Actors with Coroutine Support in Java

Vlad Serbanescu(✉), Frank de Boer, and Mohammad Mahdi Jaghoori

Centrum Wiskunde and Informatica, Amsterdam, The Netherlands
{vlad.serbanescu,frank.s.de.boer,jaghouri}@cwi.nl
http://www.cwi.nl

Abstract. In this paper, we introduce a Java library for actors integrated seamlessly with futures and supporting coroutines. Coroutines allow actors to suspend the execution of a message and possibly schedule other messages before resuming the suspended continuation. As such coroutines enhance actors as a major building block for constructing software components. The library is used together with a compiler to generate code from an application model into an executable program in Java. A formal description of the translation process is provided together with the most important library methods. We highlight the importance of having a scalable and efficient implementation by means of some typical benchmarks which model a large number of tasks, coroutines and actors.

Keywords: Asynchronous programming · Actors · Coroutines Futures · Object-orientation · Java

1 Introduction

Asynchronous programming is becoming the standard programming paradigm. The Abstract Behavioral Specification (ABS) modeling language [12] provides a formal computational model which integrates actor-based programming with futures and coroutines.

An ABS model describes a dynamic system of actors which only interact by means of asynchronous method calls. This provides a "programming to interfaces" paradigm that enables static type checking of message passing at compile time. In contrast, in the typical approach of actors such as in Scala, messages are allowed to have any type and thus it is only checked at run-time whether the receiver can handle them. In ABS a future is generated only when a method is called asynchronously. This future can be passed around and actors holding a reference to it can check the completion of the corresponding method and then, if the method is not void, get its returned value.

Actor-based models of computation in general assume a run-to-completion mode of execution of the messages [2]. ABS extends the actor-based model with *coroutines* 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.

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. Internally with support for cooperative scheduling they allow for a fine-grained and powerful internal synchronization between the different method invocations of an actor. 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 [3]. ABS has been applied to several major case studies like in the domain of cloud computing [7], simulation of railway models [14] and modeling software product lines [13].

The main contribution of this paper is a Java library called JAAC[1] together with a compiler[2] which allows generation of programs in Java from ABS sources, as well as enabling writing 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 [6] and deadlock analysis [8].

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 [1]. 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 overall architecture of the system is based on the following. We submit one main task per actor to the thread pool which iteratively selects an enabled task from its queue and runs it. We make use of 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 modelling of both asynchronous method calls and suspension of a method invocation. Suspension of a method invocation, 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.

Related Work. The library provides a scalable implementation of the ABS asynchronous programming model which integrates actors, futures and cooperative scheduling. In contrast, existing libraries in mainstream JVM languages, namely Java [20], Scala [9] and Kotlin [11], support actors, futures and coroutines mainly as independent mechanisms for asynchronous programming. Even though one

[1] https://github.com/JaacRepo/JAAC push-mechanism branch.
[2] https://github.com/JaacRepo/absCompiler.

can use them together, this in general only adds to the complexity of the program. For example in Scala, one can use a future to hold the result of an asynchronous message sent to an actor. But then one should either block the whole thread to await the completion of the future, or register a continuation that will possibly run in another thread in parallel to the actor's thread. It is the programmer's job to ensure that the thread running the continuation will not give rise to race conditions with the actor's thread. In JAAC, however, awaiting the result of an asynchronous message, which is automatically captured in a future, creates a continuation that will safely run in the actor's thread; thus no race conditions can happen.

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 [4] and Kilim [17]. 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.

The ABCL language [19,21] is a language similar to ABS in terms of asynchronous communication support and message suspension. However, messages are preempted without a user-defined condition, being based only on an assigned priority. To the best of our knowledge it does not have any formal semantics and an up to date status of its implementation available.

A straightforward way to support coroutines at source-code level in Java (see [15]) 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 [18] 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 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 (as is the case in JAAC). In other words, it is not possible to process other messages if one message is suspended.

Our solution provides support for coroutines by only making changes at the source level and provides a higher-level mechanism for scheduling and resumption tailored towards actor-based systems. Unlike Scala coroutines, suspension and release points are done through the use of the API, allowing the programmer to specify resumption based on a particular actor state or task completion (done by either the same or different actor). Therefore the programmer does not need to explicitly resume control in the code. Our solution extends the Java implementation of ABS described in [16] with a new general mechanism for spawning tasks which allows for modeling suspension of entire call stacks and in general allows for a more efficient executable in terms of scalability and performance.

The rest of this paper is organized as follows: In Sect. 2 we give an overview of the main features that the target actor-based model has. Section 3 presents the formal aspects of pre-processing and compiling continuations together with the most important method from the library API. Section 4 describes the implementation of the run-time system of the JAAC library. In Sect. 5 we show the experimental evaluation of our solution followed by the conclusions drawn in Sect. 6.

2 Coroutine Support in ABS

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 1.1 which presents a general behaviour of a pool of workers. For a detailed description of the syntax of the ABS language we refer to [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 in this manner asynchronous method calls in ABS by default return futures (see [5] 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 12 and 13). Each worker actor stores a reference to its worker pool `p` which is passed as a parameter upon instantiation (line 16). Before returning the result (line 21) the method `doWork` asynchronously calls the method `finished` of its associated `WorkerPool` reference (line 20). The suspension mechanism underlying the `await` statement in line 5 allows to schedule any 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`.

In ABS a statement of the form `await f?` suspends the executing method invocation which can only be rescheduled if the method invocation corresponding to the future `f` has computed the return value. In contrast, the evaluation of the expression `f.get` blocks all the method invocations of an actor until the return value has been computed.

A key feature of the coroutine execution of messages in ABS is that it does *not* provide an explicit command for resuming a message. Messages are resumed for execution only by the underlying scheduler. This implicit resumption by the underlying scheduler allows for an important improvement of the program quality and avoids the error-prone usage of explicit resumption, i.e., resuming a routine twice in Scala [18] raises an exception.

Listing 1.1. 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      return f.get;
10    }
11
12    Unit finished(Worker w) {
13      workers = insertElement(workers, w);
14  } }
15
16  class Worker(WorkerPool p) implements Worker{
17      Result doWork(){
18          Result r;
19          // computation
20          p ! finished(this);
21          return r;
22  }   }
```

Line 1 of Listing 1.1 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 3–5. It is important to observe that evaluation of `fut.get` will never block because of the successful execution of `await fut?` .

Listing 1.2. ABS Await sugared syntax

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

In ABS, actors which form so-called 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 1.3. We can then call this method synchronously like in line 8. 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 3 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 8. In contrast to multi-threading in Java, these call stacks cannot be interleaved arbitrarily, only one call stack in an actor is executing until it is either terminated or suspended.

Listing 1.3. Synchronous Call in ABS

```

Worker getWorker(){
   await !(emptySet(workers));
   Worker w = take(workers);
}

Result sendWork() {
   Worker w = this.getWorker();
   workers = remove(workers, w);
   Fut<Result> f = w ! doWork();
   return f.get;
}
```

3 Emulating Coroutines Through Spawning Tasks

To emulate the coroutines of ABS we introduce a new general mechanism of spawning tasks. This mechanism is a lightweight alternative to JVM threads and by generating code in a certain design pattern, the state from which to resume can be saved as part of the spawned task, such that it is equivalent to the resumption point of the coroutine. As described in Sect. 2 an actor can resume a method once the guard suspending it has been satisfied. The local environment of the method together with any possible call stack that built up before the **await** statement must be reloaded. This mechanism avoids the need to reload the local environment and call stack as will be explained later in this section.

3.1 Spawning Tasks

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 three important methods that support the simulation of ABS features in the Java language.

The first method is called **spawn** and its usage is highlighted is Listing 1.4. It is used to implement the suspension point in the `sendWork` method of the actor class `WorkerPool` in Listing 1.1. The method generates a new task (in the form of a Callable). 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). This is the syntax used on line 3 of Listing 1.4.

Note that the return type of `sendWork` is an `ABSFuture`. This is special future type defined in the library that cannot block a thread when trying to retrieve its result. Instead it is used in conjunction with **getSpawn** (to be described later in this section) to spawn a new task that uses its result once the `ABSFuture`is ready. More implementation details about the `ABSFuture` will be presented in Sect. 4.1.

The `guard` parameter (`nonEmpty`) 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 instance type of `Guard` parameter passed. As the enabling condition has to verify an actor's state, it needs to be checked every time the actor attempts to schedule the task (i.e. the block of code that starts on line 4). 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).

It is important to note that the intended target object for calling **spawn** is **this** (also known as a self call) as 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 objects as part of the guard as it would break actor semantics. The other two methods (whose usage is already shown on lines 5 and 6) represent particular cases of this method. These two methods along with their usage and parameters will be explained next in this section.

Listing 1.4. Spawn Method Intended Usage

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

The **send** method is used to model ABS asynchronous method invocations. 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.

Using a lambda expression as shown in Listing 1.5 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 itself will be stored in the internal task queue of the actor (see Sect. 4).

Listing 1.5. Sending an Asynchronous Call

```
WorkerPool pool = new WorkerPool();
ABSFuture<Result> fut = pool.send(() −> pool.sendWork());
```

The **getSpawn** method 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). Listing 1.6 shows how to model an ABS

await syntactic sugar illustrated in Listing 1.2. The task represents a `Callable` instance that in the method application in line 2 (Listing 1.6) *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 1.6. getSpawn Method Intended Usage

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

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 `Callable` 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 1.7 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 (an **await** construct is encountered), it will use **spawn** for suspension (line!3). By default tasks that are created by **spawn** or **send** are set with a LOW priority. The default priority, when **getSpawn** is used without priority arguments is 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 6 of Listing 1.4. Note that this combination of HIGH priority and STRICT does not allow scheduling of any other tasks (of the given actor).

Listing 1.7. Usage of getSpawn to emulate a synchronous call of an Actor

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

3.2 Compiler Correctness

The basic idea underlying the compiler is to model both method calls and the suspension mechanism in ABS by a general mechanism of spawning tasks. In this section we focus on the correctness of the translation of the await statement. We do so by considering a language ABS-SPAWN which is obtained from ABS by using instead of the await construct the **spawn**(g, S) statement for spawning subtasks (the unconditional release statement in ABS we view as an abbreviation of **await true**). Note that in ABS-SPAWN method invocations are thus executed in a run-to-completion mode.

The basic idea underlying the compilation of await statements then can be formalized by a formal translation of ABS statements into corresponding statements of ABS-SPAWN (we refer to the syntax of ABS in [12]). 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. We only highlight the main rules of this translation in Fig. 1 which affect the first instruction (in all other cases, e.g., that of method calls, the first instruction is not affected and the translation is only applied to its sequential continuation). For technical convenience only, we assume that repetitive (while) statements are rewritten using tail-recursion[3]. This allows us to syntactically identify the continuation of an **await** statement as its sequential continuation in the body of the method and does not require any loop-unfolding.

$$\begin{array}{rcl}
T(\epsilon) & := & \epsilon \\
T(\textbf{await}\ g;\ S) & := & \textbf{spawn}(g,\ T(S)\ ) \\
T(\texttt{if}\ b\ \{S_1\}\ \texttt{else}\ \{S_2\}; S) & := & \texttt{if}\ b\ \{T(S_1;\ S;)\}\ \texttt{else}\ \{T(S_2;\ S)\} \\
T(\texttt{case}\ e\ \{ & := & \texttt{case}\ e\ \{ \\
P_1 \Rightarrow S_1 & & P_1 \Rightarrow T(S_1;\ S) \\
P_2 \Rightarrow S_2 & & P_2 \Rightarrow T(S_2;\ S) \\
\ldots & & \ldots \\
P_n \Rightarrow S_n & & P_n \Rightarrow T(S_n;\ S) \\
\}; S & & \}
\end{array}$$

Fig. 1. Translation of ABS syntax

In Fig. 1 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

[3] In practice, running applications using tail-recursion are affected by the program's memory limits due to the buildup of the program's stack memory. The compiler of ABS into Java using the library, avoids this by converting tail-recursion into a set of spawned tasks representing each iteration of the loop.

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 also applies to the translation of the 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.

In order to establish formally the correctness of this translation we first introduce a formal operational semantics of the ABS-SPAWN language.

Operational semantics. We introduce object configurations of the form (σ, S, Q), where:

- 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.
- S represents the current statement of the active process that is run by the actor.
- Q is a (multi-)set of statements which represent suspended processes. As a special case, we introduce a run-time syntax of the form $(g \to S)$ that represents a (top-level) statement (S) that is guarded by an enabling condition (g).

As described below, to model sharing of the local variables of a method among its generated subtasks, for each method invocation fresh variables are introduced in σ for the local variables (including the formal parameters).

A global configuration G then is a set of object configurations. We highlight the following rules of the transition system for deriving transitions $G \to G'$.

Asynchronous Invocation Rule. For notational convenience only we describe the semantics of an invocation of a void method (thus abstracting from the generation of a future).

$$\{(\sigma, x!m(\bar{e}); S, Q), (\sigma', S', Q')\} \cup G \to \{(\sigma, S, Q), (\sigma'', S', Q'')\} \cup G$$

where $\sigma'(\textbf{this}) = \sigma(x)$ (i.e., $\sigma'(\textbf{this})$ is the callee of the method call $x!m(\bar{e})$), σ'' extends σ' by assigning to the fresh variables introduced for the local variables of m the values of the actual parameters $\bar{e}$ in σ, and, finally, Q'' is obtained from Q' by adding the body of m with its local variables renamed. In case of an assignment $y = x!m(\bar{e})$, we assign to the future variable a pair $(f, \bot)$, where f is the (unique) identity used as a reference to the return value which is initialised by $\bot$ (which stands for "undefined").

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

$$\{(\sigma, \mathbf{spawn}(g, S); S', Q)\} \cup G \to \{(\sigma, S', \{g \to S\} \uplus Q)\} \cup G$$

Scheduling Rule. The following rule describes the scheduling of an enabled suspended task.

$$\{(\sigma, \epsilon, \{g \to S\} \uplus Q)\} \cup G \to \{(\sigma, S, Q)\} \cup G$$

where σ validates the guard g (i.e., σ satisfies any Boolean condition of g and $\sigma(x) = (f, v)$, for some value $v \neq \perp$, for any query $x?$ of a future variable x). Note that only when the current statement has terminated a new statement is selected for execution.

It is straightforward to define a transition system for deriving transitions $G \to_{\text{abs}} G'$ modeling execution steps of an ABS program, where G (and G') now only contain ABS statements. (The main difference with the standard semantics of ABS ([12]) is the use of fresh variables for the local variables of a method instead of a local environment.)

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* statement **await** $g; S$ in G by $g \to T(S)$. 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 every global ABS configuration G we have*

$$G \to_{abs} G' \text{ iff } T(G) \to T(G')$$

4 Library Implementation in Java

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 [15]. 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.

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 detail the generation and completion of tasks and then the mechanism for task scheduling and execution.

4.1 Task Generation and Completion

Every call to **send**, **spawn** or **getSpawn** creates a new task as an instance of the class `ABSTask`(Listing 1.8) and stores it into the task queue of the actor callee. In all these cases the field `resultFuture` will contain a newly created instance of `ABSFuture`(Listing 1.9) which uniquely identifies this task and which is returned to the caller (of the **send**, **spawn** or **getSpawn** method). Upon termination of a task, the return value will be wrapped in an `ABSFuture` instance created by the `done()` method with a set `completed` flag and an assigned `value`. This new instance is subsequently assigned to the `target` field of the `ABSFuture` identifying the completed task. Note that thus upon termination of the task the `target` field of the future returned by the **send**, **spawn** or **getSpawn** method will hold a reference to the future returned by the generated task. Consequently for checking the availability and retrieval of value of a future, in general one needs to follow a chain of future references until a future with a null `target` field is found. This scheme fully integrates futures with the mechanism of spawning new tasks and supports the delegation of the computation of return values.

Listing 1.8. `ABSTask`Class

```
public class ABSTask<V> implements Serializable, Runnable {
    protected Guard enablingCondition = null;
    protected final ABSFuture<V> resultFuture;
    protected Callable<ABSFuture<V>> task;
    //implementation and functionality
}
```

Listing 1.9. `ABSFuture`Class

```
public class ABSFuture<V> {
    private V value = null;
    private boolean completed = false;
    private ABSFuture<V> target = null;
    private Set<Actor> awaitingActors = ConcurrentHashMap.newKeySet();
    //implementation and functionality
}
```

In order to avoid busy-waiting on futures, the class `ABSFuture` implements a push mechanism to notify the actors that are awaiting its completion (which are stored in the field `awaitingActors`). Whenever an actor passes a future to the **getSpawn** method, this actor is added to the list of awaiting actors of that future and will also be propagated through the `target` chain. We explain in the next subsection how the notification mechanism works together with the scheduling mechanism of actors.

Using JVM Garbage Collection. 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 `ABSFuture` class is implemented completely lock-free and therefore the chaining of futures performs very efficiently.

4.2 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 1.10. Inside the class there is a `taskQueue` which holds all tasks of an actor. Tasks are defined as instances of class `ABSTask` (for example on line 30). To allow for concurrent access and an efficient scheduling of these tasks we use a hashmap (`ConcurrentSkipListMap` on line 4) that orders tasks into buckets (queues of tasks defined by assigned priorities).

Listing 1.10. Local Actor Class

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

The implementation defines an inner class `MainTask` which is responsible for selecting an enabled task from the queue (via the `takeOrDie` method) and running it. Being a `Runnable`, the main task of an actor can be submitted to the system-wide thread pool and thus actors are put to compete for available threads in a scalable way. This fairness policy may also be fine-tuned to allow the `MainTask` to execute a fixed chunk of tasks at a time before releasing the thread in order to reduce context switches.

The `MainTask` avoids busy-waiting in cases that all tasks in the queue are disabled. In such cases, `takeOrDie` returns false and then the `MainTask` simply terminates. It is reactivated upon generation of any new task in the queue (line 33). To make sure there is no more than one instance of the `MainTask` running for an actor, we use the `mainTaskIsRunning` flag. As the task queue is accessed concurrently by the `MainTask` performing the queue traversal and other actors sending method invocations, it is important to avoid race conditions. A race condition may happen if after `MainTask` 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 `MainTask` will terminate and the new message also creates no new `MainTask`. We avoid this situation by the synchronized blocks in `takeOrDie` and `notRunningThenStart`.

In case `MainTask` 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 `MainTask`. 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 `MainTask` if it was terminated. As already stated above 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.

5 Benchmarking and Evaluation

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 [10]. 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[4] we provide implementations of several examples in the benchmark suite directly using the library, while in the compiler repository[5] we have several ABS models of these benchmarks.

5.1 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

[4] https://github.com/JaacRepo/JAAC.
[5] https://github.com/JaacRepo/absCompiler.

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 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 1.11.

Listing 1.11. Benchmark Example

```
interface Ainterface {
    Int recursive_m(Int i, Int id);
}

class A() implements Ainterface{
    Int result=0;
    Int recursive_m(Int i, Int id){
        if (i>0){
            this.recursive_m(i − 1,id);
        }else{
            Fut<Int> f = this ! compute( );
            await f ?;
        }
        return 1;
    }
    Int compute( ){
        return result + 1;  //no significant computation }  }
{ // Main block:
    Int i = 0;
    Ainterface master = new A ( );
    List<Fut<Int>> futures = Nil;
    while( i < 500){
        Fut<Int> f = master ! recursive_m (5, i);
        futures = Cons( f, futures );
        i = i + 1 ;
    }
    while ( futures != Nil ){
        Fut<Int> f1 = head(futures);
        futures = tail(futures);
        Int r = f1.get;
}  }
```

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, both in a thread-based approach and through spawning of tasks. The results are shown in Fig. 2. 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 actors 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.

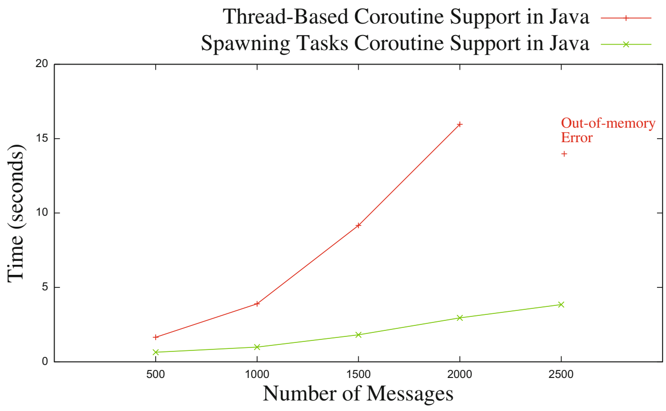

Fig. 2. Performance figures for coroutine overhead

5.2 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 1.2 and 1.3 describe in ABS the problem of arranging N queens on a $N \times N$ chess-board. It provides a master-slave model that illustrates very well the advantage of using actors together with coroutines.

Listing 1.12. NQueens Master Class Snippet

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

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 16 in Listing 1.12) 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 12 in Listing 1.13) to create a new subtask for the new board and the incremented `depth`. The master aggregates the results using a coroutine model (line 19) to await all the solutions starting at depth 0.

Listing 1.13. 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          else { //send a solution to the master
17  }   }   }
```

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 Fig. 3. 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.

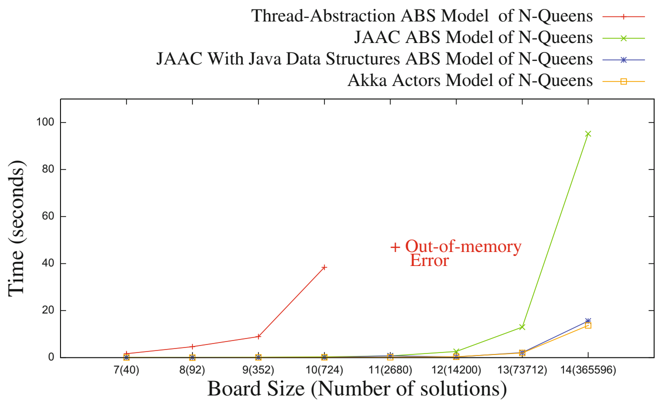

Fig. 3. Results for the N-Queens problem using two different coroutine approaches

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 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.

6 Conclusions

In this paper we have formally described a translation scheme together with a Java library for efficiently simulating the behaviour of ABS coroutines. The coroutine abstraction is a powerful programming technique in a software development context. Having a scalable JVM library with support for coroutine emulation gives us 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.

We plan to extend the library to statically type-check the message submitted via the **send** method in order to prevent the user from running unwanted code on the actors. To this end we already provide a syntactic sugar in Scala for an asynchronous invocation that can be used directly to send a message to an actor and restricts the messages to methods supported by the actor in question.

We also plan to implement a foreign language interface for ABS such that existing libraries may be used directly in the ABS model. This requires extending the type-checker to verify correct foreign types in ABS source code. This in turn would allow for creating both a model on which formal analysis tools can be applied and that would be scalable and efficient once deployed.

References

1. Azadbakht, K., de Boer, F.S., Serbanescu, V.: Multi-threaded actors. In: Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, 8–9 June 2016, pp. 51–66 (2016). https://doi.org/10.4204/EPTCS.223.4
2. Boer, F.D., et al.: A survey of active object languages. ACM Comput. Surv. **50**(5), 76:1–76:39 (2017). https://doi.org/10.1145/3122848
3. Clarke, D., Johnsen, E.B., Owe, O.: Concurrent objects à la carte. In: Dams, D., Hannemann, U., Steffen, M. (eds.) Concurrency, Compositionality, and Correctness. LNCS, vol. 5930, pp. 185–206. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11512-7_12
4. Commons, A.: Javaflow. http://commons.apache.org/sandbox/javaflow
5. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71316-6_22
6. Din, C.C., Bubel, R., Hähnle, R.: KeY-ABS: a deductive verification tool for the concurrent modelling language ABS. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 517–526. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_35

7. Flores-Montoya, A.E., Albert, E., Genaim, S.: May-happen-in-parallel based deadlock analysis for concurrent objects. In: Beyer, D., Boreale, M. (eds.) FMOODS/-FORTE -2013. LNCS, vol. 7892, pp. 273–288. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38592-6_19
8. Giachino, E., Laneve, C., Lienhardt, M.: A framework for deadlock detection in core ABS. Softw. Syst. Model. **15**(4), 1013–1048 (2016)
9. Haller, P., Odersky, M.: Scala actors: unifying thread-based and event-based programming. Theor. Comput. Sci. **410**(2), 202–220 (2009)
10. Imam, S.M., Sarkar, V.: Savina-an actor benchmark suite: enabling empirical evaluation of actor libraries. In: 4th International Workshop on Programming based on Actors Agents and Decentralized Control, pp. 67–80. ACM (2014)
11. Jangid, M.: Kotlin the unrivalled android programming language lineage. Imp. J. Interdiscip. Res. **3**(8) (2017), http://imperialjournals.com/index.php/IJIR/article/view/5491
12. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: a core language for abstract behavioral specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25271-6_8
13. Kamburjan, E., Hähnle, R.: Uniform modeling of railway operations. In: Artho, C., Ölveczky, P.C. (eds.) FTSCS 2016. CCIS, vol. 694, pp. 55–71. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-53946-1_4
14. Kamburjan, E., Hähnle, R.: Deductive verification of railway operations. In: International Conference on Reliability, Safety and Security of Railway Systems, pp. 131–147. Springer (2017)
15. Schäfer, J.: A programming model and language for concurrent and distributed object-oriented systems. Ph.D. thesis, University of Kaiserslautern (2011)
16. Serbanescu, V., Nagarajagowda, C., Azadbakht, K., de Boer, F., Nobakht, B.: Towards type-based optimizations in distributed applications using ABS and JAVA 8. In: Pop, F., Potop-Butucaru, M. (eds.) ARMS-CC 2014. LNCS, vol. 8907, pp. 103–112. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13464-2_8
17. Srinivasan, S., Mycroft, A.: Kilim: isolation-typed actors for java. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 104–128. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70592-5_6
18. Storm, E.: Scala coroutines. http://storm-enroute.com/coroutines/
19. Taura, K., Matsuoka, S., Yonezawa, A.: ABCL/f: a future-based polymorphic typed concurrent object-oriented language - its design and implementation. In: Proceedings of the DIMACS Workshop on Specification of Parallel Algorithms, pp. 275–292. American Mathematical Society (1994)
20. Wong, P.Y.H., Albert, E., Muschevici, R., Proença, J., Schäfer, J., Schlatte, R.: The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. Int. J. Softw. Tools Technol. Transf. **14**(5), 567–588 (2012). https://doi.org/10.1007/s10009-012-0250-1
21. Yonezawa, A., Briot, J.-P., Shibayama, E.: Object-oriented concurrent programming in ABCL/1. In: Agha, G., Igarashi, A., Kobayashi, N., Masuhara, H., Matsuoka, S., Shibayama, E., Taura, K. (eds.) Concurrent Objects and Beyond. LNCS, vol. 8665, pp. 18–43. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44471-9_2