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

```
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

```
abstract class Maybe<A > {
  final boolean isNothing( ) { return this instanceof Maybe_Nothing; }
  final Maybe_Nothing<A> toNothing( ) { return (Maybe_Nothing) this; }
  final boolean isJust( ) { return this instanceof Maybe_Just; }
  final Maybe_Just<A> toJust( ) { return (Maybe_Just) this; }
}
```

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

```
public final class Maybe_Nothing<A> extends Maybe<A> {

  public Maybe_Nothing( ) {
  }
  public abs.backend.java.lib.types.ABSBool eq(...) {
  ...
  }
}
```

Listing 5.4: Just Class in Java

```
package ABS.StdLib;
public final class Maybe_Just<A> extends Maybe<A> {

  public final A arg0;
  public Maybe_Just(final A arg0) {
    this.arg0 = arg0;
  }
  public abs.backend.java.lib.types.ABSBool eq(...) {
    ...
    }
}
```

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

```
public abstract class ABSObject implements ABSRef {

  protected COG __cog;
  protected final long __id;

  protected ABSObject(COG cog) {
    this.__cog = cog;
    this.__id = getFreshID( );
    this.__cog.register(this);
  }
}
```

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.

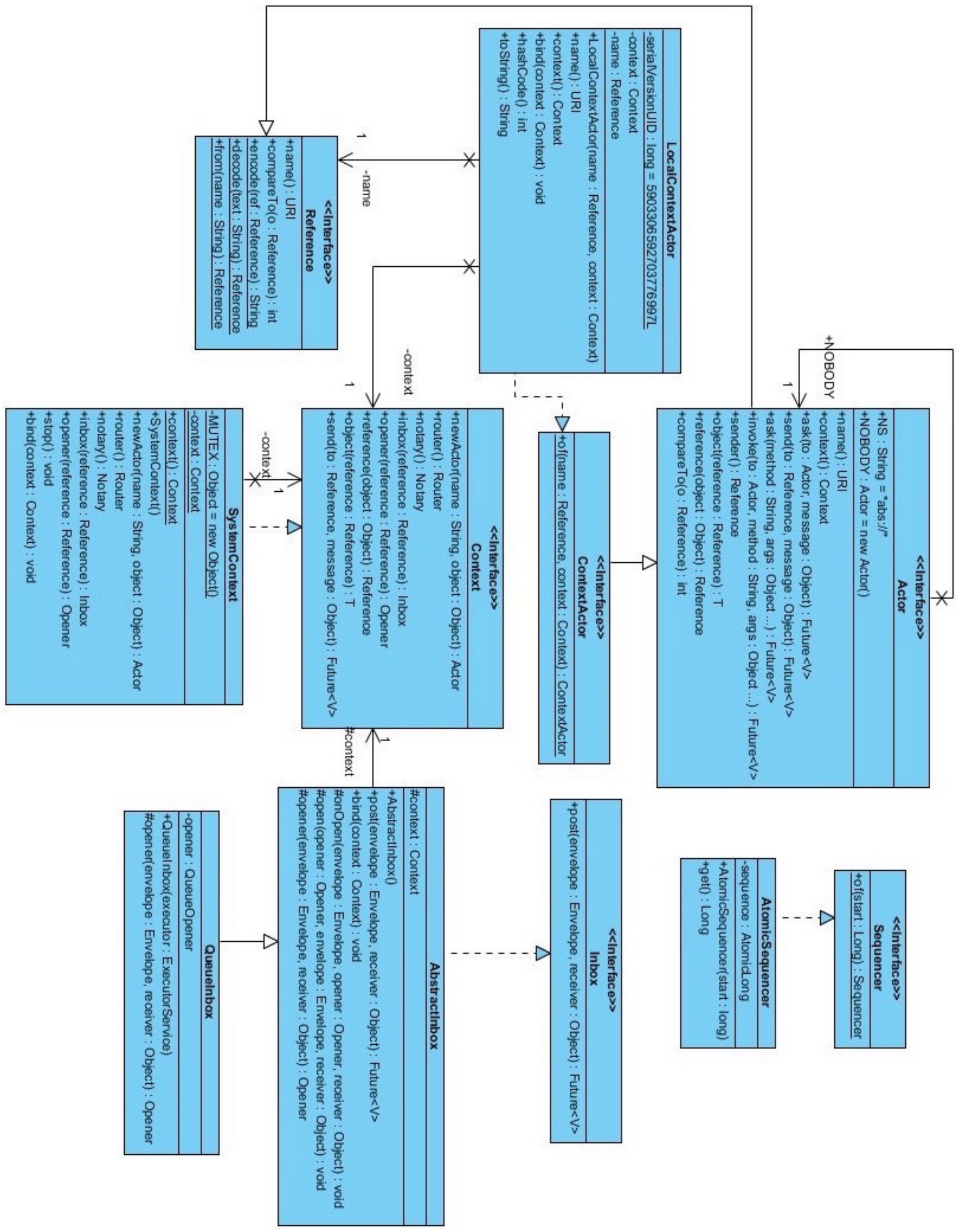
LocalContextActor
-serialVersionUID : long = 5903306592703776697L
-context : Context
-name : Reference
+LocalContextActor(name : Reference, context : Context)
+name() : URI
+context() : Context
+bind(context : Context) : void
+hashCode() : int
+toString() : String
<<Interface>> Reference
+name() : URI
+compareTo(o : Reference) : int
+encode(ref : Reference) : String
+decode(text : String) : Reference
+from(name : String) : Reference
-name
1
-context
1
<<Interface>> Actor
+NS : String = "abs://"
+NOBODY : Actor = new Actor()
+name() : URI
+context() : Context
+ask(to : Actor, message : Object) : Future<V>
+send(to : Reference, message : Object) : Future<V>
+ask(method : String, args : Object ...) : Future<V>
+invoke(to : Actor, method : String, args : Object ...) : Future<V>
+sender() : Reference
+object(reference : Reference) : T
+reference(object : Object) : Reference
+compareTo(o : Reference) : int
+NOBODY
1
<<Interface>> ContextActor
+of(name : Reference, context : Context) : ContextActor
<<Interface>> Context
+newActor(name : String, object : Object) : Actor
+router() : Router
+notary() : Notary
+inbox(reference : Reference) : Inbox
+opener(reference : Reference) : Opener
+reference(object : Object) : Reference
+object(reference : Reference) : T
+send(to : Reference, message : Object) : Future<V>
SystemContext
-MUTEX : Object = new Object()
-context : Context
+context() : Context
+SystemContext()
+newActor(name : String, object : Object) : Actor
+router() : Router
+notary() : Notary
+inbox(reference : Reference) : Inbox
+opener(reference : Reference) : Opener
+stop() : void
+bind(context : Context) : void
-context
1
#context
1
AbstractInbox
#context : Context
+AbstractInbox()
+post(envelope : Envelope, receiver : Object) : Future<V>
+bind(context : Context) : void
#onOpen(envelope : Envelope, opener : Opener, receiver : Object) : void
#open(opener : Opener, envelope : Envelope, receiver : Object) : void
#opener(envelope : Envelope, receiver : Object) : Opener
<<Interface>> Inbox
+post(envelope : Envelope, receiver : Object) : Future<V>
QueueInbox
-opener : QueueOpener
+QueueInbox(executor : ExecutorService)
#opener(envelope : Envelope, receiver : Object) : Opener
AtomicSequencer
-sequence : AtomicLong
+AtomicSequencer(start : long)
+get() : Long
<<Interface>> Sequencer
+of(start : Long) : Sequencer

Figure 5.1: ABS-API Class Diagram

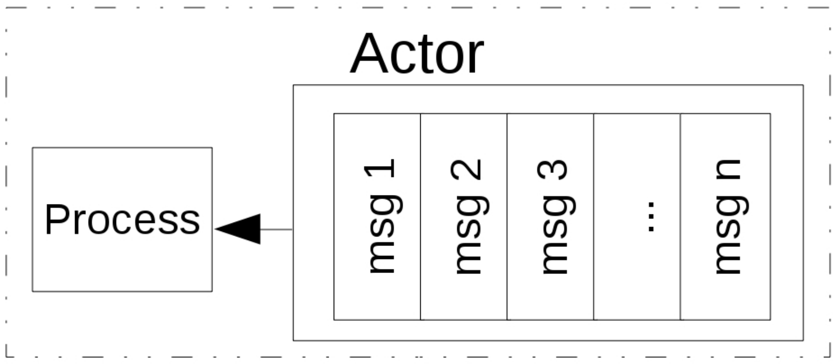

Figure 5.2: Actor Model

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.

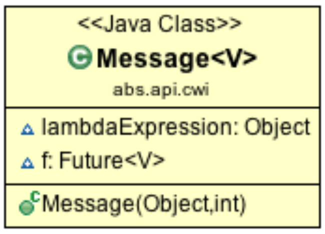

Figure 5.3: Message encapsulation

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.

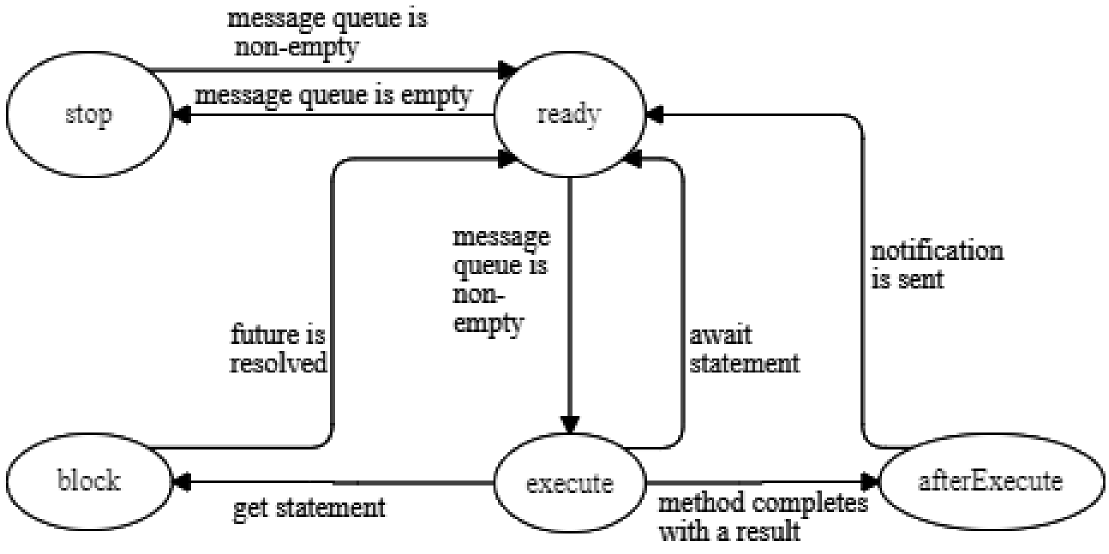

Figure 5.4: Actor State Diagram

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

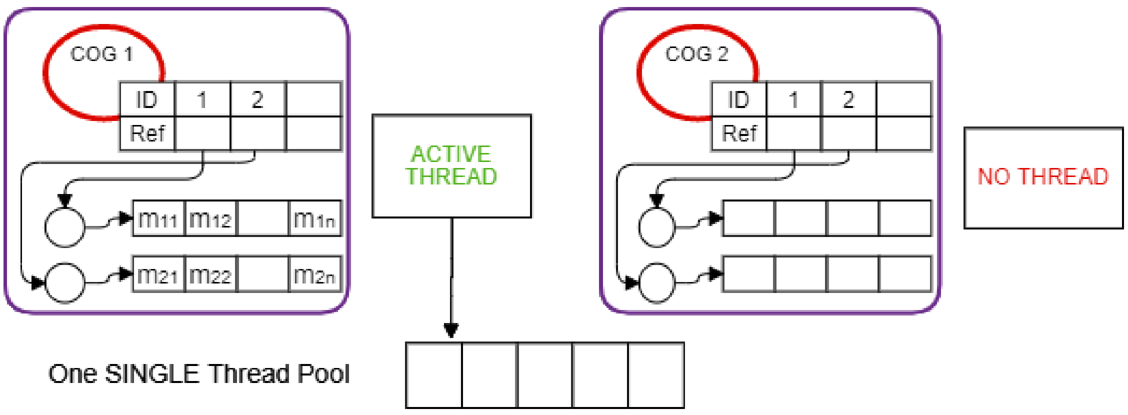

Figure 5.5: Thread Creation and Scheduling

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:

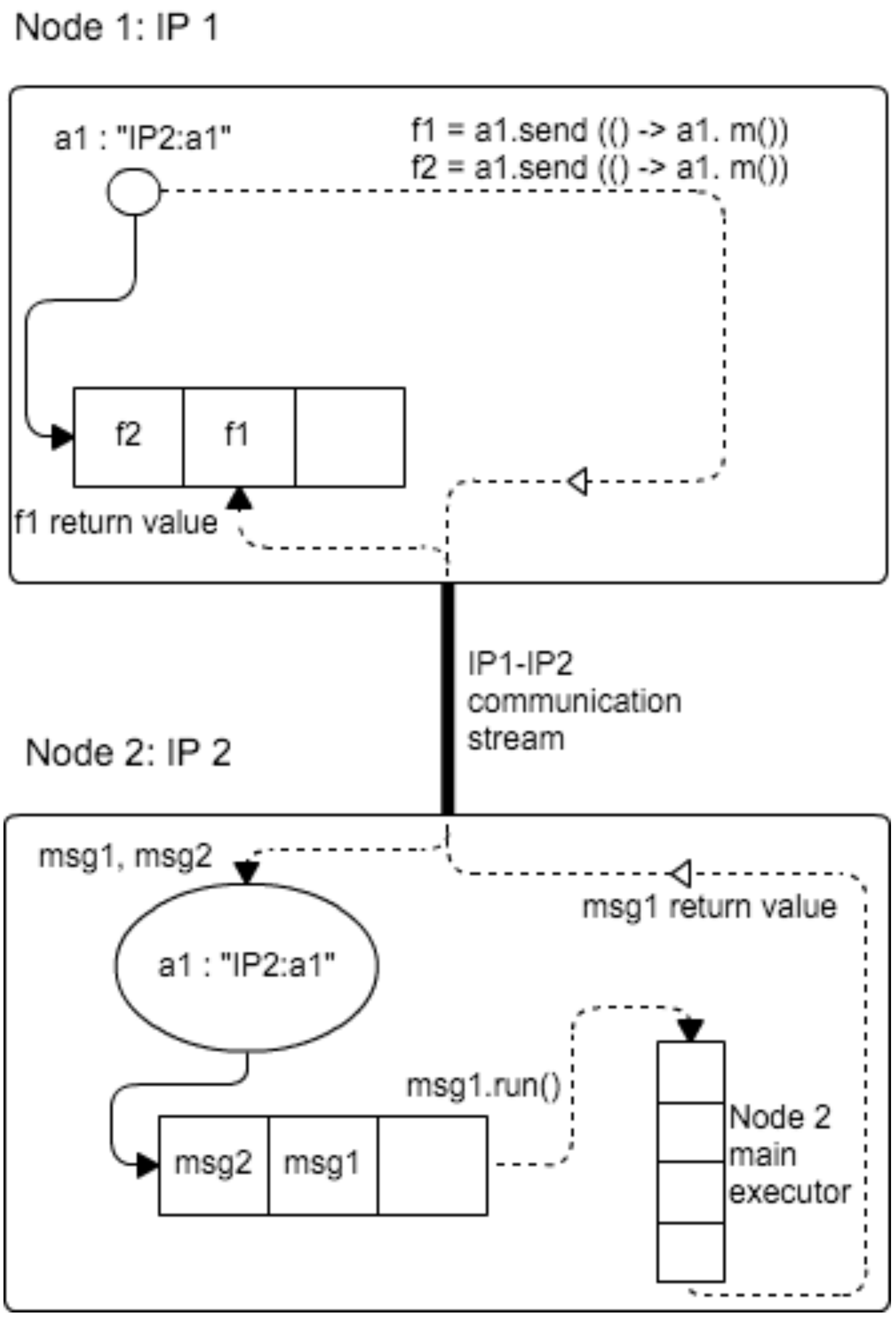

Figure 5.6: Future Flow

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

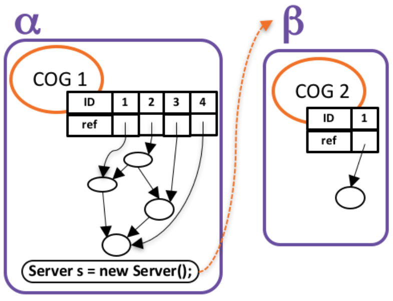

Figure 5.7: ABS new in ProActive

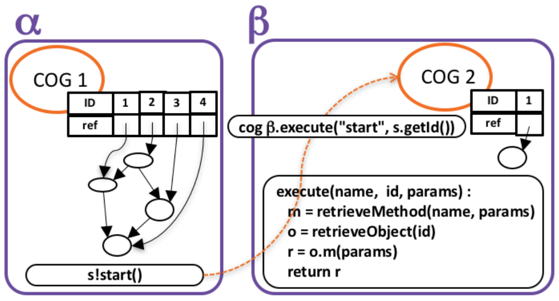
α
β
COG 1
ID 1 2 3 4
ref
s!start()
COG 2
ID 1
ref
cog β.execute("start", s.getId())
execute(name, id, params) :
m = retrieveMethod(name, params)
o = retrieveObject(id)
r = o.m(params)
return r

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).