Asynchronous Programming in the Abstract Behavioural Specification Language

Azadbakht, K.

Chapter 1

The ABS Language

1.1 Introduction

Over decades, execution of software has shifted from local computer programs with low processing power to execution of many computationally-intensive programs with massive data processing and transferring, which are physically distributed and interconnected. Execution of such programs demands efficient, rigorous software design and implementation, and powerful underlying computing and communication resources.

According to the Moore's law for decades, hardware manufacturers could double the number of transistors on a chip roughly every two years, which has been leading to proportional speed-up in the processing of a sequence of instructions for a *unicore* chip. Also in order to achieve the above speed-up, hardware manufacturers have been trying for long to reduce the sizes of transistors and the distance between them. However, the feasibility of such proportional speed-up has reached to its limit, mainly because of physical restrictions. In [8], three main obstacles are discussed that prevent the hardware manufacturers from sustaining the growing speed-up expected by Moore's law: first, the linear increase in clock frequency of a unicore chip gives rise to quadratic energy consumption. Second, such a higher-frequency processor generates more heat than the current cooling technologies can dissipate. Third, the fine-granular parallelism gained from *instruction-level parallelism* (ILP) in the streams of a single instruction seems to have reached its limit. A more fundamental obstacle is, however, that the flow of information in a computer system with a single computational core is ultimately limited to the speed of light. Therefore, the continuous effort has reached to a threshold that the production of faster unicore processors is no longer economically-viable.

Because of the above obstacles, the hardware manufacturers started prototyping and producing *multicore* (and manycore) processors as a new computer architecture around 2005 [71]. On these processors, there are multiple computational cores with dedicated and shared caches that operate on a shared memory and can process

multiple program instructions in parallel. However, a sequential program can be executed on a unicore processor with a specific frequency as fast as on a multicore processor with the same frequency, disregarding the number of cores.

The underlying idea of using a multicore processor is to improve performance by harnessing the processing power of its constituent cores. To this aim, by *parallel programming*, the *workload* is divided evenly in form of tasks which are assigned to the cores. A common parallel program (beside, e.g., data-parallelism and graphics processing) involves *communication* among the tasks (e.g., *synchronization* on a data provided by another task). With the advent of chips with higher number of cores, however, the programming means of parallelism and communication also needs to scale. As an example, the *multi-threading* in Java can be applied to programs with a few number of threads executed on the current multicore processors. However, reasoning about the correctness of multi-threaded programs is notoriously difficult in general [67], especially in the presence of a shared mutable memory which can result in software errors due to data race and atomicity violations. The problem escalates when the future multicore processors come into the picture, and thus the need for a modeling language that enables scalable parallel and distributed programming still persists.

Abstract Behavioural Specification (ABS) is a language for designing executable models of parallel and distributed object-oriented systems [48], and is defined in terms of a formal operational semantics which enables a variety of static and dynamic analysis techniques for the ABS models, e.g., deadlock detection [14,39], verification [30] and resource analysis [5]. Moreover, the ABS language is executable which means the user can generate executable code and integrate it to production — currently backends have been written to target Java, Erlang, Haskell [20] languages and ProActive [46] library.

The ABS language originated from the Creol modeling language which is in-turn influenced by SIMULA, the first object oriented language. The language is generally regarded as a modeling language rather than a programming language with the aim of software production. The main purpose of ABS is thus to construct a (usually abstract) model of the system-to-be, whose different properties can be reasoned about based on different techniques on the underlying formalism. Nevertheless, the before-mentioned ABS backends provide libraries of data structures for the language, and considering the executable nature of the ABS models (and the similarity to Java), these models can be re-used as a starting point for the software production.

ABS at the data layer is a purely functional programming language, with support for pure functions (i.e., functions that disallow side-effects), parametrically polymorphic algebraic datatypes (e.g., `Maybe<A>`) and pattern matching over those types. At the object layer sits the imperative layer of the language with the Java-reminiscing class, interface, method and attribute definitions. It also attributes the notion of *concurrent object group* (cog), which is essentially a group of objects which share control.

Therefore the language is comprised of two layers: 1) the *concurrent object layer*, an imperative object oriented language that captures the concurrency model, communication, and synchronization mechanisms of the ABS, and 2) the *functional layer*, a functional language which is used for modeling data.

From another perspective, ABS adheres to the *Globally Asynchronous Locally Sequential* model of computation. A cog forms a local computational entity, which is based on synchronous internal method activations. All objects inside a cog, which share a thread of control, can synchronously call the methods of objects inside the same group. However communication between the objects of different cogs is, in principle, asynchronous. The behavior of a cog is thus based on cooperative multitasking of external requests sent to the constituent objects.

1.2 ABS and Other Languages

The actor model has gained attention as a concurrency concept since, in contrast to thread-based concurrency, it encapsulates control flow and data. Prominent examples are Erlang [6] based on a functional programming paradigm and Akka actors [1] integrated into a modern object oriented language.

ABS, unlike the general notion of actors as in, e.g., Erlang and Akka Actors, is statically typed and supports a *programming to interface* discipline. Therefore a message, which represents an asynchronous method invocation, is statically checked if the called method on an object is supported by the interface that is implemented by the corresponding class, which gives rise to a type safe communication mechanism that is compatible with standard method calls. This also forces the fields of an object to be private, thus avoiding the incidents of reference aliasing. Unlike Java, objects in ABS are typed exclusively by interface with the usual nominal subtyping relations — ABS does not provide any means for class (code) inheritance.

In contrast to the *run-to-completion* mode of method execution, e.g., in Rebeca [69] and Akka Actors, ABS further provides the powerful feature of *cooperative scheduling* which allows an object to suspend in a controlled manner the current executing method invocation (also known as process) and schedule another invocation of one of its methods. This novel mechanism enables combining active and reactive behaviors within an object, and avoiding a common scenario where waiting for the resolution of certain messages requires the actor to be completely blocked for other activities, thus enhancing the potential concurrency and parallelism.

1.3 Model of Concurrency

The model of execution in ABS is based on the actor model [47] which is a model of concurrency with the following characterization: 1) an actor has an identity

[1] `https://akka.io`

and encapsulates its constituent data and a single thread of control, and 2) the communication between actors are via asynchronous message passing. The receiving messages are queued in the *mailbox* of the actor. The thread activates messages from the mailbox, i.e., dequeues the message and executes an internal computation correspondingly.

The identity of an actor is shared either with its creator upon creation , or it is explicitly sent as a parameter of a message. Also, there is no pre-defined global ordering where a sending actor can prioritize its message over the ones of other actors.

Active objects [55,78] are descendants of actors where messages are asynchronous method invocations and the operations of an object are defined in terms of methods that are encapsulated and exposed via interfaces. The notion of active object is an alternative to the traditional object's built-in mechanisms for multi-threaded and distributed programming, where support for structuring and encapsulation of the state space, higher-level communication mechanisms and a common model for local and distributed concurrency is missing [63].

In addition to the above, ABS active objects feature synchronization on futures and cooperative scheduling of internal processes, which are elaborated as follows:

Synchronization on futures A future of a specific type is used as a unique reference to the return value of an asynchronous method call with the same return type. The future is *unresolved* if the corresponding return value is not yet available. It is *resolved* otherwise and stores the value. The query to retrieve the value of a future (via *get* operation) synchronizes the active object on the resolution of the future, namely, the active object is blocked until the future is resolved. The value is then retrieved.

Cooperative scheduling ABS also features cooperative scheduling, where the active process of an object can deliberately yield control such that a process from the set of suspended processes of the object can be activated, i.e., explicit cooperation in contrast to the common mechanisms of thread preemption. The "release of control" happens in explicit places of the ABS model, where the potential concurrent interleavings of different processes are defined. These places are specified by **`await`** and **`suspend`** statements, for conditional and unconditional release, respectively.

1.4 Language Definition

In the following, the ABS syntax of Concurrent Object layer of Core ABS is given. We also briefly describe semantics of each syntactic structure. The syntax and semantics for the functional layer is omitted as it is not the focus of this thesis. In [48], the full formal definition of Core ABS is given.

Syntactic categories	*Definitions*		
C, I, m in Names	P	::=	$\overline{Dd}\ \overline{F}\ \overline{IF}\ \overline{CL}\ \{\overline{T}\ \overline{x};\ s\}$
g in Guard	IF	::=	**interface** $I\,\{\,\overline{Sg}\,\}$
s in Statement	CL	::=	**class** $C\,[(\overline{T}\,\overline{x})]\,[$**implements** $\overline{I}]\,\{\,\overline{T}\,\overline{x};\ \overline{M}\}$
	Sg	::=	$T\ m\ (\overline{T}\,\overline{x})$
	M	::=	$Sg\,\{\,\overline{T}\,\overline{x};\ s\,\}$
	g	::=	$b \mid e? \mid g \wedge g$
	s	::=	$s; s \mid x = rhs \mid$ **suspend** $\mid$ **await** $g \mid$ **skip**
		$\mid$	**if** $b\,\{s\}\,[$**else** $\{s\}] \mid$ **while** $b\,\{s\} \mid$ **return** e
	rhs	::=	$e \mid$ **new** [**local**] $C[(\overline{e})] \mid e!m(\overline{e}) \mid e.m(\overline{e}) \mid x.$**get**

Figure 1.1: Core ABS syntax for the concurrent object level [48]

The Concurrent Object Layer of Core ABS is given in Figure 1.1. In this grammar, an overlined entity $\overline{v}$ denotes a list of v. The IF denotes an interface. Each interface has a name I and a list of method signatures Sg. A class CL has a name C, interfaces $\overline{I}$, formal parameters and state variables $\overline{x}$ of types $\overline{T}$, and methods $\overline{M}$. (The *fields* of the class are both its parameters and state variables).

When the class is instantiated, the number, order and type of actual parameters must match those of formal parameters. This also applies to the synchronous and asynchronous method invocations. In ABS the object references are typed only with interfaces (i.e., programming to interfaces). A reference variable with type I, where I is an interface name, can hold a reference to an instance of a class C, provided that C implements I.

A method signature Sg declares the return type T of a method with name m and formal parameters $\overline{x}$ of types $\overline{T}$. M defines a method with signature Sg, local variable declarations $\overline{x}$ of types $\overline{T}$, and a statement s. Statements can have access to the fields of the current class, local variables, and the method's formal parameters. The state of a method is its local variables and the fields of the class it belongs to. A program's main block is a method body $\{\overline{T}\ \overline{x}; s\}$.

Right-hand side expressions rhs include object creation within the same cog (written "**new local** $C(\overline{e})$") and in a fresh cog (written "**new** $C(\overline{e})$"), method invocations, and expressions e. Statements are standard for sequential composition, assignment, **skip**, **if**, **while**, and **return** constructs. The statement **suspend** unconditionally releases the processor, suspending the active process. In **await** g, the guard g controls processor release and consists of Boolean conditions b and return tests x? (see below). If g evaluates to false, the processor is released and the process *suspended*. When the processor is idle, any enabled process from the object's pool of suspended processes may be scheduled. Consequently, explicit signaling is redundant in ABS.

Besides the common synchronous method calls to passive objects $e.m(\bar{e})$, ABS introduces the notion of concurrent objects (also known as active objects). These concurrent objects interact primarily via asynchronous method invocations and fu-

tures. An asynchronous method invocation is of the form of $x = e!m(\overline{e})$, where e is an object expression (i.e., an expression typed by an interface), and x is a future variable used as a reference to the return value of the asynchronous method call m, and thus the caller can proceed without blocking on the call. The method invocation itself will generate a process which is stored in the mailbox (process queue) of the object e. Futures can be passed around and can be queried for the value they contain.

There are two operations on a future expression e for synchronization on external processes in ABS. The operation $x = e.\textbf{get}$ blocks the execution of the active object until the future expression e is resolved, where its value is assigned to x. On the other hand, the statement **await** $e?$ results in releasing control by the process, where the future expression e is unresolved. This allows for scheduling another process of the same active object and as such gives rise to the notion of *cooperative scheduling*: releasing the control cooperatively so another enabled process can be (re)activated. ABS provides two other forms of releasing control: the **await** b statement which will only re-activate the process when the given boolean condition b becomes true (e.g. **await** $this.x == 3$), and the **suspend** statement which will unconditionally release control to the active object. Note that the ABS language specification does not fix a particular scheduling strategy for the process queue of active objects as the ABS analysis and verification tools will explore many (if all) schedulability options; however, ABS backends commonly implement such process queues with FIFO ordering.

When executed between objects in *different* cogs, then the statement sequence $x = o!m(\overline{e});\ v = x.\textbf{get}$ amounts to a blocking, *synchronous call* and is abbreviated $v = o.m(\overline{e})$. In contrast, synchronous calls $v = o.m(\overline{e})$ *inside* a cog have the reentrant semantics known from, e.g., Java method invocation. The statement sequence $x = o!m(\overline{e});$ **await** $x?;\ v = x.\textbf{get}$ codes a non-blocking, *preemptable call*, abbreviated **await** $v = o.m(\overline{e})$. In many cases, these method calls with *implicit* futures provide sufficiently flexible concurrency control to the modeler.

1.5 Distributed ABS

The ABS also supports distributed models at the implementation level, a cloud extension to the ABS standard language, as implemented in [20]. This extension introduces the *Deployment Component* (DC), which abstracts over the resources for which the ABS program gets to run on. In the simplest case, the DC corresponds to a Cloud Virtual Machine executing some ABS code, though this could be extended to include other technologies as well (e.g. containers, microkernels). The DC, being a first class citizen of the language, can be created (`DC dc1 = new AmazonDC(cpuSpec,memSpec)`) and called for (`dc1 ! shutdown()`) as any other ABS concurrent object. The DC interface tries to stay as abstract as possible by declaring only two methods `shutdown` to stop the DC from executing ABS code

while freeing its resources, and `load` to query the utilization of the DC machine (e.g. UNIX load). Concrete class implementations to the DC interface are (cloud) machine provider specific and thus may define further specification (CPU, memory, or network type) or behaviour.

Initially, the Deployment Component will remain idle until some ABS code is assigned to it by creating a new object inside using the expression `o = [DC: dc1] new Class(...)`, where *o* is a so-called remote object reference. Such references are indistinguishable to local object references and can be normally passed around or called for their methods. The ABS language specification and its cloud extension do not dictate a particular Garbage Collection policy, but we assume that holding a reference to a remote object or future means that the object is alive, if its DC is alive as well.

1.6 Example

In Figure 1.2 we show a model of a thread pool in ABS with a given interface and a fixed number of threads. The thread pool retrieves and executes tasks (asynchronous method invocations) that are stored in its queue and returns their corresponding futures. Once a thread terminates the execution of a task, it is assigned another task from the queue if the queue is not empty, or remains idle otherwise.

The thread pool consists of a set of active objects (i.e., instances of `Member`) that represent the threads, and one manager (i.e., an instance of `Threadpool`) that manages the thread pool. Both the manager and the members provide the same interface `Service`, which denotes the methods that can be executed by the thread pool. In this example, `Service` provides two method signatures `m1` and `m2`. The intended behaviour of these methods is implemented in `Member`. The implementation of the methods with the same signatures in `Threadpool`, however, involves relegating the call to an available member's corresponding method. It also consistently updates the list of available members. For instance, `m1` in `Threadpool` awaits the availability of a `Member` instance, removes the member from the list of available members, calls the corresponding `m1` on the member, awaits on the future resulting from the call, and finally adds the member to the list of available members again as the task is finished.

```
module threadpool;

interface Service
{
      T1 m1(...);
      T2 m2(...);
      ...
}

class Member implements Service {
      T1 m1(...)
      {
            // implementation of m1
      }
      T2 m2(...)
      {
            // implementation of m2
      }
      ...
}

class Threadpool(Int count) implements Service{

      List<Service> available = Nil;

      {
            Int i = 1;
            while(i<=count) {
                  Service thread = new Member();
                  available = cons(thread, available); i = i + 1;
            }
      }

      T1 m1(...)
      {
            Service thread = this.getThread();
            Future<T1> f = thread!m1(p1);
            // p1 is a list of arguments received by m1
            await f?;
            available = cons(thread, available);
            return f.get;
      }

      T2 m2(...)
      {
            Service thread = this.getThread();
            Future<T2> f = thread!m2(p2);
            // p2 is a list of arguments received by m2
            await f?;
            available = cons(thread, available);
            return f.get;
      }

      ...

      Service getThread() {
            await available != Nil;
            Service thread = head(available);
            available = tail(available);
            return thread;
      }
}

{ // main block
      Service threadpool = new Threadpool(numberOfThreads);
      Future<T1> f = threadpool!m1(...);
      ...
}
```

Figure 1.2: A model of thread pool