Software development by abstract behavioural specification

Serbânescu, V.

Chapter 3

Abstract Behavioural Specification Language

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

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

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

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

by executing its corresponding method.

3.1 Programming to Interfaces in ABS

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

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

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

3.2 Algebraic Data Types (ADT)

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

Listing 3.1: Prime Sieve Case Study in ABS

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

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

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

Listing 3.2: Data Type Maybe in ABS

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

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

Listing 3.3: Declaring an ADT in ABS

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

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

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

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

Listing 3.4: Constructing a recursive type in ABS

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

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

Listing 3.5: Pattern Matching in ABS

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

3.3 Asynchronous Model

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

3.3.1 Concurrent Object Groups (COGs)

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

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

3.3.2 Await and Get Construct

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

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

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

Listing 3.6: Example of a Pool of Workers

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

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

Listing 3.7: ABS Await sugared syntax

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

3.4 Synchronous Calls

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

Listing 3.8: Synchronous Call in ABS

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

3.5 Timed ABS

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

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

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

Listing 3.9: Current Time Function in ABS[absb]

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

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

Listing 3.10: Timed-ABS Statements

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

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

Listing 3.11: Case Study with Timed-ABS

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

3.6 Deployment Components

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

3.6.1 Resource Types

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

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

Listing 3.12: Defining a Deployment Component

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

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

Listing 3.13: Defined Resources

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

3.6.2 Resource Consumption

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

Listing 3.14: Resource Usage by an Instruction

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

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

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

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