Software development by abstract behavioural specification

Serbânescu, V.

Chapter 4

ABS Runtime in Java

This chapter illustrates the overall architecture of the proposed runtime and thread management system. It gives implementation details about maintaining actor semantics in the presence of coroutines. It also covers the heuristics used for propagations and delegation of futures in the context of an actor programming model. In the second part of this chapter the details of the runtime support for the extension of ABS concerning time and resources is presented.

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 [AdBS16]. 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 API is covered in Chapter 6). The overall architecture of the system is based on the following control flow: Every actor submits a main task to the thread pool which iteratively selects an enabled task from its queue and runs it. The runtime then uses 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 modeling of both asynchronous method calls and suspension of a method invocation. This suspension, whether it is 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.

4.1 Evolution of the Runtime Schemes

This section investigates the evolution of the scheme used to implement cooperative scheduling in the Java Runtime environment. It starts from a very trivial approach to using several libraries and features that the Java SDK provides. Each solution covers four main features that have an overall performance and scalability impact and have been improved on each iteration:

- Creation of the actor itself;
- Generation of asynchronous calls and message passing;
- Releasing control or suspension of a call when encountering an await;
- Saving the call stack of a synchronous call upon releasing control.

The development of the runtime focuses on efficient thread management for running multiple actors on a system. It uses the Executor interface that facilitates thread programming. The objects that implement

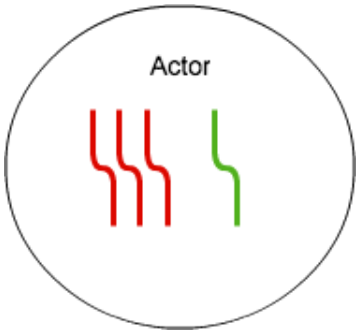

Figure 4.1: Basic process-oriented approach

this interface provide a queue of tasks and an efficient way of running tasks on multiple threads. Throughout this section we will use the terms executor and thread pool interchangeably to refer to this feature. Another notion discussed in our solution was introduced starting with the Java 8 version, and it is the construct for a lambda expression. This allows modeling a method call as a Runnable or Callable and it is referred to as a **task**.

4.1.1 Every asynchronous call is a thread and each actor has a lock

The trivial straightforward approach for implementing cooperative scheduling in the library is for an asynchronous call to generate a new native thread with its own stack and context. We would then model each actor as an object with a lock for which threads compete. The disadvantage here is that this lock-per-actor must be checked by every message handler upon start, and freed upon completion. Whenever an await statement occurs the thread would be suspended by the JVM's normal behavior. When the release condition is enabled, a suspended thread would become available and in turn compete with the other threads in order to execute on the actor. The main drawback of this approach is the large number of threads that are created, which restricts any application from having more method calls than the number of live threads that the memory of the system can handle. Figure 4.1 provides an illustration of this base *process-oriented* approach. Here the idea is that the threads in the circle belong to one actor, thus sharing a lock(the green thread holds the lock, while the red ones are waiting to acquire it). The four main features that impact the system are implemented as follows:

- The actor is initialized as an object with a lock to model actor semantics that allow one method (message) to be executed;
- Asynchronous calls are created as new threads that compete for the lock creating a performance bottleneck that leaves many threads suspended.
- Suspension of a call suspends the thread adding to more memory overhead.
- The call stack of a synchronous call is saved in the suspended thread.

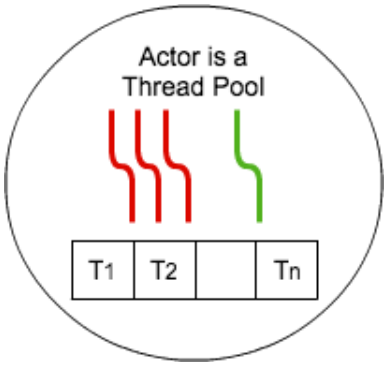

Figure 4.2: Actor-as-executor approach

4.1.2 Every actor is a thread pool.

To reduce the number of live threads in an application, we can model each invocation as a task using lambda expressions and organize each actor as a thread pool. This gives the actor an implicit queue to which tasks are submitted. We obtain a small reduction in the number of threads corresponding to the number of tasks that have been submitted, but not started. Once they are started the threads still have to compete for the actor's lock in order to execute, but the number of live threads can be restricted to the number of threads allowed by each Thread Pool, while the rest of the invocations remain in the thread pool queue as tasks. This approach is illustrated in Figure 4.2. The implementation of the four Actor features is as follows:

- The actor is initialized as an object with a lock and a thread pool preserving actor semantics similar to the previous approach;
- Asynchronous calls are created as lambda expressions and assigned new threads that compete for the lock once they start. The number of started threads is limited by the thread pool size slightly reducing the number of suspended threads competing fo the actor lock;
- Suspension of a call suspends the thread unchanged from the previous approach;
- The call stack of a synchronous call is still saved in the suspended thread.

4.1.3 Every system has a thread pool.

In the previous two approaches we modeled the concept of an actor being restricted to one task at a time by introducing a lock on which threads compete. However as all invocations are modeled as tasks that don't need a thread before they start, there is no point in starting more than one task only to have it stuck on the actor's lock. Therefore we introduce *one thread pool per system* or *the system's executor*, and instead of submitting the messages directly to the thread pool, we add an indirection by storing them into an explicit queue first and introduce as a second phase the submission to the thread pool. We assign a separate task for each actor to iterate through its associated queue and submit the next available message to the system thread pool. We will refer to this task from now on as the `Main Task` of an actor. This removes the requirement to store every message handler as a thread, after it starts and attempts to acquire a lock, saving a task as data in the heap instead. However it comes at the cost of having to manage message queues manually as we need an explicit queue for all the messages that have not yet been submitted (to the thread pool).

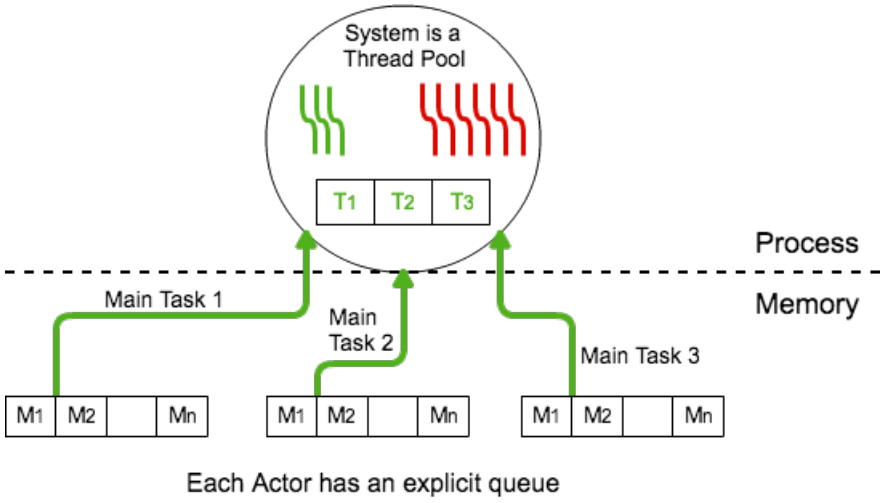

Figure 4.3: System as a thread pool

When cooperative scheduling occurs, the executing task of an actor will be suspended and therefore still live in the system as a thread so the application's live threads will be equal to the number of `await` statements in the program. The system's executor can dynamically adjust the number of available threads to run subsequent available tasks, but the application will then still be limited by the maximum number of suspended threads that can exist in the main memory. Furthermore, we still require a lock to ensure that, upon release, the resuming thread will compete with the `Main Task` associated to the actor from which it originated. This limitation is imposed to preserve actor semantics. This design is presented in Figure 4.3 and it is important to observe the migration of messages into memory and that the red threads are only tasks that have been suspended by an `await`. We observe the following changes in the four features:

- The actor is initialized as an object with a lock for threads that will be suspended by `await`, an explicit queue and an implementation of the `Main Task`, removing the thread pool-per-object;
- Asynchronous calls are created as new tasks that will be run by the `Main Task` one by one, removing the need for suspending thread to preserve actor semantics.
- Upon suspension of a call the `Main Task` restarts to iterate the actor's queue. The current call is parked as a thread that will compete with the `Main Task` once it is resumed;
- The call stack of a synchronous call is still saved in the suspended thread.

4.1.4 Fully asynchronous environment.

In the absence of synchronous calls, to eliminate the problem of having live threads when cooperative scheduling occurs, we can also use lambda expressions to turn the continuation of an await statement (which is determined by its lexical scope) into an internal method call and pass its current state as parameters to this method. Essentially what we do is allocate memory for the continuation on the heap, instead of holding a stack and context for it. We will benchmark this trade-off between the memory footprint of a native thread and the customized encoding of a thread state in memory. As there are no more suspended threads in the system to compete with the actor's `Main Task`, we can eliminate the lock per actor. The four features that affect the system are now as follows:

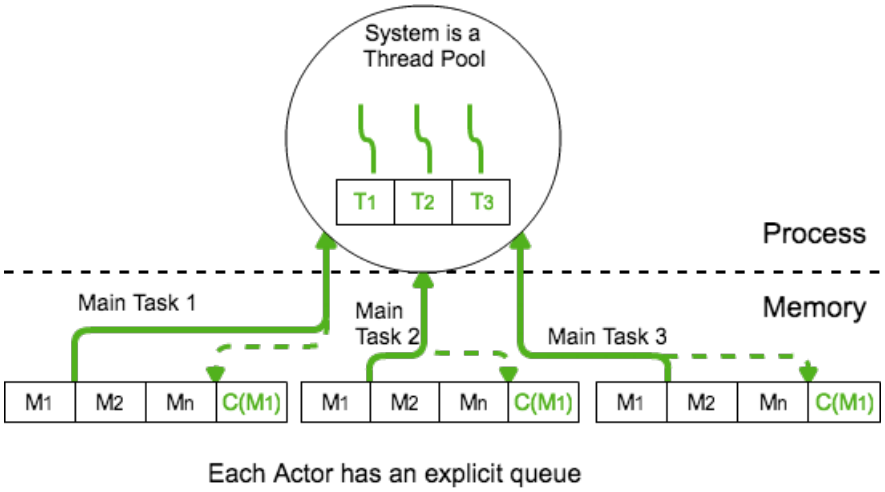

Figure 4.4: Full data-oriented approach

- The actor is initialized as an object with an explicit queue and an implementation of the `Main Task`, but without a lock;
- Asynchronous calls are created as new tasks that will be run by the `Main Task`.
- Suspension of a call first saves the continuation as a lambda expression guarded by the release condition, and the `Main Task` continues to iterate through the queue. No new threads are created any more, the suspended task is saved inside the queue of the actor;
- The process of saving the call stack of a synchronous call has to be done explicitly and is detailed in the next paragraph.

Synchronous calls context. In the presence of synchronous calls, the problem that remains is how to save this call stack without degrading performance? To do this we can try to alter the bytecode to resume execution at runtime from a particular point, but we want our approach to be independent of the runtime and be extensible to other programming languages. Therefore we try to approach the problem differently; if we can turn a continuation, that does not originate from a stack of calls, into a task, is it possible to extend this to synchronous calls as well? We know that this issue arises when methods that contain an `await` statement are called synchronously like the example in Listing 3.8. When the `await` statement is encountered, the continuation consists of both the lexical scope of the release statement that is followed by the complete synchronous call chain that has been generated(i.e. $C(M_1)$ in Figure 4.4). At compile time, when translating source-to-source from ABS to Java, we can identify all of these occurrences from the ABS code. We can mark them and transform them into asynchronous calls followed by an `await` statement on the future generated from the call. Now we can use the same approach for translating these continuations into tasks using lambda expressions and thus eliminating any suspended threads in the system. The final model of our solution is presented in Figure 4.4.

More explicitly in terms of ABS code, a fully asynchronous environment means to change a synchronous call `x = this.m( );` into an asynchronous call plus an await like `f = this!m( ); x = await f?;`. There are two problems here. First, we still need to make sure that *m( )* must be the exact next method that will run. Second, when *m( )* finishes, the actor scheduler must immediately schedule the method that is waiting for its result. Both can be implemented by changing the non-deterministic

behavior of the `Main Task`. There are certain constrains that we have to impose to preserve the chain of synchronous calls. For these we introduce priorities and strictness to the spawned tasks, which are part of the implementation details that are discussed in the next section.

4.2 Implementation Details

To summarize, 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 [Sch11]. 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. Therefore 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 the general mechanism of spawning and completion of tasks and then details of task scheduling and execution.

4.2.1 Task Generation and Completion

This section introduces two abstractions defined as classes called Task and `Future` which are used as part of the mechanism for spawning tasks. We describe in more detail some of the implementation issues of these non-blocking futures and the new delegation mechanism through Listing 4.2.

Non-Blocking Futures Every asynchronous call creates a new task as an instance of the class Task(Listing 4.1) and stores it into the task queue of the actor callee. Task instances model tasks and spawned sub-tasks representing messages and continuations. The actual computation is captured in the `task` field which is a `Callable` returning a Future. The enabling condition is only used for sub-tasks; the type `Guard` represents the possible guards presented in the ABS semantics: either a boolean expression over the actor state, a future or a symbolic time duration. In all these cases the field `resultFuture` will contain a newly created instance of `Future`(Listing 4.2) which uniquely identifies this task and which is returned to the caller. This is special future type defined in the library that cannot block a thread when trying to retrieve its result. Upon termination of a task, the return value will be wrapped in an `Future` instance with a set `completed` flag and an assigned `value`. In our approach based on the notion of a Task, futures no longer provide a blocking `get` operation that suspends a thread in the system. Instead we provide the notion of a suspension mechanism at the level of a Task which does not block the `Main Task` of an actor. Thus it allows for cooperative scheduling of other tasks by the `Main Task` of the actor.

Listing 4.1: Simplified implementation of Task class in Java.

```
1  public class Task<V> implements Serializable, Runnable {
2   protected Guard enablingCondition = null;
3   protected final Future<V> resultFuture;
4   protected Callable<Future<V>> task;
5
6   //implementation and functionality
7
8   public void run( ) {
9     try {
10      task.call( ).backLink(resultFuture);
11      // upon completion, the result is not necessarily ready
12    } catch (Throwable e) {
13      e.printStackTrace( );
14      throw new RuntimeException(e);
15    }
16  }
17 }
```

Delegation. As discussed in Section 4.1.4, a method may delegate the completion of its future to another task (its continuation). Since every task has an associated future, this gives rise to a doubly-linked list of futures. Whenever the `Main Task` runs a Task from its queue it will issue a statement task.call( ).backLink(resultFuture) (see Listing 4.1). Thus, the currently running Task, once it returns, sets the dependant of its associated future to the provided back-link (line 15). If the task successfully completed its future without delegation it can complete its dependant as well (line17). On the other hand, in case of a delegation delegation, its associated futures future will become itself a dependant to the delegated future.

Once a `complete` method is called, it will recursively propagate completion through the whole delegation chain (line 26). After that it will notify its own awaiting actors accordingly (line 28). Thus the above scheme only involves backward propagation of completed futures and avoids searching through the delegation chain when checking a particular future.

It is important to note that for every Task at least one level of such delegation happens. That is because the creation of a Task instance and its execution happen at different points in time. When the Task instance is instantiated, so is its `resultFuture` field, which is returned immediately issuer of the call or creator of a sub-task. The actor may await the completion of this future. Once the future at the end of the delegation chain actually has the computation result, the completion and notification procedure will run as explained above. As a final note, the `isDone` method and `getOrNull` methods are used internally and it is ensured that the value of the future is read only when the future is ready. This scheme fully integrates futures with the mechanism of spawning new tasks and supports the delegation of the computation of return values.

4.2.2 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 4.3. Inside the class there is a `taskQueue` which holds all tasks of an actor. Tasks are defined as instances of class Task

Listing 4.2: Details of the Future class in Java

```
1  public class Future<V> {
2
3    private V value = null;
4    private Future<V> dependant = null;
5    private AtomicBoolean completed = new AtomicBoolean(false);
6    private Set<Actor> awaitingActors = ConcurrentHashMap.newKeySet( );
7
8    void awaiting(Actor actor) {
9      awaitingActors.add(actor);
10     if (completed.get( ))
11       notifyDependant(actor);
12   }
13
14   void backLink(Future<V> target) {
15     dependant = linkedFuture;
16     if (completed.get( ))
17     dependant.complete(getOrNull( ));
18   }
19
20   void complete(V value) {
21     if (this.completed.get( ))
22       return;
23     this.value = value;
24     this.completed.set(true);
25     if (dependant != null) {
26       dependant.complete(value);
27     }
28     notifyDependants( );
29   }
30
31   boolean isDone( ) {
32     return this.completed.get( );
33   }
34
35   V getOrNull( ) {
36     return this.value;
37   }
38 }
```

(for example on line 37). To allow for concurrent access and an efficient scheduling of these tasks we use a hashmap (`ConcurrentSkipListMap` on line 5) that orders tasks into buckets (queues of tasks defined by assigned priorities).

The implementation defines an inner class `Main Task` which is responsible for selecting an enabled task from the queue (via the `takeOrDie` method) and running it. Execution of an actor is represented by an instance of `Main Task`. Being a `Runnable`, the main task of an actor can be submitted to the system-wide thread pool to a global `ExecutorService` that efficiently manages allocation of available system threads to active instances of `Main Task`. As a naive approach to iterating over the messages in the queue, one could put a while loop in the `run` method of the `Main Task`. However other actors may get no chance to run any task. In our implementation, the `Main Task` submits itself again to the executor service after having processed one task (instead of looping over all enabled tasks in one go). Thus it allows all actors to get a fair chance of executing their `Main Task`. This fairness policy may also be fine-tuned to allow the `Main Task` to execute a fixed chunk of tasks at a time before releasing the thread in order to reduce context switches. and thus actors are put to compete for available threads in a scalable way.

The `Main Task` avoids busy-waiting in cases that all tasks in the queue are disabled. If `takeOrDie` returns false, it means there are no enabled messages and the `Main Task` of this actor simply terminates. Note that the queue may happen to contain only disabled sub-tasks awaiting on boolean conditions; in that case, they can be enabled only via receiving a message from another actor. The `Main Task` is reactivated upon generation of any new task in the queue (line 41). Actor's `Main Task` will thus be reincarnated only upon receiving a new message. To make sure there is no more than one instance of the `Main Task` running for an actor, we use the `mainTaskIsRunning` flag. As the task queue is accessed concurrently by the `Main Task` performing the queue traversal and other actors sending method invocations, it is important to avoid race conditions. A race condition may happen if after `Main Task` 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 `Main Task` will terminate and the new message also creates no new `Main Task`. We avoid this situation by the synchronized blocks in `takeOrDie` and `notRunningThenStart`. Since this is a very rare case, the synchronization here does not affect performance.

In case `Main Task` 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 `Main Task`. 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 `Main Task` if it was terminated. This represents 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.

Actor Life-Cycle. Actors can be created using the **new** keyword just like normal objects. This implies that the actor life-cycle is like a plain Java object and it will be garbage collected when there are no more references to it. In other words, there is no need for any context or a central registry to keep track of actors or to stop actors explicitly. Additionally, if there are messages in an actor's queue, they will be executed even if no external reference to the actor exists. Conversely, when an actor has no messages, it puts no execution load on the system (as discussed in the implementation notes above).

COG support To support the organization of actors into concurrent groups as described in section 3.3.1, the library provides a means for actors to share the `mainTaskIsRunning` flag though the `moveToCOG` method (line 1 in Listing 4.4). This means that only one actor will have its`Main Task` running from the group sharing this flag preserving the COG notion. Certain applications also benifit from actors sharing

Listing 4.3: Details of the Local Actor class in Java

```
1  abstract class LocalActor implements Actor {
2
3  private Task<?> runningTask;
4  private final AtomicBoolean mainTaskIsRunning = new AtomicBoolean(false);
5  private ConcurrentSkipListMap<...> taskQueue = new ConcurrentSkipListMap<>( );
6
7  class MainTask implements Runnable{
8
9    public void run( ) {
10     if (!takeOrDie( )) return;
11       runningTask.run( );
12       ActorSystem.submit(this);
13     }
14   }
15
16   private boolean takeOrDie( ) {
17     synchronized (mainTaskIsRunning) {
18       / *
19       iterate through queue and take one ready task
20       if it exists set it the next runningTask and then
21       * /
22       return true;
23
24       // if the queue if empty or no task is able to run
25       mainTaskIsRunning.set(false);
26       return false;
27     }
28   }
29
30   private boolean notRunningThenStart( ) {
31     synchronized (mainTaskIsRunning) {
32       return mainTaskIsRunning.compareAndSet(false, true);
33     }
34   }
35
36   public final <V> Future<V> send(Callable<Future<V>> message) {
37     Task<V> m = new Task<>(message);
38
39     // add m to the task queue with low priority and no strictness
40     if (notRunningThenStart( )) {
41       ActorSystem.submit(new MainTask( ));
42     }
43     return m.resultFuture;
44   }
45 }
```

their task queues so the library also offers support for this (line 4) although this instruction can be optional when grouping actors together. The move `moveToCOG` method is part of the default constructor of an actor and as such will either create a new COG for the actor if the destination parameter is `null` or place the newly create actor into the destination's COG (their main tasks will share the lock for starting). The library also offers a helper method `sameCOG` such that at any point in the program the user can check if two actors are part of the same group.

Listing 4.4: Methods for COG support in the Local Actor Class

```
1  public void moveToCOG(LocalActor dest) {
2    if(dest!=null) {
3      this.mainTaskIsRunning = dest.mainTaskIsRunning;
4      this.messageQueue = dest.messageQueue;
5    }
6  }
7
8  public boolean sameCog(LocalActor dest) {
9    if(dest!=null ) {
10     return this.mainTaskIsRunning == dest.mainTaskIsRunning;
11   }
12   return false;
13 }
```

Using JVM Garbage Collection The `Main Task` of an actor dies if there are no enabled tasks or sub-tasks in the queue. We explained that Boolean guards may be re-enabled only if another message is received by the actor, therefore, the `Main Task` is reactivated only upon receiving a new message. However, as mentioned in this section, the enabling condition of a sub-task may also depend on the availability of futures. Therefore, in order to avoid actors constantly polling futures, we need a (push) mechanism to allow completed futures to notify the actors awaiting on them. A naive approach would be to keep a global table with the list of actors awaiting every future. Alternatively, we chose to distribute this table into every future to keep track of the actors awaiting its completion.

Whenever an actor creates a sub-task guarded by a future, the actor will automatically be added to the list of `awaitingActors` of that future. This is done via the `awaiting` method. Further, to notify actors of the completion of this future, the completed future sends a special wake-up message to every actor in its awaiting list. As described previously, this will reactivate the `Main Task` in case it has died. The `Main Task` of the actor then continues to process its task queue and possibly will select the newly enabled sub-task from the queue. 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 `Future` class is implemented completely lock-free and therefore the chaining of futures performs very efficiently.

4.2.3 Symbolic Time

In section 3.5 we presented how ABS models abstract time, how messages can be ordered based on how much time has passed since execution of a model has started. This extension is optional in the runtime and as such programs have the possibility to run with or without symbolic time. The extension introduces a new type of guard called `DurationGuard` that represents a new enabling condition for a task that needs to be evaluated before the task can run. Its class definition is presented in Listing 4.5. The class contains the `min` and `max` values for a duration statement as defined by the Timed-ABS model. Furthermore it contains a variable `whenCalled` to determine the logical point of time when the duration statement was called. The logic of the runtime can then use that point to compute the number of time units that need to pass before the guard is satisfied (line 13). The static method `now( )` is explained in the next paragraph in Listing 4.6.

Listing 4.5: Duration Guard Class

```
1  public class DurationGuard extends Guard {
2
3   int whenCalled, min, max;
4
5   public DurationGuard(int min, int max) {
6    this.whenCalled = TimedActorSystem.now( );
7    this.min = min;
8    this.max = max;
9   }
10
11  @Override
12  protected boolean evaluate( ) {
13   return (whenCalled+min)<=TimedActorSystem.now( );
14  }
15 }
```

In the runtime we extended the `ActorSystem` class, which only provided the management of the system-wide thread-pool. The blueprint of the class is presented in Listing 4.6. Part of this class contains support for the resource model and deployment components and is explained in Section 4.2.4. As per the semantics of the timed ABS model, processes that are explicitly suspended by Duration Guards may only execute when all other processes are blocked or suspended on either a future, a boolean state or awaiting resources. To ensure this, we require that the system knows the number of actors running at all time provided by the `runningActors` atomic variable (atomicity is needed as parallel processes that run to completion may update this variable at the same time). The system also provides a clock that measures a central logical time to which all processes have access through the `symbolicTime` atomic variable.

The system keeps an ordered mapping (`awaitingDurations`) of all suspended processes due to time such that it can optimally advance the logical clock until enough time has passed to release the "lowest-time" awaiting process. This is the functionality of the `done` method which is called at the end of every `Main Task` life cycle (see Section 4.2.2). The system first identifies the condition for advancing time (line 15: all processes are suspended). The system then selects the process or processes (actors) that require the shortest period of time to pass in order to be released, logical time advances by that amount and the actors are notified of their release through an `emptyTask` (lines 21 to 28).

4.2.4 Resource Modeling

The resource modeling feature works very similar to the COG support in the library runtime. Just like symbolic time this support is optional and is offered by the `TimedActorSystem` class in Listing 4.6. The

Listing 4.6: Timed Actor System Class

```
1  public class TimedActorSystem extends ActorSystem {
2    private static AtomicInteger symbolicTime = new AtomicInteger(0);
3    private static AtomicInteger runningActors = new AtomicInteger(0);
4    private static ConcurrentSkipListMap<Integer, List<Actor>> awaitingDurations
5                        = new ConcurrentSkipListMap<>( );
6
7    private static ConcurrentLinkedQueue<ClassDeploymentComponent> deploymentComponents
8                        = new ConcurrentLinkedQueue<>( );
9
10   static public int now( ) {
11     return symbolicTime.get( );
12   }
13
14   static public void done( ) {
15     if (runningActors.decrementAndGet( ) == 0) {
16
17       for (ClassDeploymentComponent dc : deploymentComponents) {
18         dc.replenish( );
19       }
20
21       SortedSet<Integer> keys = awaitingDurations.keySet( );
22       int advance = keys.first( );
23       List<Actor> toRealease = awaitingDurations.remove(advance);
24       keys.remove(advance);
25       advanceTime(advance);
26
27       for (Actor a : toRealease) {
28         a.send(ABSTask.emptyTask);
29       }
30     }
31   }
32
33   static void advanceTime(int x){
34     symbolicTime.addAndGet(x);
35   }
36 }
```

ABS standard library directly offers backend independent support for acquiring and awaiting on resources offered by deployment components together with support for a deployment component to self-replenish it resources. These methods work exactly like the semantics summarized in Section 3.6. The only part that is specific to each backend is determining the moment when all resources need to be replenished and this is done each moment symbolic time advances (line 18). All the modeled deployment components are stored in a hashmap for this purpose (line 8). When using the program with resource modeling support a default deployment component with infinite resources is created at the program's entry point. From that point all newly created actors are assigned a deployment component in their constructor. This can either be the same deployment component as the actor that created it, or a new one according to Listing 3.12. The constructor will then call the method in Listing 4.7 to correctly set each actor's deployment component when it is created.

Listing 4.7: Setting a reference for Resource Location in the Local Actor Class

```
1  public void setDC(Actor dc) {
2    this.dc = dc;
3  }
```