section 3.3.2 where suspension allows other enabled processes in the same COG to execute while blocking stops all processes in the same COG from executing. Note that processes in other COGs are not influenced by `duration` or `await duration`, except when they attempt to synchronize with that process.

Listing 3.10: Timed-ABS Statements

```
1 await duration(1,5)
2 duration (6,7)
```

The logic of the simulated clock is to advance is to advance as much as possible without breaking any statements while unblocking as many processes as possible. This means that when a process waits or blocks for an interval [min, max], the clock will not advance more than max, since otherwise it would miss unblocking the process. On the other hand, the clock will advance by the highest amount allowed by the model. This means that, for example, if only one process waits for (min, max), the clock will advance by max. In Listing 3.11 we present a method of the case study that models the German Railway System, a case-study that relies heavily on the time model of ABS and that is covered in Chapter 8. This listing shows the behavior of the method `insertTrain` that models the entry of a new train in the network at the moment of time `t` from the beginning of the simulation.

Listing 3.11: Case Study with Timed-ABS

```
1 Unit insertTrain(Edge e, Node n, Int t, Int v, ZugFolge resp, Train zug, ZugFolge last, Strecke str){
2   await duration(t, t);
3   resp!vorblock(last, str);
4   zug!goResp(e, n, 0, v, resp);
5 }
```

3.6 Deployment Components

Similar to the logical grouping of COGs, ABS has another extension for grouping of objects to resources. These are known as deployment components. As described in the manual [absb], deployment components are abstractions offer synchronization mechanisms based on modeling computing resources. They represent a location for multiple COGs to reside on and a list of predefined resources that can be consumed. The consumption of resources is closely related to the logic of Timed-ABS. Together with the predefined resource types ABS provides function definitions for replenishing and consuming resources, as well as synchronization mechanisms when resource requirements are less than the available resources.

3.6.1 Resource Types

Like all other class instances, an instance of a deployment component is treated the same way as any other ABS object. Deployment components are created using the `new` expression. Any other COG can be

created “on" a deployment component by using a DC annotation to the `new` statement. In Listing 3.12, each deployment component is defined with some amount of resources for each resource type. This is expressed as a mapping pair of resource type to a number, for example `map[Pair(Speed, 10), Pair(Bandwidth, 20)]`. When no amount is given for some resource type, it is considered infinite and that deployment component will never deplete that resource type. The fixed amount that is given upon instantiation is made available to all objects residing on the deployment component for one integer unit of ABS time. Therefore these properties are implicitly shared data between objects residing on the same deployment component.

Listing 3.12: Defining a Deployment Component

```
1 DeploymentComponent dc = new DeploymentComponent("Server_1", map[Pair(Speed, 10), Pair(Bandwidth, 20)]);
2 [DC: dc] Worker w = new CWorker( );
```

In ABS, a resource is a property that is countable or measurable and is influenced by program execution and the passage of time. The resources that are currently supported by ABS are illustrated in Listing 3.13.

Listing 3.13: Defined Resources

```
1 data Resourcetype = Speed | Bandwidth | Memory | Cores ;
```

3.6.2 Resource Consumption

Consumption of resources is defined through annotations preceding the code (Listing 3.14).

Listing 3.14: Resource Usage by an Instruction

```
1 //A statement consuming speed
2 [Cost: 5] skip;
3
4 //A statement consuming bandwidth.
5 [DataSize: 5] o!m();
```

The Speed resource type models execution speed. A deployment component that is provided with more speed resources must execute faster in logical time than a deployment component with less resources. This applies only to statements that are defined to consume speed which are identified using the `Cost` annotation (line line 2). Executing this statement on will attempt to consume 5 Speed resources from the deployment component where the COG was instantiated on. If the deployment component has fixed predefined Speed resources, executing the skip statement may cause suspension of the process and advancement of ABS logical clock if there are less than 5 Speed resources available. This logic has to be extended to the runtime for a proper execution of the model.

Bandwidth is an abstraction of transmission latency between two objects. This resource is consumed when a statement is annotated with `DataSize` (line 5). In order for the statement to execute successfully both the target object of the invocation and the object that called the method must be part of COGs that each reside on deployment components with a minimum of 5 Bandwidth resources. If there are not enough resources, the processes will suspend on both objects until Timed-ABS will advance the logical clock to replenish enough Bandwidth for the statement to execute.

The Memory and Cores types abstract from the size of main memory and the number of CPU cores available on a deployment component. In contrast to bandwidth and speed, memory and cores do not influence the timed behavior of the simulation of an ABS model and will not require any special changes to the runtime proposed in the next chapter.

Chapter 4

ABS Runtime in Java

This chapter illustrates the overall architecture of the proposed runtime and thread management system. It gives implementation details about maintaining actor semantics in the presence of coroutines. It also covers the heuristics used for propagations and delegation of futures in the context of an actor programming model. In the second part of this chapter the details of the runtime support for the extension of ABS concerning time and resources is presented.

One of the major challenges addressed is the development of a Java library that *scales* in the number of executing actors and (suspended) method invocations on a single JVM. To reach this goal, we represent (suspended) method invocations in Java as a kind of `Callable` objects (referred to as tasks), which are stored into actor queues [AdBS16]. This representation allows the development of a library API which encapsulates a run-time system tailored to the efficient management of the dynamic generation, storage and execution of such tasks (the API is covered in Chapter 6). The overall architecture of the system is based on the following control flow: Every actor submits a main task to the thread pool which iteratively selects an enabled task from its queue and runs it. The runtime then uses a system-wide thread pool where millions of actors can run on a limited number of threads efficiently. A key feature provided by our library is a new general mechanism for *spawning* tasks which allows an uniform modeling of both asynchronous method calls and suspension of a method invocation. This suspension, whether it is called synchronously or asynchronously, is modeled by spawning a new task in the actor queue which captures its continuation, i.e., the code to be executed upon its resumption.

4.1 Evolution of the Runtime Schemes

This section investigates the evolution of the scheme used to implement cooperative scheduling in the Java Runtime environment. It starts from a very trivial approach to using several libraries and features that the Java SDK provides. Each solution covers four main features that have an overall performance and scalability impact and have been improved on each iteration:

- Creation of the actor itself;
- Generation of asynchronous calls and message passing;
- Releasing control or suspension of a call when encountering an await;
- Saving the call stack of a synchronous call upon releasing control.

The development of the runtime focuses on efficient thread management for running multiple actors on a system. It uses the Executor interface that facilitates thread programming. The objects that implement

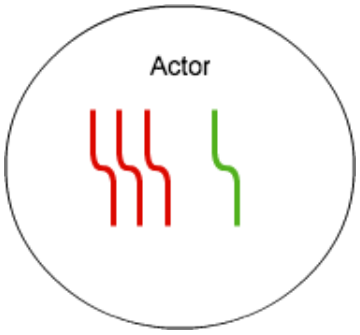

Figure 4.1: Basic process-oriented approach

this interface provide a queue of tasks and an efficient way of running tasks on multiple threads. Throughout this section we will use the terms executor and thread pool interchangeably to refer to this feature. Another notion discussed in our solution was introduced starting with the Java 8 version, and it is the construct for a lambda expression. This allows modeling a method call as a Runnable or Callable and it is referred to as a **task**.

4.1.1 Every asynchronous call is a thread and each actor has a lock

The trivial straightforward approach for implementing cooperative scheduling in the library is for an asynchronous call to generate a new native thread with its own stack and context. We would then model each actor as an object with a lock for which threads compete. The disadvantage here is that this lock-per-actor must be checked by every message handler upon start, and freed upon completion. Whenever an await statement occurs the thread would be suspended by the JVM's normal behavior. When the release condition is enabled, a suspended thread would become available and in turn compete with the other threads in order to execute on the actor. The main drawback of this approach is the large number of threads that are created, which restricts any application from having more method calls than the number of live threads that the memory of the system can handle. Figure 4.1 provides an illustration of this base *process-oriented* approach. Here the idea is that the threads in the circle belong to one actor, thus sharing a lock(the green thread holds the lock, while the red ones are waiting to acquire it). The four main features that impact the system are implemented as follows:

- The actor is initialized as an object with a lock to model actor semantics that allow one method (message) to be executed;
- Asynchronous calls are created as new threads that compete for the lock creating a performance bottleneck that leaves many threads suspended.
- Suspension of a call suspends the thread adding to more memory overhead.
- The call stack of a synchronous call is saved in the suspended thread.

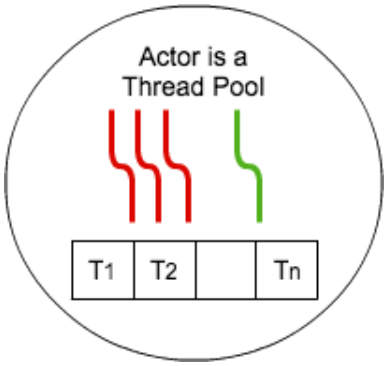

Figure 4.2: Actor-as-executor approach

4.1.2 Every actor is a thread pool.

To reduce the number of live threads in an application, we can model each invocation as a task using lambda expressions and organize each actor as a thread pool. This gives the actor an implicit queue to which tasks are submitted. We obtain a small reduction in the number of threads corresponding to the number of tasks that have been submitted, but not started. Once they are started the threads still have to compete for the actor's lock in order to execute, but the number of live threads can be restricted to the number of threads allowed by each Thread Pool, while the rest of the invocations remain in the thread pool queue as tasks. This approach is illustrated in Figure 4.2. The implementation of the four Actor features is as follows:

- The actor is initialized as an object with a lock and a thread pool preserving actor semantics similar to the previous approach;
- Asynchronous calls are created as lambda expressions and assigned new threads that compete for the lock once they start. The number of started threads is limited by the thread pool size slightly reducing the number of suspended threads competing fo the actor lock;
- Suspension of a call suspends the thread unchanged from the previous approach;
- The call stack of a synchronous call is still saved in the suspended thread.

4.1.3 Every system has a thread pool.

In the previous two approaches we modeled the concept of an actor being restricted to one task at a time by introducing a lock on which threads compete. However as all invocations are modeled as tasks that don't need a thread before they start, there is no point in starting more than one task only to have it stuck on the actor's lock. Therefore we introduce *one thread pool per system* or *the system's executor*, and instead of submitting the messages directly to the thread pool, we add an indirection by storing them into an explicit queue first and introduce as a second phase the submission to the thread pool. We assign a separate task for each actor to iterate through its associated queue and submit the next available message to the system thread pool. We will refer to this task from now on as the `Main Task` of an actor. This removes the requirement to store every message handler as a thread, after it starts and attempts to acquire a lock, saving a task as data in the heap instead. However it comes at the cost of having to manage message queues manually as we need an explicit queue for all the messages that have not yet been submitted (to the thread pool).

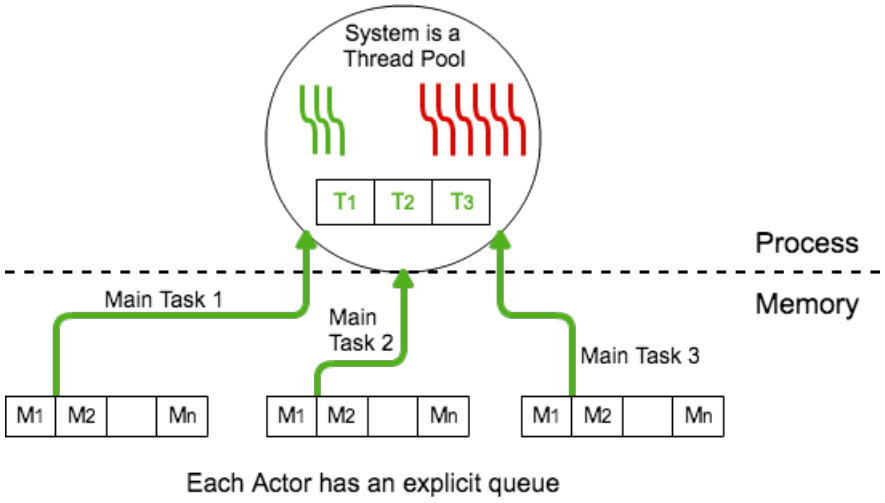

Figure 4.3: System as a thread pool

When cooperative scheduling occurs, the executing task of an actor will be suspended and therefore still live in the system as a thread so the application's live threads will be equal to the number of `await` statements in the program. The system's executor can dynamically adjust the number of available threads to run subsequent available tasks, but the application will then still be limited by the maximum number of suspended threads that can exist in the main memory. Furthermore, we still require a lock to ensure that, upon release, the resuming thread will compete with the `Main Task` associated to the actor from which it originated. This limitation is imposed to preserve actor semantics. This design is presented in Figure 4.3 and it is important to observe the migration of messages into memory and that the red threads are only tasks that have been suspended by an `await`. We observe the following changes in the four features:

- The actor is initialized as an object with a lock for threads that will be suspended by `await`, an explicit queue and an implementation of the `Main Task`, removing the thread pool-per-object;
- Asynchronous calls are created as new tasks that will be run by the `Main Task` one by one, removing the need for suspending thread to preserve actor semantics.
- Upon suspension of a call the `Main Task` restarts to iterate the actor's queue. The current call is parked as a thread that will compete with the `Main Task` once it is resumed;
- The call stack of a synchronous call is still saved in the suspended thread.

4.1.4 Fully asynchronous environment.

In the absence of synchronous calls, to eliminate the problem of having live threads when cooperative scheduling occurs, we can also use lambda expressions to turn the continuation of an await statement (which is determined by its lexical scope) into an internal method call and pass its current state as parameters to this method. Essentially what we do is allocate memory for the continuation on the heap, instead of holding a stack and context for it. We will benchmark this trade-off between the memory footprint of a native thread and the customized encoding of a thread state in memory. As there are no more suspended threads in the system to compete with the actor's `Main Task`, we can eliminate the lock per actor. The four features that affect the system are now as follows:

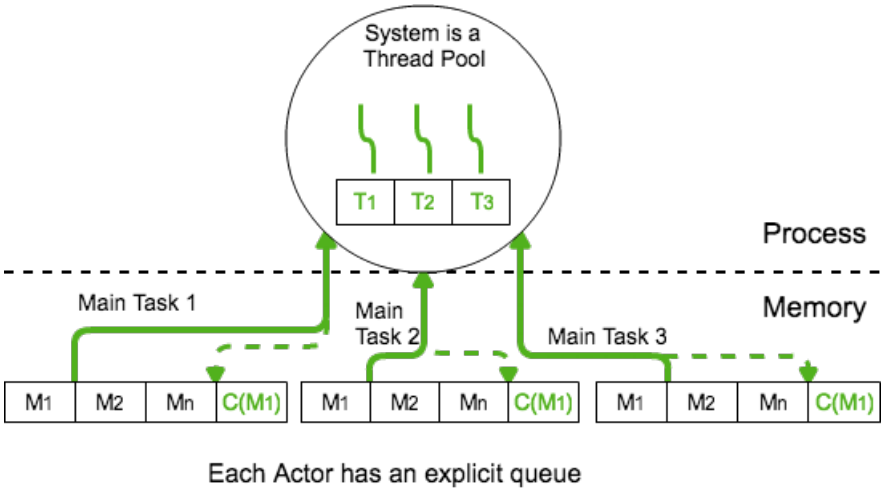

Figure 4.4: Full data-oriented approach

- The actor is initialized as an object with an explicit queue and an implementation of the `Main Task`, but without a lock;
- Asynchronous calls are created as new tasks that will be run by the `Main Task`.
- Suspension of a call first saves the continuation as a lambda expression guarded by the release condition, and the `Main Task` continues to iterate through the queue. No new threads are created any more, the suspended task is saved inside the queue of the actor;
- The process of saving the call stack of a synchronous call has to be done explicitly and is detailed in the next paragraph.

Synchronous calls context. In the presence of synchronous calls, the problem that remains is how to save this call stack without degrading performance? To do this we can try to alter the bytecode to resume execution at runtime from a particular point, but we want our approach to be independent of the runtime and be extensible to other programming languages. Therefore we try to approach the problem differently; if we can turn a continuation, that does not originate from a stack of calls, into a task, is it possible to extend this to synchronous calls as well? We know that this issue arises when methods that contain an `await` statement are called synchronously like the example in Listing 3.8. When the `await` statement is encountered, the continuation consists of both the lexical scope of the release statement that is followed by the complete synchronous call chain that has been generated(i.e. $C(M_1)$ in Figure 4.4). At compile time, when translating source-to-source from ABS to Java, we can identify all of these occurrences from the ABS code. We can mark them and transform them into asynchronous calls followed by an `await` statement on the future generated from the call. Now we can use the same approach for translating these continuations into tasks using lambda expressions and thus eliminating any suspended threads in the system. The final model of our solution is presented in Figure 4.4.

More explicitly in terms of ABS code, a fully asynchronous environment means to change a synchronous call `x = this.m( );` into an asynchronous call plus an await like `f = this!m( ); x = await f?;`. There are two problems here. First, we still need to make sure that *m( )* must be the exact next method that will run. Second, when *m( )* finishes, the actor scheduler must immediately schedule the method that is waiting for its result. Both can be implemented by changing the non-deterministic

behavior of the `Main Task`. There are certain constrains that we have to impose to preserve the chain of synchronous calls. For these we introduce priorities and strictness to the spawned tasks, which are part of the implementation details that are discussed in the next section.

4.2 Implementation Details

To summarize, a naive approach to implementing actors in ABS is to generate a thread for every asynchronous method call and introduce a lock for each actor to ensure that at most one thread per actor is executing [Sch11]. In such an approach suspending execution of a method would be as easy as parking the thread and resuming it later on. This however does not scale because an application will require a large number of JVM threads which are very expensive in terms of memory. Therefore instead of generating a thread for every asynchronous method call, in our approach such calls are stored as `Callable` objects which we call *tasks*. The overall architecture for the execution, suspension and resumption of such tasks consists of the following main basic ideas:

- A system-wide thread-pool that assigns at most one thread to each actor.
- A task queue for each actor.
- Each actor thread runs a main task which iteratively selects from its queue an enabled task and runs it.
- Newly generated tasks are stored in the corresponding actor's queue.

In the following we first describe in more the general mechanism of spawning and completion of tasks and then details of task scheduling and execution.

4.2.1 Task Generation and Completion

This section introduces two abstractions defined as classes called Task and `Future` which are used as part of the mechanism for spawning tasks. We describe in more detail some of the implementation issues of these non-blocking futures and the new delegation mechanism through Listing 4.2.

Non-Blocking Futures Every asynchronous call creates a new task as an instance of the class Task(Listing 4.1) and stores it into the task queue of the actor callee. Task instances model tasks and spawned sub-tasks representing messages and continuations. The actual computation is captured in the `task` field which is a `Callable` returning a Future. The enabling condition is only used for sub-tasks; the type `Guard` represents the possible guards presented in the ABS semantics: either a boolean expression over the actor state, a future or a symbolic time duration. In all these cases the field `resultFuture` will contain a newly created instance of `Future`(Listing 4.2) which uniquely identifies this task and which is returned to the caller. This is special future type defined in the library that cannot block a thread when trying to retrieve its result. Upon termination of a task, the return value will be wrapped in an `Future` instance with a set `completed` flag and an assigned `value`. In our approach based on the notion of a Task, futures no longer provide a blocking `get` operation that suspends a thread in the system. Instead we provide the notion of a suspension mechanism at the level of a Task which does not block the `Main Task` of an actor. Thus it allows for cooperative scheduling of other tasks by the `Main Task` of the actor.

Listing 4.1: Simplified implementation of Task class in Java.

```
1  public class Task<V> implements Serializable, Runnable {
2   protected Guard enablingCondition = null;
3   protected final Future<V> resultFuture;
4   protected Callable<Future<V>> task;
5
6   //implementation and functionality
7
8   public void run( ) {
9     try {
10      task.call( ).backLink(resultFuture);
11      // upon completion, the result is not necessarily ready
12    } catch (Throwable e) {
13      e.printStackTrace( );
14      throw new RuntimeException(e);
15    }
16  }
17 }
```

Delegation. As discussed in Section 4.1.4, a method may delegate the completion of its future to another task (its continuation). Since every task has an associated future, this gives rise to a doubly-linked list of futures. Whenever the `Main Task` runs a Task from its queue it will issue a statement task.call( ).backLink(resultFuture) (see Listing 4.1). Thus, the currently running Task, once it returns, sets the dependant of its associated future to the provided back-link (line 15). If the task successfully completed its future without delegation it can complete its dependant as well (line17). On the other hand, in case of a delegation delegation, its associated futures future will become itself a dependant to the delegated future.

Once a `complete` method is called, it will recursively propagate completion through the whole delegation chain (line 26). After that it will notify its own awaiting actors accordingly (line 28). Thus the above scheme only involves backward propagation of completed futures and avoids searching through the delegation chain when checking a particular future.

It is important to note that for every Task at least one level of such delegation happens. That is because the creation of a Task instance and its execution happen at different points in time. When the Task instance is instantiated, so is its `resultFuture` field, which is returned immediately issuer of the call or creator of a sub-task. The actor may await the completion of this future. Once the future at the end of the delegation chain actually has the computation result, the completion and notification procedure will run as explained above. As a final note, the `isDone` method and `getOrNull` methods are used internally and it is ensured that the value of the future is read only when the future is ready. This scheme fully integrates futures with the mechanism of spawning new tasks and supports the delegation of the computation of return values.

4.2.2 Task Scheduling and Execution

The `LocalActor` class implements the functionality of scheduling and executing tasks in an actor in a scalable manner that ensures fairness between the actors when competing for the system threads. The internal part of this class, which is hidden from the user of the API, is presented in Listing 4.3. Inside the class there is a `taskQueue` which holds all tasks of an actor. Tasks are defined as instances of class Task

Listing 4.2: Details of the Future class in Java

```
1  public class Future<V> {
2
3    private V value = null;
4    private Future<V> dependant = null;
5    private AtomicBoolean completed = new AtomicBoolean(false);
6    private Set<Actor> awaitingActors = ConcurrentHashMap.newKeySet( );
7
8    void awaiting(Actor actor) {
9      awaitingActors.add(actor);
10     if (completed.get( ))
11       notifyDependant(actor);
12   }
13
14   void backLink(Future<V> target) {
15     dependant = linkedFuture;
16     if (completed.get( ))
17     dependant.complete(getOrNull( ));
18   }
19
20   void complete(V value) {
21     if (this.completed.get( ))
22       return;
23     this.value = value;
24     this.completed.set(true);
25     if (dependant != null) {
26       dependant.complete(value);
27     }
28     notifyDependants( );
29   }
30
31   boolean isDone( ) {
32     return this.completed.get( );
33   }
34
35   V getOrNull( ) {
36     return this.value;
37   }
38 }
```

(for example on line 37). To allow for concurrent access and an efficient scheduling of these tasks we use a hashmap (`ConcurrentSkipListMap` on line 5) that orders tasks into buckets (queues of tasks defined by assigned priorities).

The implementation defines an inner class `Main Task` which is responsible for selecting an enabled task from the queue (via the `takeOrDie` method) and running it. Execution of an actor is represented by an instance of `Main Task`. Being a `Runnable`, the main task of an actor can be submitted to the system-wide thread pool to a global `ExecutorService` that efficiently manages allocation of available system threads to active instances of `Main Task`. As a naive approach to iterating over the messages in the queue, one could put a while loop in the `run` method of the `Main Task`. However other actors may get no chance to run any task. In our implementation, the `Main Task` submits itself again to the executor service after having processed one task (instead of looping over all enabled tasks in one go). Thus it allows all actors to get a fair chance of executing their `Main Task`. This fairness policy may also be fine-tuned to allow the `Main Task` to execute a fixed chunk of tasks at a time before releasing the thread in order to reduce context switches. and thus actors are put to compete for available threads in a scalable way.

The `Main Task` avoids busy-waiting in cases that all tasks in the queue are disabled. If `takeOrDie` returns false, it means there are no enabled messages and the `Main Task` of this actor simply terminates. Note that the queue may happen to contain only disabled sub-tasks awaiting on boolean conditions; in that case, they can be enabled only via receiving a message from another actor. The `Main Task` is reactivated upon generation of any new task in the queue (line 41). Actor's `Main Task` will thus be reincarnated only upon receiving a new message. To make sure there is no more than one instance of the `Main Task` running for an actor, we use the `mainTaskIsRunning` flag. As the task queue is accessed concurrently by the `Main Task` performing the queue traversal and other actors sending method invocations, it is important to avoid race conditions. A race condition may happen if after `Main Task` finds no enabled messages in the queue and just before it resets the `mainTaskIsRunning` flag, a new message is sent to the actor; if this happens, the current `Main Task` will terminate and the new message also creates no new `Main Task`. We avoid this situation by the synchronized blocks in `takeOrDie` and `notRunningThenStart`. Since this is a very rare case, the synchronization here does not affect performance.

In case `Main Task` terminates because all tasks in the queue are disabled and one task that was awaiting completion of a future becomes enabled, the corresponding future is made responsible for reactivating the `Main Task`. As mentioned in the previous subsection, every future has a list of awaiting actors. Upon completion, each future will send a special *empty* message to the awaiting actors. To avoid performance penalties, the actor scheduler skips such empty messages and will continue to the next message immediately. Nevertheless, this empty message will reactivate the `Main Task` if it was terminated. This represents a big advantage of this approach is that there is no need for any centralized registry of awaiting actors and also eliminates any busy waiting by actor schedulers.

Actor Life-Cycle. Actors can be created using the **new** keyword just like normal objects. This implies that the actor life-cycle is like a plain Java object and it will be garbage collected when there are no more references to it. In other words, there is no need for any context or a central registry to keep track of actors or to stop actors explicitly. Additionally, if there are messages in an actor's queue, they will be executed even if no external reference to the actor exists. Conversely, when an actor has no messages, it puts no execution load on the system (as discussed in the implementation notes above).

COG support To support the organization of actors into concurrent groups as described in section 3.3.1, the library provides a means for actors to share the `mainTaskIsRunning` flag though the `moveToCOG` method (line 1 in Listing 4.4). This means that only one actor will have its`Main Task` running from the group sharing this flag preserving the COG notion. Certain applications also benifit from actors sharing

Listing 4.3: Details of the Local Actor class in Java

```
1  abstract class LocalActor implements Actor {
2
3  private Task<?> runningTask;
4  private final AtomicBoolean mainTaskIsRunning = new AtomicBoolean(false);
5  private ConcurrentSkipListMap<...> taskQueue = new ConcurrentSkipListMap<>( );
6
7  class MainTask implements Runnable{
8
9    public void run( ) {
10     if (!takeOrDie( )) return;
11       runningTask.run( );
12       ActorSystem.submit(this);
13     }
14   }
15
16   private boolean takeOrDie( ) {
17     synchronized (mainTaskIsRunning) {
18       / *
19       iterate through queue and take one ready task
20       if it exists set it the next runningTask and then
21       * /
22       return true;
23
24       // if the queue if empty or no task is able to run
25       mainTaskIsRunning.set(false);
26       return false;
27     }
28   }
29
30   private boolean notRunningThenStart( ) {
31     synchronized (mainTaskIsRunning) {
32       return mainTaskIsRunning.compareAndSet(false, true);
33     }
34   }
35
36   public final <V> Future<V> send(Callable<Future<V>> message) {
37     Task<V> m = new Task<>(message);
38
39     // add m to the task queue with low priority and no strictness
40     if (notRunningThenStart( )) {
41       ActorSystem.submit(new MainTask( ));
42     }
43     return m.resultFuture;
44   }
45 }
```

their task queues so the library also offers support for this (line 4) although this instruction can be optional when grouping actors together. The move `moveToCOG` method is part of the default constructor of an actor and as such will either create a new COG for the actor if the destination parameter is `null` or place the newly create actor into the destination's COG (their main tasks will share the lock for starting). The library also offers a helper method `sameCOG` such that at any point in the program the user can check if two actors are part of the same group.

Listing 4.4: Methods for COG support in the Local Actor Class

```
1  public void moveToCOG(LocalActor dest) {
2    if(dest!=null) {
3      this.mainTaskIsRunning = dest.mainTaskIsRunning;
4      this.messageQueue = dest.messageQueue;
5    }
6  }
7
8  public boolean sameCog(LocalActor dest) {
9    if(dest!=null ) {
10     return this.mainTaskIsRunning == dest.mainTaskIsRunning;
11   }
12   return false;
13 }
```

Using JVM Garbage Collection The `Main Task` of an actor dies if there are no enabled tasks or sub-tasks in the queue. We explained that Boolean guards may be re-enabled only if another message is received by the actor, therefore, the `Main Task` is reactivated only upon receiving a new message. However, as mentioned in this section, the enabling condition of a sub-task may also depend on the availability of futures. Therefore, in order to avoid actors constantly polling futures, we need a (push) mechanism to allow completed futures to notify the actors awaiting on them. A naive approach would be to keep a global table with the list of actors awaiting every future. Alternatively, we chose to distribute this table into every future to keep track of the actors awaiting its completion.

Whenever an actor creates a sub-task guarded by a future, the actor will automatically be added to the list of `awaitingActors` of that future. This is done via the `awaiting` method. Further, to notify actors of the completion of this future, the completed future sends a special wake-up message to every actor in its awaiting list. As described previously, this will reactivate the `Main Task` in case it has died. The `Main Task` of the actor then continues to process its task queue and possibly will select the newly enabled sub-task from the queue. The only extra references we need for the actors (i.e., in addition to what is used in the program) are the ones required for the notification mechanism for futures. Once the future is completed and notifications are sent, these extra references are deleted. Therefore we can leave the entire garbage collection process to the Java Runtime Environment as no other bookkeeping mechanisms are required. This way we do not need to keep a registry of the actors like the `context` in Scala and Akka. In this setup we completely encapsulate the generation and completion of futures and they are an integral part of the asynchronous method invocation and return, and as such are not exposed to the user of the API. Furthermore, the `Future` class is implemented completely lock-free and therefore the chaining of futures performs very efficiently.

4.2.3 Symbolic Time

In section 3.5 we presented how ABS models abstract time, how messages can be ordered based on how much time has passed since execution of a model has started. This extension is optional in the runtime and as such programs have the possibility to run with or without symbolic time. The extension introduces a new type of guard called `DurationGuard` that represents a new enabling condition for a task that needs to be evaluated before the task can run. Its class definition is presented in Listing 4.5. The class contains the `min` and `max` values for a duration statement as defined by the Timed-ABS model. Furthermore it contains a variable `whenCalled` to determine the logical point of time when the duration statement was called. The logic of the runtime can then use that point to compute the number of time units that need to pass before the guard is satisfied (line 13). The static method `now( )` is explained in the next paragraph in Listing 4.6.

Listing 4.5: Duration Guard Class

```
1  public class DurationGuard extends Guard {
2
3   int whenCalled, min, max;
4
5   public DurationGuard(int min, int max) {
6    this.whenCalled = TimedActorSystem.now( );
7    this.min = min;
8    this.max = max;
9   }
10
11  @Override
12  protected boolean evaluate( ) {
13   return (whenCalled+min)<=TimedActorSystem.now( );
14  }
15 }
```

In the runtime we extended the `ActorSystem` class, which only provided the management of the system-wide thread-pool. The blueprint of the class is presented in Listing 4.6. Part of this class contains support for the resource model and deployment components and is explained in Section 4.2.4. As per the semantics of the timed ABS model, processes that are explicitly suspended by Duration Guards may only execute when all other processes are blocked or suspended on either a future, a boolean state or awaiting resources. To ensure this, we require that the system knows the number of actors running at all time provided by the `runningActors` atomic variable (atomicity is needed as parallel processes that run to completion may update this variable at the same time). The system also provides a clock that measures a central logical time to which all processes have access through the `symbolicTime` atomic variable.

The system keeps an ordered mapping (`awaitingDurations`) of all suspended processes due to time such that it can optimally advance the logical clock until enough time has passed to release the "lowest-time" awaiting process. This is the functionality of the `done` method which is called at the end of every `Main Task` life cycle (see Section 4.2.2). The system first identifies the condition for advancing time (line 15: all processes are suspended). The system then selects the process or processes (actors) that require the shortest period of time to pass in order to be released, logical time advances by that amount and the actors are notified of their release through an `emptyTask` (lines 21 to 28).

4.2.4 Resource Modeling

The resource modeling feature works very similar to the COG support in the library runtime. Just like symbolic time this support is optional and is offered by the `TimedActorSystem` class in Listing 4.6. The

Listing 4.6: Timed Actor System Class

```
1  public class TimedActorSystem extends ActorSystem {
2    private static AtomicInteger symbolicTime = new AtomicInteger(0);
3    private static AtomicInteger runningActors = new AtomicInteger(0);
4    private static ConcurrentSkipListMap<Integer, List<Actor>> awaitingDurations
5                        = new ConcurrentSkipListMap<>( );
6
7    private static ConcurrentLinkedQueue<ClassDeploymentComponent> deploymentComponents
8                        = new ConcurrentLinkedQueue<>( );
9
10   static public int now( ) {
11     return symbolicTime.get( );
12   }
13
14   static public void done( ) {
15     if (runningActors.decrementAndGet( ) == 0) {
16
17       for (ClassDeploymentComponent dc : deploymentComponents) {
18         dc.replenish( );
19       }
20
21       SortedSet<Integer> keys = awaitingDurations.keySet( );
22       int advance = keys.first( );
23       List<Actor> toRealease = awaitingDurations.remove(advance);
24       keys.remove(advance);
25       advanceTime(advance);
26
27       for (Actor a : toRealease) {
28         a.send(ABSTask.emptyTask);
29       }
30     }
31   }
32
33   static void advanceTime(int x){
34     symbolicTime.addAndGet(x);
35   }
36 }
```

ABS standard library directly offers backend independent support for acquiring and awaiting on resources offered by deployment components together with support for a deployment component to self-replenish it resources. These methods work exactly like the semantics summarized in Section 3.6. The only part that is specific to each backend is determining the moment when all resources need to be replenished and this is done each moment symbolic time advances (line 18). All the modeled deployment components are stored in a hashmap for this purpose (line 8). When using the program with resource modeling support a default deployment component with infinite resources is created at the program's entry point. From that point all newly created actors are assigned a deployment component in their constructor. This can either be the same deployment component as the actor that created it, or a new one according to Listing 3.12. The constructor will then call the method in Listing 4.7 to correctly set each actor's deployment component when it is created.

Listing 4.7: Setting a reference for Resource Location in the Local Actor Class

```
1  public void setDC(Actor dc) {
2    this.dc = dc;
3  }
```

Chapter 5

From ABS to Java

This chapter extensively covers the possibilities of offering all of the ABS features in a software development context. These include the concurrency model, both object-oriented principles and functional programming, cooperative scheduling, symbolic time simulation and resource modeling. These possibilities are discussed through two main approaches for translating ABS into program code (the Java language). These approaches are by either constructing a **compiler** from ABS code to Java code or by providing a **library** and **API** to use and develop programs directly in Java using ABS features. The second part of the chapters covers how the solution for the Java backend has evolved to support the different features of ABS discussed in Chapter 3, as well as promote ABS paradigm as a full-fledged programming concept to be adopted by industry.

5.1 Compiler vs Library approach

The discussion is divided into two general perspectives for software engineering. The first perspective focuses more on the practical side of the software engineering cycle, or how the user's experience changes by using either the compiler or library. The second section will present the theoretical and formal analysis perspective of the user that is developing an application. We discuss these two approaches in terms of three dimensions:

Programming In programming practice, as a developer, one wants to limit the learning path of new programming technologies, languages and patterns. At the same time, as the features of ABS are introduced into the programming code, a certain amount of work is required to understand how to correctly use these features and maximize their performance benefits. If ABS is used together with a compiler, all of the features will be readily available and straightforward to use. On one hand, a user with a more formal background and already familiarized with a ABS will benefit a lot from this approach. On the other hand, a user with industrial experience and a background in programming languages, will require some time to understand the syntax and semantics of programming in ABS.

Working with ABS features directly using a library directly will be easier to learn by a developer with experience in programming languages while also being able to maximize the performance of the program using these features. On the opposite side of the spectrum, an experienced user of ABS or other formal languages will require time to learn how to use the library and design patterns to program directly using the library and emulate the features of ABS. This latter approach brings ABS closer to programming patterns and approaches that are already available and used in software engineering and such makes ABS more likely to be adopted by industry as a full-fledged programming language.

One of the main challenges of promoting ABS as a general language for software development is its efficiency when the code is running on a real machine, using the system threads, main memory and I/O interfaces. For example the core part of the ABS language offers limited support for working with complex and efficient data structures for various purposes like storing or searching for data. In the ABS standard library there is little support for Hashmaps (which are implemented as associative lists) and Arrays. In general it is the language backend the creates a correspondence between the existing data structures of the language and the ABS library data types, or the data types are simply translated as functional data types with possible performance penalties (e.g. search complexity in a Hashmap is O(n)). To have a standardized way of translating ABS, in the last approach that uses a compiler, we simply do not create any correspondence between existing types implemented in the ABS Standard library and translate everything exactly according to the grammar and operational semantics of ABS. All the definitions that exist in ABS standard library, such as resource modeling, timed-ABS, predefined data structures, data types and functions are translated exactly like all the other source code by the user. To allow the coder of ABS to use Java or Scala language specific packages, an extension of the type-checker is required to allow foreign language imports. The type-checker of ABS needs to be extended such that when the imported interfaces are used in the code they will be parsed correctly.

Using the approach of providing a library with the core ABS mechanisms like cooperative scheduling and asynchronous communication, the developer has a lot more flexibility when coding. All the language specific libraries and packages available in Java and Scala can be used directly with the library containing ABS features. The API is simply another program library that can be integrated into any code or any application that uses an actor programming model similar to Scala or Akka actor libraries. There is no need to extend or modify any type-checking or parsing as the ABS features are not any language extensions. It comprises of normal Java Interfaces or Scala traits with method signatures. One of the more difficult aspects of using ABS cooperative scheduling or asynchronous messages through the library will be learning certain patterns of code similar to using other libraries like sockets or files (e.g. a certain programming sequence is required to open a socket). For example an asynchronous call in ABS will require the library user to first create a message using lambda expression or an instance of Runnable/Callable. Then this instance needs to be sent to a different actor as a method argument. To ease the use of this pattern some syntactic sugaring constructs are provided in the Scala library such that an synchronous method call is done in a similar pattern to a synchronous one.

Another important feature that ABS already has incorporated in the language are deltas and product lines. This feature allows compile-time modification of the code depending on parameters passed to the compiler. The current version of the parser and type-checker is backend independent with respect to precessing deltas and product lines. As such, using the ABS compiler to Java does not require any modifications or extensions. The API on the other hand does not have support for this feature directly, but Java has a corresponding reflection feature which can be used in a similar way.

In terms of debugging code, using ABS together with a compiler, requires a separate debugger for the ABS language. The new debugger needs to be integrated with the code generator such that the code that is debugged in JVM has corresponding lines in ABS regardless of whether there are foreign imports in the language or not. When directly programming with ABS features when coding with the library API, the entire debugging process is left to the JVM debugger without any correlations.

Analysis For the ABS language several tools have been developed to apply formal development and analysis techniques, e.g. functional correctness [DBH15] and deadlock analysis [GLL16]. Writing software using the ABS language directly and testing it using such tools can result in much more reliable and robust software. This is a significant advantage that the compiler presents, as a user with a formal background can use all the existing state-of-the-art tools developed for ABS. These tools include resource analysis, deadlock detection and correctness proofs and then still generate executable code to run the application in a real environment.

To allow the usage of these tools on an application written directly using the library, the code needs to have a mapping to ABS. This will require some research into model extraction techniques to obtain the ABS model from the written code. For example the latest version of the API to be used directly in Scala allows a one-to-one mapping of asynchronous calls from ABS to Scala. This mapping is provided in the library with some helper definitions to allow the same behavior as in ABS with exactly the same syntax. This allows compile-time checking of the code to ensure asynchronous calls are invoked on methods that are exposed by the interfaces or classes. However, several more features in ABS have syntax and semantics that differ from the library implementation. Thus, this requires effort to modify either the existing tools of analyzing ABS code or to extract the ABS corresponding code from either Java or Scala code. In this case, using the ABS code together with the compiler is so far the better option when it comes to using the formal verification tools.

The timed-ABS extension can be used to simulate behavior of actors that require a certain chronological ordering of execution. The compiler has support for translating this feature into symbolic time and using the spawn mechanism to schedule and suspend tasks according to a logical clock. The library runtime offers the implementation of symbolic time together with constructs for spawning tasks with duration guards as explained in Section 4.2.3. As such this feature is available for usage in both programming approaches depending on the developer's background (formal or software).

Another program analysis feature available in both approaches is the resource modeling explained in 4.2.4. Programming with this feature using ABS and a compiler is straightforward as the compiler links the ABS standard library to any program and correctly compiles any ABS sources that require this support. To program directly using the library, the user has to have a compiled version of the same standard library linked in the program to use the methods of acquiring and replenishing resources. As such, this feature is easier to use by a programmer more familiar with ABS, but nonetheless it is available for both approaches.

Run-time verification Evaluation and benchmarks are an integral part of software development and we want to compare how easy it is to test and benchmark an application either written in ABS together with a compiler or an application written directly using the library. When running an application written in ABS, the code first needs to be compiled into Java code , which is usually very large and difficult to read (although it is not required) due to the some of the restrictions that will be described in Chapter 7. The next step is to then compile and run the code in the JVM, which runs normally as the ABS program entry point will correspond in Java to the `main` method. A big disadvantage to running an application directly from ABS code is that for any run-time errors that occur, the JVM will point the generated Java code. The user then has to find the point in the ABS code where this error occurs or use the separate ABS debugger. Whenever the user makes changes to the code to solve any runtime errors or change functionality this two-step process needs to be repeated to re-run the code. Running code written in Scala or Java eliminates the step of compiling from ABS and also offers a much easier interpretation of runtime exceptions and stack traces as the fault points have direct correspondence in the code.

Another important aspect of testing an application is using its results. The core ABS language offers in its standard library support for console I/O so many applications can be evaluated by printing results to the standard output. However, ABS does not support other means of retrieving results like files, inter-process communication or the possibility to internally time the execution of a program. To use the results from running an ABS application or profiling it, one needs to use bash programs to redirect console results or time the application after running the generated Java program. Using the library directly offers more means of aggregating and analyzing results directly in the program rather then only after running it.

5.2 The "Bloody Battle of Backends"

This section covers how the solution for providing ABS together with a compiler has evolved through several Java backends providing complete or partial support to ABS features. As the ABS language covers a lot of modeling purposes, programming paradigms, concurrency and distributed features as well as support for programming both real applications and simulation platforms, each backend had a separate focus of translating ABS. Initially ABS was composed of the core language features and through each iteration of the research projects new language extensions and features were added (again we refer to [JHS+12] for the core semantics and to [absa] for the current manual with all the language features). With each extension and research project there came requirements for which a different language became suitable Currently ABS has 4 language backends available : Maude [WAM+12], Erlang [AVWW93, GJSS14], Haskell [BdB16, ABdBMM16] and JVM. Throughout this section we will focus on the iterations that the backend and runtime of ABS for the JVM has gone through.

5.2.1 Original Java Backend

The **Original Java Backend** [Sch11] had the main purpose of supporting the full core ABS features. This included the functional features of ABS, asynchronous programming, cooperative scheduling and COGs. The goal was to provide a correct translation and behavior of each ABS feature. This presented some challenges in the translation process which are highlighted below. As the translated code is very complex, the Listings in this section will only show the most important parts of the generated code.

Algebraic Data Types(ADT) and Functional Features Translating ADT and using pattern matching on them posed a significant challenge in Java. This was due to the fact that this data type that was defined either as simple data type or as a data constructor with arguments. These latter constructs could have an undefined length, as arguments could be ADT themselves. We look at translation of the `Maybe` data type in the ABS standard library shown in Listing 5.1.

Listing 5.1: Data Type Maybe in ABS

```
1 data Maybe<A> = Nothing  Just(A fromJust);
```

The original Java backend translates all ADT declarations as abstract classes (Listing 5.2) with methods to identify or cast each possible type defined (in this case `Nothing` or `Just`). Each simple data type (i.e. `Nothing`) is a subclass of the abstract class as the definition in Listing 5.3. If the declared type is a data constructor that has arguments, then it is translated as a subclass with corresponding fields and a Java constructor with the arguments as parameters (line 5 in Listing 5.4).

Listing 5.2: Maybe Class in Java

```
1 abstract class Maybe<A > {
2   final boolean isNothing( ) { return this instanceof Maybe_Nothing; }
3   final Maybe_Nothing<A> toNothing( ) { return (Maybe_Nothing) this; }
4   final boolean isJust( ) { return this instanceof Maybe_Just; }
5   final Maybe_Just<A> toJust( ) { return (Maybe_Just) this; }
6 }
```

To translate the pattern matching feature of ABS into Java, each data type must have an implementation that first verifies that two ADTs (translated as objects) are of the same instance. Data types must then also provide an implementation that tests the equality between two instances of that data type. This is quite straightforward for simple data types, where the two methods (`eq` in Listing 5.3) essentially test the `instanceof` equality. However for data constructors this implementation needs to test the full depth o the data constructor which may require a large recursive call stack to the `eq` method, seriously affecting

performance and possibly even limiting the size of an ADT definition, though it is still a formally correct translation [NdB14].

Listing 5.3: Nothing Class in Java

```
1  public final class Maybe_Nothing<A> extends Maybe<A> {
2
3    public Maybe_Nothing( ) {
4    }
5    public abs.backend.java.lib.types.ABSBool eq(...) {
6    ...
7    }
8  }
```

Listing 5.4: Just Class in Java

```
1  package ABS.StdLib;
2  public final class Maybe_Just<A> extends Maybe<A> {
3
4    public final A arg0;
5    public Maybe_Just(final A arg0) {
6      this.arg0 = arg0;
7    }
8    public abs.backend.java.lib.types.ABSBool eq(...) {
9      ...
10     }
11 }
```

Objects Representation and Interaction The semantics of ABS objects defined them as actors running messages from their queue one at a time. These actors can be grouped in COGs and thus are they have to share the same thread of execution. To translate this functionality, all actors the original Java backend extend a superclass class for all ABS objects as in Listing5.5. This class provides the basic initialization any object instance and assigns it a COG. This superclass will be present in this form or an extended form through all iterations of the Java backend, however in the original Java backend such object abstractions exist for every notion of ABS including predefined data types (Int, Boolean, Rational), method calls(synchronous or asynchronous), ADTs (as shown in the previous paragraphs), object references (classes and interfaces) and most importantly active processes that are running like the execution of a method call within an actor. This object hierarchy is very stable and ensures correct code generation from an ABS model, but has significant performance penalties as it does not use existing Java libraries and constructs such as basic one-to-one mapping to method calls or primitives.

Cooperative Scheduling The solution for correctly translating suspension and resumption of messages was as detailed in the previous chapter and based on a basic thread-based approach. Every message that is generated via a synchronous or asynchronous call is a particular type of object in the backend. In the original solution these objects are an extension of the Thread class and the main idea behind the concurrency model was to assign a lock for each COG and have threads competing for execution. When messages are suspended or COGs are blocked, the generated code will use the readily available JVM runtime to manage suspended threads, saved call stacks and context switches, ensuring a correct execution of the ABS model.

Listing 5.5: ABSObject Class in Java

```
1  public abstract class ABSObject implements ABSRef {
2
3    protected COG __cog;
4    protected final long __id;
5
6    protected ABSObject(COG cog) {
7      this.__cog = cog;
8      this.__id = getFreshID( );
9      this.__cog.register(this);
10   }
11 }
```

Resource Management, Time Models and Actor Location The ABS features for modeling time, resources and actor location are not part of the core ABS language and were only introduced later when the language and first backends became more stable. In the original Java backend which targeted the core concurrency model there was no support added for these features. As users of ABS focused more on the performance of the generated programs, research in the Java backend first went towards improving thread management and scalability. Thus the original Java backend did not support resource and time, but it was integrated as part of the another backend to support actor location and scalability, which is described in Section 5.2.3.

5.2.2 ABS-API using Java 8

The second iteration of the Java Backend was focused on reducing code size and improving its readability bringing it closer to the ABS code. It was also focused on improving performance in a highly parallelizable application, but restricted to one machine. The research conducted in this thesis contributed to this solution named as the ABS-API library. The ABS-API library [SNA+14, SAB+15] was introduced as a solution to translate ABS code into production code initially for parallel applications. Starting with Java 8, there were new features that allow wrapping of method calls into lightweight lambda expressions such that they can be put into a scheduling queue of an Executor Service. Running objects were now mapped to a single thread, significantly reducing the number of idle Threads at runtime.

ADT, Functional Features, Resource Management and Time Models One of the biggest drawbacks in terms of performance for the original Java backend was the fact that ADT had to be translated as objects and pattern matching on them would require indefinite recursive calls to completely match the data type. The ABS-API completely forgoes support for data constructor ADT and only translates simple data type as enumerators in Java. Thus pattern matching for these ADT only requires `switch` statements and greatly improves performance of ABS models that do not require the use of complex ADT when running in a real-life environment. As with the original Java backend, the ABS-API was focused only on the core features of ABS and did not implement support for the simulation of resources and time. However it did introduce a solution to differentiate between local and remote actors and how object references and futures were exchanged between actors. This solution along with the mechanism for propagation of futures are presented in the next section.

Objects Representation and Interaction The ABS-API was developed to mitigate the performance issues of translating ABS code into Java code. The added features in Java 8 allow for a more efficient

and easy to use implementation of the actor model in ABS. This API provided an intuitive mapping between ABS objects and Java threads and a more straightforward correspondence between ABS and Java. First, methods in an interface were declared as Defender Methods using the **default** keyword. This allowed actors to have a default behavior and optionally override this behavior to suit a specific function. For instance, in Java 8 `java.util.Comparator` provides a default method reversed( ) that creates a reversed-order comparator of the original one. Such a default method implementation eliminated the need for any implementing class to provide such behavior by inheritance. Second, the introduction of Java Functional Interfaces and lambda expressions was a fundamental change in Java 8. All interfaces that contain only one abstract method were now functional interfaces that at runtime could be turned into lambda expressions. This meant that the same lambda expression can be statically cast to a different matching functional interface based on the context. This was a fundamental new feature in Java 8 that facilitated application of a functional programming paradigm in an object-oriented language. This API made use of these new features available in Java 8 because many of the interfaces found in the Java libraries were marked as functional interfaces, most important of which to this research were `java.lang.Runnable` and `java.util.concurrent.Callable`. This meant that a lambda expression could replace an instance of Runnable or Callable at runtime. Therefore a lambda expression equivalent of a Runnable or a Callable could be treated as a queued message of an actor and executed. Finally, Java Dynamic Invocation and execution with method handles enabled JVM to support efficient and flexible execution of method invocations in the absence of static type information. This feature has been validated performance-wise over anonymous inner classes and the Java Reflection API. Thus, lambda expressions were compiled and translated into method handle invocations rather reflective code or anonymous inner classes. A sketch of the overall ABS-API class diagram is shown in Figure 5.1

A Java API for the implementation of ABS objects and their interactions had the following three features. First, one actor should be able to asynchronously send an arbitrary message in terms of a method invocation to a receiver actor. Second, sending a message can optionally generate the equivalent of an ABS future that the sending actor can use to refer to the return value. Finally, an object during the processing of a message should have a context reference to the sender of a message in order to reply to the message via another message. All these characteristics co-existed without requiring any modification of the intended interface, for an object to act like an actor. The ABS-API library had a fundamental interface namely the Actor Interface. This interface provided a set of default methods, namely the `run` and `send` methods, which the implementing classes have access to. A default implementation for a single machine was called `LocalContextActor`. This contained a queue that supported concurrent access as ABS semantics imposed that an actor can receive messages in its queue from multiple other actors that have a reference to it. The default `run` method took a message from the queue and executed the message accordingly. The default (overloaded) `send` method stored the sent message in the corresponding queue. In ABS, futures are used to control synchronization. In the ABS-API messages that were expected to return a result were modeled as instances of Callable and a future was created by the `send` method which is returned to the caller, while those messages that needed to run in parallel without a future reference to the outcome were modeled as instances of Runnable.

For running on a single machine, an actor had a unique identifier and was deployed in a local context (similar to Akka actors), where it could send messages via lambda expressions to other actors registered in the same context. Its configuration is illustrated in Figure 5.2. Each actor was modeled to have its own queue of messages and run them on a single process. Abstracting from the syntax of the actual parameters, in ABS an asynchronous invocation of a method `m` of an actor `a` is described by a statement `Fut f=a!m( )`, where f is a future used to store the return value (assuming that m contains a return statement). In this iteration of the backend this invocation will be stored in a queue of the called actor `a`. The `LocalContextActor` encapsulated the entire behaviour of the actor that comprised of the continuous running cycle, the task message queue, the single thread restriction and the cooperative scheduling of its suspended tasks.

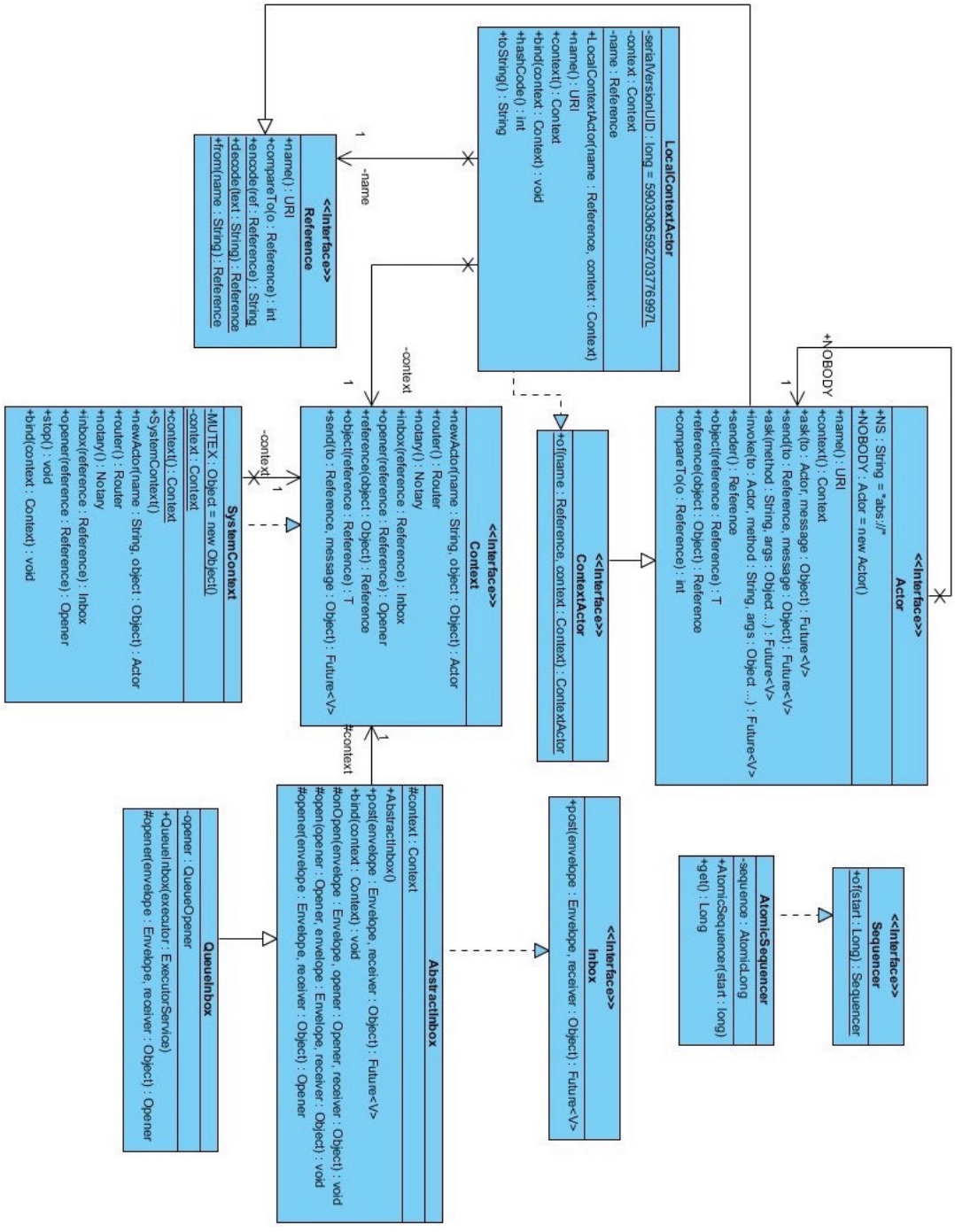

Figure 5.1: ABS-API Class Diagram

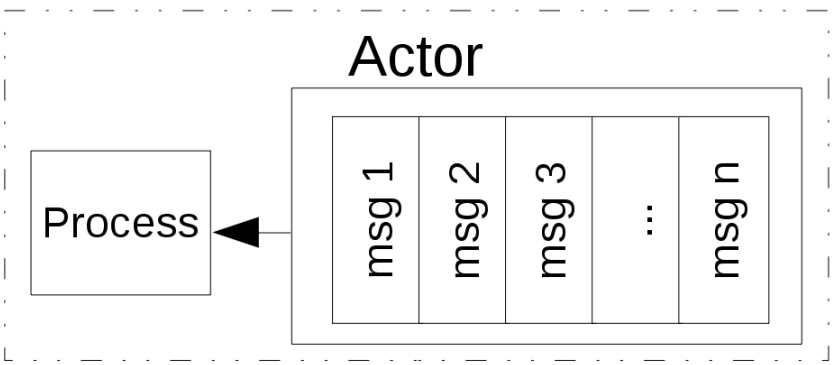

Figure 5.2: Actor Model

The general class that replaced message representation from the original Java backend and avoided using a thread for each message is presented in Figure 5.3. The class simply creates a lambda expression that takes the form of a Java Runnable or Callable and subsequently a wrapper future which may be used for synchronization purposes. This class did not extend from `Thread` and was just a Java object without any extra memory allocated.

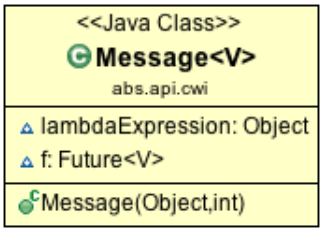

Figure 5.3: Message encapsulation

Cooperative Scheduling Again, abstracting from the syntax of the actual parameters, ABS features synchronous method calls for self calls and objects within the same COG. These statements are described in the standard manner of the form `x = a.m( )` or `x = this.m( )`. In case of a call to a different actor `a` the semantics of such a synchronous call can be translated by `Future f = a!m( ); x = f.get`, for some future f (of the appropriate type). In case the actor calls its own defined method `m` the translation depends on whether cooperative scheduling may be encountered. Futures can be passed around as references and provide a `get` operation described by `f.get` which results in the value returned and blocks the current actor if the corresponding method invocation has not yet computed the return value.

The ABS-API also had the purpose to formalize actor-based programming which implies asynchronous message passing together with the ever-growing object-oriented software engineering approach. Using asynchronous message passing and a corresponding actor programming methodology which abstracts invocation from execution (e.g. thread-based deployment), it fully supported and emphasized the programming to interfaces discipline. For a single machine, a thread, called a *Sweeper* was introduced. This thread was available across all actors, that continuously checked all actors that had messages ready for execution. It then submitted the message at each queue head to an executor service to minimize thread explosion in case of a large number of actors. Actor execution was demand-driven as shown in Figure 5.4,

with a single thread that is spawned into the state *ready* once the first message was received in the actors queue, moves to state *execute* and runs all messages in the queue and goes into state *stop* once the queue becomes empty, restarting once another message is inserted in the queue. This makes actors completely independent from each other unless they explicitly call the synchronization mechanisms *get* and *await*. This approach represented the basis for the demand-driven `Main Task` present in the JAAC runtime.

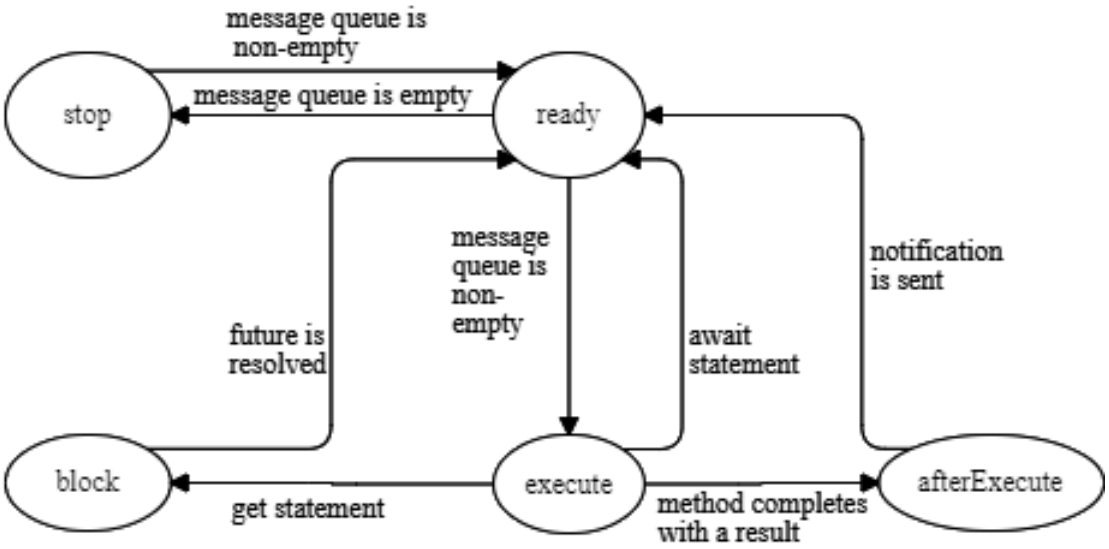

Figure 5.4: Actor State Diagram

Furthermore this minimized the number of threads to just those that required saving the call stack of suspended methods within an actor caused by the await statement. The solution to achieve this was to add a central context for all actors in the system and followed this execution sequence:

- Each asynchronous call/invocation is a message delivered in the corresponding object's queue.
- All objects in the same Concurrent Object Group (COG) compete for one Thread.
- A *Sweeper Thread* decides which task should be created and be available for execution from the available queues.
- A thread pool executes available tasks based on a work stealing mechanism.
- On every await statement, we try to optimally suspend the message thread until the continuation of the call is released.

A key feature of the *Sweeper* thread was that it allowed efficient scheduling of tasks within an actor. It prevented redundant thread creation by having suspended tasks of an actor given priority once they were released to compete for the actor's lock. This *Sweeper Thread* however became a bottleneck when the number of actors was very large while also making actors dependent on each other. This happened especially in applications that had a large number of small computations in actors as the *Sweeper* thread completed a very big cycle at the end of each task. This was also true for programs where tasks were very large, which meant that the *Sweeper* thread would occupy CPU while waiting for actors to become

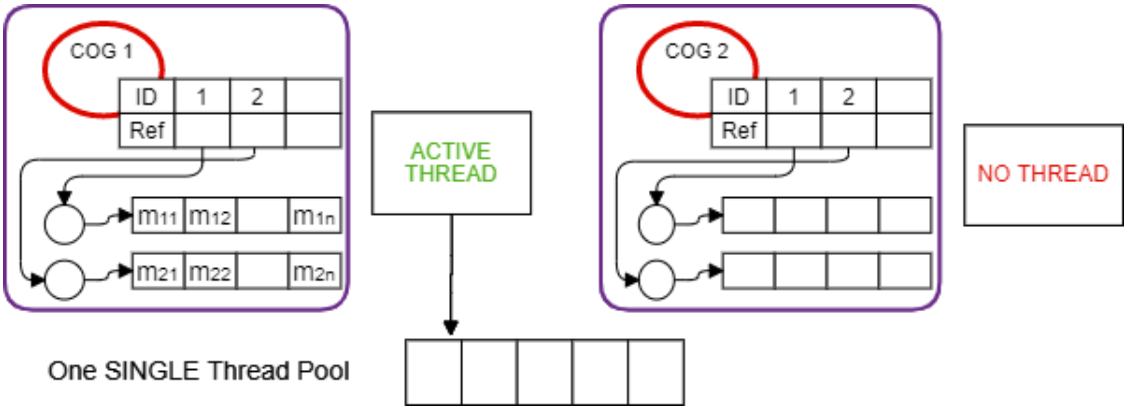

Figure 5.5: Thread Creation and Scheduling

active again. The existence of the *Sweeper* thread was to support actors' cooperative scheduling property, without creating a lot of threads in the system, but still had some drawbacks.

The *Sweeper* thread was thus removed from the model of the API and replaced with a new mechanism to support cooperative scheduling. First of all when a *get* statement is called on a future, the actor moves to the state *block* until the future is resolved. If an *await* statement is encountered, the actor invokes another exposed method *await* which receives a boolean condition or a future to suspend on and a continuation in the form of a lambda expression. The actor will then store a mapping of the continuation and the condition or future in a separate map as either *futureContinuations* or *conditionContinuations* specific to each actor and moves to state *ready*. The main executor introduced in the library is now responsible when, a thread completes, to run the *afterExecute* method which verifies if it has to send a notification to the actor from which it came to avoid a busy-waiting thread that may do this work. The method has to search each actor's continuation maps to identify the continuations that may have been resolved by this method (either an existing boolean condition or the actual future that has been resolved).

Figure 5.5 shows how active objects in this implementation are demand-driven and only create a physical thread once they have a method to execute inside their queue and the `ExecutorService` has a thread available in its pool. Finally, when all messages are blocked or the active object's queue is empty the thread is ended and released back to the pool of the `ExecutorService`. The impact of this approach is evaluated on an ABS benchmark comparing the performance of the Java 8 backend with the ProActive backend for ABS discussed in Section 5.2.3. The only drawback of this implementation approach is that if a method call contains a recursive stack of synchronous calls, this stack needs to be saved when encountering an `await` statement, which cannot be realized with lambda expressions. To solve this problem, such a call stack can be modeled by continuation functions and then saved as lambda expressions. These lambda expressions are ordered such that, once released, the continuations can execute and maintain the correct logic of the program.

Actor Location An extension the ABS-API library that supported an actor programming model in a distributed setting, with enhanced cooperative scheduling, distributed future control and garbage collection was proposed as part of the research in this thesis. The library had a new and simple format for classifying actors based on their intended behavior. It introduced a class hierarchy of actors running on a local host and actors whose functionality is reachable from a remote host. This hierarchy handled garbage collection and controlled the peer-to-peer communications between remote hosts, as well as offering a clear separation between an application running on a single machine or in a distributed environment.

The library was extended to ABS-API-Remote and had added support for identifying actors by a

URI object which is defined by an IP and port allowing both local and remote communication. Actors were aware of their location, as well as other actors identified by the same IP, allowing for optimizations depending on whether two actors work on shared or distributed memory. This optimization was referred as **Location Aware Actors** or **Distributed Actors**. The member queue however was restricted to one data structure per IP name in the application. Each actor that had the same IP in its identification key had a reference to this queue. The default `run` method behaved very similar as in the previous solution, the only difference being that it polls a message from the shared queue, before checking its type and executing the message. The default `send` method is the direct translation the `actor!method( )` statement from ABS and stores the method call represented by the lambda expression given as an argument in the corresponding shared queue and returns a Future to control synchronization.

Actors were classified based strictly on the machine on which they reside and thus based on whether they are allocated in memory on the machine. The API had to maintain data specific to the machine. Firstly, it contained a customized ThreadPoolExecutor to which the actors residing on the machine submitted their tasks. This ThreadPoolExecutor was available to all actors on one machine and implemented the *afterExecute* method (state). Secondly, the class also contained a map of all the actors that were initialized on one machine such that remote messages can be routed to the correct actor. Finally, the class contained a table of the socket streams with all other machines in the system that grew and shrunk dynamically, as more machines were added to the system. An important observation is to notice that socket streams were initialized only when a remote actor belonging to a node that was previously unknown was instantiated in the system and a *listener* thread was assigned to the stream processing either incoming messages to the machine and as such, only if the setting was distributed. The machines communicated through serialized messages and objects. The main features of the ABS-API-Remote library were:

- An actor is identified by a URI object and is aware of its location with regards to the other actors in the system
- Sending a message is done by either placing the lambda expression in the shared queue if the actor is local or sending the message to a different VM if the actor is remote.
- An object during the processing of a method should have a context reference (defined by the URI) to the sender of a message in order to reply to the request.

All these characteristics must co-exist without requiring any modification of the intended interface, for an object to act like an actor.

The ABS-API-Remote was focused for programming with distributed actors. A more detailed illustration in Figure 5.6 explains how an actor is referenced on both a local and remote machine. In this setting we have an actor *a1* with a unique global identifier which is a Java URI that is "IP2:a1", where IP2 is the IP address of Node 2 on which the actor was instantiated and a1 is a unique identifier of the actor. Node 1 has a reference to this actor and its interface which contains method *m( )* is also available. An important objective of our solution is to avoid actors entering a busy-waiting loop in order to check the resolution of futures. To achieve this, we insert a *remoteUncompletedFutures* data structure which retains all the futures that were generated by calls to remote actors. The machine then sends a serialized lambda expression of the asynchronous method call to the socket. Each machine is aware of the senders of incoming messages, therefore when an actor completes a remote call, the *serialized result* of the actor can be sent back as a reply. This behavior is part of the *afterExecute* method of the machine's main executor and is illustrated by the state *afterExecute* in the state diagram of the actor Figure 5.4. To allow remote actors to identify which reply belongs to which future in the queue we introduce a naming scheme in the form of "IP:f" where IP is the address of the actor that will complete the future and f the unique global identifier (futureID) of the future.

The general mechanism is best described in terms of an example scenario with two asynchronous calls to the same actor:

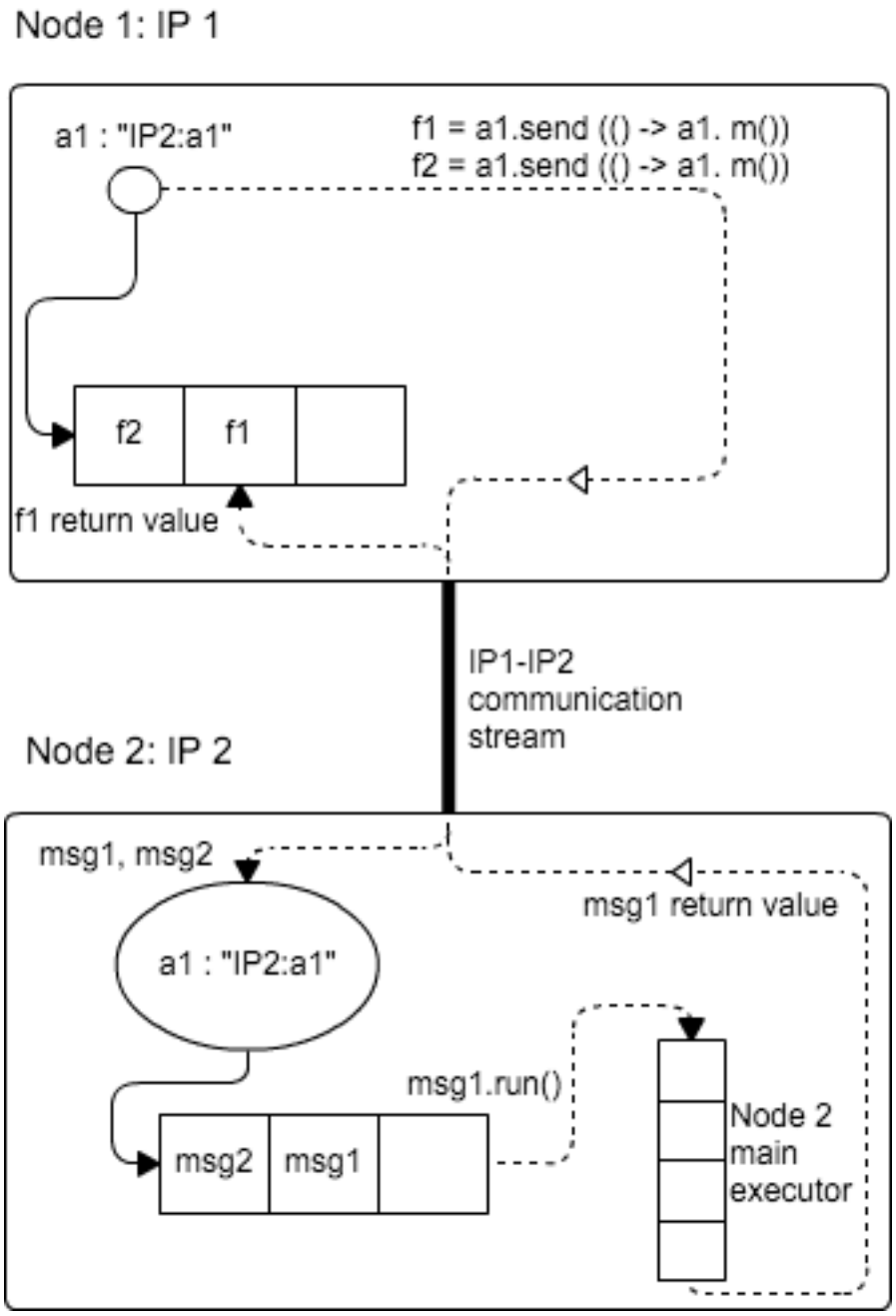

Figure 5.6: Future Flow

1. Node 1 sends the following sequence of messages to actor *a1* on Node 2.
   - A futureID "IP2:f1" identifying the future that will be generated by the following asynchronous method call.
   - A pair $< IP2 : a1, m() >$ representing the first asynchronous method call to actor *a1*.

- A futureID "IP2:f2" identifying the future that will be generated by the next asynchronous method call.
- A pair $< IP2 : a1, m() >$ representing the second asynchronous method call to actor *a1*.

2. The two uncompleted futures *f1* and *f2* and their identifiers are stored as mapping as *remoteUncompletedFutures*.
3. Actor *a1* receives from the socket stream the two identifiers and two messages *msg1* and *msg2* in the same order and inserts them in the message queue.
4. Actor *a1* schedules *msg1* and *msg2* in a FIFO order on Node 2 main executor unless rescheduled by an *await* statement.
5. When either message has finished executing, the *afterExecute* method of the main executor sends back the corresponding futureID within a either pair $< IP2 : f1, result >$ or $< IP2 : f2, result >$back to the socket stream where the message came from.
6. The socket stream forwards the result to Node 1.
7. Future *f1* or *f2* is completed with the received result depending on the futureID.

5.2.3 ProActive Backend for ABS

The ProActive Backend for ABS [RH14, HR16] was developed based on the original Java Backend to generate ProActive code from an ABS source with the main purpose of running ABS programs in a real distributed setting. Its development and research was made in parallel with the ABS-API, and as such did not integrate any of the optimizations described in the previous section. Instead it provided a full integration of the ProActive Language for distributed systems [] into the original Java backend.

ADT, Functional Features, Resource Management and Time Models This implementation used the full support of the original Java backend for ADTs and thus the representation of these types took the form of classes and subclasses for each ADT. Each class needs the methods described in Listings 5.3 and 5.4 to compare or verify equality for both simple ADTs or ADTs with data constructors. Furthermore instances of these ADTs are normal Java objects and pattern matching on them requires a complete recursive stack of `eq` methods. The ProActive backend was mainly focused on a correct translation and behaviour of ADTs in Java. As the ProActive backend for ABS used mainly the features of the old Java backend and was only tailored towards real distributed systems it did not add any support for symbolic time or resource simulation. Its support was strictly based on the core ABS features and added the ProActive extension for objects to be able to interact according to ABS semantics regardless of physical location.

Objects Representation, Interaction and Location With a target towards distributed deployment, the ProActive backend had an important challenge to overcome when representing ABS objects. The semantics of ABS make all objects accessible by any other object as long as one holds a reference to it (whether it is passed as a method argument or through the return type of a method). A trivial translation would have been a one-to-one correspondence between ProActive objects and instances of ABS classes, but this would have created the same overhead as in the original Java backend with a thread associated to each object. Instead this backend proposed a mapping from ABS COGs to ProActive objects and thus to Java threads. This was very intuitive as an ABS COG is restricted to running a maximum of one object from its group. As such COGs were translated to active objects that held the entry point for all the ABS objects in their group. ABS objects were in turn translated as passive objects as a new step towards optimizing performance of ABS models translated to Java. Inside an active object representing an ABS

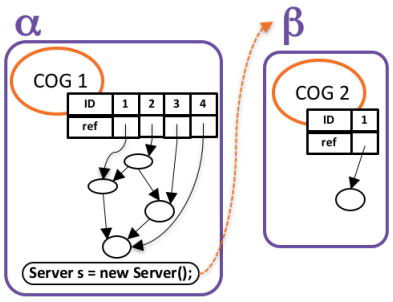

Figure 5.7: ABS new in ProActive

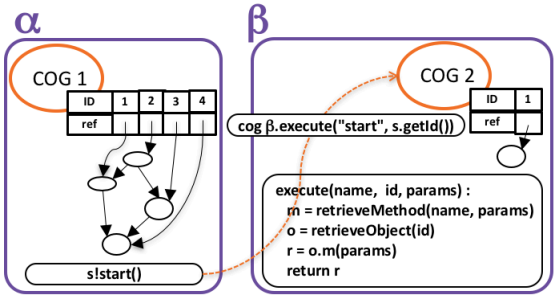

Figure 5.8: ABS asynchronous method call in ProActive

COG there was a mapping between object identifiers and object references such that all objects pertaining to the same group could access each other. At the same time objects in different COGs would require inter-COG communication which was translated as communication between two active objects vis Java Remote Method Invocation (RMI) [PM01]. This configuration of objects is illustrated in Fig. 5.7.

An asynchronous call was translated as an equivalent remote call to the ProActive object representing the COG which contained the target object. This target object would be identified through a unique ID and execute the method through Java's reflection feature as in Listing 5.8. Another challenge when configuring an ABS backend for distributed support is how references to objects are passed to remote target objects as method arguments. All method parameters are copied which is safe for immutable types like ADTs or primitives. For object references, the copying procedure requires the object's unique identifier and its reference to its hosting COG. This is sufficient due to ABS actor semantics which require all interactions between objects on different COGs to be asynchronous.

Cooperative Scheduling Cooperative scheduling in this backend was done by tuning the scheduling of requests in ProActive objects. A simplified version of this tuning is part of the JAAC runtime and API and described in Section 6.1.2. In ProActive, however the requests are still each assigned a thread and each COG has its own thread pool to deal with the issues of saving call stacks. Another important aspect that derived from this solution is the presence of a single queue of requests for each ProActive object (which resembled an ABS COG). This concept was adopted in the runtime of JAAC where newly instantiated objects can reference the queue of the object where the `new` call was issued (this will be covered in depth in Section 7.1.1).

Chapter 6

ABS Runtime as a Library API

The core of the JAAC library is written in Java and can thus be used in both Scala and Java. The API directly provided in the Java library may at some places look too permissive, in the sense that the user can misuse the API, for example, to run arbitrary Callables on an actor. We include here the expected usage of each part of the API. The first part of this chapter aims at explaining the power of JAAC in allowing integrated usage of actors, futures and coroutines. In the second part the usage of the API in Scala, an extension called ASCOOP, is explained through the use of several small examples that follow the newly introduced constructs. These constructs enforce at compile time the intended usage as we will describe in Section 6.2.

This chapter covers the library API in Java and Scala that allows to extend the programming to interfaces paradigm to actors in a seamless manner. It means sending a message to an actor is allowed only if the actor defines a corresponding method publicly; in other words, sending a message can be seen an asynchronous call to the corresponding method. This gives programmers much more rigor in programming with actors, compared to the typical approach of actors such as in Akka, where messages are allowed to have any type and the interface of an actor (in terms of what messages it can receive) is not well defined. The extension ASCOOP not only enables static type checking of messages sent and received by actors at compile time, but also allows defining hierarchies of actor classes.

The API also provides a design pattern for cooperative scheduling at the programming level of tasks and sub-tasks within an actor. In this chapter, we use the term *task* to denote the execution of the body of the method invoked by the corresponding message. A task, once started, does not necessarily produce its final result upon completion. It can spawn *sub-tasks* (also referred to as a *continuation*) that will run in the same actor interleaved with other tasks and sub-tasks. The result that should be produced by the original task may thus depend on the completion of these sub-tasks. A sub-task can be given an enabling condition (also referred to as *guard*) such that it will be scheduled only when the condition is satisfied. An enabling condition can be based on the actor reaching a particular state, or a future being available as explained next or a logical time condition if the programmer uses the timed-model explained in Section 4.2.3.

The API provides a tight integration of futures with actors. This means that, first of all, sending a message or spawning a sub-task creates a future variable that will be completed when the corresponding task or sub-task has finished. Then, one can spawn a sub-task guarded by this future that will naturally model the continuation in terms of what should happen after the future is ready. The continuation is guaranteed to run on the actor's single thread of execution and thus preserving the actor semantics. This is different from using futures and actors together in for example Akka or Scala where it is left to the programmer to make sure that futures do not break actor semantics (see related work). The integration of actors and futures gives us a powerful means for the expression and analysis of fine-grained run-time dependencies between tasks through cooperative scheduling within an actor.

6.1 JAAC API through an Example Program

Throughout this section we will show the use of the API through the implementation of the `WorkerPool` actor introduced in Chapter 3. To declare a new actor, one must extend from the `LocalActor` class[1]. This way, actor instantiation is like the instantiation of normal objects and is simply specified by the **new** keyword in Java. This class defines its state in terms of fields and can also have constructors and methods like normal classes in Java. An example actor is shown in Listing 6.1. It describes an implementation using the JAAC library of the corresponding actor class `WorkerPool` in Listing 3.6.

Listing 6.1: Definition of an actor

```
1 class WorkerPool extends LocalActor {
2   Set<Worker> workers;
3
4   public WorkerPool( ) { / * initialize the pool * / }
5   public Future<Result> sendWork( ) { / * method body * / }
6   public Future<Void> finished(Worker w) { / * method body * / }
7 }
```

Methods that can be called asynchronously must return a value of type `Future`. This is different from ABS. When a method in ABS would return a type `T`, a corresponding method defined in JAAC should return type `Future<T>`. This `Future` wraps the return value, for example, `doWork( )` method is expected to return a value of type `Future<Result>`. When no result is expected, `Future<Void>` is used. This allows for *chaining* calls in JAAC, e.g., a method can call another method and return the future obtained from that call. On the other hand, when the method wants to just return a value, the value needs to be lifted into a future containing that value. For this purpose, one can use the two variants of the static `done( )` method provided by the `Future` class in Listing 6.2. These methods create a new instance of a completed future: one with a result put into it, and one for futures of type Void.

Listing 6.2: Future class with its helper methods

```
1 public class Future<V> {
2   public static <T> Future<T> done(T value) { ... }
3   public static Future<Void> done( ) { ... }
4 }
```

Message parameters are defined as method parameters and even though we cannot enforce using immutable data structures, the actor is not supposed to mutate the arguments passed to it. The methods inherited from `LocalActor` (see Listing 6.3) are explained in the sequel.

6.1.1 Library Methods.

The API provides several methods that can be used both by the compiler from ABS to Java or as a standalone Java library. We highlight the methods that support the simulation of ABS features in the Java language in Listing 6.3.

Spawn. The first method used to offer the emulation support of coroutines and the internal spawning mechanism of tasks is called spawn and its usage is highlighted is Listing 6.4. Its intended usage is only on a reference to `this` as actor semantics impose that all communication between actors is done through asynchronous calls. To implement the suspension point in the `sendWork` method of the actor class `WorkerPool` in Listing 3.6, the spawn method is used to generate a new task (in the form of a Callable) on line 3 of Listing 6.4.

[1] `LocalActor` is used for non-distributed applications which run on a single JVM.

Listing 6.3: API exposed by an actor

```
1  abstract class LocalActor {
2
3   // method used for sending asynchronous calls
4   <V> Future<V> send(Callable<Future<V>> task) {...}
5
6   // helper methods enabling coroutine implementation
7   <V> Future<V> spawn(Guard guard, Callable<Future<V>> task) {...}
8   <T, V> Future<T> getSpawn(Future<V> f, CallableGet<T, V> task) {...}
9   <T, V> Future<T> getSpawn(Future<V> f, CallableGet<T, V> task,
10  int priority, boolean strictness){...}
11 }
```

The `guard` parameter represents the associated enabling condition that can be either the completion of a future or a condition based on the actor's internal state. Here the abstract class `Guard` allows for multiple types of enabling conditions to be evaluated. The aforementioned enabling conditions are subclasses of `Guard` known as `FutureGuard` and `PureExpressionGuard`. The static overloaded method `convert` creates instances of these subclasses depending on the parameter passed. The enabling condition has to *guard* the continuation (the block of code that starts on line 4) and needs to be checked every time the actor attempts to schedule the task. Therefore we transform this enabling condition into a guard from a lambda expression that verifies if the set of available workers is non-empty (line 2). As a note, guards always refer to the local environment of an object (either future references or local variables and fields) and thus should not be passed to different target objects (hence the usage of spawn only on **this**). The other two methods (whose usage is already shown on lines 5 and 8) represent particular cases of this method. These two methods along with their usage and parameters will be explained in the next paragraphs.

Listing 6.4: Spawn Method Intended Usage

```
1  public Future<Result> sendWork( ) {
2   Guard nonEmpty = Guard.convert(( ) -> ! workers.isEmpty( ));
3   return spawn(nonEmpty, ( ) -> {
4   Worker w = workers.pop( );
5   Future<Result> f = w.send( ( ) -> w.doWork( ));
6   return getSpawn(
7     f, (r)->{
8       return Future.done(r);}, HIGH, STRICT);
9     }
10  );
11 }
```

Send. Asynchronous method invocations are modeled by invocations of the `send` method provided by `LocalActor`. According to its signature in Listing 6.3, the send method takes a `Callable`. It is a particular case of spawning a task without a guard. Unlike spawn, its intended use can be both a self call or a different target object (as it does not have a guard). Without a guard the newly spawned task will be ready for execution on the target object. It is also important to note that actor semantics of ABS impose that the spawned task be a method exposed by the target object's interface. As of now the library does not enforce this semantics, but we recommend as a general programming practice to avoid sending a task

represented by an arbitrary block of code to the target object.

In Java 8 lambda expressions have been introduced to allow a block of code to be passed as an argument and be treated as data (implicitly converted into a Callable or Runnable). Using a lambda expression as shown in Listing 6.5 we model an asynchronous invocation of the `sendWork` method of the newly created `WorkerPool`. The send method returns a future that will eventually contain the result of running the `Callable` parameter. The `task` parameter itself will be stored in the internal task queue of the actor (see Section 4.2.2).

Listing 6.5: Sending an Asynchronous Call

```
1 WorkerPool pool = new WorkerPool( );
2 Future<Result> fut = pool.send(( ) -> pool.sendWork( ));
```

getSpawn. This method is used in Listing 6.6 to model an ABS `await` syntactic sugar illustrated in Listing 3.7. It is a particular case of spawning a task that is used only together with a future guard. This guard's result is passed as a parameter to the spawned task making it available to use once the future is complete (the task is ready to be scheduled). This method generates an instance of type `CallableGet` with a future instance of `Future<Result>` as the associated enabling condition.

The type `CallableGet` represents a `Callable` instance that in the method application in line 2 (Listing 6.6) *implicitly* binds its parameter `result` to the (completed) value stored in the future `fut`. This provides a cleaner, more intuitive way of retrieving and using a future's result as part of the block of code to be run by the actor when the future completes.

Listing 6.6: getSpawn Method Intended Usage

```
1 Future<Result> fut = w.send( ( ) -> w.doWork( ));
2 getSpawn(fut, (result) -> {...});
```

6.1.2 Call Stack and Priorities.

In ABS an asynchronous method invocation may in general generate a stack of synchronous calls. In the JAAC API we can model such a call stack by generating for each call a corresponding task as a `CallableGet` instance that represents the code to be resumed after the return of the call and that is parameterized by an enabling condition on a future uniquely associated with the call (see Listing 6.7 for a simple example of a synchronous call). To ensure that these tasks are executed in the right order, that is, tasks belonging to different call stacks are not interleaved, we assign them a HIGH priority. But if one of these instances should suspend, it will get the LOW priority, i.e., the same as normal tasks. By default tasks that are created by spawn or send are set with a LOW priority. Moreover, the default priority, when getSpawn is used without priority arguments is also LOW.

The default scheduling policy of an actor is to schedule one of the enabled tasks with highest priority. If all such tasks are disabled the scheduler moves to the next priority. As a result, when a synchronous call returns, the task representing its return will have priority over all other tasks (note that the enabling conditions of these tasks ensure the LIFO execution of the tasks representing a call stack).

The additional `strictness` parameter allows the following refinement of the scheduling policy: an enabled task of a lower priority can only be scheduled if all higher priority tasks are disabled and *non strict*. As an example of the use of this additional parameter, the modeling of the `f.get` is illustrated on line 8 of Listing 6.4. Note that this combination of HIGH priority and STRICT does not allow scheduling of any other tasks (of the given actor).

Listing 6.7: Usage of getSpawn to emulate a synchronous call of an Actor

```
1  public Future<Worker> getWorker( ) {
2   Guard nonEmpty = Guard.convert(( ) -> ! workers.isEmpty( ));
3   return spawn(
4    nonEmpty, ( ) -> {
5     Worker w = workers.pop( );
6     return Future.done(w);
7    }
8   );
9  }
10
11 Future<Result> sendWork( ) {
12  Future<Worker> fw = this.getWorker( );
13  return getSpawn(
14   fw, (w)->{
15    Future<Result> f = w ! doWork( );
16    return f;
17   },
18   HIGH, NON_STRICT
19  );
20 }
```

6.2 ASCOOP Scala API

In this section we present the extension o the JAAC library to ASCOOP. The runtime core of ASCOOP is written in Java employing efficiently the available executor services for thread pooling and using very little synchronization locks, thus giving us a very high performance. Using Scala features for writing domain-specific languages, ASCOOP improves the general programmability of actor-based systems in Scala, enabling one to write simpler programs without compromising performance. The extension of the API involves the following three main aspects.

6.2.1 Actors Typed with Interfaces

One of the main concepts of object-oriented programming is that classes encapsulate their internal implementation and present a public interface to the outside (whether or not they actually implement an explicit interface declaration). This allows for a clear separation of concerns, enables static typing, and additionally caters for inheritance and type hierarchies.

The concept of actors can be viewed as bringing together abstract data types (as represented by classes) and the concurrency control (as represented by threads). It is therefore very natural that actors can define their public interface in exactly the same way as classes do (cf. [JHS+12]). In this section, we show how actors define their interface and how they communicate. For simplicity in this section, we assume the fire-and-forget approach to message passing; in other words, we assume for now that actor messages do not produce any return values. In the next section, we will elaborate further how to make messages return results and how those results can be received by the sender of a message.

Defining an Interface We use the term interface to refer to the concept and the term trait as it is used in Scala. In the context of Scala, we may use these terms interchangeably. To define an explicit interface

Listing 6.9: Actor Definition

```
1  class PingActor(pongActor: PongActor) extends PingInterface {
2
3    private var pingsLeft = 0
4
5    override def start(iterations: Int) = messageHandler { /* code */ }
6    override def stopped = messageHandler { /* code */ }
7    override def pong = messageHandler { /* code */ }
8    private def ping: Future[Void] = messageHandler { /* code */ }
9  }
```

for an actor, one can extend the TypedActor trait to define the messages an actor can receive. These messages are defined as normal methods with the restriction that they must have a return type `Future`. In general the `Future` is parameterized with a type that would denote the result type of the message, but for now we assume only Void, which means these messages produce no return value. In itself, since these methods return a type `Future`, it makes it clear on the caller side that calling them could result in an asynchronous invocation (more on this in the next sections).

Listing 6.8: Ping-Pong Interfaces Definition

```
1  trait PingInterface extends TypedActor {
2
3    def start(iterations: Int): Future[Void]
4    def stopped: Future[Void]
5    def pong: Future[Void]
6  }
7
8  trait PongInterface extends TypedActor {
9
10   def stop(sender: PingInterface): Future[Void]
11   def ping(sender: PingInterface): Future[Void]
12 }
```

In the ping-pong example depicted in Listing 6.8, the start message will start a round of ping and pong messages being sent back and forth for the given number of iterations. After that, it will send a stop message and expects a stopped message back. As can be seen in the method signatures, the Ping actor is expected to know its counterpart, while the Pong actor receives a reference to its counterpart on every message. This demonstrates how in general actor references are typed by proper types and can be sent as parameters to messages.

Defining an Actor with Message Handlers Once a trait is defined for an actor like in the previous section, one can create a concrete actor by writing a class that extends that trait (see Listing 6.9). Note that when programming in Scala, one is not forced to write a trait always, but can immediately start defining an actor by creating a class that extends the TypedActor trait.

As with defining normal classes in Scala, users can define fields and methods inside actors. Mutable fields (those defined as var) will constitute the actor state. All public methods of an actor must be defined using the messageHandler helper: this turns every call to this method into an asynchronous message passed to the actor. Obviously an actor may define any other internal methods, but in order to preserve the actor semantics, all methods not defined as message handlers must be only accessible to the actor itself.

Listing 6.10: Method Definition Using Message Handler

```
1  override def pong: Future[Void] = messageHandler {
2    if (pingsLeft > 0) { this.ping }
3    Future.done
4  }
5
6  private def ping: Future[Void] = messageHandler {
7    pongActor.ping(this)
8    pingsLeft -= 1
9    Future.done
10 }
```

Unfortunately, without explicit support from Scala, we cannot enforce this (in the same way other existing actor libraries cannot do that). The internal methods of an actor may be called synchronously, or may also be defined as message handlers, which means invoking them will schedule them as an asynchronous task that will be later executed.

As an important observation, using our library API, whether invoking a method corresponds to an asynchronous task or a normal synchronous call (as a stack-based method invocation in Scala) is determined at the definition of the method, namely, by using (or not) the messageHandler helper. On the caller side, nevertheless, the fact that invocations of message handler methods return a `Future` type makes it clear to the caller that the computation linked to the method is not necessarily executed synchronously and the result may be available later in the future. This is not contradicting the fact that futures may be passed around (as first-class citizens) and anyone may spawn sub-tasks to be executed using their result (see next section).

Programming a Message Handler As mentioned before, the main characteristic of a message handler is that it should return a `Future` type. For now, let us assume that messages return no result, so they should return Future[Void]. To create an instance of this type, one should use the static done method in the `Future` class. Listing 6.10 illustrates how a method is defined such that its invocation will be asynchronous.

Sending a message, as explained above, is by calling the corresponding method, e.g., in the listing above, the actor sends a message ping to itself by invoking this.ping method. Messages can have parameters, e.g., the ping message takes an actor reference of type PingActor, so we can send this message to pongActor instance, for example, like pongActor.ping(this). The message handler ping additionally changes the actor local state by decrementing the state variable pingsLeft. This significantly simplifies programming from a user perspective (compared to the Java API), as it removes the need to use lambda expressions or making sure parameters are immutable.

Await on Boolean Condition. An actor can spawn internal sub-tasks that await on a Boolean enabling condition, called a guard. This is achieved by using the on (guard) execute {sub−task} syntax. These sub-tasks are scheduled only if the associated guard is satisfied. In Listing 6.11, right after starting the ping-pong, the actor adds a sub-task that will run only when the required number of ping-pong messages have been exchanged. Similar to normal methods, sub-tasks should also end with done, in other words, they must also result in a future. As with the case of sending asynchronous tasks, specifying a boolean condition is much more intuitive in the Scala API compared to the Java one.

Listing 6.11: Awaiting on a Boolean Condition

```
1  override def start(iterations: Int): Future[Void] = messageHandler {
2   val t1 = System.currentTimeMillis
3   this.ping
4   on (pingsLeft == 0) execute {
5    val t2 = System.currentTimeMillis
6    pongActor.stop
7    Future.done
8   }
9   Future.done
10 }
```

Listing 6.12: Sieve: Returning Values using Futures

```
1  class SimpleSieve(prime: Int) extends TypedActor {
2
3   var next: Option[Sieve] = None
4
5   def divide(toDivide: Int): Future[Option[Int]] = messageHandler {
6    if (toDivide % prime == 0) {
7     Future.done(None)
8    } else {
9     next match {
10     case None =>
11      next = Some(new Sieve(toDivide))
12      Future.done(Some(toDivide))
13     case Some(nextPrime) =>
14      nextPrime.divide(toDivide)
15    }
16   }
17  }
18 }
```

6.2.2 Actors with Integrated Futures

In this section, we explain how message handlers are used to return the result of their computation encapsulated inside a future variable, and how the sender of a message can await the result to be available and spawn a sub-task to use the future result. This section uses the Eratosthenes Sieve as an example. In this example, we define actors that represent prime numbers and can check divisibility of an input by their prime number. The divide method, given a number, returns the same number as an Option if it is a prime, otherwise, returns None. Furthermore, when the next prime is found a new actor is created for that number.

Completed Futures and Delegated Futures

In the previous section, the static done method was used to create an instance of an Future[Void]. The overloaded done method can be used to wrap a value inside a future such that it can be returned. We illustrate this in Listing 6.12.

Listing 6.13: Sieve Main Method

```
1  object SieveMain extends TypedActor {
2
3    def main(args: Array[String]): Unit = {
4      val two = new Sieve(2)
5      val futures = for (i <− 3 to 1000) yield {two.divide(i)}
6      sequence(futures) onSuccess {
7        results: List[Option[Int]] =>
8        val primes = 2 +: results.flatten
9        ActorSystem.shutdown( )
10       Future.done
11     }
12   }
13 }
```

At lines 7 and 12, we know that the number is not (resp. is) a prime so the result is immediately known and returned. But not always the result of a message handler is ready at the end of its execution. For example, when the input number is not divisible by this prime, but there is a next prime that should be tried, the actor sends a message to the next prime actor that will eventually produce the desired result. Similarly, spawning a sub-task (using the on-execute construct introduced in 6.2.1 or onSuccess hook introduced below) returns also a future. Message handlers can choose to return the future created as a result of sending a message or spawning a sub-task. This can be seen as *delegating* the completion of a future associated with a task to a spawned sub-task or a message sent (see line 14). Below, we explain the details of how the library can efficiently implement this new mechanism for delegating futures.

Sub-tasks Awaiting Futures Once we have a reference to a future, the main question is how do we get access to the value inside it. Our approach is to allow actors to spawn sub-tasks with an enabling condition (known as a guard) based on the availability of a given future. The syntax to do this, as demonstrated in the listing 6.13, is using the onSuccess method on the future variable. The example code above shows how to start the sieve program and how to ensure that the computation is completed by awaiting the completion of all futures.

Given a future fut of type T, the generic syntax for writing a sub-task is:
fut onSuccess {v: T => sub−task}. Here the value inside the completed future will be accessible using the v variable in the sub-task. Additionally, the library provides the sequence function that can be applied on a list of futures to make one future that will hold the list of all results.

Our syntax looks similar to onSuccess method in Akka, but the crucial difference is that in Akka the call-back will run in a separate thread and thus the programmer must make sure that the code in the call-back may not close over the actor state. In our case, the sub-task will be scheduled in the same actor and is therefore completely safe. In other words, semantically it is similar to the combination of the ask and pipe patterns in Akka but with the advantage that one can write the sub-task inline, thus keeping the program logic simpler. Finally, compared to the Await method in Akka, no thread is blocked for the sub-tasks allowing any arbitrary number of such sub-tasks to exist.

Additionally, ASCOOP provides another way of spawning a sub-task by using the syntax fut blockingOnSuccess {v: T => sub−task}. This variant differs from onSuccess by that the actor will not execute any other tasks or sub-tasks before this sub-task is enabled and executed. Semantically it is like a blocking get on the future with the difference that it blocks the actor bot does not block the current thread and therefore has no performance penalties if one needs to use it.

Chapter 7

ABS to Scala Compiler

Together with the runtime presented in Chapter 4, a source-to-source translation from ABS to Scala is provided to support both the object-oriented and functional programming paradigms of ABS. The most important features of the compiler are the correct translation of the core ABS semantics together with the coroutines feature of ABS, the compilation of the Timed-ABS model extension and the translation of the ABS extension that models deployment components and consumption of resources.

7.1 Translating Core ABS and its Extensions to the Proposed Runtime

To provide a compiler for ABS for all the core features discussed in Chapter 3 a lot of the work that already existed for state-of-the-art backends for Maude and Erlang in [absc] was reused. For parsing and type-checking the Core ABS language we used the grammar and type system already in place to obtain the Abstract Syntax Tree(AST) of the program. The syntactical and semantic analysis steps are the same for every backend, the proposed compiler reuses these steps. The code generation step is backend specific and generates Scala code that uses the proposed Java Runtime. This involves using a visitor pattern to traverse the AST and generate code for each tree node. Most of the nodes do not present a difficult task, as interfaces, classes, method signatures or bodies, control statements and expressions all have one-to-one correspondents in Java or Scala. The exception to all these are the algebraic data types and the control statement *case* (discussed in Section 3.2) which provide the functional part of ABS. For this paradigm of ABS the translation must be done to the Scala language discussed in depth section 7.2. However, the rest of the code-generation for the core ABS is one of the main challenges as it has to take into account all of the particularities of ABS discussed in Chapter 3, which are:

- Instantiating objects as actors: all of the objects created using the `new` command behave like actors and need to be separated in their own COGs unless indicated by the `local` option which specifies sharing a COG with the parent object.
- Asynchronous communication: method invocations created using the "!" operator must be created as lambda expressions and passed to the called object/actor.
- Continuations: when release points are met, actors need to know from which point and in what state to resume execution of either the actor itself or a suspended message.
- Cooperative Scheduling: execution flow must be controlled to follow the semantics in Section 3.3 when encountering a `get` or `await` construct.

7.1.1 Objects as Actors in COGs

ABS objects are not just regular JVM objects, and require some changes when they are instantiated. First, all interface declarations need to extend the **Actor** interface and all class declarations need to extend the **Local Actor** default implementation such that newly instantiated objects can behave like actors. Extending this base interface and class allows for the calling of the default constructor that initializes each object with a message queue and a `MainTask` to iterate through it. It also makes available for each object the methods `send`, `spawn` and `getSpawn` to be used for asynchronous communication and cooperative scheduling). A simple layout of an interface and class declaration in ABS is given in Listing 7.1. According to ABS semantics, all declarations need to be comparable, whether they are defined as ADTs or objects, mainly due to the definitions and assumptions made in the ABS Standard Library. With respect to objects, there is no particular rule to comparing objects, so this is left to the default ordering that Scala makes for objects. For this to happen, all parent interfaces need to extend the Scala trait `Ordered` as shown in Listing 7.2.

Listing 7.1: An Interface and Class in ABS

```
1 interface Ping {
2   ...
3 }
4
5 class PingImpl(...) implements Ping {
6   ...
7 }
```

Second, we need to differentiate between the `new` and `new local` constructs. The presence of the `local` option puts the newly created ABS object into the same COG as the object that created it, while the absence of this option will place it a new COG. What this means for the generated code is that the actors have to create new or share the `messageQueue` or `mainTaskIsRunning` variables discussed in Chapter 4 such they can emulate placement on a new COG or the same one. To add this support, the generated code for each class adds a parameter `destCOG` as shown in Listing 7.2. This reference may point to the actor that called `new local` such that the the `messageQueue` or `mainTaskIsRunning` will be shared, otherwise if the reference is null, the variables will be created by the parent constructor. Depending on the location of the new object, its constructor will call moveToCOG(callerofNewLocal) or moveToCOG(null) (line 10) . The call to this method is very important as it needs to be done immediately after calling the super( ) constructor of the LocalActor class. The generation of a statement like **new** local Ping(...); will look as **new** Pingthis,...);)

Listing 7.2: COG and DC Assignment

```
1  trait Ping extends Actor with Ordered[Actor] {
2    ...
3  }
4
5  class PingImpl(var destCOG : LocalActor,var destDC : Actor,...)
6      extends LocalActor with Ping {
7
8   //constructor
9   {
10    moveToCOG(destCOG);
11    setDc(destDC);
12  }
13 }
```

Deployment Components Very closely related to COG assignment is the option to generate code with resource model support. This generation is optional and can be activated as an argument in the command line. To achieve this, the compiler automatically generates a second parameter for every ABS class definition (`destDC`). Again this parameter determines on which resource the newly created object will be placed through calling setDC(targetDC) (line 11). An ABS `new` construct with resource model support can optionally have an annotation such as in Listing 3.12. This construct looks like [DC: dc] Worker w = **new** CWorker( ) In this case the compiler will generate the a statement with the correct deployment component location for the newly created object through a statement **new** CWorker(null, dc, ...).

7.1.2 Asynchronous Communication

A very important focus of the code generation is translating the "!" operator. In ABS its syntax is $AsyncCall ::= PureExp\ !\ SimpleIdentifier(PureExp, PureExp)$ and the semantics is to create a new task in the COG that contains the target object identified by $PureExp$. This means that the caller task proceeds independently and in parallel with the callee task, without waiting for the result. The result of evaluating an asynchronous method call expression (Listing 7.3) is a future of type (**Fut**<T>), where T is the return type of the callee method m.

Listing 7.3: Asynchronous method call in ABS

```
1 Fut<Int> f = o!m(e);
```

This type of invocation needs to be wrapped as a lambda expression and sent as a message. The problem here is that lambda expressions only take read-only parameters inside the functions, so all any variables passed as function arguments need to be redeclared before creating the lambda expression. In Java this restriction is defined in terms of the final predicate.

Listing 7.4: Asynchronous method call code generations

```
1 final Integer e1 = e;
2 msg: Callable[ABSFuture[Integer]] = ( )=>o.m(e1);
3 f:ABSFuture[Integer] = o.send(msg);
```

To have the correct code generation of **Fut**<Int> f = o!m(e), where e is a variable of type `Int`, into the code in Listing 7.4 we need to take the following steps:

1. Redeclare all variables that need to be passed to the lambda expression.
2. Create the lambda expression as either a Runnable or Callable.
3. Call the `send` method of the actor to which the asynchronous message is being sent and pass the lambda expression as the argument.

Duplicate Variable Names and Renaming When pre-processing continuations the starting point can be within any scope depth of the method, thus becoming the outer scope of the method wrapping the continuation. The problem that arises here is that the variables declared in this inner scope in the initial code are considered "dead" once the scope ends, therefore their names may be reused. The same variables however are now in the most outer scope of the method that represents the continuation, resulting in duplicate names and variable re-declaration once the names are reused. Also when sweeping the parameters, it is possible that a parameter that is passed to a continuation signature may be re-declared with the same name inside the continuation (i.e. in the case of preprocessing a while block. With all these considerations we need a pattern to rename variables to avoid these name collisions.

7.2 Translation of ABS to Scala

This section covers the biggest challenges that were encountered translating the ABS Standard Library when developing the compiler with the Scala backend support. The standard library is actually a good benchmark for the functional paradigm of ABS as many of the predefined data types are ADTs and many of the defined functions use case expressions and pattern matching. Another topic covered in this section the efficient translation of built-in data types that are dependent on the backend in which they are translated. The challenge for the Scala backend is to find as many corresponding types in Scala that have they same semantic behavior that is specified in the ABS manual. The same goes for built-in functions where ABS imposes backend specific semantics for which it is also better to identify an existing scala function (either in the standard library or otherwise), rather than defining a specific one for the backend.

7.2.1 Sweeping Parameters

In Scala the parameters passed to a method are immutable (val) and because a continuation may change swept parameters we create new local variables to save each parameter. To identify a temporary state from which a message needs to resume, we have to *SWEEP*all of the variables that live in that scope. There are two parts to the sweeping process:

1. During pre-processing, we have to sweep all the variables for their types that live in that scope in order to create the correct method signature.
2. During code generation, we have to sweep for all variables as the pairs of type:value that live in the scope such that we can make the correct method call to be execute when the message is released.

The sweeping process is presented in Figure 7.1. We consider that when *SWEEP* is inside a method definition, we assume it returns tuples of type: value, but when it is inside a method call we assume that it only returns value.

$$
\begin{aligned}
SWEEP(T\ m(\overline{p})\{S\}) &::= \overline{p} \cup SWEEP(\underline{S}) \\
SWEEP(\underline{S_1}; S_2) &::= SWEEP(\underline{S_1}) \\
SWEEP(S_1; \underline{S_2}) &::= SWEEP(\underline{S_2}) \\
SWEEP(case\ e1\ \{P_1 => \{\underline{S_1}\}\}) &::= PSP(P_1),\ await\ or\ get\ statement\ in\ S_1 \\
SWEEP(T\ v_1[= exp]) &::= (type : T,\ value : v_1) \\
SWEEP(\_) &::= \{\emptyset\} \\
PSP(Literal) &::= \{\emptyset\} \\
PSP(QU) &::= \{\emptyset\} \\
PSP(Var) &::= (type : \_inferred,\ value : var) \\
PSP(QU(\overline{p})) &::= \{PSP(a) | a \in \overline{p}\} \\
PSP(\_) &::= \{\emptyset\}
\end{aligned}
$$

Figure 7.1: Parameter Sweeping for a continuation triggered when encountering a given **await**

7.2.2 Compiling Algebraic Data Types and Pattern Matching

These particular features of ABS are the main reason behind using Scala as the backend for ABS rather than Java. Scala provides in the core language constructs the means for translation support to the functional programming paradigm described in Section 3.2. For the translation of an ADT declaration, the example of the `Maybe` data type is very useful **data** Maybe<A> = Nothing | Just(A fromJust);. This examples covers both types of ADT declarations: simple data types and data constructor types. This declaration also incorporates a generic parameter that needs to be correctly translated. The snippet for the compiled Scala code is given in Listing 7.5. The defined type itself (`Maybe`) must be an abstract class such that it cannot be instantiated and only serve as a reference type.

Listing 7.5: Translation of the Maybe ADT

```
1 abstract class Maybe[A<%Ordered[_ >: A]] extends Ordered[Maybe[A]] {
2   var rank : Int;
3 }
```

Like ABS objects, ADTs also have to be comparable using a simple rule that orders the defined types based on the order in which they appear in the declaration. An important observation to make here is that ADTs do not have a superclass and unlike objects ABS does not force different ADTs to be comparable. However, a `rank` (line 2) field needs to be defined to allow comparison between the types defined for an ADTs based on their declaration order. A big advantage of using Scala is that the compiler forces the instantiation of this field in all subclasses and as such can yield an error if anything went wrong in the code generation of the declared ADT types. Coming back to this example the types are in the following relation: $Nothing < Just(...)$. If two ADTs are of the same simple type, they are obviously equal, while two ADTs that are of the same constructor type are compared recursively based on their constructor arguments in a depth-first manner. With these aspects in mind, Listing 7.6 gives an overview of the translation of the two types that the `Maybe` ADT has.

The two types are declared as sub case classes of the `Maybe` class such that they can be referenced correctly. The benefit of using case classes is that they provide a default equals method and as such the pattern matching feature of ABS can have a correspondent to the `match` construct in Scala. On lines 2 and 15 the `rank` variable is initialized with respect to the order of the declarations in the ABS code. In the case of the data constructor type `Just` the `compare` method has to compare the arguments for a proper ordering of two `Just` types (line 20).

7.2.3 Translating the Built-in Types of ABS

In the ABS standard library there are several data types and functions that only have a particular semantics attached to them, without any definition or specification. The general purpose of the Scala backend is to provide an efficient execution platform for ABS, and as such these built-in types must not be redefined in Scala with the expected semantics, but rather already have an existing correspondent type. These are the built-in data types and they are defined in Table 7.1 taken from the ABS Manual. This table is updated with the corresponding type in Scala for a straightforward and efficient translation.

Rational Numbers There are two exceptions to these Scala corresponding types. The first one is for futures which have a much more extensive behavior and semantics in ABS and for this purpose the corresponding type is the `Future` class that is part of the ABS runtime in Java. The second problematic type is the `Rational` type which does not have any pre-defined class in Scala and requires definition of the type in the ABS runtime. A layout of this class is presented in Listing 7.7. The first challenge with this class is that although a rational number may be small, it may be represented by a numerator and denominator that exceed the size `Int` type in Scala. Added to this is that any mathematical operation

Listing 7.6: Translation of the Nothing and Just Types

```
1  case class Nothing[A<%Ordered[_ >: A]]( ) extends Maybe[A] {
2   final var rank : Int = 0;
3
4   def compare( that : Maybe[A]): Int= {
5    if(this.rank == that.rank) {
6     return 0;
7    }
8    else {
9     return this.rank-that.rank;
10   }
11  }
12 }
13
14 case class Just[A<%Ordered[_ >: A]](var fromJust : A) extends Maybe[A] {
15  final var rank : Int = 1;
16
17  def compare( that : Maybe[A]): Int= {
18   if(this.rank == that.rank) {
19    var jthat:Just[A] = that.asInstanceOf[Just[A]]
20    return fromJust.compare(jthat.fromJust)
21   }
22   else {
23    return this.rank-that.rank;
24   }
25  }
26 }
```

Table 7.1: ABS Built-in data types

Name	Description	Example	Corresponding Type in Scala
Unit	The empty (void) type	Unit	Unit
Bool	Boolean values	True, False	Boolean
Int	Integers of arbitrary size	0, -15	Int
Rat	Rational numbers	1/5, 22/58775	-
String	Strings	"Hello World"	String
Fut <A>	Futures	the return type of an asynchronous call	Future< A >
Float	Floating point numbers	3.14, -100.5	Float

may also cause the resulting representation to overflow before it is simplified. For these reasons, both the numerator and denominator are defined as `BigInt` (lines 5 and 6) to preserve correctness of the compiled model. However this adds a significant performance penalty to programs using a large number of rationals, and it is strongly recommended whenever possible to use the newly introduced ABS built-in type `Float` which has a corresponding primitive in Scala with the same name. This is especially important for models where high performance is required.

Listing 7.7: The Rational Class in the ABS Runtime

```
1  class Rational(n: BigInt, d: BigInt) extends Ordered[Rational] {
2    require( d != 0 )
3
4    private val g = Rational.gcd(n.abs, d.abs)
5    val numer: BigInt = n/g * d.signum
6    val denom: BigInt = d.abs/g
7
8    def this(n: Int) = this(n, 1)
9    def this(rational: Rational)= this(rational.numer,rational.denom);
10
11   override def toString = ""+ numer + (if (denom == 1) "" else ("/"+denom))
12
13   // default methods (for all math operations)
14   def +(that: Rational): Rational =
15     new Rational( number * that.denom + that.number * denom,
16         denom * that.denom )
17   def +(that: Int): Rational =
18     new Rational( numer + (that * denom), denom )
19
20   ...
21 }
```

Another challenge specific to the Rational type is that, because it does not have a pre-defined type in Scala, it cannot be used together with other numerical types like the semantics of ABS require. As such the Java ABS runtime has to define this behaviour for mathematical operations (line 18) and comparison operations as shown in Listing 7.8. In these examples the methods are defined for operations between rationals and integers and the same behaviour is defined for operations between rationals and floats.

Finally there will be certain instances where an integer will have to be "promoted" to a rational such as on an assignment statement Rational x = 0;. The Rational object in Listing 7.9 defines in Scala

Listing 7.8: The Comparison Methods in the Rational Class

```
1  def equals(that: Rational) = {
2    (this.numer==0 && that.numer==0)||
3    (this.numer == that.numer && this.denom == that.denom)
4  }
5
6  def equals(that: Int) = {
7    (this.numer == that && this.denom == 1) || (this.numer == 0 && that == 0)
8  }
9
10 override def equals(obj: scala.Any): Boolean = {
11   if(obj.isInstanceOf[Int]){
12     return equals(obj.asInstanceOf[Int])
13   }
14   if(obj.isInstanceOf[Rational]){
15     return equals(obj.asInstanceOf[Rational])
16   }
17   return super.equals(obj)
18 }
19
20 override def compare(that: Rational) = {
21   val c = this.numer * that.denom - that.numer * this.denom
22   if(c.isValidInt)
23     c.toInt
24   else if(c<0)
25     Int.MinValue
26   else
27     Int.MaxValue
28 }
```

this implicit conversion (line 2), together with a constructor that creates a rational from an integer parameter(line 6). Finally in order to reduce a rational number to its simplest form and also to avoid storing too many `BigInt` numbers the Rational object provides a function for calculating the greatest common divisor (gcd on line 3) to be applied straight after constructing the number. Another performance issue that appears when using Rational numbers is that they are immutable, but they are not Scala primitives so a large number of these objects are created when a lot of computation is performed with them. As they are no pools of constants defined like String and Integer pools it is best to avoid using this built-in type in favor of Float to model high-performance applications.

Listing 7.9: The Rational Object in the ABS Runtime in Java

```
1 object Rational {
2   implicit def intToRational(x: Int) = new Rational(x)
3   private def gcd(a: BigInt, b: BigInt) : BigInt = if (b == 0) a else gcd(b, a % b)
4
5   def apply(numer: Int, denom: Int) = new Rational(numer, denom)
6   def apply(numer: Int) = new Rational(numer)
7 }
```

7.3 Compiler Correctness

Implementing cooperative scheduling in ABS that scales in the number of active objects and messages provides a major challenge (as described in Chapter 4). Furthermore, it also complicates considerably reasoning about the correctness of ABS programs. This is evidenced by that no sound and complete proof system for ABS has been introduced yet. The proof system in [DO14] for example is shown to be complete only for the sublanguage that does not support cooperative scheduling. ABS itself has a formal semantics, but the languages to which it is compiled, Java and Scala, do not have a full language formal semantics. Therefore it is difficult to provide a formal correctness proof for the whole compiler and all the challenges described in this chapter. As such, the focus is on investigating correctness of certain aspects of the compiler, mainly the translation of the expressive power of cooperative scheduling in ABS. This section shows that the powerful abstraction of cooperative scheduling in ABS can be modeled by a run to completion model of active objects.

The essence of this compilation process is the spawning mechanism the translation is generalized to a run to completion model described by a language GAC (Guarded ACtor) which provides a guarded-command language ([Dij78]) for the description of the method bodies. The formal translation of ABS into GAC is given by an intermediate language ABS-SPAWN which uses an explicit spawn operation to model the execution of await statements. In the GAC language the operation of spawning local processes can be modeled directly by asynchronous self-calls. Any source code written in ABS consists of a set of classes, and each class consists of a set of method definitions. In this section we abstract from the nominal type system of ABS and its functional layer, and focus on the control flow of ABS programs. Figure 7.2 presents the formal syntax of ABS statements which are used to describe the method bodies.

The expression *e* denotes a local side-effect free expression (that is, its evaluation only depends on the local state of the actor and does not affect this local state). For the purpose of providing a smooth and clear proof, we can abstract from its syntax (which in general involves the functional layer of ABS). For notational convenience we assume that every method call (asynchronous or synchronous self call) returns a value. We assume these values typed according to the type system of ABS. A slight modification of Listing 3.8 is shown is Listing 7.10 where it is imposed in the formal syntax that returned expressions are side-effect free. As such, the return instructions is separated into the instructions on lines 10 and !11.

Further we restrict a guard g of an **await** statement by either a local side-effect free Boolean condition *b* or a single a future variable. It is not difficult to see that this restriction does not restrict the expressive

S	::=	ϵ	empty statement
	|	$x = e$	basic assignment
	|	$y = x!m(\bar{e})$	asynchronous method call
	|	$y = m(\bar{e})$	synchronous method call
	|	$x =$ **new** $C(\bar{e})$	object creation
	|	**await** g	await statement
	|	$x = y$.**get**	get statement
	|	**if** b $\{S\}$ **else** $\{S\}$	conditional statement
	|	$S; S$	sequential composition
	|	**return** e	return statement

Figure 7.2: Syntax for ABS statements.

Listing 7.10: Synchronous Call in ABS

```
1  Worker getWorker( ){
2    await !(emptySet(workers));
3    Worker w = take(workers);
4  }
5
6  Result sendWork( ) {
7    Worker w = this.getWorker( );
8    workers = remove(workers, w);
9    Fut<Result> f = w ! doWork( );
10   Result r = f.get;
11   return r;
12 }
```

power since any **await** statement on a guard which consists of a Boolean condition and a set $\{x_1, \ldots, x_n\}$ of futures can be implemented by a sequential composition:

```
1  await x1?; ...; await xn?; await b;
```

because a future is single-write shared data (note that the **await** on the Boolean condition should indeed be executed last).

For technical convenience, we also abstract from the so-called Concurrent Object Groups (COG) as provided by the ABS language. However, it is not difficult to generalize the main result to the language including COG's. More importantly, it should be noted that the syntax does not include the usual **while** statement. A detailed discussion of the challenges of translating **await** statements occurring in the body of a **while** statement will be presented in section 7.3.2. Note however that the **while** statement can be modeled by tail recursion, using synchronous self calls.

To allow for this tail-recursion we assume for the ABS language to feature synchronous self calls of the form $m(\bar{e})$.

7.3.1 ABS Operational Semantics

This section presents a different approach to the semantics of the ABS language using variable renaming of local variables instead of local environments. This allows for a simple definition of a process as the statement to be executed, which in turn allows for a transparent way of modeling cooperative scheduling.

We assume given an ABS program P where object configurations are of the form (σ, S, Q):

- σ assigns values to the instance variables (fields) of the class (we treat the keyword **this** as a distinguished instance variable identifying the object) and all the fresh variables generated for the local variables of the different method invocations. For any side-effect free expression e (including Boolean conditions b) we denote by $\sigma(e)$ the value of e in σ.
- S represents the current statement of the active process that is run by the actor denoted by $\sigma(\mathbf{this})$.
- Q is a (multi-)set of statements which represent suspended processes.

We define a global configuration G as a pair (F, O) where F is a partial function which assigns to each future identity f in its domain a value $F(f)$ and O is a set of configurations (as defined above). By $F(f) = \perp$ we denote that the future f has not been completed yet. For handling the completion of futures by return statements, we introduce an implicit formal parameter **dest** which holds the value returned by the method invocation. In the rules below we assume some mechanism for generating fresh variables. Generating fresh variables is needed to distinguish between the **dest** variable of each method and also to avoid name clashes when renaming local variables. To make the use of the **dest** variable explicit, we replace every **return** statement in a method body with the auxiliary statement **return** *e to* **dest**.

The following rule describes the operation semantics of an assignment.

Assignment Rule

$$(F, \{(\sigma, x = e; S, Q)\} \cup O) \rightarrow (F, \{(\sigma[x = \sigma(e)], S, Q)\} \cup O)$$

Here and in the sequel we denote by $\sigma[x = v]$ the update of σ which assigns the value v to the variable x.

Asynchronous Invocation Rule The following rule describes the semantics of an asynchronous method call.

$$(F, \{(\sigma, y = x!m(\bar{e}); S, Q), (\sigma', S', Q')\} \cup O)$$
$$\rightarrow$$
$$(F[f = \perp], \{(\sigma[y = f], S, Q), (\sigma'', S', Q'')\} \cup O)$$

where:

- f is a new future which does not exist in the domain of F (and thus $F[f = \perp]$ denotes the function which results from extending the domain of F by assigning $\perp$ to f)
- $\sigma(x) = \sigma'(\mathbf{this})$.
- S' is the current statement of the active process run by the target object x (the callee in $x!m(\bar{e})$)
- Q'' extends Q' with the body of method m where all the formal parameters (including the distinguished variable **dest**) are replaced by fresh (that is, not in use in (σ', S', Q')) variables.
- σ'' results from assigning the values of the actual parameters $\sigma(\bar{e})$ to the corresponding fresh local variables. Additionally $\sigma''(\mathbf{dest}') = f$ where $\mathbf{dest}'$ is the fresh local variable corresponding to the destiny variable **dest**.

Synchronous Self Call Rule

$$(F, \{(\sigma, x = m(\bar{e}); S, Q)\} \cup O) \rightarrow$$
$$(F[f = \perp], \{(\sigma', S'; x = f.\mathbf{get}; S, Q)\} \cup O)$$

where:

- f is a new future which does not exist in the domain of F (and thus $F[f = \perp]$ denotes the function which results from extending the domain of F by assigning $\perp$ to f).
- f is the future that will hold the result of the asynchronous method invocation m.
- S' is obtained by renaming the local variables in the body of method m (as above, including the variable **dest**) by fresh variables and σ' assigns to these fresh variables the values of the actual parameters $\sigma(\bar{e})$. Additionally $\sigma'(\mathbf{dest'}) = f$ where $\mathbf{dest'}$ is the fresh local variable corresponding to the destiny variable **dest**. This translation of the body replaces the return statement with an auxiliary statement **return** *e to* **dest**. Freshness is defined as a variable not in use by any statement in S' and Q' .

Note that we thus use simple inlining which works because we introduce fresh variables for the formal parameters of methods. We use a future in order to get a uniform semantics for returning a value for both synchronous calls and asynchronous calls. The get operation will always be enabled (because of the assumption that any method body will end with a return statement), but it is used here instead of an await because we want the process to proceed, as otherwise we would have a release point which breaks the call stack.

In the case of a method which is declared void then the syntax would be **return null** *to* **dest**. By means of this convention, every suspended statement is uniquely identified by its destiny variable,so that we can model Q as a set..

Object Instantiation Rule

$$(F, \{(\sigma, x \ = \ \mathbf{new}\ C(\bar{e}); S, Q)\} \cup O) \rightarrow$$

$$(F, \{(\sigma[x = o], S, Q), (\sigma', S', \emptyset)\} \cup O)$$

where:

- o is a fresh object identity (not appearing as value of a variable in the initial configuration).
- S' is the constructor method body.
- $\sigma'(this) = o$ and σ' assigns to the formal parameters of the constructor method the values $\sigma(\bar{e})$.

Note that each object configuration assigns a new object identity to the instance variable $this$. This explains the usage of union in object configurations.

Conditional Statement Rule The conditional statement has the following two rules.

$$(F, \{(\sigma, \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2; S, Q)\} \cup O) \rightarrow$$

$$(F, \{(\sigma, S_1; S, Q)\} \cup O)$$

where $\sigma(b) = true$.

$$(F, \{(\sigma, \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2; S, Q)\} \cup O) \rightarrow$$

$$(F, \{(\sigma, S_2; S, Q)\} \cup O)$$

where $\sigma(b) = false$.

Return Rule

$$(F, \{\sigma, \textbf{return } e \ \ to \ \textbf{dest}', Q)\} \cup O) \rightarrow (F[f = \sigma(e)], \{(\sigma, \epsilon, Q)\} \cup O)$$

where $f = \sigma(\textbf{dest}')$

Get Rule

$$(F, \{(\sigma, x = y.\textbf{get}; S, Q)\} \cup O) \rightarrow (F, \{(\sigma[x = F(\sigma(y))], S, Q)\} \cup O)$$

where $F(\sigma(y)) \neq \perp$.

Await Rule

$$(F, \{(\sigma, \textbf{await } g; S, Q)\} \cup O) \rightarrow (F, \{(\sigma, \epsilon, \{\textbf{await } g; S\} \cup Q)\} \cup O)$$

where ϵ represents the empty statement, denoting that the current executing statement has ended. This rule "blindly" suspends the current statement without evaluating the guard. The evaluation of the guards will be performed in the context of the scheduling rule below.

Scheduling Rule The following rules schedule enabled **await** statements of a Boolean and a future variables respectively. For suspended statements that start with an **await** we have the following two rules.

$$(F, \{(\sigma, \epsilon, \{\textbf{await } b; S\} \cup Q)\} \cup O) \rightarrow (F, \{(\sigma, S, Q)\} \cup O)$$

where $\sigma(b) = true$

$$(F, \{(\sigma, \epsilon, \{\textbf{await } y; S\} \cup Q)\} \cup O) \rightarrow (F, \{(\sigma, S, Q)\} \cup O)$$

where y is a future variable such that $F(\sigma(y)) \neq \perp$.

For any other suspended statement that is in Q, e.g., that resulted from an asynchronous call, we have the following rule:

$$(F, \{(\sigma, \epsilon, \{S\} \cup Q)\} \cup O) \rightarrow (F, \{(\sigma, S, Q)\} \cup O)$$

7.3.2 ABS-SPAWN

In this section we introduce the ABS-SPAWN language which is obtained from the ABS language discussed above by replacing the await statement with a statement **spawn**(g, S), the so-called **spawn** statement, for spawning a new local process that executes the statement S. In ABS-SPAWN method invocations are thus executed in a run-to-completion mode.

For the operational semantics of the ABS-SPAWN language we introduce the run-time syntax construct $(g \rightarrow S)$ that represents a suspended statement (S) that is guarded by an enabling condition (g). We thus make a distinction between the statement that spawns a process and resulting generated suspended process. The guard in $(g \rightarrow S)$ is the enabling condition for scheduling the corresponding statement S for execution. For its semantics we used the same notions for a global configuration and object configuration, as introduced for the semantics of the ABS language, The semantics of the ABS-SPAWN language results from the semantics of ABS by replacing the **Await Rule** with the rule **Spawning Subtasks** and changing the **Scheduling Rule**, as described above.

Spawning Subtasks Spawning a sub-task simply consists of adding a corresponding statement with an enabling condition to the set Q of suspended processes:

$$(F, \{(\sigma, \textbf{spawn}(g, S); S', Q)\} \cup O) \rightarrow (F, \{(\sigma, S', \{(g \rightarrow S)\} \cup Q)\} \cup O)$$

Scheduling Rule The following rules describe the scheduling of an enabled suspended task. The first two rules are for statement suspended by **await**.

$$(F, \{(\sigma, \epsilon, \{(b \rightarrow S)\} \cup Q)\} \cup O) \rightarrow (F, \{(\sigma, S, Q)\} \cup O)$$

where $\sigma(b) = true$.

$$(F, \{(\sigma, \epsilon, \{(y \rightarrow S)\} \cup Q)\} \cup O) \rightarrow (F, \{(\sigma, S, Q)\} \cup O)$$

where y is a future variable and $F(\sigma(y)) \neq \bot$.

The last rule is for statements that are suspended as a result of an asynchronous invocation and is the same as in the ABS operational semantics:

$$(F, \{(\sigma, \epsilon, \{S\} \cup Q)\} \cup O) \rightarrow (F, \{(\sigma, S, Q)\} \cup O)$$

Translating ABS into ABS-SPAWN We next introduce a formal translation from ABS programs into ABS-SPAWN programs. This translation is applied to every class in the ABS program. For each class, every method body is viewed as a sequential composition of the first instruction followed by its (sequential) continuation and translated accordingly.

$$\begin{aligned}
T(\epsilon) &:= \epsilon \\
T(x = e;\ S) &:= x = e; T(S) \\
T(\textbf{await}\ g;\ S) &:= \textbf{spawn}(g,\ T(S)\ ) \\
T(\textbf{if}\ b\ \{S_1\}\ \textbf{else}\ \{S_2\}; S) &:= \textbf{if}\ b\ \{T(S_1;\ S)\}\ \textbf{else}\ \{T(S_2;\ S)\} \\
T(y\ =\ x!m(\bar{e});\ S) &:= y = x!m(\bar{e});\ T(S) \\
T(y\ =\ m(\bar{e});\ S) &:= y\ =\ m(\bar{e});\ T(S) \\
T(x\ =\ \textbf{new}\ C(\bar{e}); S) &:= x = \textbf{new}\ C(\bar{e});\ T(S) \\
T(x = y.\textbf{get};\ S) &:= x = y.\textbf{get};\ T(S) \\
T(\textbf{return}\ e;\ S) &:= \textbf{return}\ e;\ T(S)
\end{aligned}$$

Figure 7.3: Translation of ABS into ABS-SPAWN

The scheme is applied using a bottom-up approach starting at the level of statements using Figure 7.3. The scheme is then lifted to the level of method bodies. Finally a translation of a class simply consists of the translation of its method definitions.

In Figure 7.3 the empty statement is denoted by ϵ (we assume here the syntactical equivalence $S; \epsilon \equiv S$). The translation of an await construct with guard g followed by a (sequential) continuation S results simply in a spawn statement with two parameters: the guard g and the task representing the translation applied to the continuation ($T(S)$). A conditional statement is translated by "absorbing" the sequential continuation that follows into the two branches of the statement. This general pattern also would apply to, for example, the translation of the ABS case statement (or pattern matching statement) where the continuation has to capture for each possible pattern (P_i) both the block to be executed on that pattern branch (S_i) as well as the rest of the control flow that follows the statement (S). The translation thus captures the whole syntactic continuation that follows an await statement as the new task to be spawned. Therefore the translation of the method containing the await statement will terminate directly after having spawned the corresponding subtask, thus emulating an implicit suspension point.

The While Statement. We describe next the problem of translating a repetitive loop or the while statement. Intuitively, to capture the syntactic continuation that follows an await statement occurring in the body of the while statement, the translation could simply “unfold" the loop. However this would result in a recursive translation. Instead, we can model while statements by means of a tail recursive method. Note that such a method should capture in its formal parameters the execution context (that is, all the local variables used in the loop body).

Listing 7.11: While Loop in ABS

```
1  {  List<Fut<Int>> futuresList = Nil;
2
3   //ABS code that fills the futuresList with
4   //futures resulting from asynchronous calls
5
6   this.sum=0;
7   while( !emptyList( futuresList ) ){
8    Fut<Int> f = head( futuresList );
9    await f?;
10   Int x = f.get;
11   this.sum = this.sum + x;
12   futuresList = tail( futuresList );
13  }
14  if( this.sum > 0 ){
15   //do work
16  }
17 }
```

To describe this in more detail, we look at an example in Listing 7.11 that computes the sum of numbers generated by asynchronous calls whose results are captured in a list of futures. In ABS, lists are part of the functional layer and all functions applied on them (`head, tail, emptyList`) are side-effect free. We note that in this particular program the variables `x, f` are local variables declared inside the repetitive loop, `futuresList` is a local variable defined in the method’s body prior to the loop scope and `sum` is a class member variable.

This repetitive loop can naturally be “unfolded" by defining a new method `m` with a formal parameter of type `List< Fut<Int»`, as we observe it is the only local variable declared prior to the loop. This is shown in Listing 7.12. We can see that this way of “unfolding"" the loop works because the state of execution (in this case, the continuously processed list) is passed to the next call as formal parameters. Listing 7.13 then shows the translation of tail-recursive method modeling the while statement. Note that the syntactic continuation of the await statement is captured in this translation by the recursive call.

Listing 7.13: Translation Tail Recursion

```
1  Unit m(List<Fut<Int>> futuresList){
2   if(!emptyList(futuresList)){
3    Fut<Int> f = head(futuresList);
4    spawn( f?, {
5     Int x = f.get;
6     this.sum = this.sum + x;
7     futuresList = tail(futuresList);
8     m(futuresList)
9    } );
10  }
11 }
```

Listing 7.12: Re-written While Loop in ABS using tail recursion

```
1  //new method
2  Unit m(List<Fut<Int>> futuresList){
3   if(!emptyList(futuresList)){
4    Fut<Int> f = head(futuresList);
5    await f?;
6    Int x = f.get;
7    this.sum = this.sum + x;
8    futuresList = tail(futuresList);
9    m(futuresList);
10  }
11 }
12
13 {  //original method scope
14  this.sum=0;
15  m(futuresList);
16  if(this.sum > 0){
17   //do work
18  }
19 }
```

To conclude the presentation of ABS-SPAWN, we apply the translation scheme to the `WorkerPool` class written in ABS in Listing 3.6. The resulting code in ABS-SPAWN is illustrated in Listing 7.14.

Listing 7.14: The Worker Pool Class in ABS-SPAWN

```
1  class WorkerPool(){
2   Set<Worker> workers;
3
4   Result sendWork() {
5    spawn ( !( emptySet(workers) ) , {
6     Worker w = take(workers);
7     workers = remove(workers, w);
8     Fut<Result> f = w ! doWork();
9     spawn (f?, {
10     Result result = f.get;
11     return result;
12    } ) ;
13   } );
14  }
15
16  Unit finished(Worker w) {
17   workers = insertElement(workers, w);
18  }
19 }
```

Correctness of the ABS Translation In order to show the correctness of the above translation of ABS programs into ABS-SPAWN programs, we use G to denote a global ABS configuration as well as

ABS-SPAWN configurations. We introduce the notation:

$$G \rightarrow_{\text{abs}} G'$$

to differentiate between transitions in pure ABS and transitions in ABS-SPAWN which are denoted as:

$$G \rightarrow G'$$

Let $T(G)$, for any global ABS configuration G, denote the result of applying the translation to all the *executing* ABS statements in G and translating any *suspended* **await** statement **await** $g; S$ in G by $g \rightarrow T(S)$. We now can state the following theorem which states the correctness of the translation of **await** statements in ABS, the proof of which proceeds by a straightforward case analysis of the first instruction of an executing statement.

Theorem 1. *For any configuration G of an ABS program we have:*

$$G \rightarrow_{abs} G' \textit{ iff } T(G) \rightarrow T(G')$$

Proof. The proof proceeds by a case analysis of the transition rules. We treat the following main cases. We only need consider those statements that are affected by the translation, because for statements like the *assignment* and *empty* statement, the semantics of ABS coincides with that of ABS-SPAWN. We only consider the main case of translating the **await** statement, because the translation of the conditional statement is correct because of standard programming equivalences.

Figure 7.4: Execution of an Await Statement

$$\begin{array}{ccc} \Sigma[(\sigma, \textbf{await } g;\ S, Q)] & \xrightarrow[\text{ABS}]{} & \Sigma[(\sigma, \epsilon, \{\textbf{await } g;\ S\} \uplus Q)] \\ \Big\downarrow T & & \Big\downarrow T \\ T(\Sigma)[(\sigma, \textbf{spawn}(\ g;\ T(S)), T(Q))] & \xrightarrow[\text{ABS-SPAWN}]{} & T(\Sigma)[(\sigma, \{g \rightarrow T(S)\} \uplus T(Q))] \end{array}$$

The proof is divided into two parts. The first part is presented by the diagram in Figure 7.4 and treats the translation of the **await** statement that appears as an instruction in a context Σ, that is, $\Sigma[(\sigma, S, Q)]$ describes a global configuration (F, O), with $(\sigma, S, Q) \in O$. The upper transition corresponds to the application of the **Await Rule**, the result of which, namely that the process **await** $g;\ S$ is added to the suspended processes Q (of the executing active object), is denoted by the corresponding global configuration $\Sigma[(\sigma, \epsilon, \{\textbf{await } g;\ S\} \uplus Q)]$. The lower transition results from the definition of the translation scheme to global configurations, and a corresponding application of the **Spawning Tasks** rule in ABS-SPAWN.

Figure 7.5: Scheduling a Suspended Statement

$$\begin{array}{ccc} \Sigma[(\sigma, \epsilon, \{\textbf{await } g;\ S\} \uplus Q)] & \xrightarrow[\text{ABS}]{} & \Sigma[(\sigma, S, Q)] \\ \Big\downarrow T & & \Big\downarrow T \\ T(\Sigma)[(\sigma, \epsilon, \{g \rightarrow T(S)\} \uplus T(Q))] & \xrightarrow[\text{ABS-SPAWN}]{} & T(\Sigma)[(\sigma, T(S), T(Q))] \end{array}$$

Conversely, the second part is presented by the diagram in Figure 7.5 and shows the correctness of translating the **await** statement as part of a suspended process which conforms to the semantics of the **Scheduling Rules** in ABS and ABS-SPAWN, respectively.

7.4 The GAC Language

It is worthwhile to note that by the above translation scheme we can actually embed ABS in a language *without* any await statements (that allow for cooperative scheduling) by encoding **spawn**(g, S) itself as an asynchronous self call of the form **this**$!m()$, where $m()$ is an unique method name with defining body $g \to S$.

In Figure 7.6 we introduce so-called guarded command statements (following [Dij78]) as statements for describing the method bodies in ABS.

S	$::=$	ϵ	empty statement
	$\mid$	$x = e$	basic assignment
	$\mid$	$y = x!m(\bar{e})$	asynchronous method call
	$\mid$	$y = m(\bar{e})$	synchronous method call
	$\mid$	$x =$ **new** $C(\bar{e})$	object creation
	$\mid$	$x = y.$**get**	get statement
	$\mid$	$(*)\Box_{i=1}^{n}\ g_i \to \{S_i\}$	guarded command
	$\mid$	**case** $e\ \{e \Rightarrow S\}$	case statement
	$\mid$	$S; S$	sequential composition
	$\mid$	**return** e	return statement

Figure 7.6: ABS guarded command statements.

The semantics of the statement $\Box_{i=1}^{n}\ g_i \to S_i$ consists of a non-deterministic selection of one of the statements S_i for which the associated guard g_i is enabled. It blocks the execution of the active object if none of the guards are enabled. A guard itself in the GAC language consists of a Boolean condition and a set of futures. Such a guard is enabled if the Boolean condition holds and all its futures are completed (that is, for all of them the return value has been produced). Its iterated version (indicated by the asterisk) consists of repeatedly executing the marked guarded choice as long as one of its guards is enabled. It terminates as soon as none of the guards is enabled. Formally, the semantics of the guarded command statements is described by the following rules (the semantics of the other statements are described as in the ABS semantics).

Guarded Choice Rule

$$(F, \{(\sigma, \Box_{i=1}^{n}\ g_i \to \{S_i\}; S, Q)\} \cup O) \to (F, \{(\sigma, S_j; S, Q)\} \cup O)$$

provided g_j is enabled in σ and F.

For its iterated version we have the following two transitions.

Iterated guarded Choice Rule

$$(F, \{(\sigma, * \Box_{i=1}^{n}\ g_i \to \{S_i\}; S, Q)\} \cup O) \to$$

$$(F, \{(\sigma, S_j; * \Box_{i=1}^{n}\ g_i \to \{S_i\}; S, Q)\} \cup O)$$

provided g_j is enabled in σ.

$$(F, \{(\sigma, * \Box_{i=1}^{n}\ g_i \rightarrow \{S_i\}; S, Q)\} \cup O) \rightarrow (F, \{(\sigma, S, Q)\} \cup O)$$

provided none of the g_j is enabled in σ and F.

Further, we have the following scheduling rules.

Scheduling Rules

$$(F, \{(\sigma, \epsilon, \{\Box_{i=1}^{n}\ g_i \rightarrow \{S_i\}; S\} \cup Q)\} \cup O) \rightarrow (F, \{(\sigma, S_j; S, Q)\} \cup O)$$

provided g_j is enabled in σ and F.

$$(F, \{(\sigma, \epsilon, \{* \Box_{i=1}^{n}\ g_i \rightarrow \{S_i\}; S\} \cup Q)\} \cup O) \rightarrow$$

$$(F, \{(\sigma, S_j; * \Box_{i=1}^{n}\ g_i \rightarrow \{S_i\}; S, Q)\} \cup O)$$

provided g_j is enabled in σ and F.

The resulting GAC language thus follows a strict run to completion mode of execution of the methods by active objects, like the Rebeca language)[Sir07]). Differently from the Rebeca language, it features guarded command statements which allow to associate an enabling condition with a suspended process. Note that such a suspension mechanism avoids modeling a suspended process $g \rightarrow S$ by a recursive method definition

$$m()\{\textbf{if}\ g\ \{S\}\ \textbf{else}\ \{\textbf{this}!m()\}\}$$

which involves busy waiting (and which assumes testing a future as a Boolean condition).

As an overall conclusion we illustrate in Figure 7.15 the translation of the `WorkerPool` into GAC. Note the need to wrap the statements of the guarded commands into separate methods. Thus these methods can be called asynchronously and stored as suspended messages into the queue of the `WorkerPool` until their guards are enabled. As such, execution of other enabled statements can continue without blocking the actor.

Listing 7.15: The Worker Pool Class in GAC

```
1  class WorkerPool(){
2    Set<Worker> workers;
3
4    Result sendWork() {
5      ! emptySet(workers) → this ! m1( workers );
6    }
7
8    Result m1(Set<Worker> workers){
9      Worker w = take(workers);
10     workers = remove(workers, w);
11     Fut<Result> f = w ! doWork();
12     f → this ! m2( f );
13   }
14
15   Result m2(Fut<Result> f){
16     Result result = f.get;
17     return result;
18   }
19
20   Unit finished(Worker w) {
21     workers = insertElement(workers, w);
22   }
23 }
```

Chapter 8

Case Studies and Benchmarks

This chapter covers all the use cases of the integration of formal methods into software development using ABS, Java 8 and Scala. The first part covers several typical benchmarks for actor-based programming that validate the usage of ABS together with the Scala Backend compared to the existing state-of-the art Erlang backend that is the community's most stable and supports all of ABS's features. At the same time these benchmarks validate the Java Runtime System and ASCOOP as a standalone library compared to other state of the art Actor libraries like Scala and Akka. In the second part, a case study that uses Map Reduce is presented to underline the effectiveness of the runtime system on a single machine. It is compared to the ProActive backend which has stable distributed support, yet uses the thread-abstraction model. This case study makes of a comparative analysis of both backends, with advantages and disadvantages of each one. The third part of this chapter shows a very useful example of having support for ABS together with resource modeling support for simulating a cache coherence memory model. Again this case study is compared to the Erlang backend in terms of how well the program runs with high memory and CPU loads. The last part of this chapter presents a case study in which ABS together with symbolic time support in order to visualize the progression of a railway model. This case study highlights the importance of discrete time simulation extension of ABS.

8.1 Cooperative Scheduling Benchmarks

This section shows the comparison of having coroutine support available in Java through either thread-abstraction or spawning tasks. The comparison is first made through an example that relies heavily on coroutines, such that we can measure the overhead that programming with coroutines has on a program. The second example is selected from the Savina benchmark for programming with actors [IS14]. All the benchmarks are ran a core i5 machine which supports hyper-threading and 8GB of RAM on a single JVM. In the library repository[1] we provide implementations of several examples in the benchmark suite directly using the library, while in the compiler repository [2] we have several ABS models of these benchmarks.

8.1.1 Coroutine "Heavy" Benchmark

First the library is evaluated in terms of the impact that programming with coroutines has on performance. The first benchmark involves a large number of suspension and release points in an actor's life cycle in order to compare the spawning approach to the thread-abstraction approach when translating from

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

ABS to Java. In Java using threads and context switches heavily limits the application to the number of native threads that can be created. To measure the improvement provided by our Java library features we use a simple example that creates a recursive stack of synchronous calls. A sketch of the ABS model is presented in Listing 8.1.

Listing 8.1: Benchmark Example

```
1  interface Ainterface {
2    Int recursive_m(Int i, Int id);
3  }
4
5  class A( ) implements Ainterface{
6
7    Int result=0;
8
9    Int recursive_m(Int i, Int id){
10     if (i>0){
11       this.recursive_m(i − 1,id);
12     }else{
13       Fut<Int> f = this ! compute( );
14       await f ?;
15     }
16     return 1;
17   }
18
19   Int compute( ){
20     return result + 1; //no significant computation
21   }
22 }
23
24 { // Main block:
25   Int i = 0;
26   Ainterface master = new A ( );
27   List<Fut<Int>> futures = Nil;
28
29   while( i < 500){
30     Fut<Int> f = master ! recursive_m (5, i);
31     futures = Cons( f, futures );
32     i = i + 1 ;
33   }
34   while ( futures != Nil ){
35     Fut<Int> f1 = head(futures);
36     futures = tail(futures);
37     Int r = f1.get;
38   }
39 }
```

The model creates an Actor of type “A" and sends a large number of messages to it to execute a method `recursive_m(5,id)`. This method creates a call chain of size 5 before sending an asynchronous message to itself to execute method compute( ) and awaits on its result. Although simple, this example allows us to benchmark the pure overhead that arises from having a runtime system with coroutine support,

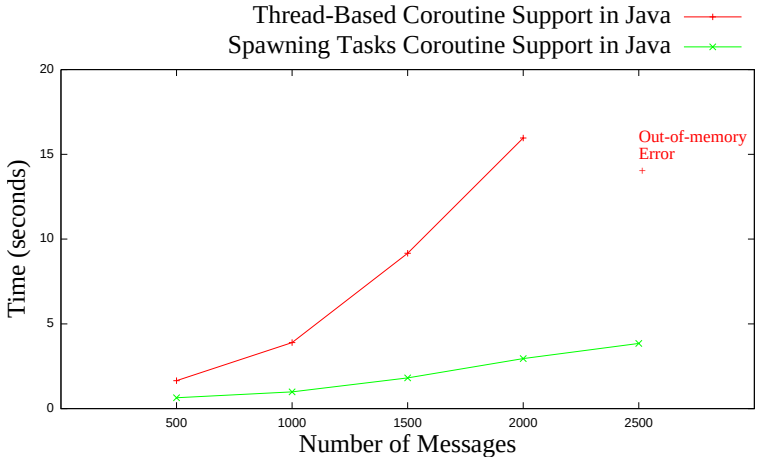

Figure 8.1: Performance figures for Coroutine Overhead

both in a thread-based approach and through spawning of tasks. The results are shown in Figure 8.1. The performance figures presented are for one actor that is running 500-2500 method invocations. It is important to observe that each invocation generates 2 tasks in the actor's queue, so as the number of calls increases, the number of tasks doubles. The figures show that the trade-off for storing continuations and context as tasks into heap memory instead of saving them in native threads removes limitations on the application and significantly reduces overhead.

8.1.2 NQueens Benchmark

From the Savina test suite, we selected the NQueens problem as it is a typical problem with both memory operations and CPU-intensive tasks. Listings 8.2 and 8.3 describe in ABS the problem of arranging N queens on a $N \times N$ chessboard. It provides a master-slave model that illustrates very well the advantage of using actors together with coroutines.

The benchmark divides the task of finding all the valid solutions to the N queens problem into subtasks sent to a fixed number of workers. The `board` is defined as a list of integers where the index of each element represents the line (equivalent to the `depth` of the board) and the number represents the column. Each subtask sent to a worker (line 21 in Listing8.2) requires finding all possible valid solutions of placing the next queen on a `board` filled up to the current `depth`. Once an intermediary solution is found the worker sends an asynchronous call to the `master` (line 12in Listing8.3) to create a new subtask for the new board and the incremented `depth`. The master aggregates the results using a coroutine model (line 24) to await all the solutions starting at depth 0.

We ran the benchmark with a board size varying from 7 to 14 with a fixed number of 4 workers. The results compare the implementations of the NQueens problem and are shown in Figure 8.2. The first two implementations are direct translations in Java from ABS source code with the two co-routine approaches (thread-abstraction and JAAC(spawning)). It is important to observe that as the board size increases, the number of solutions grows from 40 to 14200. The results show that using thread abstraction (where each method invocation generates a corresponding thread) the time taken grows exponentially and cannot complete once the board size reaches 11 while the approach that uses tasks remains unaffected.

The next result (the blue plot) shows the improvement brought by using Java data structures. These results are at a comparable level with the Savina implementation using Akka actors(orange plot). ABS has limited support for data structures (offering only lists, sets and associative lists that can be used as

Listing 8.2: NQueens Master Class Snippet

```
1  class Master (Int numWorkers, Int threshold, Int boardSize, ...) implements IMaster {
2
3   List<IWorker> workers = Nil;
4
5   //... constructor and initializations
6   {
7    Int i = 0;
8    while (i <= numWorkers) {
9    IWorker w = new Worker(this,threshold,size);
10   workers = Cons(w,workers);
11   i = i+1;
12  }
13
14
15  this!sendWork(Nil, 0, ...); // triggers computation
16  }
17
18  //method for receiving solutions
19  Unit sendWork(List<Int> board, Int depth, ...){
20   Fut<Unit> f = nth(workers,messageCounter)!
21   nqueensKernelPar(board,depth,priorities);
22   messageCounter = (messageCounter + 1) % numWorkers;
23   if(depth==0){
24    await f? ;
25    //handling program completion
26   }
27  }
28 }
```

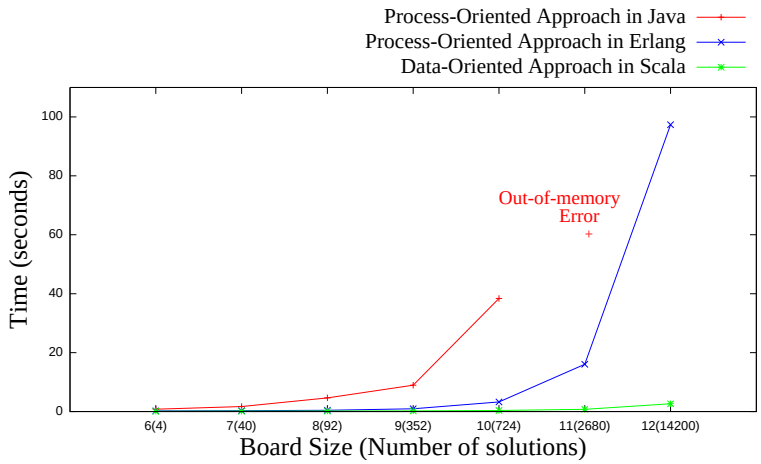

Figure 8.2: Results for the N-Queens problem using two different coroutine approaches

Listing 8.3: NQueens Worker Class Snippet

```
1  class Worker(IMaster master, Int threshold, Int size) implements IWorker {
2
3   Unit nqueensKernelPar(List<Int> board, Int depth, ...) {
4    Int i = 0;
5    if (size != depth) {
6     if (depth >= threshold) {
7      //handle the rest of the solution sequentially and send it to the master
8     } else {
9      while (i < size) {
10      List<Int> newboard = appendright(board,i);
11      if (boardValid(0, newboard,depth+1)) {
12       master!sendWork(newboard,depth+1, ...);
13      }
14      i = i+1;
15     }
16    }
17   }
18   else { //send a solution to the master
19   }
20  }
21 }
```

maps) and by changing the `board` from an ABS list to a Java Array we obtain a significant improvement. This enforces the need of a foreign language interface for ABS to be used a full-fledged programming language.

8.2 Benchmarking the ASCOOP Library

In this section, we evaluate the ASCOOP library in terms of the overhead of creating actors, asynchronous message passing between them, and suspended sub-tasks in actors. To this end, we use some of the examples in the Savina benchmark for programming with actors [IS14], each focusing on one aspect. We compare the execution time of our ASCOOP implementation with those available in the Savina Suite for standard Scala actors and Akka actors. All benchmarks are run on a core i5 machine with 2 CPU that support hyperthreading and 8GB memory.

We use the Ping Pong benchmark for exchanging a large number of messages between two actors. This tests how well each of the libraries handles message passing to the correct actor and how quickly an actor receives a message and processes it. Then we use the Fibonacci benchmark to see how well the library behaves with an exponentially growing number of actors. Finally, we take two typical benchmarks for CPU and memory operations: the Sieve of Eratosthenes and the NQueens problem.

For each of the benchmarks we offer two implementations using ASCOOP Actors. The first one is a typical actor implementation similar to the other libraries with a fire-and-forget approach. This approach, additionally, requires some sort of termination protocol and a prior known condition of termination to ensure correct completion of the application. In the second approach we use the cooperative scheduling feature of ASCOOP actors. This reduces the need for extra message passing to explicitly return the result of a computation; instead, the sender awaits the availability of a future and gets the result from there. Similarly, the termination of the program can be determined by completion of the future corresponding

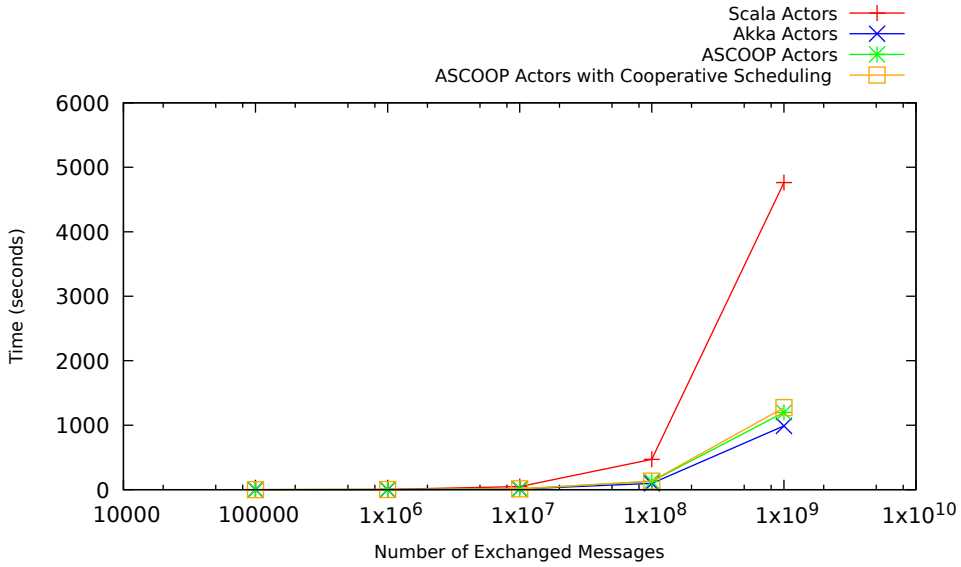

Figure 8.3: Performance figures Ping Pong Benchmark

to the task that starts the whole benchmark, but this can be tweaked to a different task or actor state depending on the benchmark.

8.2.1 Message Passing Overhead

The first benchmark we look at is the Ping Pong benchmark where two actors exchange a fixed number of empty messages. These messages do not perform any computations so this benchmark strictly tests only the overhead of communication between actors. The results are presented in Figure 8.3.

Listing 8.4: Pong Actor code in ASCOOP

```
1  class PongActor extends TypedActor {
2
3    var pongCount = 0
4
5    def ping(sender: PingActor): Future[Void] = messageHandler {
6      sender.pong
7      pongCount += 1
8      done
9    }
10
11   def stop: Future[Void] = messageHandler {
12     println("Pong: pongs = " + pongCount)
13     done
14   }
15 }
```

From the results we see that up to 10^9 messages the overhead of both ASCOOP implementations is comparable to that of Akka and significantly better than Scala Actors. What we want to stress is that

even for a simple program like this, the size of the code is reduced from 150 to 110 lines of code (loc). For example we compare the implementation of the Pong Actor with Akka and ASCOOP Actors in Listings 8.4 and 8.5. We can observe that explicit message types and pattern matching (on line 7 of Listing 8.5) are not longer required, as they are replaced by method definitions for each type of message. This also gives the code more readability and modularity as an actor no longer has just a single method for processing all messages. Also in Akka each type of message needs to be defined separately in PingPongConfig and this is a requirement that is specific to each benchmark.

Listing 8.5: Pong Actor code in Akka

```
1  class PongActor extends AkkaActor[Message] {
2
3    private var pongCount: Int = 0
4
5    override def process(msg: PingPongConfig.Message) {
6      msg match {
7        case message: PingPongConfig.SendPingMessage =>
8          val sender = message.sender.asInstanceOf[ActorRef]
9          sender ! new PingPongConfig.SendPongMessage(self)
10         pongCount = pongCount + 1
11       case _: PingPongConfig.StopMessage =>
12         exit( )
13       case message =>
14         val ex = new IllegalArgumentException("Unsupported message: " + message)
15         ex.printStackTrace(System.err)
16 } } }
```

The implementation with cooperative scheduling also uses the code sequence presented previously in Listing 6.11 to ensure correct termination of the program. In Listings 8.6 and 8.7 we compare the termination of the benchmark in ASCOOP and Akka. This is part of the Ping Actor implementation.

Listing 8.6: Ping Actor code in ASCOOP

```
1  class PingActor(pongActor: PongActor) extends PingInterface {
2
3    var pingsLeft = 0
4    var t1 = 0L
5
6    override def start(iterations: Int) = messageHandler {
7      t1 = System.currentTimeMillis
8      pongActor.ping(this)
9      pingsLeft = iterations - 1
10
11     on (pingsLeft == 0) execute {
12       val t2 = System.currentTimeMillis
13       pongActor.stop
14       println(s"Finished in" +{t2-t1}+ " milliseconds")
15       ActorSystem.shutdown( )
16       done
17     }
18     done
19 } }
```

Listing 8.7: Ping Actor code in Akka

```
1  private class PingActor(count: Int, pong: ActorRef) extends AkkaActor[Message] {
2
3   private var pingsLeft: Int = count
4
5   override def process(msg: PingPongConfig.Message) {
6    msg match {
7     //...other case branches for prcessing messages
8     case _: PingPongConfig.SendPongMessage =>
9      if (pingsLeft > 0) {
10      self ! PingMessage.ONLY
11     } else {
12      pong ! StopMessage.ONLY
13      exit( )
14     }
15   }
16 }
```

The termination that uses ASCOOP Actors delegates the verification of the number of pings left to issue to the underlying scheduler (see Section 4.2.2) instead of verifying this state upon every execution of a ping as Akka Actors implementation shows(line 9 in Listing 8.7). As such the termination in ASCOOP is based on a particular actor state that the user can specify(line 11 of Listing 8.6) and needs not to be checked upon every ping.

8.2.2 Actors Overhead

The second benchmark calculates the nth number of the Fibonacci sequence. The sequence is defined with two starting numbers with the value of 1 and each number that follows is the sum of the two preceding ones. The implementation in the Savina benchmark calculates number n by creating two actors to calculate numbers $n-1$ and $n-2$ and then adding them together. These two actors in turn will each create two more actors (for calculating the sums of $n-2$ *and* $n-3$ and $n-3$ *and* $n-4$ respectively) and so on. This sequence is repeated until $n=1$ or $n=2$ which both have value 1. This benchmark is very suitable for testing the performance of a program with a large number of actors as it will create 2^n actors. We observe the results in Figure 8.4, which show the overhead for up to 2^33 actors.

The implementation with cooperative scheduling for this benchmark is shown in Listing 8.8. The completion of the Fibonacci number (starting on line 10) is scheduled to run only when the two previous numbers have been computed by futures ff1 and ff2. This example best illustrates how much code can be reduced using cooperative scheduling as a normal fire-and-forget implementation would have required two separate response messages before computing the number.

8.2.3 CPU and Memory benchmarks

The next two benchmarks are typical benchmarks for programming with actors that perform a lot of CPU and memory operations. The Sieve of Eratosthenes [O'N09] calculates prime numbers from the first n candidates [Tip13, Bok87, pri]. The implementation first creates two actors:one that generates candidates starting from 3 and skipping even numbers. The candidates are passed to a second actor that determines if they are prime based on the prime numbers it has already identified by checking if they are divisors of the current candidate. If a number has no divisors it is stored in an array inside the actor. To increase parallelism, each actor stores a fixed number of primes, which once exceeded will create a new actor

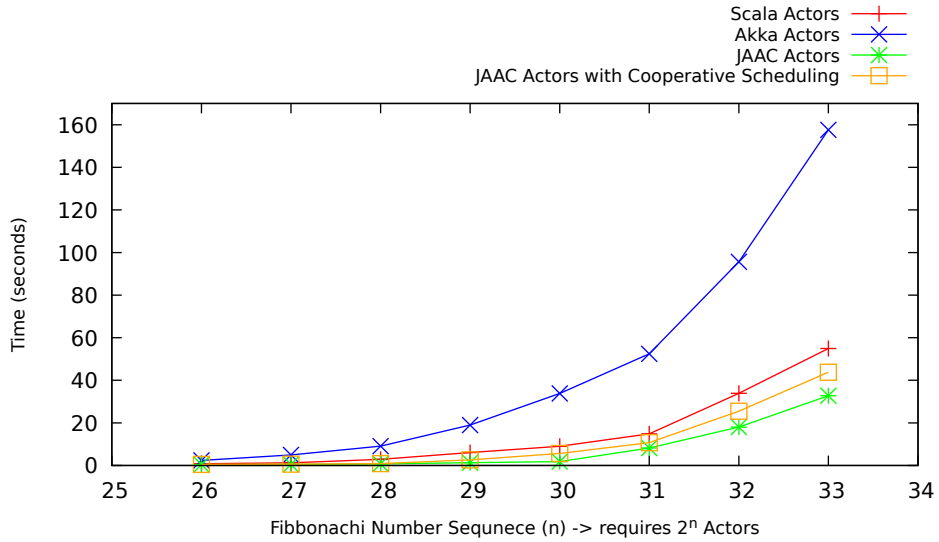

Figure 8.4: Performance figures Fibonacci Benchmark

Listing 8.8: Fibonacci Actor code in ASCOOP with Cooperative Scheduling

```
1  class FibActorAwaiting extends TypedActor{
2    def request(n: Int): Future[Int] = messageHandler {
3      if (n <= 2) {
4        Future.done(1)
5      }
6      else {
7        val ff1 = (new FibActorAwaiting).request(n - 1)
8        val ff2 = (new FibActorAwaiting).request(n - 2)
9        List(ff1, ff2) onSuccessAll {
10         ns =>
11         Future.done(ns.head + ns.tail.head)
12       }
13     }
14   }
15 }
```

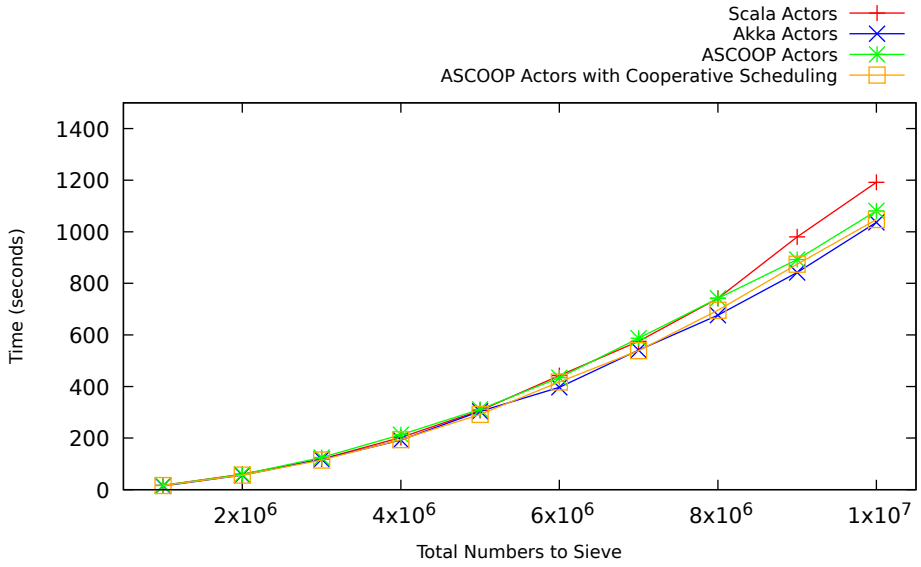

Figure 8.5: Performance figures Eratosthenes Sieve Benchmark

with the same behaviour. Once all candidates up to n have been processed, the total number of primes is aggregated from each array. Am important observation is that this is an actor-based implementation and comparison of the case study and it is not meant to be the fastest implementation of the sieve, as there exist many algorithms of segmenting the data to obtain better parallelism and scaling [yS05].

The results of this benchmark are shown in Figure 8.5, with each actor storing a fixed number of 2000 primes before an new actor is created. In this benchmark the number of messages and actors is significantly lower and we can see that there is no particular implementation that is constantly better in terms of performance. However when it comes to writing the actual application, once again we observe that the size of the code compared to the Savina benchmarks in Scala and Akka has been reduced from 160 loc to 110 loc.

The NQueens benchmark is reused because the master-slave model relies heavily on the cooperative scheduling properties. The benchmark divides the task of finding all the valid solutions to the N queens problem to a fixed number of workers that at each step have to find an intermediary solution of placing a queen K on the board before relaying the message back to the master which then assigns the next job of placing queen $K + 1$ to another worker. As the search space becomes smaller, a threshold is imposed where the worker has to sequentially find the complete solution up to N queens before sending a message back to the master.

We ran the benchmark with a board size varying from 11 to 15 with a fixed number of 4 workers. The results are shown in Figure 8.6. It is important to observe that as the board size increases, the number of solutions grows from 2680 - 2,279,184.

These results also have very small differences in performance and it is very important to know that the ASCOOP model for this benchmark is 150 lines of code in total, while the benchmark written directly using the Scala or Akka libraries is around 400 lines of code. Furthermore the ASCOOP implementation with cooperative scheduling does not require that the number of solutions to be found is known a priori, as the master actor can simply wait on the future that generated all of the searches (line 9 of Listing 8.9) and generate a sub-task to terminate the application once it completes (line 10).

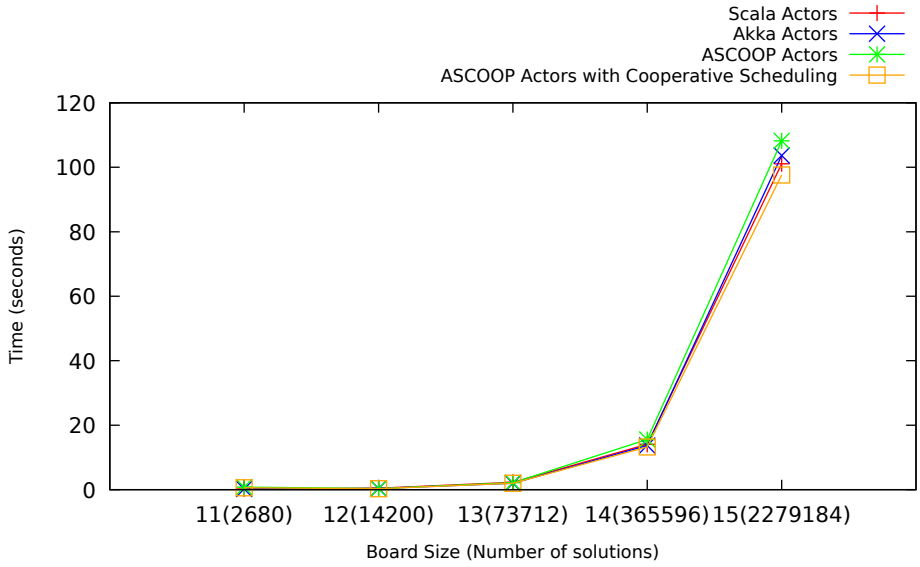

Figure 8.6: Performance figures Nqueens Benchmark

Listing 8.9: NQueens Master Methods for Initialization and Completion

```
1  def sendWork(list: Array[Int], depth: Int, priorities: Int):
2    Future[Iterable[Array[Int]]] = messageHandler {
3   val worker = new Worker(threshold, size)
4   worker.nqueensKernelPar(list, depth, priorities)
5  }
6
7  def init : Future[Void] = messageHandler {
8   val inArray: Array[Int] = new Array[Int](0)
9   val f = this.sendWork(inArray, 0, priorities)
10  f onSuccess(result => {
11    println(s"Found␣{result.size}␣solutions")
12  }
13 }
```

8.3 Map Reduce

The solution provided in this thesis is tailored towards the efficient integration of actors, futures and coroutines on a single-machine. As we have seen in Chapter 5, there are significant challenges to maintaining this efficiency in a distributed setting. This section covers a comparative analysis between the research and solution proposed in this thesis and the research conducted in [Roc16] where ABS is extended with distributed support.

Through the use of a representative HPC application: pattern matching of a DNA sequence programmed in the MapReduce model [DG08] we compare the performance of the two models deployed on an increasing number of nodes. The case study is computation-intensive and yet does no have a lot of active object communication. On one hand, ProActive relies on physical threads for active objects and

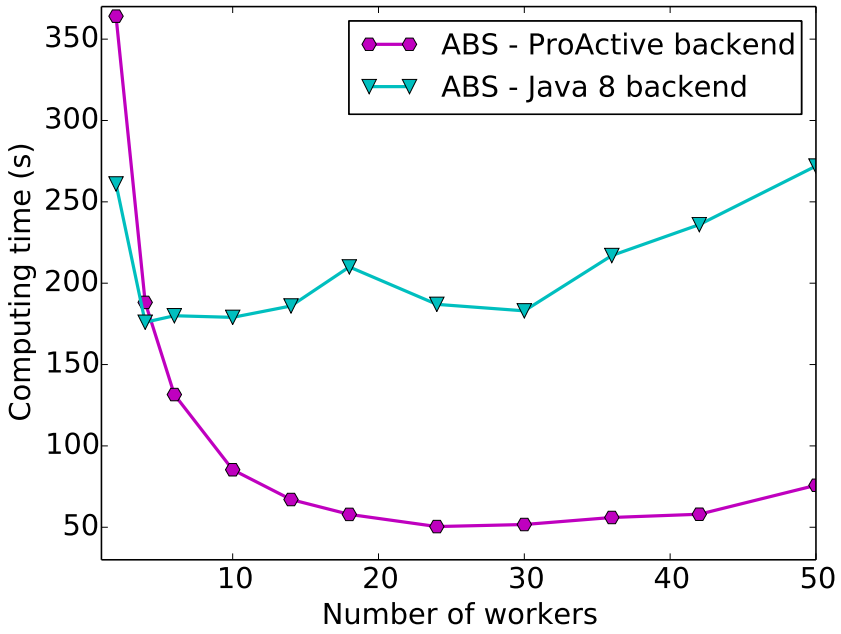

Figure 8.7: Execution time of DNA-matching ABS application

data copying occurs in each communication, thus ProActive is better suited for computation-intensive scenarios than for communication-intensive ones. On the other hand, the JAAC backend is perfectly suited when the number of nodes does not exceed the machine's cores as the premise of the `LocalActor` implementation is to communicate with other actors at no additional cost. We compared performance of the code generated with the ProActive backend for ABS run in distributed mode, the code generated with the Java 8 backend of ABS run on a single machine, and native code written manually in ProActive for the same algorithm.

The application creates `Map` instances (workers) each in their own COG. The search pattern is of 250 bytes inside a database of 5 MB. Each worker queries and processes the maximum matching sequence of a chunk while a reducer aggregates and prints out the global maximum matching sequence. When deploying with ProActive for the best mapping to cores, each machine is instantiated with two workers (as it has two dual-core CPUs). One of the biggest drawbacks of ABS was its standard library data structures, so in both generated programs, we manually replaced the translation of functional ABS types (integers, booleans, lists, maps) with standard Java types to avoid search spaces of very high complexities (i.e a search in an ABS functional hash map is $O(n)$).

The hardware setup for this case study was with machines that have 2 dual core CPUs at 2.6GHz, and 8 GB of RAM[1].

Fig. 8.3 shows the execution times of both ABS backends for the application ranging from 2 to 50 workers, and 1 to 25 physical machines with ProActive. The execution times of the ProActive backend are sharply decreasing for the first few added machines and then decrease at a slower rate. The first instance added in the JAAC backend also improves significantly the performance of the program and the JAAC backend performs better with one or two workers. Thanks to the efficient thread management of the Java

[1]We use a cluster of Grid5000 platform [BCC+06]: `http://grid5000.fr`

8 backend, the performance stays stable until 30 workers. With a high number of instances, the degree of parallelism becomes harmful. In contrast, increasing the degree of parallelism for the ProActive backend results in a linear speedup, because it balances the load between machines and benefits from distribution.

8.4 Cache coherency

This section covers a case study where ABS is used to simulate the memory model of a multi-core architecture. It is a model of the entire multi-core system that includes computing cores, a scheduler, the bus and the different memory levels that include cache levels and the main memory. The simulation models how data is loaded and written into main memory and how it passes through the caches when different operations are executed. For the research conducted in this thesis the operations abstract from the actual data by working only with address requests like loading an address for reading or writing purposes. Operations are grouped together in sets to represent tasks that are to be executed on a core. It is important to understand that the entire system is a model that is independent from the actual system it runs on. All the operations corresponding to loading addresses, looking them up in the cache levels, evicting addresses to create space in a cache or flushing addresses to main memory for maintaining persistent data are all programmed in the ABS source code. As such at the end of executing the model, the user can observe the state of the system such as the data stored in the cache or the number of hits and misses when an address is required.

This case study illustrates a powerful usage of the resource modeling extension of ABS. Each core is modeled as an ABS object that can executed its assigned tasks in parallel to other cores. The simulated core scheduler assigns tasks to the available cores in the system in a round-robin manner. Using the resource provisioning model, core objects can each be assigned a Deployment Component with a finite amount of resources (Listing 8.10), thus ensuring a load-balancing factor between the cores. On lines 6- 10 the model determines whether a new Deployment Component (referenced by `dc`) is instantiated with an infinite (0) or finite amount of resources. Then upon setup of the system's cores (lines 14 or 18), these are assigned the newly created component with finite resources if this was the configured setup.

Listing 8.10: Assigning finite resources to a new core object

```
1  class Config(Maybe<Rat> resources) implements IConfig {
2
3   Unit runConfig(Int nCores_, ...){
4    / * other configuration setups * /
5    while (nCores > 0) {
6     Rat rc = case resources {
7      Nothing => 0;
8      Just(x) => x;
9     };
10    DeploymentComponent dc = new DeploymentComponent(name, map[Pair(Speed,rc)]);
11    / * ... * /
12    case resources {
13     Nothing => {
14      ICore c = new Core(name,s,l1);
15      //core setup Deployment Component with infinite resources
16     }
17     _ => {
18      [DC: dc] ICore c = new Core(name,s,l1);
19      //core setup on a Deployment Component with finite resources
20 } } } } }
```

Even more powerful is the fact the by giving the Deployment Components different speeds (as in line 10 of the previous listing) or the tasks different costs by using annotations (line 4 Listing 8.11), the model can simulate cores that execute faster or tasks than take longer to run (thus the reason for abstracting from actual data as far as execution speed is concerned). A simple algorithm of assigning equal resources to all Deployment Components on which `Core` objects are instantiated and giving all tasks the same Cost will simulate complete parallelism in the system: a core will execute a second task only after all cores have executed the first task. Together with a round-robin scheduler, the model also simulates perfect load-balancing between all the cores.

Listing 8.11: Assigning taks execution a particular cost

```
1  class Core( ... ) implements ICore {
2
3    Unit run( ) {
4     [Cost:1] skip;
5     case currentTask {
6       //execution of current task assigned to the core
7     }
8    }
9  }
```

The Scala backend was compared with the Erlang backend for ABS in this case study. This backend is the most stable and well-maintained backend of ABS and it offers full support for all of ABS features including error handling and custom-defined schedulers, however it does use the thread-abstraction approach for asynchronous communication and cooperative scheduling. In Listing 8.8 the performance times for simulating a system with 2 cores and three levels of cache are presented. In this first simulation (the red and green plots) there is no usage of resource models assigned to ABS `Core` objects and as such these can work at non-deterministic speeds resulting in an uneven number of tasks executed between the cores. In the second simulation (the orange and blue plots) the ABS model uses Deployment Components as explained in Listing 8.10 to obtain a load balance of the tasks executed on the two `core` objects. There are two important observations to withdraw from these results:

1. The tasks spawning approach allows the ABS model to run much faster as the number of tasks simulated in the system grows
2. The simulation of the load balancing and resources creates a small overhead in order to provide a much more realistic version of the memory system.

8.5 German Railway Example

The final case study involves testing and validating support for the time model of ABS in the runtime library. The case study simulates physical behaviour of trains and events occur happen on the railway track and its components such as signals, stations and switches. For a detailed formal description of the model we refer to [HM16, KH16, KH17]. The main focus of using this example in the research is its usage of the time model of ABS. Throughout the source model several events occur after a fixed amount of time units passes and in turn trains have to react to those events in correctly establish capacity and safety properties.

The railway is modeled as an oriented graph where nodes represent fixed points of information flow (PIF) and are connected by edges that represent the tracks between them. In short PIFs are structural elements where trains can receive or send information such as a signal or a track clearance detection device or a gradient change in the track. Both tracks and nodes are represented as ABS objects as in

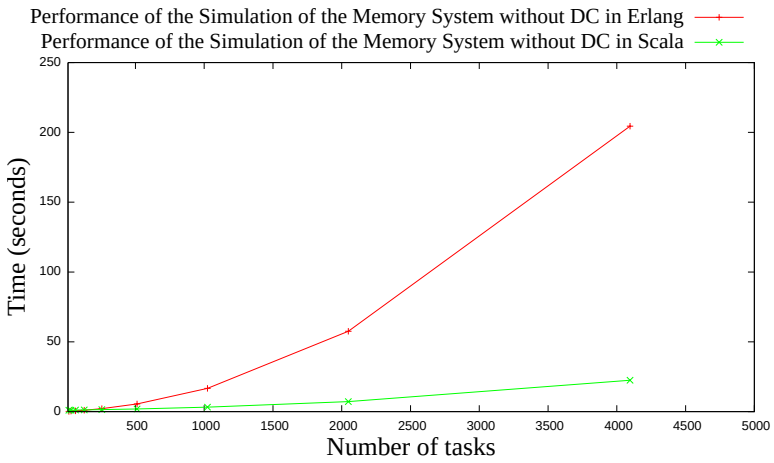

Figure 8.8: Results for the simulation of the memory system

Listing 8.12. A `NodeImpl` object contains a list of connecting **Edges** that make up the graph and represent the tracks on the railway. The `EdgeImpl` class represents tracks with a particular length `l` and the two nodes it connects(`frNode` and `toNode`). Multiple lines between the same two nodes may be represented by edges.

Listing 8.12: Graph and Edge Classes

```
1  class NodeImpl(Int x, Int y, String name) implements Node {
2
3    List<Edge> edges = Nil;
4
5  }
6  class EdgeImpl(..., Node frNode, Node toNode, Int l, String name) implements Edge {
7
8  }
```

Trains are also ABS objects are represented by the their front and back position relative to the most recent node they passed. A train has a series of attributes to describe its behavior such as speed, accelerations state and length as well as fixed attributes such as maximum acceleration and brake retardation. An outline of the class is presented in Listing 8.13. Trains are modeled to drive on simulation events. At every PIF where the train is active, it computes its next event and the time until this event must be processed. This is where the time-model of ABS is used by calling the `advance` method whenever an event is computed to simulate the amount of time that passes.

Listing 8.13: Train Class

```
1  class TrainImpl(App app, String name, Int length) implements Train {
2    //attributes that define a train's state and behaviour
3
4    Unit advance(Rat r){
5      await duration(round(r), round(r));
6    }
7  }
```

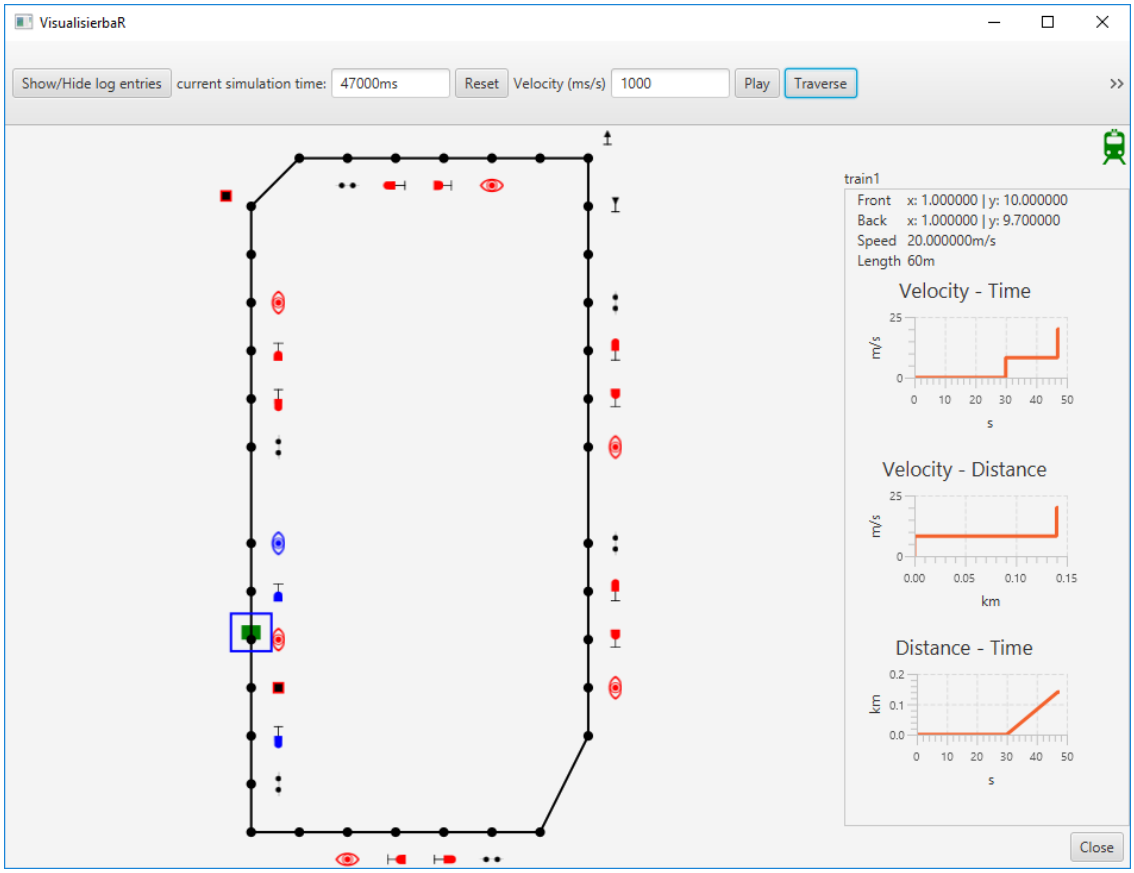

Figure 8.9: The railway model through the visualization tool

The model was run with the ABS Scala backend with time support and a very important result that was obtained is that the output of the model is deterministic and the simulation always yields that same result given the same input. The output can then be processed using a visualization tool developed in [KS19]. Figure 8.9 presents a snapshot of the model where the user can observe and monitor the behaviour of trains by advancing time in the simulation model.

Chapter 9

Conclusions and Future Work

Throughout the research conducted in my thesis I proposed and validated solutions to unify two the domain of formal modeling and verification with the domain of software development and engineering. I discovered that in general researchers and professors with a more theoretical background are willing to adopt a mainstream programming language like Java or Scala into their research with more graduate students having acquired experience in programming during their studies. On the other hand industry is much more reserved to integrate formal methods and newly proposed language models, despite their strong academic backing, mainly due to the agile approach that focuses on getting products out quickly and making improvements on a continuous basis. As such my research slowly turned towards making formal languages, specifically ABS and its features, more approachable to software development and making it easier to learn and use by programmers. The research also focused on integrating ABS's features into the software development cycle. These included features for concurrent and functional programming, as well as features for modeling real-time systems with deadlines and resource provisioning for distributed systems.

This thesis formally describes a translation scheme together with a Java library for efficiently simulating the behaviour of ABS. The coroutine abstraction is a powerful programming technique in a software development context with a very strong research effort both in model proposals and implemented libraries. Having a scalable JVM library with support for coroutine emulation gives a basis for industrial adoption of the ABS language for component-based software engineering. It makes ABS a powerful extension of Java with support for formal verification, resource analysis and deadlock detection. The proposed library was stress-tested with a benchmarks tailored towards cooperative scheduling that show significant improvements of saving continuations including the call stack as data in memory instead of using a process-oriented approach by means native Java threads. With a better approach to the problem of using coroutines in programming comes the need to also compare it with state of the art solutions and therefore the library was tested against benchmarks for CPU-intensive applications to show that this feature does not have a negative impact on performance compared to existing actor libraries like the Akka Actor library.

There are however certain things that theoretical models cannot account for in software development. The Java language is a mainstream language with countless libraries and technologies developed around it that ABS is by no means intending to replace. This was the basis for extending the Java library to ASCOOP as a standalone library that directly offered ABS's functional and concurrent features in Scala. This extended library seamlessly integrates actors with futures and support for cooperative scheduling. The programming to interfaces paradigm enables common advantages of object-oriented programming including static type checking of message passing between actors as well as type hierarchies of actors. Tested with the Savina benchmarks the library is proves to be scalable in the number of actors, number of

messages communicated, and number of suspended sub-tasks. Comparisons with Akka and Scala actors show comparable performance while the simpler programming paradigm and statically typed messages lead to more ease of programming and less chance of runtime errors. The obtained reduction in the code is in part due to cooperative scheduling but also due to ASCOOP not requiring a context setup for actors and explicit build-up of messages and subsequent pattern matching on them.

The research towards unifying software development and formal models will continue in these two main directions covering several new aspects. At the translation level of ABS, the plan is to extend the ABS type-checker to verify the Foreign Language Interface constructs directly in ABS. ABS also requires the development of a debugger to enable profiling and visualization of concurrent programs. In both research directions there is still a need to further develop the research presented in Chapter 5 with respect to implementation of distributed actors. To allow systems of distributed actors, ASCOOP is extending the way completion of futures is propagated into a distributed push-based approach. Additionally, ASCOOP needs a mechanism to allow programmers to specify blocking tasks such that they can be scheduled on a separate executor; otherwise there is a risk that all available threads get blocked and normal CPU-intensive tasks get delayed. ASCOOP will also provide a more functional style of programming with futures as well as more integral error handling mechanisms.

Summary

The development process of any software has become extremely important not just in the IT industry, but in almost every business or domain of research. The effort in making this process quick, efficient, reliable and automated has constantly evolved into a flow that delivers software incrementally based on both the developer's best skills and the end user's feedback.

Software modeling and modeling languages have the purpose of facilitating product development by designing correct and reliable applications. The concurrency model of the Abstract Behavioural Specification (ABS) Language with features for asynchronous programming and cooperative scheduling is an important example of how modeling contributes to the reliability and robustness of a product. By abstracting from the implementation details, program complexity and inner workings of libraries, software modeling, and specifically ABS, allow for an easier use of formal analysis techniques and proofs to support product design. However there is still a gap that exists between modeling languages and programming languages with the process of software development often going on two separate paths with respect to modeling and implementation. This potentially introduces errors and doubles the development effort.

The overall objective of this research and 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. The contributions of this thesis are divided in three parts aimed at achieving this overall objective.

The first part is about developing a runtime that implements the proposed concurrency model of ABS its features with the main focuses being 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.

The second part of this research brings 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.

The third part focuses on 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.

Samenvatting

Het ontwikkelproces van welke software dan ook is heden ten dage uitermate belangrijker dan voorheen, niet alleen in de IT sector maar tevens in het bedrijfsleven en vrijwel alle onderzoeksrichtingen. Met oog op het verbeteren van dit ontwikkelproces wat betreft schakelsnelheid, efficientie, betrouwbaarheid, en automatiseerbaarheid, is er in de afgelopen decennia een werkwijze ontstaan, waarin software stapsgewijs opgeleverd wordt, gebaseerd op de competenties van de ontwikkelaar en terugkoppeling van de eindgebruikers.

Softwaremodelering en modeleringstalen hebben als doel het ontwikkelprocess te vergemakkelijken door middel van ontwerpbeschrijvingen van correcte en betrouwbare softwaretoepassingen. Een invloedrijk voorbeeld van softwaremodelering, wat betreft het verbeteren van de betrouwbaarheid en weerbarstigheid van softwareproducten, is het 'concurrency' model van de Abstract Behavioral Specification (ABS) taal. Deze modeleringstaal heeft als kenmerken de ondersteuning voor asynchrone programma's en coĂ¶peratieve uitvoering van programma's. Door van implementatiedetails, programmacomplexiteit en inhoud van programmabilbiotheken te abstraheren, zijn formele analyse technieken en bijbehorende bewijstechnieken aanzienlijk eenvoudiger toe te passen op software modellen dan op software an sich, en dit geldt specifiek ook voor ABS. Echter blijft er hierdoor afstand tussen de gebruikte software modellen en de daadwerkelijke software: het ontwikkelproces bewandelt twee losse paden, de een en ander overeenkomstig aan softwaremodelering in een modeleringstaal en implementatie in een programmeertaal. Dit leidt tot foutgevoeligheid en dubbel werk.

Het uiteindelijke doel van het onderzoek zoals beschreven in dit proefschrift is het overbruggen van de afstand tussen softwaremodellen en software zoals uitgedrukt in programmeertalen. Dit biedt een soepele integratie van formele methoden en twee van de meest bekende en gebruikte programmeertalen voor softwareontwikkeling: Java en Scala. Het onderzoek richt zich voornamelijk op sequentiele en parallelizeerbare applicaties, maar een apart onderdeel richt zich ook op een theoretische beschouwing van gedistribueerde applicaties. Dit onderzoek is een eerste stap naar een programmeertaal met grondige ondersteunig voor formele methoden. Het gepresenteerde werk is verdeeld in drie delen, die elk een onderdeel vormen van het gemeenschappelijke doel.

Het eerste deel beschrijft de ontwikkeling van een programmaomgeving, dat het voorgestelde gestrengelde gelijktijdigheidsmodel van ABS implementeert, doelgericht op de efficiĂ«ntie en schaalbaarheid ervan. Deze omgeving benut optimaal het beheer van de zogenaamde parallellopende leidraden ('threads' in het Engels) in de Java programmeertaal en integreert zowel acteursobjecten als toekomstobjecten met het asynchrone programmeren.

Het tweede deel van dit onderzoek stelt de constructen van hoger niveau, in het bijzonder de constructen die asynchroniciteit en parallellisme modelleren in ABS, beschikbaar binnen de programmeeromgeving van de Java programmeertaal door middel van een applicatieprogrammeringsinterface (API). De basis van deze API, genaamd JAAC, is gegeven als een Java programmaonderdeel en stelt daarmee constructen bloot aan vrij gebruik, veelal niet beperkt door types te checken van asynchrone aanroepen, en is daardoor vatbaar voor de mogelijkheid ongewenste programmatuur uit te voeren binnen acteursobjecten. Deze API is vervolgens uitgebreid, genaamd ASCOOP, als Scala programmaonderdeel, zodanig dat alle (a)synchrone

methode-aanroepen nu wel bij het type checken door de Scala compilator gecontroleerd worden, met als resultaat dat alleen de toegestane programmatuur kan worden uitgevoerd binnen acteursobjecten.

Het derde deel richt zich op de ontwikkeling van een compilator van softwaremodellen met als doeltaal Java, en als uitgebreide doeltaal Scala, dat een formeel correcte vertaling van programmagedrag vertoont. Deze vertaalslag ondersteunt volledig de semantiek van de kern van de modelleringstaal, inclusief acteursobjecten en groepen van acteursobjecten als softwarecomponenten. De ondersteuning van de compilator includeert de constructies voor asynchrone communicatie, het suspenderen van co-routines, en de continuering daarvan. Ten slotte vertaalt de compilator taal extensies van ABS voor tijdsafhankelijke modellen en modellen met consumptiemetingen van (geheugen)bronnen.

Acknowledgements

I would like to dedicate this book to my family for which I continue the legacy of having a PhD with my parents sharing three between them, and my brother pursuing this goal as well. They are my role-models of love, hard work, professionalism and support. I want to dedicate it to my grandparents who have started this family legacy and I feel are watching over me all day long and keeping me safe in all situations and close calls I went through life. I want to thank all my family members that spread over four generations from my grandfather's sister, my aunt and uncle and their family, my sister-in-law and her family, my cousins and the youngest generation, my four wonderful nieces and nephews. I want to thank all of the friends I made and have throughout this world that were at one point part of this amazing journey. They include professors, mentors, colleagues and student alumni as I consider them all friends for life. Finally I want to dedicate this book to my friend Arya who has been by my side two years of this amazing journey before passing away.

Curiculum Vitae

Experience

- 1 May 2020-present **Senior Full Stack Engineer** *ING*
- 1 October 2018 - 30 April 2020 **Software Engineer** *bGrid*
- 1 October 2013 - 30 September 2018 **PhD in Computer Science** *Centrum Wiskunde en Informatica (CWI) and Leiden Advanced Institute of Computer Science (LIACS).*

Education

- 2011-2013 **Master of Science in Parallel and Distributed Computer Systems** *University Politehnica of Bucharest - first year, Vrije University of Amsterdam - second year*
- 2007-2011 **Bachelor of Science in Computers** *University Politehnica of Bucharest, Romania*

Bibliography

[AAFM+14] Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, Germán Puebla, and Guillermo Román-Díez. Saco: Static analyzer for concurrent objects. In *TACAS*, volume 14, pages 562–567, 2014.

[ABdBMM16] Elvira Albert, Nikolaos Bezirgiannis, Frank de Boer, and Enrique Martin-Martin. A formal, resource consumption-preserving translation of actors to haskell. *arXiv preprint arXiv:1608.02896*, 2016.

[absa] `http://docs.abs-models.org/`.

[absb] `https://abs-models.org/manual/`.

[absc] `https://github.com/abstools/abstools`.

[AdBdV18] Keyvan Azadbakht, Frank S. de Boer, and Erik de Vink. Deadlock detection for actor-based coroutines. In Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik de Vink, editors, *Formal Methods*, pages 39–54, Cham, 2018. Springer International Publishing.

[AdBH+13] Elvira Albert, Frank de Boer, Reiner Hähnle, Einar Broch Johnsen, and Cosimo Laneve. Engineering virtualized services. In *Proceedings of the Second Nordic Symposium on Cloud Computing & Internet Technologies*, pages 59–63. ACM, 2013.

[AdBH+14] Elvira Albert, Frank S. de Boer, Reiner Hähnle, Einar Broch Johnsen, Rudolf Schlatte, Silvia Lizeth Tapia Tarifa, and Peter Y. H. Wong. Formal modeling and analysis of resource management for cloud architectures: an industrial case study using real-time ABS. *Service Oriented Computing and Applications*, 8(4):323–339, 2014.

[AdBS16] Keyvan Azadbakht, Frank S. de Boer, and Vlad Serbanescu. Multi-threaded actors. In *Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8-9 June 2016.*, pages 51–66, 2016.

[Agh85] Gul A Agha. Actors: A model of concurrent computation in distributed systems. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1985.

[ASdB15] Keyvan Azadbakht, Vlad Serbanescu, and Frank de Boer. High performance computing applications using parallel data processing units. In Mehdi Dastani and Marjan Sirjani, editors, *Fundamentals of Software Engineering*, pages 191–206, Cham, 2015. Springer International Publishing.

[AVWW93] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. Concurrent programming in erlang. 1993.

[BBVB+01] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. 2001.

[BCC+06] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, et al. Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *The International Journal of High Performance Computing Applications*, 20(4):481–494, 2006.

[BdB16] Nikolaos Bezirgiannis and Frank de Boer. Abs: a high-level modeling language for cloud-aware programming. In *International Conference on Current Trends in Theory and Practice of Informatics*, pages 433–444. Springer, 2016.

[BdBJ+13] Joakim Bjørk, Frank S de Boer, Einar Broch Johnsen, Rudolf Schlatte, and S Lizeth Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 9(1):29–43, 2013.

[Bok87] Shahid H Bokhari. Multiprocessing the sieve of eratosthenes. *Computer*, 20(4):50–58, 1987.

[BSH+17] Frank De Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5):76:1–76:39, October 2017.

[But97] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[CJO10] Dave Clarke, Einar Broch Johnsen, and Olaf Owe. *Concurrent Objects à la Carte*, pages 185–206. Springer Berlin Heidelberg, 2010.

[Com] Apache Commons. Javaflow, 2009. *URL http://commons. apache. org/sandbox/javaflow*.

[CW10] Martin J Chorley and David W Walker. Performance analysis of a hybrid mpi/openmp application on multi-core clusters. *Journal of Computational Science*, 1(3):168–174, 2010.

[DBCJ07] Frank S De Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *Programming Languages and Systems*, pages 316–330. Springer, 2007.

[DBH15] Crystal Chang Din, Richard Bubel, and Reiner Hähnle. Key-abs: A deductive verification tool for the concurrent modelling language ABS. In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 517–526, 2015.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[Dij78] Edsger W Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. In *Programming Methodology*, pages 166–175. Springer, 1978.

[DO14] Crystal Chang Din and Olaf Owe. A sound and complete reasoning system for asynchronous communication with shared futures. *J. Log. Algebr. Meth. Program.*, 83(5-6):360–383, 2014.

[FMAG13] Antonio E Flores-Montoya, Elvira Albert, and Samir Genaim. May-happen-in-parallel based deadlock analysis for concurrent objects. In *Formal Techniques for Distributed Systems*, pages 273–288. Springer, 2013.

[FRCH+19] Kiko Fernandez-Reyes, Dave Clarke, Ludovic Henrio, Einar Broch Johnsen, and Tobias Wrigstad. Godot: All the benefits of implicit and explicit futures. 2019.

[GJSS14] Georg Göri, Einar Broch Johnsen, Rudolf Schlatte, and Volker Stolz. Erlang-style error recovery for concurrent objects with cooperative scheduling. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 5–21. Springer, 2014.

[GLL16] Elena Giachino, Cosimo Laneve, and Michael Lienhardt. A framework for deadlock detection in core ABS. *Software and System Modeling*, 15(4):1013–1048, 2016.

[Gup12] Munish Gupta. *Akka essentials*. Packt Publishing Ltd, 2012.

[Hal12] Philipp Haller. On the integration of the actor model in mainstream technologies: the scala perspective. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*, pages 1–6. ACM, 2012.

[hat] `http://www.hats-project.eu/node/113`.

[HFW84] Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Continuations and coroutines. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 293–298, New York, NY, USA, 1984. ACM.

[HM08] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41(7), 2008.

[HM16] Reiner Hähnle and Radu Muschevici. Towards incremental validation of railway systems. In *International Symposium on Leveraging Applications of Formal Methods*, pages 433–446. Springer, 2016.

[HO09] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2):202–220, 2009.

[HR16] Ludovic Henrio and Justine Rochas. From modelling to systematic deployment of distributed active objects. In *Proc. Coordination Models and Languages: 18th Intl. Conf. (COORDINATION)*, volume 9686. 2016.

[IS12] Shams M Imam and Vivek Sarkar. Integrating task parallelism with actors. In *ACM SIGPLAN Notices*, volume 47, pages 753–772. ACM, 2012.

[IS14] Shams M Imam and Vivek Sarkar. Savina-an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *4th International Workshop on Programming based on Actors Agents & Decentralized Control*, pages 67–80. ACM, 2014.

[Jan17] Manish Jangid. Kotlin-the unrivalled android programming language lineage. *Imperial Journal of Interdisciplinary Research*, 3(8), 2017.

[JHS+12] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. Abs: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects*, pages 142–164. Springer, 2012.

[JST12a] Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa. Modeling resource-aware virtualized applications for the cloud in real-time ABS. In *Formal Methods and Software Engineering - 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012. Proceedings*, pages 71–86, 2012.

[JST12b] Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa. Modeling resource-aware virtualized applications for the cloud in real-time abs. In *Formal Methods and Software Engineering*, pages 71–86. Springer, 2012.

[Kan02] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[KH16] Eduard Kamburjan and Reiner Hähnle. Uniform modeling of railway operations. In *FTSCS*, pages 55–71. Springer, 2016.

[KH17] Eduard Kamburjan and Reiner Hähnle. Deductive verification of railway operations. In *International Conference on Reliability, Safety and Security of Railway Systems*, pages 131–147. Springer, 2017.

[KS19] Eduard Kamburjan and Jonas Stromberg. Tool-supported visualization and interactive prototyping of railway operations, 2019.

[KSA09] Rajesh K Karmani, Amin Shali, and Gul Agha. Actor frameworks for the jvm platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 11–20. ACM, 2009.

[Lam] Lambda expression deocumentation. `https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html`. Accessed: 2018-09-30.

[LDMA09] Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul Agha. A framework for state-space exploration of java-based actor programs. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 468–479, Washington, DC, USA, 2009. IEEE Computer Society.

[Lea00] Douglas Lea. *Concurrent programming in Java: design principles and patterns*. Addison-Wesley Professional, 2000.

[Mar02] Robert C Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[NAB+11] Bogdan Nicolae, Gabriel Antoniu, Luc Bougé, Diana Moise, and Alexandra Carpen-Amarie. Blobseer: Next-generation data management for large scale infrastructures. *J. Parallel Distrib. Comput.*, 71:169–184, February 2011.

[NdB14] Behrooz Nobakht and Frank S de Boer. Programming with actors in java 8. In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, pages 37–53. Springer, 2014.

[O'N09] MELISSA E O'NEILL. The genuine sieve of eratosthenes. *Journal of Functional Programming*, 19(01):95–106, 2009.

[PM01] Esmond Pitt and Kathy McNiff. *Java. rmi: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[PR94] Jenny Preece and H Dieter Rombach. A taxonomy for combining software engineering and human-computer interaction measurement approaches: towards a common framework. *International journal of human-computer studies*, 41(4):553–583, 1994.

[pri] `http://primesieve.org`.

[PS12] Guillaume Pierre and Corina Stratan. ConPaaS: a platform for hosting elastic cloud applications. *IEEE Internet Computing*, 16(5):88–92, September-October 2012.

[PWB09] Kai Petersen, Claes Wohlin, and Dejan Baca. The waterfall model in large-scale development. In *International Conference on Product-Focused Software Process Improvement*, pages 386–400. Springer, 2009.

[RH14] Justine Rochas and Ludovic Henrio. A ProActive Backend for ABS: from Modelling to Deployment. Research Report RR-8596, INRIA, September 2014.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified modeling language reference manual, the*. Pearson Higher Education, 2004.

[Roc16] Justine Rochas. *Execution support for multi-threaded active objects: design and implementation*. PhD thesis, Université Côte d'Azur, 2016.

[SAB+15] Vlad Serbanescu, K Azadbakht, F Boer, C Nagarajagowda, and B Nobakht. A design pattern for optimizations in data intensive applications using abs and java 8. *Concurrency and Computation: Practice and Experience*, 2015.

[ŞAdB16] Vlad Şerbănescu, Keyvan Azadbakht, and Frank de Boer. *A Java-Based Distributed Approach for Generating Large-Scale Social Network Graphs*, pages 401–417. Springer International Publishing, Cham, 2016.

[SB] Vlad Serbanescu and Frank S. Boer. On the nature of cooperative scheduling in active objects. In Submission.

[SBJ18a] Vlad Serbanescu, Frank S. Boer, and Mahdi Jaghoori. Actors with coroutine support in java. In *Proceedings 15th International Conference on Formal Aspects of Component Software, Pohang, South Korea 10-12 October 2018*. Springer, 2018.

[SBJ18b] Vlad Serbanescu, Frank S. Boer, and Mahdi Jaghoori. Ascoop: Actors in scala with cooperativescheduling. In *Proceedings 21st IEEE International Conference on Computational Science and Engineering*. IEEE, 2018.

[Sch97] Ken Schwaber. Scrum development process. In *Business object design and implementation*, pages 117–134. Springer, 1997.

[Sch11] Jan Schäfer. *A Programming Model and Language for Concurrent and Distributed Object-Oriented Systems*. PhD thesis, University of Kaiserslautern, 2011.

[Sir07] Marjan Sirjani. Rebeca: Theory, applications, and tools. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 102–126, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[SM01] M Sirjani and A Movaghar. An actor-based model for formal modelling of reactive systems: Rebeca. *Technical Report*, 2001.

[SM08] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *European Conference on Object-Oriented Programming*, pages 104–128. Springer, 2008.

[SNA+14] Vlad Serbanescu, Chetan Nagarajagowda, Keyvan Azadbakht, Frank de Boer, and Behrooz Nobakht. Towards type-based optimizations in distributed applications using abs and java 8. In *Adaptive Resource Management and Scheduling for Cloud Computing*, pages 103–112. Springer, 2014.

[SOHL+98] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference (Vol. 1): Volume 1-The MPI Core*, volume 1. MIT press, 1998.

[SP15] Andrei Sfrent and Florin Pop. Asymptotic scheduling for many task computing in big data platforms. *Information Sciences*, 319:71–91, 2015.

[SPCA14] V. Serbanescu, F. Pop, V. Cristea, and G. Antoniu. Architecture of distributed data aggregation service. In *Advanced Information Networking and Applications (AINA), 2014 IEEE 28th International Conference on*, pages 727–734, May 2014.

[SPCA15] Vlad Serbanescu, Florin Pop, Valentin Cristea, and Gabriel Antoniu. A formal method for rule analysis and validation in distributed data aggregation service. *World Wide Web*, 18(6):1717–1736, 2015.

[Sto] Enroute Storm. Scala coroutines, 2011. *URL http://storm-enroute.com/coroutines/*.

[Tip13] Sever Tipei. Composing with sieves: Structure and indeterminacy in-time. In *ICMC*. Citeseer, 2013.

[VLFGL01] Gregor Von Laszewski, Ian Foster, Jarek Gawor, and Peter Lane. A java commodity grid kit. *Concurrency and Computation: practice and experience*, 13(8-9):645–662, 2001.

[VPT+15] Mihaela-Andreea Vasile, Florin Pop, Radu-Ioan Tutueanu, Valentin Cristea, and Joanna Kołodziej. Resource-aware hybrid scheduling algorithm in heterogeneous distributed computing. *Future Generation Computer Systems*, 51:61–71, 2015.

[WAM+12] Peter YH Wong, Elvira Albert, Radu Muschevici, José Proença, Jan Schäfer, and Rudolf Schlatte. The abs tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *International Journal on Software Tools for Technology Transfer*, 14(5):567–588, 2012.

[WDS11] Peter YH Wong, Nikolay Diakov, and Ina Schaefer. Modelling distributed adaptable object oriented systems using hats approach: A fredhopper case study. In *2nd International Conference on Formal Verification of Object-Oriented Software*, volume 7421, 2011.

[yS05] T Oliveira y Silva. Fast implementation of the segmented sieve of eratosthenes, 2005.