Asynchronous Programming in the Abstract Behavioural Specification Language

Azadbakht, K.

Chapter 5

Multi-Threaded Actors

5.1 Introduction

Object-oriented programs organize data and corresponding operations by means of a hierarchical structure of classes. A class can be dynamically instantiated and as such extends the concept of a module. Operations are performed by corresponding method calls on class instances, namely objects. In most object-oriented languages, like Java, method calls are executed by a thread of control which gives rise to a stack of call frames. In a distributed setting, where objects are instantiated over different machines, remote method calls involve a *synchronous rendez-vous* between caller and callee.

It is generally recognized that *asynchronous* communication is better suited for distributed applications. In the Actor-based programming model of concurrency [2] actors communicate via asynchronous messages. In an object-oriented setting such a message specifies a method of the callee and includes the corresponding actual parameters. Messages in general are queued and trigger execution of the body of the specified method by the callee, when dequeued. The caller object proceeds with its own execution and may synchronize on the return value by means of *futures* [27].

In [64] JCoBox, a Java extension with an actor-like concurrency model based on the notion of concurrently running object groups, the concept of coboxes is introduced which integrates thread-based synchronous method calls with asynchronous communication of messages in a *Globally Asynchronous, Locally Sequential* manner. More specifically, synchronous communication of method calls is restricted to objects belonging to the same cobox. Objects belonging to the same cobox share control, consequently within a cobox at most one thread of synchronous method calls is executing. Only objects belonging to different coboxes can communicate via asynchronous messages.

Instead of sharing control, in this chapter we introduce an Actor-based language which features new programming abstractions for parallel processing of messages. The basic distinction the language supports is that between the instantiation of an

Actor class which gives rise to the initialization of a group of active objects sharing a queue and that which adds a new active object to an existing group. Such a group of active objects sharing a message queue constitutes a multi-threaded actor which features the parallel processing of its messages. The distinction between actors and active objects is reflected by the type system which includes an explicit type for actors and which is used to restrict the communication between actors to asynchronous method calls. In contrast to the concept of a cobox, a group of active objects sharing a queue has its own distinct identity (which coincides with the initial active object). This distinction further allows, by means of simple typing rules, to restrict the communication between active objects to synchronous method calls. When an active object fetches a message from the shared message queue, the object starts executing a corresponding thread in parallel with all the other threads. This basic mechanism gives rise to the new programming concept of a *Multi-threaded Actor (MAC)* which provides a powerful Actor-based abstraction of the notion of a *thread pool*, as for example, implemented by the Java library *java.util.concurrent.ExecutorService*. We further extend the concept of a MAC with a powerful high-level concept of synchronized data to constrain the parallel execution of messages.

In this chapter we provide a formal operational semantics like Plotkin [61], and a description of a Java-based implementation for the basic programming abstractions describing sharing of message queues between active objects. The proposed run-time system is based on the ExecutorService interface and the use of *lambda expressions* in the implementation of asynchronous execution and messaging.

Related work Since Agha introduced in [2] the basic Actor model of concurrent computation in distributed systems, a great variety of Actor-based programming languages and libraries have been developed. In most of these languages and libraries, e.g., Scala [42], Creol [49], ABS [48], JCoBox [64], Encore [21], ProActive [22], AmbientTalk [74], Rebeca [69], actors execute messages stored in their own message queue. The Akka library for Actor-based programming however does support sharing of message queues between actors. In this chapter we introduce a new corresponding Actor-based programming abstraction which integrates a thread-based execution of messages with event-based asynchronous message passing.

Our work complements in a natural manner that of [64] which introduces groups of actors sharing control. Another approach to extending the Actor-based concurrency model is that of Multi-threaded active objects (MAO) [44] and Parallel Actor Monitors (PAM) [65] which allow the parallel execution of the different method invocations within an actor. Another approach is followed in the language Encore which provides an explicit construct for describing parallelism within the execution of one method [35]. In contrast to these languages, we do allow the parallel execution of different asynchronous method invocation inside a group of active objects which provides an overall functionality as that of an actor, e.g., it supports an interface

for asynchronous method calls and a unique identity. Further we provide a new high-level language construct for specifying that certain parameters of a method are *synchronized*, which allows a fine-grained parameter-based scheduling of messages. In contrast, the more coarse-grained standard scheduling of methods as provided by Java, PAM, and MAO, and JAC [43] in general only specify which methods can run in parallel independent of the actual parameters. [77] also shows the notion of Microsoft COM (Component Object Model)'s multi-threaded apartment. In this model, calls to methods of objects in the multi-threaded apartment can be run on any thread in the apartment. It however lacks the ability of setting scheduling strategies (e.g. partial order of incoming messages in the next section). Multi-threaded actors offer a higher level of abstraction to parallel programming and can be viewed as similar to the OpenMP [25] specification for parallel programming in C, C++ and Fortran.

The rest of this chapter is organized as follows: In section 5.2, an application example is established, by which we introduce the key features of MAC. Section 5.3 describes the syntax of MAC and the type system. Section 5.4 presents the operational semantics. In Section 5.5 we show the implementation of MAC in the Java language and explain its features through an example. We draw some conclusions in Section 5.6 where we briefly discuss extensions and variations describing static group interfaces, support for the cooperative scheduling of the method invocations within an actor (as described in for example [49]), synchronization between the threads of a MAC, and encapsulation of the active objects belonging to the same actor.

5.2 Motivating Example

In this section, we explain an example which is used in the rest of the chapter to show the notion of MAC. We also raise a challenge regarding this example which is solved later in our proposed solution. We present a simple concurrent bank service where the requests such as withdrawal, checking, and transferring credit on bank accounts are supported. The requests can be submitted in parallel by several clients of the bank. The system should respect the temporal order of the submitted requests on the same accounts. For instance, checking the credit of an account should return the amount of credit for the account after withdrawal, if there is a withdrawal request for that account which precedes the check request. The requests can be sent asynchronously. Therefore, respecting temporal order of two events means that there is a happens-before relation between termination of the execution of the former event and starting the execution of the latter.

Existing technologies are either not able to implement this property or they need ad-hoc explicit synchronization mechanism which can be complicated and erroneous. Using locks on accounts (e.g. *synchronized* block in Java) may cause deadlock or violate the ordering, unless managed explicitly at the lower level, since two accounts are involved in transferring credit. Another approach is to implement the scheduler

in PAM [65] to support such ordering which raises synchronization complexities. The last alternative we investigate in this section is that to implement the service as a thread pool (e.g. ExecutorService in Java), where the above ordering is respected explicitly via passing the *future* variable corresponding to the previous task, to the current one. The variable is then used to force the happens-before relation by suspending the process until the future is resolved (e.g. *get* method in Java). One challenge is that the approach requires that the submitter knows and has access to the future variables associated to the previous task (or tasks in case the task being submitted is a *transfer*). The other challenge is that, in a parallel setting with multiple concurrent source of task submitters, how to provide such knowledge. Last but not least, the approach first activates the task by allocating a thread and then the task may be blocked which imposes overhead, while a desirable solution forces the ordering upon the task activation. As shown in the rest of the chapter, we provide the notion of MAC which overcomes this issue only via annotating the parameters based on which we aim to respect the temporal order.

5.3 Syntax of MAC

Figure 5.1 specifies the syntax. A *MAC* program P defines interfaces and classes, and a main statement. An interface *IF* has a name I and method signatures *Sg*. A class *CL* has a name C, interfaces $\overline{I}$ that C implements (that specify the possible types for its instances), formal parameters and attributes $\overline{x}$ of type $\overline{T}$, and methods $\overline{M}$. A multi-threaded actor consisting of a group of active objects (a MAC) which share a queue of messages of type I is denoted by **Actor**<I>. The type **Fut**<T> denotes futures which store return values of type T. The *fields* of the class consist of both its parameters and its attributes. A method signature *Sg* declares a method with name m and the formal parameters $\overline{x}$ of types $\overline{T}$ with optional **sync**<l> modifier which is used to indicate that the corresponding parameter contains synchronized data. The user-defined label l allows to introduce different locks for the same data type. Informally, a message which consists of such synchronized data can only be activated if the specified data has not been locked.

Statements have access to the local variables and the fields of the enclosing class. Statements are standard for sequential composition, assignment, **if** and **while** constructs. The statement e.**get**, where e is a future variable, blocks the current thread until x stores the return value. Evaluation of a right-hand side expression **new** $C(\overline{e})$ returns a reference to a new active object within the same group of the executing object, whereas **new actor** $C(\overline{e})$ returns a reference to a new actor which forms a new group of active objects. By $e.m(\overline{e})$ we denote a synchronous method call. Here e is assumed to denote an active object, i.e., e is an expression of some type I, whereas $e!m(\overline{e})$ denotes an asynchronous method call on an actor e, i.e., e is of some type **Actor**<I>.

Listing 5.1 contains an example of an actor *bank* which implements a bank

$$
\begin{aligned}
T &::= \text{Bool} \mid I \mid \textbf{Actor}\langle I\rangle \mid \textbf{Fut}\langle T\rangle \\
P &::= \overline{IF}\ \overline{CL}\ \{\overline{T}\ \overline{x}; s\} \\
CL &::= \textbf{class}\ C[(\overline{T}\ \overline{x})]\ \textbf{implements}\ \overline{I}\{\overline{T}\ \overline{x}; \overline{M}\} \\
IF &::= \textbf{interface}\ I\{\overline{Sg}\} \\
Sg &::= \overline{[\textbf{sync}\langle l\rangle]}\ T\ m(\overline{[\textbf{sync}\langle l\rangle]}\ \overline{T}\ \overline{x}) \\
M &::= Sg\{\overline{T}\ \overline{x}; s\} \\
s &::= x = e \mid s; s \mid e.\textbf{get} \mid \textbf{if}\ b\{s_1\}\textbf{else}\{s_2\} \mid \textbf{while}\ b\{s\} \\
e &::= \textbf{null} \mid b \mid x \mid \textbf{this} \mid \textbf{new}\ [\textbf{actor}]\ C[(\overline{e})] \mid e.m(\overline{e}) \mid e!m(\overline{e}) \\
b &::= e? \mid b \mid b \wedge b
\end{aligned}
$$

Figure 5.1: Syntax

service. The services provided by a bank are specified by the interface `IEmployee` which is implemented by the class `Employee`. A bank is created by a statement

```
Actor <IEmployee> bank = new actor Employee().
```

New employees can be created on the fly by the addEmp method. The actual data of the bank is represented by the instances of the class `Account` which implements the interface `IAccount` and which contains the actual methods for transferring credit, checking and withdrawal. A simple scenario is the following:

```
(1) Fut<Int> f = bank!createAcc(...);
(2) Int acc1 = f.get;
(3) Fut<Bool> f3 = bank!withdraw(acc1, 50);
(4) Fut<Int> f2 = bank!check(acc1);
```

Line 1 models a request to create an account by an asynchronous method call. The result of this call is a number of the newly created account. Lines 3 and 4 then describe a withdrawal operation followed by a check on this account by means of corresponding asynchronous method calls. These calls are stored in the message queue of the actor *bank* and dispatched for execution by its empoyees, thus allowing a parallel processing of these requests. However, in this particular scenario such a parallel processing of requests involving the same account clearly may give rise to inconsistent results. For example a main challenge in this setting arises how to ensure that the messages are *activated* in the right order, i.e., the order in which they have been queued. Note that the *execution* of messages can be synchronized by means of standard synchronization mechanisms, e.g., synchronized methods in Java. Another approach is to use transactional memory to recover from inconsistent states. However both approaches do not guarantee in general that the messages are activated in the right order because they do not provide direct control of their activation.

By declaring in Listing 5.1 all the parameters of the methods of the interface `IEmployee` which involve account numbers as synchronized by means of a single lock "a" we ensure mutual exclusive access to the corresponding accounts. More specifically, the selection for execution of a queued message which contains a request to withdraw a certain amount for a specified account, for example, requires that (1) no employee is currently holding the lock "a" on that account and (2) no preceding message in the queue requires the lock "a" on that account. Similarly, a message which contains a transfer request, which involves two accounts, requires that (1) no employee is currently holding the lock "a" on one of the specified accounts and (2) no preceding message in the queue requires the lock "a" on one of these accounts. The formal details of this synchronization mechanism is described in the following section.

Listing 5.1: Syntax Example

```
interface IEmployee {
        IAccount createAcc(...);
        Bool transfer(sync<a> Int accNum1, sync<a> Int accNum2, Int amount);
        Bool withdraw(sync<a> Int accNum, Int amount);
        Int check(sync<a> Int acc);
}

interface IAccount {
        Bool transfer(IAccount acc2, Int amount);
        Bool withdraw(Int amount);
        Int check();
}

class Employee implements IEmployee {
        Int createAcc(){
                Int accNum = ...;
                IAccount acc = new Account(accNum, ...); \\account creation
                return accNum;
        }
        Bool transfer(Int accNum1, Int accNum2, Int amount){
                IAccount acc1 = getAccount(accNum1);
                IAccount acc2 = getAccount(accNum2);
                acc1.transfer(IAccount acc2, Int amount);...
        }
        Bool withdraw(Int acc, Int amount){
                IAccount acc1 = getAccount(acc1);
                acc.withdraw(Int amount);...
        }
        Int check(Int accNum){
                IAccount acc = getAccount(accNum);
                acc.check();...
        }
        Unit addEmp(){ ...
                IEmployee emp = new Employee();
        }
        IAccount getAccount(Int accNum){...}
}

class Account(Int acn, ...) implements IAccount {
        ...
}
```

5.4 Operational Semantics

Runtime concepts We assume given an infinite set of active object and future references, with typical element o and f, respectively.

We assume distinguished fields *myactor*, I, and L which denote the identity of the actor, the type of the active object, and the set of pairs of synchronized entries locked by the active object, respectively. A local environment τ assigns values to the local variables (which includes the distinguished variables *this* and *dest*, where the latter is used to store the future reference of the return value). A closure $c = (\tau, s)$ consists of a local environment τ and a statement s. A thread t is a sequence (i.e., a stack) of closures. A process p of the form (o, t) is a runtime representation of an active object o with an active thread t. An actor a denotes a pair (o, P) consisting of an object reference o, uniquely identifying the actor as a group of active objects, and a set of processes P. A set A denotes a set of actors. By e we denote an event $m(\overline{v})$ which corresponds to an asynchronous method call with the method name m and values $\overline{v}$. For notational convenience, we simply assume that each event also includes information about the method signature. A queue q is a sequence of events. A (global) context γ consists of the following (partial) functions: γ_h, which denotes for each existing object its local state, that is, an assignment of values to its fields; γ_q, which denotes for each existing object identifying an actor its queue of events, and, finally, γ_f, which assigns to each existing future its value ($\perp$, in case it is undefined).

Some auxiliary functions and notations. By $\gamma[o \leftarrow \sigma]$ we denote the assignment of the local state σ, which assigns values to the fields of o, to the object o (affecting γ_h); by $\gamma[o.x \leftarrow v]$ we denote the assignment of the value v to the field x of object o (affecting γ_h); by $\gamma[o \leftarrow q]$ we denote the assignment of the queue of events q to the object reference o (affecting γ_q); and, finally, by $\gamma[f \leftarrow v]$ we denote the assignment of value v to the future f (affecting γ_f). By *act-dom*(γ) and *fut-dom*(γ) we denote the actors and futures specified by the context γ. We assume the evaluation function $val_{\gamma,\tau}(e)$. The function *sync-call*$(o, m, \overline{v})$ generates the closure corresponding to a call to the method m of the actor o with the values $\overline{v}$ of the actual parameters. The function *async-call*$(o, m, \overline{v})$ returns the closure corresponding to the message $m(\overline{v})$, where $\overline{v}$ includes the future generated by the corresponding call (which will be assigned to the local variable *dest*), o denotes the active object which has been scheduled to execute this method. In both cases we simply assume that the class name can be extracted from the identity o of the active object (to retrieve the method body). The function *init-act*$(o, \overline{v}, o')$ returns the initial state of the new active object o. The additional parameter o' denotes the the actor identity which contains o, which is used to initialize the field *myactor* of o. The function $sg(m(\overline{v}))$ returns the signature of the event $m(\overline{v})$. Finally, $sync_m(\overline{v})$ returns the synchronized arguments of event $m(\overline{v})$ together with their locks (i.e., the arguments

specified by **sync**<l> modifier in the syntax where l is the lock).

The Transition Systems Figure 5.2 gives a system for deriving local transition of the form: $\gamma, (o, t) \rightarrow \gamma', (o, t')$ which describes the effect of the thread t in the context of γ. Rules (ASSIGN-LOCAL) and (ASSIGN-FIELD) assign the value of expression e to the variable x in the local environment τ or in the fields $\gamma_h(o')$, respectively. o' is the identity of the active object corresponding to the current closure. Rules (COND-TRUE) and (COND-FALSE) evaluate the boolean expression and branch the execution to the different statements depending on the value from the evaluation of boolean expression e. Rule (SYNC-CALL) addresses synchronous method calls between two active objects. A synchronous call gives the control to the callee after binding the values of actual parameters to the formal parameters and forming a closure corresponding to the callee. The closure (τ_0, s_0), which represents the environment and the statements of the called method, is placed on top of the stack of closures. Rule (SYNC-RETURN) addresses the return from a synchronous method call. We assume that **return** is always the last statement of a method body. Therefore, the rule consists of obtaining the value v of the return expression e, updating the variable which holds the return value on the caller side with v, and removing the closure of the callee from the stack. Rule (NEW-ACTOB) creates a new active object in the same actor by allocating an identity to the new active object and extending the context γ_h with the fields of the active object.

Rule (READ-FUT) blocks the active object o until the expression e is resolved, i.e., if e is evaluated to a future which is equal to $\perp$ then the active object blocks.

Rule (NEW-ACTOR) creates a new actor o' and sends the special event *init* to it with the class name C and the values $\overline{v}$ obtained by evaluating the actual parameters of the constructor. This event will initialize the actor with one active object of type C with the parameters $\overline{v}$. Rule (ASYNC-CALL) sends a method invocation message to the actor o' with the new future f, the method name m, and the values $\overline{v}$ obtained by evaluating the expressions $\overline{e}$ of the actual parameters. The rule updates γ to place the message in the queue of the target actor o' and also to extend the set of futures with f with the initial value $\perp$.

Rule (SCHED-MSG) addresses the activation of idle objects of an actor. The rule specifies scheduling a thread for the idle object o by binding an event from the queue of the actor o' to which the active object o belongs, and removing the event from the queue. The $q \backslash m(\overline{v})$ removes the first occurrence of message $m(\overline{v})$ from the queue.

$$\frac{v = val_{\gamma,\tau}(e)}{\gamma, (o, t.(\tau, x = e; s)) \to \gamma, (o, t.(\tau[x \leftarrow v], s))} \text{ ASSIGN-LOCAL}$$

$$\frac{o' = \tau(this) \qquad v = val_{\gamma,\tau}(e)}{\gamma, (o, t.(\tau, x = e; s)) \to \gamma[o'.x \leftarrow v], (o, t.(\tau, s))} \text{ (ASSIGN-FIELD)}$$

$$\frac{val_{\gamma,\tau}(e) = \text{False}}{\gamma, (o, t.(\tau, \textbf{if } e \textbf{ then } \{s1\} \textbf{ else } \{s2\}; s)) \to \gamma, (o, t.(\tau, s2; s))} \text{ (COND-FALSE)}$$

$$\frac{o' = val_{\gamma,\tau}(e) \qquad \overline{v} = val_{\gamma,\tau}(\overline{e}) \qquad (\tau_0, s_0) = \textit{sync-call}(o', m, \overline{v})}{\gamma, (o, t.(\tau, x = e.m(\overline{e}); s)) \to \gamma, (o, t.(\tau, x =?; s).(\tau_0, s_0))} \text{ (SYNC-CALL)}$$

$$\frac{val_{\gamma,\tau}(e) = \text{True}}{\gamma, (o, t.(\tau, \textbf{if } e \textbf{ then } \{s1\} \textbf{ else } \{s2\}; s)) \to \gamma, (o, t.(\tau, s1; s))} \text{ (COND-TRUE)}$$

$$\frac{v = val_{\gamma,\tau}(e)}{\gamma, (o, t.(\tau, x =?; s).(\tau_0, \textbf{return } e)) \to \gamma, (o, t.(\tau, x = v; s))} \text{ (SYNC-RETURN)}$$

$$\frac{val_{\gamma,\tau}(e) \neq \bot}{\gamma, (o, t.(\tau, e.\textbf{get}; s)) \to \gamma, (o, t.(\tau, s))} \text{ (READ-FUT)}$$

$$\frac{v = val_{\gamma,\tau}(e) \qquad f = \tau(dest)}{\gamma, (o, (\tau, \textbf{return } e)) \to \gamma[f \leftarrow v, o.L \leftarrow \emptyset], (o, \epsilon)} \text{ (ASYNC-RETURN)}$$

$$\frac{o' \notin dom(\gamma_h)}{\gamma, (o, t.(\tau, x = \textbf{new } C(\overline{e}); s)) \to \gamma[o' \leftarrow \textit{init-act}(o', val_{(\gamma,\tau)}(\overline{e}), \gamma_h(o.myactor))], (o, t.(\tau[x \leftarrow o'], s))} \text{ (NEW-ACTOB)}$$

$$\frac{o' \notin \textit{act-dom}(\gamma) \qquad \overline{v} = val_{\gamma,\tau}(\overline{e})}{\gamma, (o, t.(\tau, x = \textbf{new actor } C(\overline{e}); s)) \to \gamma[o' \leftarrow \textit{init}(C, \overline{v})], (o, t.(\tau[x \leftarrow o'], s))} \text{ (NEW-ACTOR)}$$

$$\frac{f \notin \textit{fut-dom}(\gamma) \qquad \overline{v} = val_{\gamma,\tau}(\overline{e}) \qquad o' = val_{\gamma,\tau}(e) \qquad \gamma_q(o') = q}{\gamma, (o, t.(\tau, x = e!m(\overline{e}); s)) \to \gamma[f \leftarrow \bot, o' \leftarrow q.m(\overline{v}, f)], (o, t.(\tau, x = f; s))} \text{ (ASYNC-CALL)}$$

$$\frac{o' = \gamma_h(o.myactor) \qquad \gamma_q(o') = q \qquad m(\overline{v}) = \textit{select}(\gamma_h(o.I), \textit{lock}(\gamma, o'), q) \qquad (\tau, s) = \textit{async-call}(o, m, \overline{v})}{\gamma, (o, \epsilon) \to \gamma[o' \leftarrow q \backslash m(\overline{v}), o.L \leftarrow \textit{sync}_m(\overline{v})], (o, (\tau, s))} \text{ (SCHED-MSG)}$$

Figure 5.2: Operational Semantics at the Local Level

The event selection mechanism is underspecified, provided that it respects the temporal order of events in the queue that use the same synchronized data with the same locks. For instance, suppose given events with the order *m1, m2, m3, m4* and *m5* in the queue of an actor with the required set of pairs of lock and data: {*(l, v1)*}, {*(l', v1)*}, {*(l, v1), (l, v2)*}, {*(l, v2)*}, and {*(l, v3)*} for the events respectively (Recall that each synchronized entry is a pair consisting of a data value and a user-defined lock which is specified in the program by the **sync**<*l*> modifier on the method parameters). The actor also contains more than one active object. If event *m1* is activated then event *m2* can be scheduled in parallel since the required lock for *v1* is different. However, *m3* cannot be scheduled unless *m1* is terminated. Event *m4* also cannot be activated in parallel with *m1*, even though *v2* is free, since *m3* which requires *v2* precedes *m4* in the queue. However *m5* can be activated in parallel with *m1*. The semantics of the *select* function is defined as follows:

$$select(I, L, m(\bar{v}).q) = \begin{cases} m(\bar{v}) & \texttt{in case } L \cap sync_m(\bar{v}) = \emptyset \wedge Sg(m(\bar{v})) \in I \\ select(I, L \cup sync_m(\bar{v}), q) & \texttt{otherwise} \end{cases}$$

where $L \subseteq Labels \times Data$ and $select(I, L, \epsilon) = \perp$ (where $\perp$ stands for undefined). The signature of selected method requires to be supported by the active object type, I. The set of synchronized entries of the message, $sync_m(\bar{v})$, also requires to be mutually disjoint with the union of synchronized entries of the actor and the synchronized arguments of the messages preceding to the message in the queue. The binding proceeds then by assigning the set of synchronized entries of the method to the field L of the object. $Lock(\gamma, o) = \bigcup\{o'.L | \gamma_h(o'.myactor) = o\}$ returns the synchronized entries of the actor o, that is, the union of synchronized entries of its objects, represented by field L of each object.

Rule `(ASYNC-RETURN)` evaluates the expression e and assigns the resulting value v to the future f associated to the method call. The return statement belongs to an asynchronous method invocation if there is only one closure in the thread stack (i.e., the closure generated by `(SCHED-MSG)`). The set L of synchronized entries associated to the invocation are also released by assigning $\emptyset$ to the field L of the active object. Then the closure is removed and the active object o becomes idle.

Figure 5.3 gives the rules for the second level, the actor level. Rule `(PROCESS-UPDATE)` specifies that if the domain of the heap remains the same then only the current process is updated. Rule `(PROCESS-CREATE)`, on the other hand, shows that if the domain of the heap has been extended with a new active object o' then a new idle process p'' for the active object o' is introduced to the processes of the actor.

Figure 5.4 gives the rules for the third level, the system level. Rule `(ACTOR-UPDATE)` specifies that if the domain of γ remains the same then only the current actor is updated. Rule `(ACTOR-CREATE)`, on the other hand, shows that if the domain of γ has been extended then a corresponding new actor configuration a'' is added to the system. Note that this actor is identified by the reference which has

been added to γ. This reference is also used to identify the initial active object of the newly created actor.

$$\frac{\gamma, p \to \gamma', p' \quad dom(\gamma_h) = dom(\gamma_{h'})}{\gamma, (o, P \cup \{p\}) \to \gamma', (o, P \cup \{p'\})} \text{(PROCESS-UPDATE)}$$

$$\frac{\gamma, p \to \gamma', p' \quad o' \in dom(\gamma_{h'}) \backslash dom(\gamma_h) \quad p'' = (o', \epsilon)}{\gamma, (o, P \cup \{p\}) \to \gamma', (o, P \cup \{p', p''\})} \text{(PROCESS-CREATE)}$$

Figure 5.3: Operational Semantics at the Actor Level

We have the following the description of the initial state for the operational semantics in the local, actor, and system level respectively:

$$p_0 = (\_, (\tau_{main}, s_{main})) \qquad a_0 = (\_, \{p\}) \qquad A_0 = \{a\}$$

The p_0 represents a process with the context τ_{main} for the main body and its statement s_{main}. The process is considered to be an active object with the anonymous identity which is denoted by underscore. The a_0 represents an anonymous actor with the underscore identity in the system and the process p_0 in the process set. The gamma is initialized as the following,

$$\gamma[\_ \leftarrow \{myactor \leftarrow \_\}]$$

as the active object state for p_0. Any object which is created in the main body is a free object, an active object that belongs to the anonymous actor. All the objects which are created by a free object are also free objects. The field *myactor* of all the free objects is equal to underscore. The anonymous actor does not receive any event as it has no identity in the program.

We conclude this section with the following basic operational property of synchronized data:

Theorem 2. *First, let Object*$(a) = \{o \mid (o, t) \in P,$ for some process $p\}$ *denote the set of objects in a which contains the set processes* P*. For every configuration* A

$$\frac{\gamma, a \to \gamma', a' \quad act\text{-}dom(\gamma) = act\text{-}dom(\gamma')}{\gamma, A \cup \{a\} \to \gamma', A \cup \{a'\}} \text{(ACTOR-UPDATE)}$$

$$\frac{\gamma, a \to \gamma', a' \quad o \in act\text{-}dom(\gamma') \backslash act\text{-}dom(\gamma) \quad \gamma'_q(o) = q.init(C, \overline{v}) \quad a'' = (o, \{(o, \epsilon)\})}{\gamma, A \cup \{a\} \to \gamma'[o \leftarrow q, o \leftarrow init\text{-}act(o, \overline{v}, o)], A \cup \{a', a''\}} \text{(ACTOR-CREATE)}$$

Figure 5.4: Operational Semantics at the System Level

Table 5.1: The interface for group management

`poolSize()` Returns the number of threads in the actor's pool of suspended threads.
`groupSize()` Returns the number of internal actors in the group.
`groupThreadNumber()` Returns the number of threads (active and suspended) in the group.

reachable from the initial configuration A_0 *we have* $o.L \cap o'.L = \emptyset$ *for any* $o, o' \in$ *Object*(a) $(o \neq o')$

This invariant property follows immediately from the definition of the select function. It expresses that at run-time there are no two distinct asynchronous method invocations which require the same synchronized data.

5.5 Experimental Methodology and Implementation

In this section we present the implementation of the MAC in a widely used, mainstream programming language, the Java language. The implementation has to take into account the transparency of parallel computation from the user's perspective and the functions that are exposed by the abstract class. The outline of the implementation is presented in Listing 5.4.

As shown in the operational semantics in section 5.4, the default policy schedules the idle objects non-deterministically. However, there is the possibility to overload the policy using the runtime information to allow a preferential selection of the active objects. Furthermore, the current selection method presented is minimal, in the sense that it can be overloaded with different arguments to provide more selection options based on application specific requirements.

To this aim, each new actor in a group is a subclass of class Group which provides an interface S that can be used for specifying different scheduling policies (e.g. addressing load balancing concerns). The internal actors can call the methods in S synchronously. Table 5.1 describes the methods in the S.

5.5.1 Actor Abstract Class

The Java module creates an abstract class, **Actor**, that provides a runtime system for queuing and activation of messages. It exposes two methods to the outside

world for interaction, namely *send(Object message)* and *getNewWorker(Object... parameters)*. This layout is used to allow a clear separation between internal object selection, message delivery and execution. This class is the mediator between the outside applications and the active objects defined by the internal interface **ActiveObject**. These Active Objects will be assigned execution of the requests sent to the actor. Our abstract class contains a queue of *availableWorkers* and a set of *busyWorkers* separating those objects that are idle from those that have been assigned a request. Parallel execution and control is ensured through a specific Java Fork Join Pool *mainExecutor* that handles the invocations assigned to the internal objects and is optimized for small tasks executing in parallel. The class uses a special queue, named *messageQueue*, that is independent of the thread execution. It is used to store incoming messages and model the **shared queue** of the group. This message queue is initialized with a comparator(ordering function) that selects the first available message according to the rule `(SCHED-MSG)` specified in Section 5.4. To use this abstract class as a specific model, it needs to be extended by each interface defined in our language in order to be initialized as an Actor.

The default behavior of the exposed method *getNewWorker(Object... parameters)* is to select a worker from the *availableWorkers* queue. The workers are inserted in a first-in-first-out(FIFO) order with a blocking message delivery if there is no available worker (i.e. the *availableWorkers* queue is empty). While the behavior of this method is hidden from the user, it needs to be exposed such that the user has a clear view of the selection, before sending a request.

An interesting observation here is that the **Actor** interface extends the *Comparable* such that once the abstract class is extended a specific or a natural ordering of the workers can be made in the queue. When overriding the *getNewWorker(Object... parameters)*, additional arguments can be processed to offer an **ActorGroup** with several types of Actors available for selection and concurrency control. For this implementation example, our abstract class offers the signature *getNewWorker(Object... parameters)* with a simple non-deterministic selection.

The second exposed function, *send(Object message, Set<Object> data*, takes the first argument in the form of a lambda expression, that models the request. The format of the lambda expression must be

```
() -> ( getNewWorker() ).m()
```

The second argument specifies a set of objects that the method *m()* needs to lock and maintain data consistency on. Therefore when a request is made for a method *m()* the runtime system must also select an *Active Object* from the *availableWorkers* queue to be assigned the request, as well a set of data that needs concurrency control. Execution is then forwarded to the *mainExecutor* which returns a Java Future to the user for synchronization with the rest of the application outside the actor. The selection of the *ActiveObject* is important to form the lambda expression that saves the application from having a significant number of suspended threads if the set of data that is required is locked.

The application outside of the Actor sends requests asynchronously and must be free to continue execution regardless of the completion of the request. To this end we provide the class **Message** illustrated in Listing 5.2 which creates an object from the arguments of the *send* method and initializes a future from the lambda expression.

Listing 5.2: Message Class in Java

```
import java.util.Set;
import java.util.concurrent.Callable;
import java.util.concurrent.ForkJoinTask;
import java.util.concurrent.atomic.AtomicInteger;

public class Message {

    static final AtomicInteger queuePriority = new AtomicInteger(0);

    String name;
    Object lambdaExpression;
    Set<Object> syncVariables;
    ForkJoinTask<?> f;
    AtomicInteger preemptPriority;
    int priority = 0;

    public Message(Object message, Set<Object> variables, String name) {
        this.lambdaExpression = message;
        this.syncVariables = variables;
        this.name = name;
        this.preemptPriority = new AtomicInteger(0);
        priority = queuePriority.getAndAdd(1);

        f = null;
        if (message instanceof Runnable)
            f = ForkJoinTask.adapt((Runnable) message);
        if (message instanceof Callable<?>)
            f = ForkJoinTask.adapt((Callable<?>) message);
    }

    public Message(Message m) {
        this(m.lambdaExpression, m.syncVariables, m.name);
        this.preemptPriority.set(m.preemptPriority.get());
    }

    @Override
    public String toString() {
        return name + "␣" + syncVariables + "␣:<␣" + priority + ","
                + preemptPriority + "␣>";
    }
}
```

This class contains the specific parts of a message which are the *lambdaExpression*, the *syncData* on which the request need exclusive access and the Future *f* which captures the result of the request. To maintain a temporal order on messages synchronize on the same messages the class also contains a static field *queuePriority* which determines a new message's *priority* upon creation and insertion in the queue.

The Actor runs as a process that receives requests and runs them in parallel while maintaining data-consistency throughout its lifetime. The abstraction is data-oriented as it is a stateful object maintaining records of all the data that its workers are processing. It contains a set of *busyData* specifying which objects are

currently locked by the running active objects. An internal method, named *reportSynchronizedData* is defined to determine if a set of data corresponding to a possible candidate message for execution is intersecting with the current set of *busyData*. This method is used as part of the comparator defined in the *messageQueue* to order the messages based on their availability. The main process running the Actor is then responsible to take the message at the head of the queue and schedule it for execution and add the data locked by the message to the set of *busyData*. It is possible that at some point during execution, all messages present in the *messagesQueue* are not able to execute due to their data being locked by the requests that are currently executing. To ensure that our Actor does not busy-wait, we forward all the messages into a *lockedQueue* such that the Actor thread suspends.

The Actor is a solution that makes parallel computation transparent to the user through the internal class implementation of its worker actors. These objects are synchronized and can undertake one assignment at a time. Each request may have a set of synchronized variable to which it has exclusive access while executing. At the end of the execution, the active object calls the *freeWorker(ActiveObject worker, Object ... data)* method that removes itself from the *busyWorkers* set and becomes available again by inserting itself in the *availableWorkers* queue. At this point, the *lockedQueue* is flushed into the *messageQueue* such that all previously locked messages may be checked as candidates for running again. All of the objects that were locked by this ActiveObject are also passed to this method such that they can be removed from the *busyData* set and possibly release existing messages in the newly filled *messageQueue* for execution. This control flow is illustrated in an example in the next section, however our motivation is to modify this module into an API and use it as a basis for a compiler from the modeling language to Java.

5.5.2 Service Example and Analysis

Listing 5.3 shows the implementation of a **Bank** service as an **Actor**. As a default behavior, whenever a new concrete extension of an Actor is made, the constructor or the *addWorkers* method may create one or more instances of the internal Active Object. The behavior of *getNewWorker(Object... parameters)* is overridden to ensure the return of a specific internal Active Object with exposed methods, in this case the **BankEmployee**. This internal class implements the general *Active Object* interface and exposes a few simple methods of a general Bank Service. The methods *withdraw, deposit, transfer* and *checkSavings* all perform their respective operations on one or more references of the internal class Account a reference which is made available through the method *createAccount*. The MAC behavior is inherited from the Actor and only the specific banking operations are implemented.

To test the functionality, as well as the performance of the MAC we implement a simple scenario that creates a fixed number of users each operating on their own bank account. We issue between 100 and 1 million requests distributed evenly over the

Listing 5.3: Bank Class in Java

```
public class Bank extends Actor {

    public void addWorkers(int n){
        for (int i = 0; i < n; i++) {
            availableWorkers.add(new BankEmployee());
        }
    }

    @Override
    public BankEmployee getNewWorker(Object... parameters) {
        BankEmployee selected_worker = null;
        try {
            selected_worker = (BankEmployee) availableWorkers.take();
            busyWorkers.add(selected_worker);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return selected_worker;
    }

    class BankEmployee implements ActiveObject {

        public Account createAccount() {
            Account a = new Account();
            Bank.this.freeWorker(this);
            return a;
        }

        public boolean withdraw(Account n, int x) {
            boolean b = n.withdraw(x);
            Bank.this.freeWorker(this, n);
            return b;
        }

        protected boolean deposit(Account n, int x) {
            boolean b = n.deposit(x);
            Bank.this.freeWorker(this, n);
            return b;
        }

        public boolean transfer(Account n1, Account n2, int amount) {
            boolean b = n1.transfer(n2, amount);
            Bank.this.freeWorker(this, n1, n2);
            return b;
        }

        public int checkSavings(Account n) {
            int res = n.checkSavings();
            Bank.this.freeWorker(this, n);
            return res;
        }

        class Account {
            //Acount processing methods
} } }  }
```

fixed number of accounts. To ensure that some messages have to respect a temporal order and forced await execution of prior requests on the same account we issue sets of 10 calls for each account. This also ensures that the selection rule `(SCHED-MSG)` does not become too large of a bottleneck as in the case of issuing all operations for one bank account at a time. We measure the time taken to process the requests based on a varying number of Active Objects inside in the **Bank** Service. The performance figures for a MAC with 1,2 and 4 available *Active Objects* is presented in Figure 5.5

The results validate our solution in the sense that the time:message ratio is almost linear with very little overhead introduced by the message format and the selection function. Furthermore the benefit of parallelism is maintained with the increasing volume of request issued to the service. To emphasize this we computed

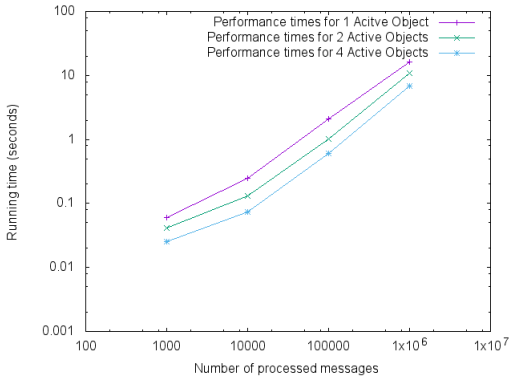

Figure 5.5: Performance Times for processing 100-1M messages

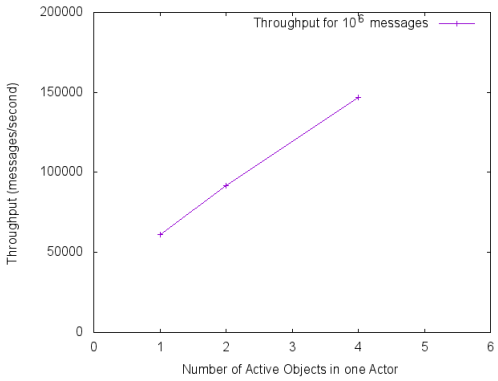

Figure 5.6: Throughput for processing 1M messages

the throughput of the service in relation to the number of *Active Objects* running and present it in Figure 5.6. From these results we can infer the scalability of the MAC for parallel computation.

5.6 Conclusion and Future Work

In this chapter we have introduced the notion of multi-threaded actors, that is, an actor-based programming abstraction which allows to model an actor as a group of active objects which share a message queue. Encapsulation of the active objects which share a queue can be obtained by simply not allowing active objects to be passed around in asynchronous messages. Cooperative scheduling of the method invocations within an active object (as described in for example [49]), can be obtained by introduction of a lock for each active object. In general, synchronization mechanisms between threads is an orthogonal issue and as such can be easily integrated, e.g., lock on objects, synchronized methods (with reentrance), or even synchronization by the compatibility relationship between methods as defined in [43] and [44]. Other extensions and variations describing dynamic group interfaces can be considered along the lines of [50].

Future work will be dedicated toward the development of the compiler which allows importing Java libraries, and further development of the runtime system, as well as benchmarking on the performance. Other work of interest is to investigate into dynamic interfaces for the multi-threaded actors and programming abstractions for application-specific scheduling of multi-threaded actors.

Listing 5.4: Actor Abstract Class in Java

```
public abstract class Actor implements Runnable {

    protected ForkJoinPool mainExecutor;
    protected PriorityBlockingQueue<Message> messageQueue;
    protected ConcurrentLinkedQueue<Message> lockedQueue;
    protected PriorityBlockingQueue<ActiveObject> availableWorkers;
    protected Set<ActiveObject> busyWorkers;
    protected Set<Object> busyData;

    public Actor() {
        //initialization of internal data structures
    }

    @Override
    public void run() {

        while (true) {
            try {
                Message message = messageQueue.take();
                if (reportSynchronizedData(message.syncData)) {
                    synchronized (busyData) {
                        busyData.addAll(message.syncData);
                    }
                    mainExecutor.submit(message.f);
                } else {
                    this.lockedQueue.offer(newM);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
    }}

    // message format: ()->getWorker().m()
    public <V> Future<V> send(Object message, Set<Object> data) {
        Message m = new Message(message, data, name);
        messageQueue.put(m);
        return (Future<V>) m.f;
    }

    public ActiveObject getNewWorker(Object... parameters) {
        ActiveObject selected_worker = null;

        try {
            selected_worker = availableWorkers.take();
            busyWorkers.add(selected_worker);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        return selected_worker;
    }

    private boolean reportSynchronizedData(Set<Object> data) {
        Set<Object> tempSet = new HashSet<Object>();

        synchronized (busyData) {
            tempSet.addAll(busyData);
            tempSet.retainAll(data);

            if (tempSet.isEmpty()) {
                return true;
            }
            return false;
    } }

    protected void freeWorker(ActiveObject worker, Object... data) {

        synchronized (busyData) {
            busyWorkers.remove(worker);
            availableWorkers.offer(worker);

            messageQueue.addAll(lockedQueue);
            lockedQueue.clear();


            for (Object object : data) {
                busyData.remove(object);
    } } }

    public interface ActiveObject extends Comparable<ActiveObject> {
        // the active objects in charge of requests
    }
}
```