Software development by abstract behavioural specification

Serbânescu, V.

Chapter 6

ABS Runtime as a Library API

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

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

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

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

6.1 JAAC API through an Example Program

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

Listing 6.1: Definition of an actor

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

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

Listing 6.2: Future class with its helper methods

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

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

6.1.1 Library Methods.

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

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

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

Listing 6.3: API exposed by an actor

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

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

Listing 6.4: Spawn Method Intended Usage

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

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

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

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

Listing 6.5: Sending an Asynchronous Call

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

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

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

Listing 6.6: getSpawn Method Intended Usage

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

6.1.2 Call Stack and Priorities.

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

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

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

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

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

6.2 ASCOOP Scala API

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

6.2.1 Actors Typed with Interfaces

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

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

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

Listing 6.9: Actor Definition

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

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

Listing 6.8: Ping-Pong Interfaces Definition

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

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

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

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

Listing 6.10: Method Definition Using Message Handler

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

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

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

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

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

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

Listing 6.11: Awaiting on a Boolean Condition

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

Listing 6.12: Sieve: Returning Values using Futures

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

6.2.2 Actors with Integrated Futures

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

Completed Futures and Delegated Futures

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

Listing 6.13: Sieve Main Method

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

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

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

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

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

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