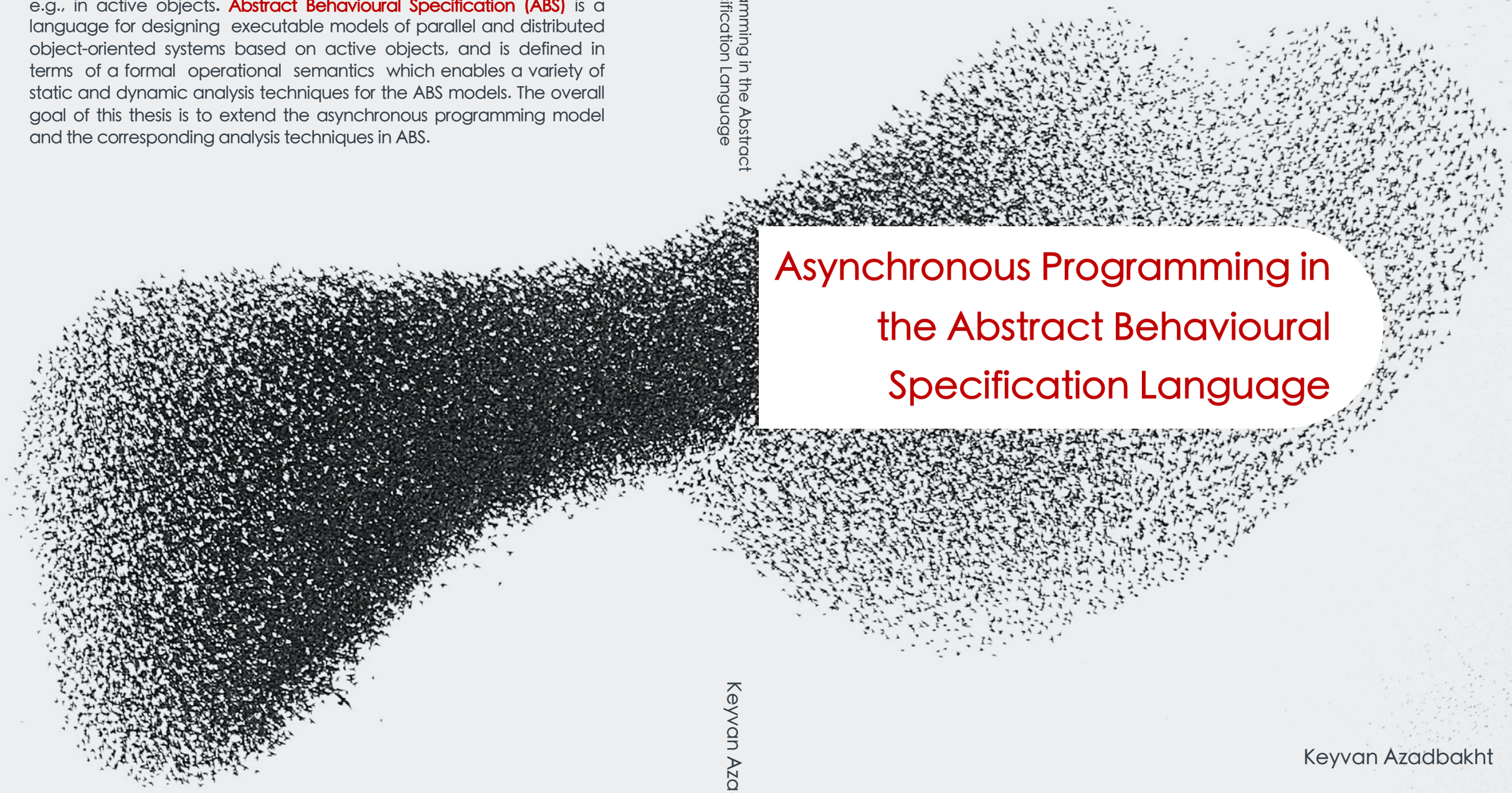
Chip manufacturers are rapidly moving towards so-called manycore chips with thousands of independent processors on the same silicon real estate. Current programming languages can only leverage the potential power by inserting code with low level concurrency constructs, sacrificing clarity. Alternatively, a programming language can integrate a thread of execution with a stable notion of identity, e.g., in active objects. **Abstract Behavioural Specification (ABS)** is a language for designing executable models of parallel and distributed object-oriented systems based on active objects, and is defined in terms of a formal operational semantics which enables a variety of static and dynamic analysis techniques for the ABS models. The overall goal of this thesis is to extend the asynchronous programming model and the corresponding analysis techniques in ABS.

Asynchronous Programming in the Abstract Behavioural Specification Language



Asynchronous Programming in the Abstract Behavioural Specification Language

Keyvan Azadbakht

Keyvan Azadbakht

Asynchronous Programming in the Abstract Behavioural Specification Language

Keyvan Azadbakht

Asynchronous Programming in the Abstract Behavioural Specification Language

Proefschrift

ter verkrijging van
de graad van doctor aan de Universiteit Leiden
op gezag van de Rector Magnificus prof. mr. C.J.J.M. Stolker
volgens besluit van het College voor Promoties
te verdedigen op woensdag 11 december 2019
klokke 11:15 uur

door

Keyvan Azadbakht

geboren te Kohdasht, Iran
in 1987

PhD Committee

Promotor:		
	Prof. dr. F.S. de Boer	
Co-promotor:		
	Dr. E.P. de Vink	Eindhoven University of Technology
other members:		
	Prof. dr. ir. F. Arbab	
	Dr. M.M. Bonsangue	
	Dr. L. Henrio	CNRS, Paris
	Prof. dr. A. Plaat	
	Prof. dr. M. Sirjani	Malardalen University, Sweden

The work in this thesis has been carried out at the Center for Mathematics and Computer Science (CWI) in Amsterdam and Leiden Institute of Advanced Computer Science (LIACS) at Leiden University. This research was supported by the European project FP7-612985 UpScale (From Inherent Concurrency to Massive Parallelism through Type-based Optimizations).

Cover design: Mostafa Dehghani

Contents

Prologue

This manuscript studies the Abstract Behavioural Specification (ABS), a formal language for designing executable models of parallel and distributed object-oriented systems [48]. ABS is defined in terms of a formal operational semantics which enables a variety of static and dynamic analysis techniques for ABS models, e.g., deadlock detection [14, 39], verification [30] and resource analysis [5].

The overall goal of this thesis is to extend the asynchronous programming model and the corresponding analysis techniques in ABS. Based on the different results, the thesis is structured as follows: Part I gives a preliminary overview of the ABS. In part II, we apply an extension of ABS with a notion of shared memory which preserves encapsulation to a case study, where we provide a parallel and distributed model of *preferential attachment* which is used to simulate large-scale social networks with certain mathematical properties. Encapsulation is preserved by a single-write policy. In Part III, we formally extend ABS to enhance both asynchronous programming by data streaming between processes, and parallelism by multi-threading within an actor. Finally in part IV, a new technique based on predicate abstraction is introduced to analyze the ABS models for the absence of deadlock within an actor.

Validation. This work has been carried out in the context of the UpScale Project, an EU-funded project where the vision was:

> to provide programming language support to efficiently develop applications that seamlessly scale to the available parallelism of manycore chips without abandoning the object-oriented paradigm and the associated software engineering methodologies.

In particular, the above extension of ABS concerning streaming of data has been validated by the case study on *preferential attachment* for the efficient multicore and distributed simulation of large-scale social networks. The results of this thesis have been separately validated by the peer-reviewed scientific publications listed in Table 1.

Table 1: List of publications used in the thesis.

Publication	Chapter
Azadbakht, Keyvan, et al. "A high-level and scalable approach for generating scale-free graphs using active objects." Proceedings of the 31st Annual ACM Symposium on Applied Computing. ACM, 2016.	2
Azadbakht, Keyvan, Nikolaos Bezirgiannis, and Frank S. de Boer. "Distributed network generation based on preferential attachment in ABS." International Conference on Current Trends in Theory and Practice of Informatics. Springer, Cham, 2017.	3
Azadbakht, Keyvan, Nikolaos Bezirgiannis, and Frank S. de Boer. "On Futures for Streaming Data in ABS." International Conference on Formal Techniques for Distributed Objects, Components, and Systems. Springer, Cham, 2017.	4
Azadbakht, Keyvan, Frank S. de Boer, Nikolaos Bezirgiannis, and Erik de Vink. "A formal actor-based model for streaming the future." Science of Computer Programming 186 (2019): 102341.	4
Azadbakht, K., Frank S. de Boer, Vlad Serbanescu. "Multi-threaded actors." In: Proceedings 9th Interaction and Concurrency Experience, ICE 2016, Heraklion, Greece, volume 223 of EPTCS, pp. 51–66 2016.	5
Azadbakht, Keyvan, Frank S. de Boer, and Erik de Vink. "Deadlock Detection for Actor-Based Coroutines." International Symposium on Formal Methods. Springer, Cham, 2018.	6

Part I

Background: Abstract Behavioural Specification

This part consists of the following chapter:

Chapter 1 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. In this chapter, we give a brief overview of ABS with the main focus on syntax, semantics, and the model of concurrency. Finally, a simple example that represents a model of thread pool in ABS is given.

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

Part II

Case Study: Preferential Attachment in ABS

This part consists of the following chapters:

Chapter 2 The Barabasi-Albert model (BA) is designed to generate scale-free networks using the preferential attachment mechanism. In the preferential attachment (PA) model, new nodes are sequentially introduced to the network and they attach preferentially to existing nodes. PA is a classical model with a natural intuition, great explanatory power and interesting mathematical properties. Therefore, PA is widely-used for network generation. However the sequential mechanism used in the PA model makes it an inefficient algorithm. The existing parallel approaches, on the other hand, suffer from either changing the original model or explicit complex low-level synchronization mechanisms. In this chapter we investigate a high-level Actor-based model of the parallel algorithm of network generation and its scalable mult-core implementation in the ABS language.

Chapter 3 Generation of social networks using Preferential Attachment (PA) mechanism introduced in previous chapter features interesting mathematical properties which only appear in large-scale networks. However generation of such extra-large networks can be challenging due to memory limitations. In this chapter, we investigate a distributed-memory approach for PA-based network generation which is scalable and which avoids low-level synchronization mechanisms thanks to utilizing the powerful programming model and proper programming constructs of the ABS language.

Chapter 2

Preferential Attachment on Multicore Systems

2.1 Introduction

Social networks in the real world appear in various domains such as, among others, friendship, communication, collaboration and citation networks. Social networks demonstrate nontrivial structural features, such as power-law degree distributions, that distinguish them from random graphs. There exist various network generation models that synthesize artificial graphs that capture properties of real-world social networks. Some existing network generative models are the Erdos-Renyi (ER) [33] model of random graphs, the Watts-Strogatz (WS) [75] model of Small-world networks, and the Barabasi-Albert model of scale-free networks. Among these models, Barabasi-Albert model, which is based on Preferential Attachment [18], is one of the most commonly used models to produce artificial networks, because of its explanatory power, conceptual simplicity, and interesting mathematical properties [73]. The need for efficient and scalable methods of network generation is frequently mentioned in the literature, particularly for the preferential attachment process [4, 9, 19, 23, 41, 58–60, 73, 79]. Scalable implementations are essential since massive networks are important; there are fundamental differences between the structure of small and massive networks even if they are generated according to the same model, and there are many patterns that emerge only in massive networks [56]. Analysis of the large-scale networks is of importance in many areas, e.g. data-mining, network sciences, physics, and social sciences [16]. The property that we have focused on in this chapter is the degree of the nodes and by preferential attachment (PA) we mean degree-based preferential attachment. In PA-based generation of the networks, each node is introduced to the existing graph preferentially based on the degrees of the existing nodes, i.e., the more the degree of an existing node, the higher the probability of choosing it as the target of a new connection.

The PA-based parallel and distributed versions of generating the scale-free graphs

are based on a partitioning of the nodes and a parallel process for each partition which adds edges to its nodes. The edges are generated by random selection of target nodes. The data structure prescribed in the *Copy Model* [54] guarantees that the selection of the target is done consistently, e.g., the probability distribution of selecting the target nodes in the parallel version should remain the same as the distribution in the sequential one. However, from the point of view of the control flow, the following main problem arises: random selection of the target node requires synchronization between the parallel processes. The process that hosts the randomly selected target node has possibly not determined the node yet. However the process hosting the source node must be informed about the target node in the future once it is determined.

random selection requires synchronization between the parallel processes, i.e., the target nodes are not resolved yet. and the need for conflict resolution, namely, the selection of a node which has already been selected as a target of the given source node.

The main contribution of this chapter is a high-level Actor-based model for the PA-based generation of networks which avoids the use of low-level intricate synchronization mechanisms. A key feature of the Actor-based model itself, so-called *cooperative scheduling*, however, poses a major challenge to its implementation. In this chapter, we discuss the scalability of a multicore implementation based on Haskell which manages cooperative scheduling by exploiting the high-level and first-class concept of continuations [62]. Continuations provide the means to "pause" (part of) the program's execution, and programmatically switch to another execution context; the paused computation can be later resumed. Thus, continuations can faithfully implement cooperative scheduling in a high-level, language-builtin manner.

The rest of the chapter is organized as follows. The description of the Actor-based modeling framework which is used to model the PA-based generation of massive networks is given in section 2.2. Section 2.3 elaborates on parallelizing the PA model. Section 2.4 mentions the related works. Finally we conclude in section 2.5.

2.2 The Modeling Framework

We propose an Actor-based modeling framework that supports concurrency and synchronization mechanisms for concurrent objects. It extends the ABS language and, apart from the ABS functional layer which includes algebraic data types and pattern matching, it additionally features global arrays as a mutable data structure shared among objects. This extension fits well in the multicore setting to decrease the amount of costly message passing, and also to simplify the model. In general, this feature can cause complicated and hard-to-verify programs. Therefore the model only allows using this feature in a disciplined manner, which restricts the array to initially *unresolved* slots with single-write access (also known as *promises* in

languages like Haskell and Scala[1]), to avoid race conditions. In this chapter, and chapter 3 and 4 the initial value 0 denotes the unresolved state of a slot.

2.3 Parallel Model of the PA

In this section we present the solution for the PA problem utilizing the idea of active objects cooperating in a multicore setting. For the solution we adopt the *copy model*, introduced in [54]. We first introduce the main data structure of the proposed approach which is based on the graph representation in *copy model*. Next we present the basic synchronization and communication mechanism underlying our solution and its advantages over existing solutions.

2.3.1 The Graph Representation

We introduce one shared array, *arr*, as the main data structure that holds the representation of the graph. The array consists of the edges of the graph. Each (i, j) where $i, j > 0$ and $j = i + 1$, and $j \ mod \ 2 = 0$ shows an edge in the graph between $arr[i]$ and $arr[j]$ (Figure 2.1(a)).

According to the PA, each node is added to the existing graph via a constant number of edges (referred to as m) targeting *distinct* nodes. There is also an initial clique, a complete graph with the size of $m0$ where $(m0 \geqslant m)$, which is stored at the beginning of the array. Therefore the size of the array is calculated based on the number of nodes, num, and the number of edges that connect each new node to the existing distinct nodes, m. The connections of a new node are established via a probability distribution of the degrees of the nodes in the existing graph, that is, the more the degree of the existing node, the more the probability of choosing it as the target. For instance, if the node n is the new node to be added to the graph with the existing graph with $[1..n-1]$ nodes then, according to equation 2.1, the probability distribution of choosing the existing nodes is $[p_1..p_{n-1}]$. ($deg(i)$ gives the degree of the node i in the existing graph)

$$p_i = \frac{deg(i)}{\sum_{j=1}^{n-1} deg(j)} \qquad \sum_{i=1}^{n-1} p_i = 1 \tag{2.1}$$

As mentioned, the connections for the new node should be distinct. Therefore if a duplicate happens the approach retries to make a new connection until all the connections are distinct. This graph representation provides the above mentioned probability distribution since the number of occurrences of each node in the array is equal to its degree. Figure 2.1(b) represents the position of node n in the graph array, where $m = 3$. In order to add node n to the existing graph containing $n - 1$ nodes, with the assumption that $m = 3$, targets are selected randomly from the slots

[1] `https://docs.scala-lang.org/sips/completed/futures-promises.html`

that are located previous to the node n (with the principle shown in Figure 2.2). It is obvious that self-loop cannot happen, i.e., an edge whose source and target are the same. Figure 2.1(c) illustrates an optimization on the array so that the array only contains the targets of the edges since the sources for each node are calculable. The array is half size as the one in Figure 2.1(b). Each slot in the array can have one of two states: *resolved* or *unresolved*. In the former case it contains the node number which is greater than zero, and in the latter it contains zero.

A sequential solution for the generation of such graphs consists of processing the array from left to right to resolve all the slots. The parallel alternative, on the other hand, is to have multiple active objects processing partitions of the array in parallel. As shown in the following equations, we distinguish between the following uses of indices. At the lowest level we have the indices of the slots. The next level is the id of the nodes. Each node contains a sequence of slots. Finally at the top level we have the id of the partitions. Each partition contains nodes and consequently slots. In the proposed approach the partitions satisfy the following equations which express that the sets of indices of the partitions are mutually disjoint (equation 2.3); that their union is equal to the whole array (equation 2.2); furthermore, the sequence of slots of each node must be placed in one partition (equation 2.4) so that one active object resolves the new node and race conditions are avoided for the checking duplicates:

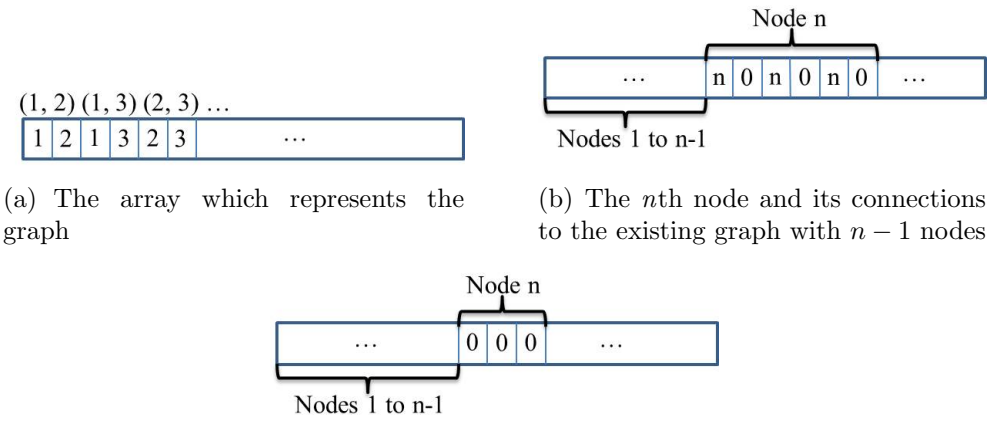
$$\bigcup_{i=1}^{w} par_i = G \tag{2.2}$$

$$\forall (1 \leq (i \neq j) \leq w).par_i \cap par_j = \emptyset \tag{2.3}$$

$$\forall i, j \in G.(node(i) = node(j)) \rightarrow (par(i) = par(j)) \tag{2.4}$$

where G is the global set containing all the indices of the shared array, w holds the number of partitions, par_i is the set which holds the indices of the ith partition of the array, $node(i)$ is a function that returns the node id to which the slot of the array with index i belongs, and $par(i)$ is a function that returns the partition id to which the index i belongs. Note that indices that belong to a specific node differ from the occurrences of that specific node in the array. The former indices are the slots that represents the edges that are created during introducing the *new* node to the graph, which its size is constant (denoted by m), while the latter changes during the graph generation.

There are different approaches to partition the array so that they hold the above equations, such as Consecutive and Round Robin Node Partitioning (CSP and RRP respectively). As it is shown in [4], RRP is more efficient and it is observed a better load balancing among processors as well as less unresolved chains of dependencies which leads to less computational overhead. Therefore we have utilized RRP to partition the array among active objects.



(a) The array which represents the graph

(b) The nth node and its connections to the existing graph with $n-1$ nodes

(c) A memory usage optimization to the array based on *copy model*

Figure 2.1: The array representing the graph

2.3.2 Synchronization of Chains of Unresolved Dependencies

Each active object only resolves (i.e. writes to) the slots which belong to its own partition . Nevertheless it can read all the slots throughout the array. In the parallel solution, an active object may select a slot as the target which is not resolved yet since either the other active object responsible for the target slot has not yet processed it or the target slot may wait for another target itself (see dependency chains in figure 2.2). The way waiting for unresolved slots is managed is crucial for the complexity of the model and its scalability. Next we describe the two main approaches to deal with unresolved dependencies (Figure 2.3):

Synchronization by communication: Active object A processes its own partition of the array and for each randomly selected, unresolved slot it sends a request to the object B responsible for the target. When object B processes the request, it checks whether the slot is resolved. If it is not then it stores the information of the request (e.g. the sender id, the slot requiring the value of the target) in a corresponding data structure. Because B is the only object which writes to the target slot when it

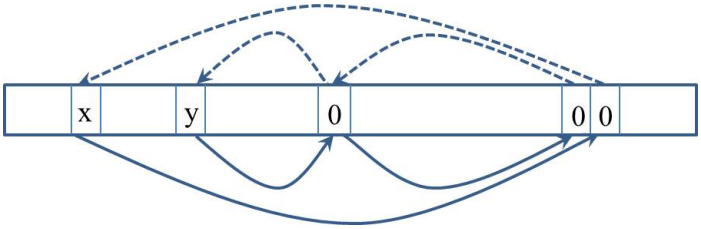


Figure 2.2: An example of the general sketch of dependencies (right to left) and computations (left to right)

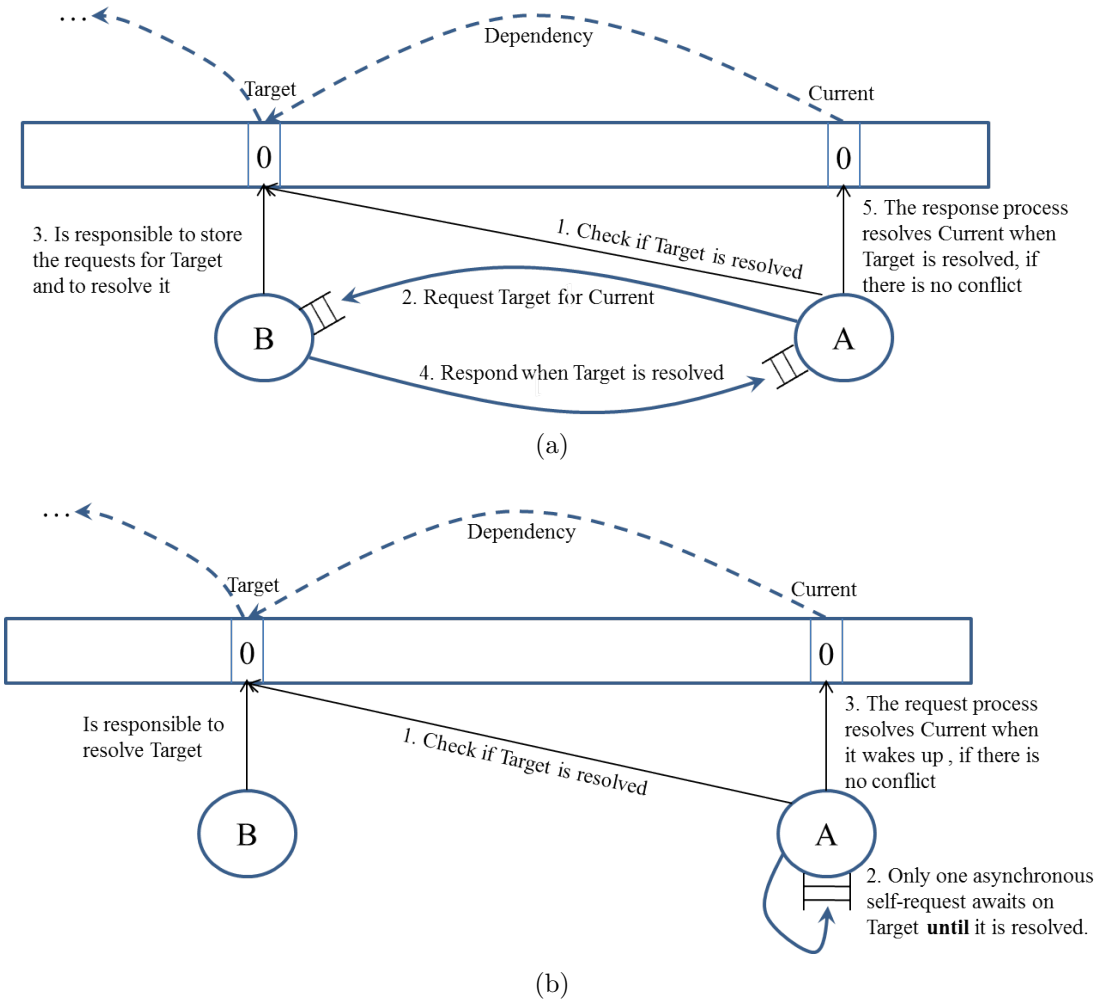


Figure 2.3: Two different solutions for the PA problem (the second one is the proposed approach)

is resolved, it suffices that B answers all the stored requests waiting for the resolved target by broadcasting the value of the slot. As such this approach exploits the wait-notify pattern rather than busy-wait polling, and it can be efficient depending on how the programmer implements the data structure. However, this approach involves a low-level user-defined management of the requests through the explicit user-defined implementation of the storage and retrieval of the requests. Note that in this approach there are *exactly* two messages that have to be passed for each request for an unresolved slot (Figure 2.3(a)).

Synchronization by Cooperative Scheduling: Active object A processes its own partition of the array and for each unresolved randomly selected slot it sends an asynchronous *self* request called "request" for the target value. When object A schedules and processes the request "request" it checks whether the slot is resolved or not (by a Boolean condition) and if not it *awaits* on this condition. This means

that the request process is *suspended*. It is notified when the boolean condition evaluates to *true* and stored back in the object's queue of active processes (Figure 2.3(b)). This approach also avoids busy-waiting and follows the wait-notify pattern. However, the key feature in this approach is the use of cooperative scheduling in which the executing process of an active object can release conditionally the control cooperatively so that another process from the queue of the object can be executed. The continuation of the process which has conditionally released the control will be stored into a separate queue of suspended processes. These processes are stored again in the object's queue for execution when they are notified. The Haskell implementation of this mechanism takes care of the low-level storage, execution and suspension of the processes generated by asynchronous messages. The ABS code itself, see below, remains high-level by means of its programming abstractions describing asynchronous messaging and conditional release of control.

2.3.3 The Actor-based Model of PA

The main part of the encoding of our proposed approach is depicted as a pseudo-code in Figure 2.4. The full implementation of the model is provided online[2]. The worker objects are active objects which resolve their corresponding partitions. To this aim, each worker goes through its own partition and it checks a randomly selected target for each of its slots (note that m denotes the number of connections, or slots in the array, per node). Since we use the optimized array representation (shown in Figure 2.1(c)), half of the array that corresponds to sources of the edges are not part of the array and the values (i.e., node numbers) are calculable from the index. However those indices can be targeted. The auxiliary function $result(arr, target)$ checks the target index. If it is calculable, then it calculates and returns the value without referring to the array. Otherwise, the value of the index should be retrieved from the array. In such case, if the target slot is already resolved then the worker takes the value and resolves the slot of the `current` index in case there is no conflict. If it is not resolved yet then it calls the *request* method asynchronously. The *request* method awaits on the target until it is resolved. Then it uses the value of the resolved target to resolve the current slot, if there is no duplicate. In case of a duplicate, the algorithm selects another target randomly in the same range as the previous one.

Note that the calls to the request method in lines 8 and 22 are asynchronous (denoted by exclamation mark) and synchronous (denoted by dot) respectively. The asynchronous call is introduced so as to spawn one process per each unresolved dependency. In the synchronous call, however, there is no need to spawn a new process since the current process is already introduced for the corresponding unresolved dependency. Note that suspension of such a process thus involves in general an entire call stack, which poses one of the major challenges to the implementation of ABS, but which is dealt with in Haskell by the high-level and first-class concept

[2] `https://github.com/kazadbakht/PA/blob/master/src/ParProRR.abs`

of continuations (described in more detail below).

```
1: Each active object O executes the following in parallel
2: run(...) : void
3: for each Node i in the partition do
4:     for  j = 1 to m do
5:         target ← random[1..(i − 1)2m]
6:         current = (i − 1)m + j
7:         if result(arr, target) = 0 then
8:             this ! request(current, target)
9:         else if duplicate(result(arr, target), current) then
10:            j = j − 1                          ▷ Repeat for the current slot j
11:        else
12:            arr[current] = result(arr, target)                    ▷ Resolved
13:
14:
15: request(target : Int, current : Int) : void
16: await (result(arr, target) ≠ 0)
17:                                  ▷ At this point the target is resolved
18: value = result(arr, target)
19: if duplicate(value, current) then
20:     target = random[1..(target/m)2m]
21:                                  ▷ Calculate the target for the current again
22:     this.request(target, current)
23: else
24:     arr[current] = value                                          ▷ Resolved
25:
26:
27: duplicate(target : Int, current : Int) : Boolean
28: for each i in (indices of the node to which current belongs) do
29:     if arr[i] == value then
30:         return True
31: return False
```

Figure 2.4: The sketch of the proposed approach

2.4 Related Work

There exist some attempts to develop efficient implementations of the PA model [4, 9, 19, 41, 58, 59, 73, 79]. Some existing works focus on more efficient implementations of the sequential version [9, 19, 73]. Such methods propose the utilization of data-structures that are efficient with respect to memory consumption and time complexity. Few existing methods are based on a parallel implementation of the PA

model [4, 59, 79], among which some methods [59, 79] are based on a version of the PA model which does not satisfy its basic criteria (i.e., consistency with the original model). The approach in [4] requires complex synchronization and communication management and generates considerable overhead of message passing. This stems from that this latter approach is not developed for a multicore setting but for a distributed one. However our focus is to have a high-level parallel implementation of the original PA model utilizing the computational power of multicore architectures [12].

2.5 Conclusion and Future Work

We showed that the PA-based generation of networks allows a high-level multicore implementation using the ABS language and its Haskell backend that supports cooperative multitasking via continuations and multicore parallelism via its lightweight threads. An experimental validation of a scalable distributed implementation of our model based on Haskell is presented in [12].

Future work will be dedicated toward optimizations of the Haskell runtime system for the ABS. Other work of interest is to formally restrict the use of shared data structures in the ABS to ensure encapsulation. One particular approach is to extend the compositional proof-theory of concurrent objects [31] with foot-prints [28] which capture write accesses to the shared data structures and which can be used to express disjointness of these write accesses.

Chapter 3

Preferential Attachment on Distributed Systems

3.1 Introduction

Massive social networks are structurally different from small networks synthesized by the same algorithm. Furthermore there are many patterns that emerge only in massive networks [56]. Analysis of such networks is also of importance in many areas, e.g. data-mining, network sciences, physics, and social sciences [16]. Nevertheless, generation of such extra-large networks necessitates an extra-large memory in a single server in the centralized algorithms.

The major challenge is generating large-scale social networks utilizing distributed-memory approaches where the graph, generated by multiple processes, is distributed among multiple corresponding memories. Few existing methods are based on a distributed implementation of the Preferential Attachment model (PA, chapter 2) among which some methods are based on a version of the PA model which does not fully capture its main characteristics. In contrast, we aim for a distributed solution which follows the original PA model, i.e., preserving the same probability distribution as the sequential one. The main challenge of a faithful distributed version of PA is to manage the complexity of the communication and synchronization involved.

In a distributed version, finding a target node in order for the new node to make connection with may cause an unresolved dependency, i.e., the target itself is not yet resolved. However this kind of dependencies must be preserved and the to-be-resolved target will be utilized when it is resolved. How to preserve these dependencies and their utilization give rise to low-level explicit management of the dependencies or, by means of powerful programming constructs, high-level implicit management of them.

The main contribution of this chapter is a new distributed implementation of an ABS model of PA. In this chapter, we show that ABS can be used as a powerful programming language for efficient implementation of cloud-based distributed

applications.

This chapter is organized as follows: Section 3.2 elaborates on the high-level proposed distributed algorithm using the notion of cooperative scheduling and futures. In Section 3.3, implementation-specific details of the model are presented. Finally, Section 3.4 concludes the chapter.

Related Work. In chapter 2 (section 2.4) the existing related work for sequential and parallel implementations of PA is presented. The work in this chapter is inspired by the work in [4] where a low-level distributed implementation of PA is given in MPI: the implementation code remains closed source (even after contacting the authors) and, as such, we cannot validate their presented results (e.g, there are certain glitches in their weak scaling demonstration), nor compare them to our own implementation.

Since efficient implementation of PA is an important and challenging topic, further research is called for. Moreover, our experimental data are based on a high-level model of the PA which abstracts from low-level management of process queues and corresponding synchronization mechanism as used in [4].

In [68], a high-level distributed model of the PA in ABS has been presented together with a high-level description of its possible implementation in Java. However, as we argue in Section 3, certain features of ABS pose serious problems to an efficient distributed implementation in Java. In this chapter, we show that these problems can be solved by a run-time system for ABS in Haskell and a corresponding source-to-source translation. An experimental validation of a scalable distributed implementation based on Haskell is presented in [10].

3.2 Distributed PA

In this section, we present a high-level distributed solution for PA which is similar to the ones proposed for multicore architectures in [12] (chapter 2) and distributed architectures in [4, 68], in a sense that they adopt *copy model* introduced in [54] to represent the graph. The main data structure used to model the graph which represents the social network is given in section 2.3.1. We use the same data structure for distributed PA as well.

The sequential algorithm of PA based on copy model is fairly straightforward and the unresolved slots of the array are resolved from left to right. The distributed algorithms however introduce more challenges. First of all, the global array should be distributed over multiple machines as local arrays. The indices of the global array are also mapped to the ones in the local arrays according to the partitioning policy. Secondly, there is the challenge of *unresolved dependencies*, a kind of dependency where the target itself is not resolved yet since either the process responsible for the target has not processed the target slot yet or the target slot itself is dependent on another target slot (chain of dependencies). Synchronization between the processes

to deal with the unresolved dependencies is the main focus of this chapter. Next we present the basic synchronization and communication mechanism underlying our distributed approach and its advantages over existing solutions.

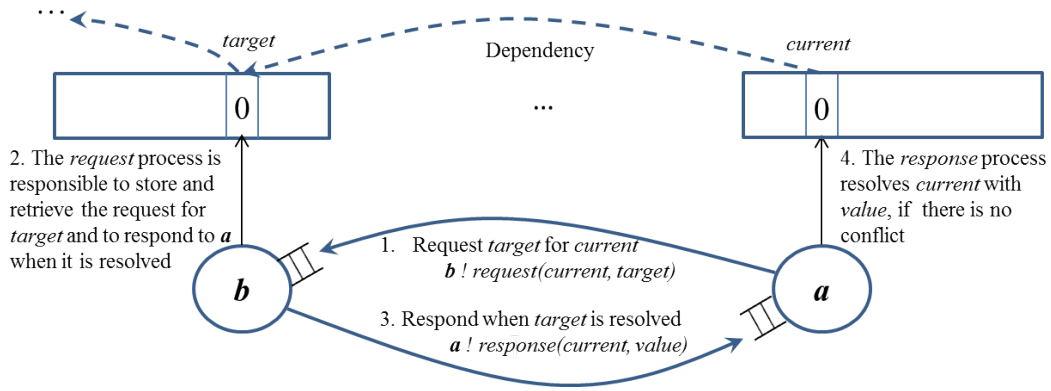
3.2.1 The Distributed ABS Model of PA

Two approaches are represented in Figure 3.1 which illustrate two different schemes of dealing with the unresolved dependencies in a distributed setting. In order to remain consistent with the original PA, both schemes must keep the unresolved dependencies and use the value of the target when it is resolved. Scheme A (used in [4]) utilizes message passing. If the target is not resolved yet, actor b explicitly stores the request in a data structure until the corresponding slot is resolved. Then it communicates the value with actor a. Actor b must also make sure the data structure remains consistent (e.g., it does not contain a request for a slot which is already responded).

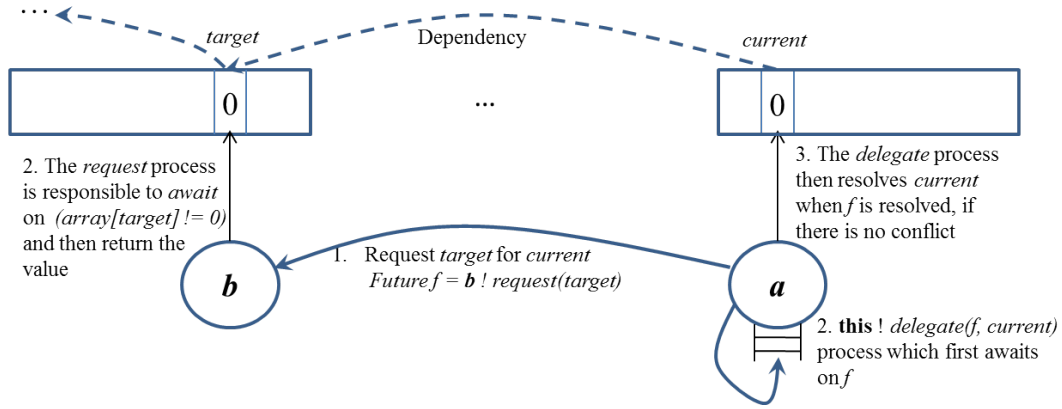
In addition to message passing, scheme B utilizes the notion of *cooperative scheduling*. Instead of having an explicit data structure, scheme B simply uses the *await* statement on $(target \neq 0)$. It suspends the request process until the target is resolved. The value is then communicated through the return value to actor a. Also *await* f? is skipped if the future f is resolved, and suspends the current process otherwise. This statement is used to synchronize on the return value of a called method. The above-mentioned await constructs eliminates the need for an explicit user-defined data structure for storing and retrieval of the requests. The following section describes an ABS implementation of the scheme B.

An ABS-like pseudo code which represents scheme B in the above section is given in Figure 3.2. The full implementation of the model is provided online[1]. The main body of the program, which is not mentioned in the figure, is responsible to set up the actors by determining their partitions, and sending them other parameters of the problem, e.g., n and m. Each actor then processes its own partition via *run* method. The function *whichActor* calculates and returns the index of the actor containing the target slot, based on n, m and the partitioning method. The request for the slot is then sent asynchronously to the actor and the future variable is sent as a parameter to the *delegate* function where the future value is obtained and checked for conflict. If there is no conflict, i.e., the new target is not previously taken by the source, then the slot is written with the target value. Recall that the one global array is divided into multiple local arrays, one per actor. Based on the partitioning method, n and m there is a mapping from the global indices to the local ones. The function *whichSlot* maps an index of the global array to the index of a local array. The *request* method is responsible to map the global index of the target to the local index function (via *whichSlot*) and *awaits* on it and returns the value once the slot is resolved. Note that, based on the same optimization of the array size discussed in chapter 2, the

[1] `https://github.com/kazadbakht/PA/blob/master/src/DisPA.abs`



(a) Simple message passing



(b) Message passing with futures and cooperative scheduling

Figure 3.1: The process of dealing with unresolved dependencies in an actor-based distributed setting

method *result(arr, target)* checks if the value for the index *target* is calculable then it returns the calculated value. Otherwise it checks the corresponding array index in the array and returns the value.

```
1: Each actor O executes the following in parallel
2: Unit run(...)
3: for each node i in the partition do
4:     for j = 1 to m do step
5:         target ← random[1..(i − 1)2m]
6:         current = (i − 1)m + j
7:         x = whichActor(target)
8:         Fut < Int > f = actor[x] ! request(target)
9:         this ! delegate(f, current)
10:
11:
12: Int request(Int target)
13: localTarget = whichSlot(target)
14: await (result(arr, localTarget) ≠ 0)
15:                                   ▷ At this point the target is resolved
16: return result(arr, localTarget)
17:
18:
19: Unit delegate(Fut < Int > f, Int current)
20: await f?
21: value = f.get
22: localCurrent = whichSlot(current)
23: if duplicate(value, localCurrent) then
24:     target = random[1..(current/m)2m]
25:                                   ▷ Calculate the target for the current again
26:     x = whichActor(target)
27:     Fut < Int > f = actor[x] ! request(target)
28:     this.delegate(f, current)
29: else
30:     arr[localCurrent] = value                                  ▷ Resolved
31:
32:
33: Boolean duplicate(Int value, Int localCurrent)
34: for each i in (indices of the node to which localCurrent belongs) do
35:     if arr[i] == value then
36:         return True
37: return False
```

Figure 3.2: The sketch of the proposed approach

3.3 Implementation

The distributed algorithm of Figure 3.2 is implemented directly in ABS, which is subsequently translated to Haskell code [10], by utilizing the ABS-Haskell [20] transcompiler (source-to-source compiler). The translated Haskell code is then linked against a Haskell-written parallel and distributed runtime API. Finally, the linked code is compiled by a Haskell compiler (normally, GHC) down to native code and executed directly.

The performance results of an experimental validation of the proposed approach in ABS-Haskell transcompiler is presented in [10]. The parallel runtime treats ABS active objects as Haskell's lightweight threads (also known as green threads), each listening to its own concurrently-modifiable process queue: a method activation pushes a new continuation to the end of the callee's process queue. Processes awaiting on futures are lightweight threads that will push back their continuation when the future is resolved; processes awaiting on boolean conditions are continuations which will be put back to the queue when their condition is met. The parallel runtime strives to avoid busy-wait polling both for futures by employing the underlying OS asynchronous event notification system (e.g. epoll, kqueue), and for booleans by retrying the continuations that have part of its condition modified (by mutating fields) since the last release point.

For the distributed runtime we rely on Cloud Haskell [32], a library framework that tries to port Erlang's distribution model to the Haskell language while adding type-safety to messages. Cloud Haskell code is employed for remote method activation and future resolution: the library provides us means to serialize a remote method call to its arguments plus a static (known at compile time) pointer to the method code. No actual code is ever transferred; the active objects are serialized to unique among the whole network identifiers and futures to unique identifiers to the caller object (simply a counter). The serialized data, together with their types, are then transferred through a network transport layer (TCP,CCI,ZeroMQ); we opted for TCP/IP, since it is well-established and easier to debug. The data are de-serialized on the other end: a de-serialized method call corresponds to a continuation which will be pushed to the end of the process queue of the callee object, whereas a de-serialized future value will wake up all processes of the object awaiting on that particular future.

The creation of Deployment Components is done under the hood by contacting the corresponding (cloud) platform provider to allocate a new machine, usually done through a REST API. The executable is compiled once and placed on each created machine which is automatically started as the 1st user process after kernel initialization of the VM has completed.

The choice of Haskell was made mainly for two reasons: the ABS-Haskell backend seems to be currently the fastest in terms of speed and memory use, attributed perhaps to the close match of the two languages in terms of language features:

Haskell is also a high-level, statically-typed, purely functional language. Secondly, compared to the distributed implementation sketched in Java [68], the ABS-Haskell runtime utilizes the support of Haskell's lightweight threads and first-class continuations to efficiently implement multicore-enabled cooperative scheduling; Java does not have built-in language support for algebraic datatypes, continuations and its system OS threads (heavyweight) makes it a less ideal candidate to implement cooperative scheduling in a straightforward manner. On the distributed side, layering our solution on top of Java RMI (Remote Method Invocation) framework was decided against for lack of built-in support for asynchronous remote method calls and superfluous features to our needs, such as code-transfer and fully-distributed garbage collection.

3.3.1 Implementing Delegation

The distributed algorithm described in Section 3 uses the concept of a *delegate* for asynchronicity: when the worker actor demands a particular slot of the graph array, it will spawn asynchronously an extra delegate process (line 9) that will only execute when the requested slot becomes available. This execution scheme may be sufficient for preemptive scheduling concurrency (with some safe locking on the active object's fields), since every delegate process gets a fair time slice to execute; however, in cooperative scheduling concurrency, the described scheme yields sub-optimal results for sufficient large graph arrays. Specifically, the worker actor traverses its partition from left to right (line 3), spawning continuously a new delegate in every step; all these delegates cannot execute until the worker actor has released control, which happens upon reaching the end of its `run` method (finished traversing the partition). Although at first it may seem that the worker actors do operate in parallel to each other, the accumulating delegates are a space leak that puts pressure on the Garbage Collector and, most importantly, delays execution by traversing the partitioned arrays "twice", one for the creation of delegates and one for "consuming them".

A naive solution to this space leak is to change lines 8,9 to a synchronous instead method call (i.e. `this.delegate(f,current)`). However, a new problem arises where each worker actor (and thus its CPU) continually blocks waiting on the network result of the request. This intensely sequentializes the code and defeats the purpose of distributing the workload, since most processors are idling on network communication. The intuition is that modern CPUs operate in much larger speeds than commodity network technologies. To put it differently, the worker's main calculation is much faster than the round-trip time of a request method call to a remote worker. Theoretically, a synchronous approach could only work in a parallel setting where the workers are homogeneous processors and requests are exchanged through shared memory with memory speed near that of the CPU processor. This hypothesis requires further investigation.

```
1: Unit run(...)
2: for each node i in the partition do
3:     for j = 1 to m do step
4:         target ← random[1..(i − 1)2m]
5:         current = (i − 1)m + j
6:         x = whichActor(target)
7:         Fut < Int > f = actor[x]! request(target)
8:         aliveDelegates = aliveDelegates + 1
9:         this! delegate(f, current)
10:        if aliveDelegates = maxBoundWindow then
11:            await aliveDelegates <= minBoundWindow
```

Figure 3.3: The modified run method with window of delegates.

We opted instead for a middle-ground, where we allow a window size of delegate processes: the worker process continues to create delegate processes until their number reaches the upper bound of the window size; thereafter the worker process releases control so the delegates have a chance to execute. When only the number of alive delegate processes falls under the window's lower bound, the worker process is allowed to resume execution. This algorithmic description can be straightforwardly implemented in ABS with boolean awaiting and a integer counter field (named *this.aliveDelegates*). The modification of the `run` is shown in Figure 3.3; Similarly the `delegate` method must be modified to decrease the *aliveDelegates* counter when the method exits.

Interestingly, the size of the window is dependent on the CPU/Network speed ratio, and the Preferential Attachment model parameters: nodes (n) and degree (d). In [10], the performance results of the PA model presented in this chapter in the Haskell backend are given. We empirically tested and used a fixed window size of $[500, 2000]$. Finding the optimal window size that keeps the CPUs busy while not leaking memory by keeping too much delegates alive for a specific setup (cpu,network,n,d) is planned for future work.

3.4 Conclusion and Future Work

In this chapter, we have presented a high-level distributed-memory algorithm that implements synthesizing artificial graphs based on Preferential Attachment mechanism. The algorithm avoids low-level synchronization complexities thanks to ABS, an actor-based modeling framework, and its programming abstractions which support *cooperative scheduling*. The experimental results for the proposed algorithm presented in [10] suggest that the implementation scales with the size of the distributed system, both in time but more profoundly in memory, a fact that permits the generation of PA graphs that cannot fit in memory of a single system.

For future work, we are considering combining multiple request messages in a single TCP segment; this change would increase the overall execution speed by having a smaller overhead of the TCP headers and thus less network communication between VMs, and better network bandwidth. In another (orthogonal) direction, we could utilize the many cores of each VM to have a parallel-distributed hybrid implementation in ABS-Haskell for faster PA graph generation.

Part III

Enhancing Parallelism

This part consists of the following chapters:

Chapter 4 Asynchronous Actor-based software programming has gained increasing attention as a model of concurrency and distribution. Many modern distributed software applications require a form of continuous interaction between their components which consists of streaming data from a server to its clients. In this chapter, we extend the basic model of asynchronous method invocation and return in order to support the streaming of data [13]. We introduce the notion of "future-based data streams" by augmenting the syntax, type system, and operational semantics of ABS. The application involving future-based data streams is illustrated by a case study on social network simulation.

Chapter 5 In this chapter we introduce a new programming model of multi-threaded actors which feature the parallel processing of their messages [15]. In this model an actor consists of a group of active objects which share a message queue. We provide a formal operational semantics, and a description of a Java-based implementation for the basic programming abstractions describing multi-threaded actors. Finally, we evaluate our proposal by means of an example application.

Chapter 4

Futures for Streaming Data

4.1 Introduction

Since the rapid growth in big data, data streaming is widely used in many distributed applications, e.g., telecommunications, event-monitoring and detection, and sensor networks. Data streaming is a client/server pattern which, in essence, consists of a continuous generation of data by the server and a sequential and incremental processing of the data by the client. Data streams are naturally processed differently from batch data. Functions cannot operate on data streams as a whole, as the produced data can be unlimited. Hence, new programming abstractions are required for the continuous generation and consumption of data in the streams.

Data streaming is highly relevant in modern distributed systems. Actor-based languages are specifically designed for describing such systems [1]. They provide an event-driven model of concurrency where messages are communicated asynchronously and processed by pattern matching mechanism [7]. Concurrent objects generalize this model to *programming to interface* discipline by modeling messages as asynchronous method invocations. The main contribution of this chapter is to integrate data streaming mechanism with concurrent object systems.

In this chapter, we extend the ABS language in order to support the streaming of data between a server and its clients. We introduce "future-based data streams" which integrates futures and data streams, and which specifies the return type of so-called *streaming* methods. Upon invocation of such a method a new future is created which holds a reference to the generated stream of data. Data items are added to the stream by the execution of a *yield* statement. Such a statement takes as parameter an expression the value of which is added to the stream, *without* terminating the execution of the method. The *return* statement terminates the execution of a streaming method, and is used to signal the end of data streaming. Even though no new data is produced, the existing data values in the stream buffer can be retrieved by the consumers.

The values generated by the server (the streaming method) can be obtained

incrementally and sequentially by a client by querying the future corresponding to this method invocation. By the nature of data streaming, it is natural to restrict the streaming to the asynchronous method calls. Therefore there is no support for synchronous invocation of streaming methods.

In this chapter, we introduce two different implementations of streams tailored to different forms of parallel processing of data streams. Obtaining data from a *destructive* stream involves the removal of the data, whereas in a *non-destructive* stream the data persists. Which of the implementation is used is determined by the caller of the streaming method (the *creator* of the stream) which is not necessarily the consumer of the data stream. The creator can then provide the consumers with a reference to the stream. Both the streaming method (producer) and the consumers which hold a reference to the data stream are *not* exposed to the underlying implementation of the stream, i.e., these different implementations are not represented by different *types* of data streams. This allows for a separation of concerns between the generation and processing of data streams, on the one hand, and their orchestration, on the other hand. This also enables reusability of the implementation of producers and consumers for both consumption approaches.

A preliminary discussion of the overall idea underlying this chapter is given in [11]. As an extension, in this chapter we introduce the different implementations of data streams, an operational semantics for both implementations of streams, a new type system which formalizes the integration of futures and data streams, and a proof of type-safety. Further, we show how the basic mechanism in ABS of cooperative scheduling of asynchronously generated method invocations itself can be used to implement data streams and the cooperative scheduling of streaming methods.

As a proof of concept, exploiting a prototype implementation for supporting future-based data streams on top of ABS, we present the usage of the above-mentioned feature in the implementation of a distributed application for the generation of distributed PA (chapter 2 and 3). The notion of data streaming abstracts from the specific implementation of ABS. In our case, we make use of the distributed Haskell backend of ABS [20] for the case study on future-based data streams reported in this chapter.

The overall contribution of this chapter is a formal model of streaming data in the ABS language, which fully complies and generalizes the asynchronous model of computation underlying the ABS language. Since ABS is defined in terms of a formal operational semantics which supports a variety of formal analysis techniques (e.g., deadlock detection [39] and [14]), we thus obtain a general formal framework for the modeling and analysis of different programming techniques for processing data streams, e.g., *map-reduce* and *publish-subscribe* [34]. To the best of our knowledge, our work provides a first formal type system and operational semantics for a general notion of streaming data in a high-level actor-based programming language.

Plan of the chapter This chapter is organized as follows: the notion of a future-based data stream is specified as an extension of ABS in Section 4.2. In section 4.3, it is shown that the well-typedness of a program in the extended ABS is preserved. Section 4.4 discusses the usage of streams in a distributed setting. In section 4.5, an implementation of data streams is given as an API written in ABS. In Section 4.6, a case study on social network simulation is discussed, which uses the proposed notion of streams. Related works are discussed in section 4.7. Finally we conclude in section 4.8.

4.2 Future-Based Data Streams

In this section, we define future-based data streaming in the context of the ABS language. A *streaming* method is statically typed, namely, the return type of the method is followed by the keyword **`stream`**, specifying that the method streams values of that type. As mentioned before, ABS features a programming to interfaces discipline. Therefore the caller can asynchronously call a streaming method, provided that the interface of the callee includes the method definition.

Data streaming is defined as a stream of return values from a callee to the data consumers of the stream in an asynchronous fashion. An invocation of a streaming method creates a stream. The callee first create an empty stream, and then produces and stores data to the stream buffer via the **`yield`** statement. The caller assigns the invocation to a variable of type **`Stream`**`<T>` for the return type `T` **`stream`** of the callee. The stream variable can be passed around. Therefore different variables in multiple processes (a process is the execution of an asynchronous method call) may refer to the same stream and retrieve data from it.

We distinguish between two different kinds of streams: *destructive* and *non-destructive* streams. The kind of stream is determined by the caller upon the invocation of the streaming method. In destructive streams, values are retrieved from a FIFO queue which stores the data produced but not yet consumed. Querying availability of data values in an empty queue gives rise to a cooperative release of control (further discussed below). Also an attempt to take a value from a stream where the callee is terminated (and thus no further data streaming will take place) gives rise to the execution of a block of statements specified by programmer for this reason, thus avoiding the generation of a corresponding error (see below). Parallel processes which have access to the same destructive stream compete for the different data items produced. Consequently, the parallel processing of destructive data streams gives rise to *race conditions*, in the sense that different order of requests to read from a stream may correspondingly give rise to different data values. Note that at most one process can destructively read a specific data value. On the other hand, a non-destructive stream allows complete *sharing* of all the data produced which are only *read* to be processed. As described in more detail below, non-destructive streams maintain access by means of cursors at different positions of the buffer which allows

for its asynchronous parallel processing.

Abstracting from the typing information, to be discussed in more detail below, the syntax of our proposed extension of ABS, i.e., that of future-based data streams, is specified in Figure 4.1, where e denotes an expression (i.e., a variable name, etc), $\overline{e}$ denotes a sequence of expressions, x is a variable name, m is a method name, and s denotes a sequence of statements.

$$\begin{aligned} s ::= \; & s; s \mid x = [\textbf{nd}]\ e!m(\overline{e}) \mid \textbf{yield}\ e \mid \textbf{return} \mid \textbf{suspend} \mid \\ & \textbf{await}\ e?\ \textbf{finished}\ \{s\} \mid x = e.\textbf{get finished}\ \{s\} \end{aligned}$$

Figure 4.1: Syntax

In the asynchronous invocation $x = [\textbf{nd}]\ e!m(\overline{e})$ of a streaming method, the optional keyword **nd** indicates the creation of a new non-destructive stream.

Execution of the **yield** statement, which can only be used in the body of a *streaming* method, consist of queuing the value of the specified expression.

Execution of the **return** statement by a streaming method indicates termination of the data generation which is signaled to the consumers of the stream by queuing the special value η.

The active process unconditionally releases control by **suspend**. The object is then idle and can activate a suspended process. The **await-finished** statement allows to check the buffer of the stream denoted by the expression e in the following manner: if there is at least one proper value, different from the signal η, in the buffer, the statement is skipped. In case the buffer is empty, the current process *suspends* such that the object can activate another process. The statement s is executed in case the buffer only contains the signal η.

The **get-finished** statement allows to actually retrieve (in case of a destructive stream) or read (in case of a non-destructive stream) a next data value. It however *blocks* the whole object when the buffer is empty. As above, statement s is executed when the buffer only contains the signal η.

In **await-finished** and **get-finished**, the keyword **finished** and its following block can be omitted if the block is empty.

We next illustrate the difference in the behaviour of destructive and non-destructive access to a stream by the following simple toy example. Consider the streaming method `m()`:

```
Int stream m() {
   yield 1; yield 2; return;
}
```

This method adds 1 and 2, followed by a termination token to the resulting stream buffer.

The following snippet asynchronously calls the above method definition `m()` on some object `o` which gives rise to two references `r1` and `r2` to the resulting destructive stream.

```
Stream<Int> r1 = o!m();
Stream<Int> r2 = r1;
```

The following code uses the above references, with the assumption that it is the only process that consumes data items of the stream:

```
    Int x, y, z;
(1) Int x = r1.get finished {x = -1};
(2) Int y = r2.get finished {y = -1};
(3) Int z = r1.get finished {z = -1};
```

Once the process corresponding to the method call `m()` on (possibly remote) object `o` is executed and the results are provided to the stream, the values 1, 2, and -1 are assigned to `x, y` and `z`, respectively. These values are consumed from the stream and assigned to the variables incrementally as soon as they are provided by `m()`. In the above code, the object possibly blocks on any of the three statements, if a value (whether an integer or the terminating token) is not yielded to the stream yet. The statement `(1)` destructively reads 1 from the stream via `r1` and assigns it to `x`. The statement `(2)` destructively reads 2 from the same stream via the other reference `r2` and assigns it to `y`. However, the statement `(3)` runs the **`finished`** statement which assigns -1 to `z`, since it reads the terminating token (i.e., the stream is already terminated). Any further *get* operations on *every* variable referring to the stream also read the terminating token.

To show how a non-destructive stream works in the same setting, suppose we use the following references `r1` and `r2` in the above code (note that the keyword **`nd`** denotes that the resulting stream is non-destructive).

```
Stream<Int> r1 = nd o!m();
Stream<Int> r2 = r1;
```

With the same incremental production of values and blocking mechanism, in this setting the values 1, 1, and 2 are assigned to `x, y` and `z`, respectively. The statement `(1)` non-destructively reads 1 from the stream via `r1` and assigns it to `x`. The statement `(2)` non-destructively reads 1 from the same stream via the other reference `r2` (with its own cursor to the stream) and assigns it to `y`. Finally, the statement `(3)` assigns 2 to `z`, since the cursor of `r1` is already moved forward by statement `(1)`. Note that any number of further *get* operations on `r1` will read the terminating token. It is important to observe the role of cursors per each stream variable that gives rise to such behaviour.

Note that the assignment of a non-destructive reference (`r2 = r1`) is different from the standard ABS assignment in the sense that, in addition to the assignment of the reference to stream, it also assigns the cursor. Based on this design, the

copying is required as each stream variable represents a new access to the stream to *all* data values from the position its cursor denotes.

4.2.1 Design Decisions

Integration of streams with ABS, where we enjoy the advantages of both, roots in the ever-growing application of data streaming in different domains. The consumption approaches of the stream (i.e., destructive or non-destructive) are not fixed in the streams in form of different data types. Instead, the creator of the stream determines the consumption approach of the stream instance, in order to maintain generality. Note that the creator of the stream is not necessarily the consumer of the stream, and by design, it can be considered as part of the producer process (e.g., using *factory method* design pattern) that forces one of the above consumption approaches to the consumers.

We support both destructive and non-destructive data streams, as they can be naturally used to implement, respectively, *one-of-n* semantics (only one consumer reads a given data as in, e.g., *data parallelism* model), and *one-to-n* message delivery (a given data can be read by all such consumers as in, e.g., one-to-many *trainer and learners* and *publish/subscribe* model). Also integration of data streaming and cooperative scheduling enables enhancing concurrency and parallelism on the consumer side.

Note that the above two approaches allow for designing a third hybrid consumption approach where, at the intra-object level, every access to the stream buffer is via an object field (shared variable), and at the inter-object level, the cursor is copied (i.e., via passing parameters in method invocations).

4.2.2 Example of Destructive Streams

The code example in Figure 4.2 illustrates the use of ABS destructive data streams in modeling a parallel *map-reduce* processing of a data stream. The mapping step maps each streamed data value of type `T` to a data value of type `Int`, and the reduction step calculates the average of those integers.

An ABS program is a set of interface and class definitions, followed by the main block of the program, which is an anonymous block at the end of the program. The main block is the initial run-time process (similar to `public static void main` in java). Each class implements at least one interface. The type of a reference variable to an object can only be an interface, and the object must be an instance of a class that implements the interface. Every object instance is an active object, namely, it features a dedicated thread of control, and can have (at most) one active process among its processes. Each process of an object is initiated by an asynchronous call of a method of the object.

```
interface IMapper<T> {
 Int stream map(Stream<T> s);
}

interface IReducer<T> {
 Pair<Int, Int> reduce(Stream<T> s);
}
interface IPar<T> {
 Int start(Stream<T> s, Int num);
}
interface IProd<T> {
  T stream streamer();
}
class Mapper()
implements IMapper<T> {
 Int stream map(Stream<T> s) {
  Bool last = False;
  while(last==False){
    T v = s.get finished {last=True};
    if (last == False) {
      yield v.value();
    }
  }
  return;
 }
}
class Reducer()
implements IReducer<T> {
 Pair<Int, Int> reduce(Stream<Int> s)
 {
  Bool last = False;
  Int count = 0;
  while(last==False){
    Int v = s.get
      finished {last=True};
    if (last == False){
      count = count + 1;
      sum = sum + v;
    }
  }
  return Pair(sum, count);
 }
}
```

```
class Producer implements IProd<T>
{
  T stream streamer() {
    // yields a seq of data of type T
    return;
  }
}
class Par implements IPar<T> {
 Int start(Stream<T> s, Int num) {
  Int m = 1;
  Int sum = 0, avg = 0, count = 0;
  List<Fut<Pair<Int, Int>>> l = Nil;
  while(m<=num) {
    IMapper<T> p = new Mapper();
    Stream<Int> s2 = p!map(s);
    IReducer<T> q = new Reducer();
    l = Cons(q ! reduce(s2), l);
    m=m+1;
  }
  while (l != Nil) {
   Pair<Int,Int> pair = head(l).get;
   case (pair) {
    Pair(a, b) => {
        sum = sum + a;
        count = count + b;
    }
   }
   l = tail(l);
  }
  if (count > 0) return sum / count;
  else return 0;
 }
}
{// Main block
 IProd<T> producer = new Producer();
 Stream<T> s = producer ! streamer();
 IPar<T> par = new Par();
 Int average = par.start(s, 4);
}
```

Figure 4.2: Parallel data processing based on Map-Reduce data model

The program is composed of four interfaces: `IProd` types a class with a streaming method to stream the data values of type `T` to be processed. The interface `IPar` types a class for spawning multiple chains of active objects for map-reduce processing. Each chain is a pipeline processing of the data values retrieved from the stream which is shared among the chains. The interfaces `IMapper` and `IReducer` type the objects that form a pipeline chain. These interfaces are implemented by four classes `Producer`, `Par`, `Mapper` and `Reducer`, respectively. The above definitions are followed by the main block of the program. As shown in the main block, the general idea is that the data values of the stream `s` will be processed in parallel by `num` computationally identical pipelines, and the aggregated result, which is the average of those values, is returned as the final result. Runtime control and data flow of the example are also illustrated in Figure 4.3, where each thread represents a process created by an asynchronous method call.

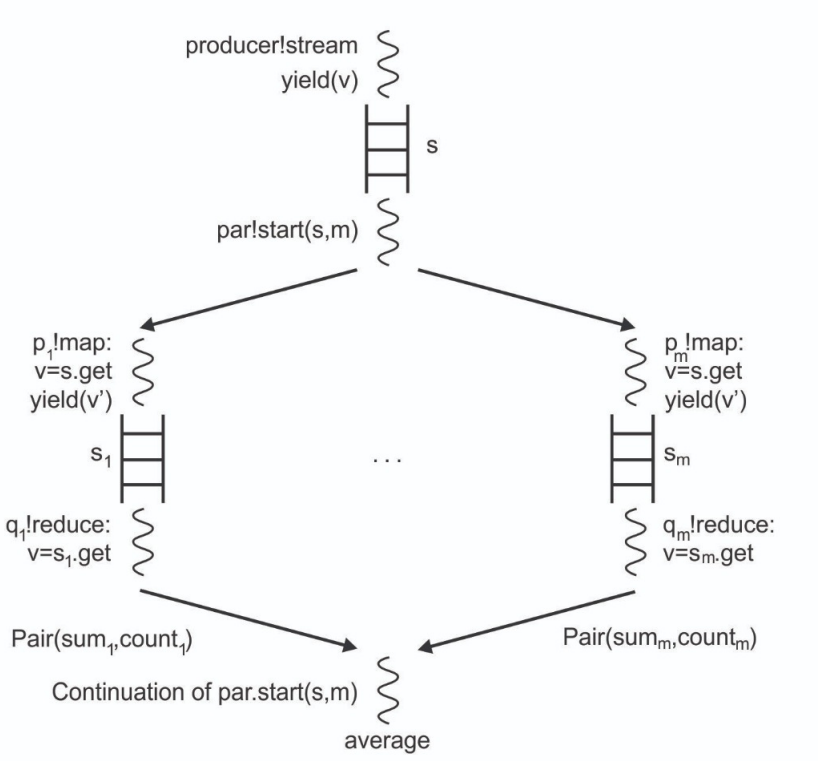


Figure 4.3: Control and data flow

The asynchronous invocation of method `streamer` on `producer` in the main block returns a reference `s` to a stream. The method `start` provided by `IPar<T>` enables parallel processing of the stream by creating multiple chains (`num`) of two active objects of type `IMapper<T>` and `IReducer<T>`, where the former retrieves values from `s`, and yields a mapped value of type integer to an intermediate stream `s2`, and the latter consumes those integers from `s2` and reduces them to a pair which is the `sum` and the `count` of those integers processed by one chain. The futures

of the pairs resulting from calling `reduce` in different chains are stored in list `l`. The elements of the list are then used as a synchronization means, namely, awaiting until each process resolves the corresponding future by providing the return value. Finally the average is calculated by the `start` from the reduced pairs.

Similar to parallel map-reduce transformations on streams in languages like Scala, the following pseudo-code can be used as a simplified abstract replacement for the code in 4.2:

```
s.par(num).map(_.value).average();
```

where a sequence of *transformation* methods (e.g., *map* and *filter*) followed by a *reduce* method (e.g., *count* and *average*) can be executed in parallel by *num* threads (modeled by active objects) on stream `s`.

Note that our implementation utilizes two ways of parallelism: 1) *horizontal parallelism*, which is achieved by creating multiple chains of active objects, e.g., p_i and q_i and intermediate streams s_i in Figure 4.3, and 2) *vertical parallelism*, which is achieved by pipeline processing, e.g., the process *map* in p_i that yields values to s_i runs in parallel with *reduce* in q_i that consumes the values immediately upon their availability.

4.2.3 Example of Non-Destructive Streams

In the example specified in Fig. 4.4, we represent a basic means of *publish/subscribe* communication via non-destructive streams in a social network such as Twitter. An object of class `Member` denotes a member in the network that can follow and be followed by multiple members. The main idea is to implement each member object such that: 1) it can follow multiple members by reading their stream of posts 2) its stream of posts can be read by multiple members that follow the member 3) it can post new items to its stream. The object naturally needs to interleave these tasks. To this aim, each member is modeled as an actor with a process to post new items to its stream (`share`), a set of processes one per each member it follows, in order to read their streams (`follow`), and a set of processes from other members that request to follow the member (`request`). These processes can be interleaved by the ABS *cooperative scheduling*. The active process can cooperatively release control conditionally, e.g., the **`await`** statement in `follow` which checks whether there is no new post to be read from a specific member, or unconditionally, e.g., the **`suspend`** in `share` after posting a new item gives rise to unconditional release of control. In both cases, other processes of the member object can be activated.

The method `follow` sends a request to a member denoted by the argument p. The data (i.e., posts) can be retrieved from the resulting stream r of the member `p` by the current member. In other words, the current member object *follows* object p. Further, a followed member returns a reference to the same data stream for all the followers, denoted by r in the the class `Member`. Each follower uses its corresponding

cursor to read from the stream belonging to the followed member. Note the difference between the return types of `share` and `request`. The former is a streaming method that creates and populates a stream, and can *only* be called asynchronously with the return type **`Stream`**`<Post>`, whereas the latter is a non-streaming method that returns a reference to an existing stream, and returns **`Stream`**`<Post>` or **`Fut`**`<`**`Stream`**`<Post>>`, respectively, depending on being called synchronously or asynchronously.

```
interface IMem {
  Unit run();
  Unit follow(IMem p);
  Stream<Post> request();
}

class Member implements IMem
{
  Stream<Post> r;
  // r is a stream of
  // posts for followers

  Unit run() {
    if (r == null)
      r = nd this!share();
  }

  Post stream share() {
    Post post;
    while(True) {
      // Next post is ready
      yield post;
      suspend;
    }
    return;
  }

 Unit follow(IMem p) {
   Fut<Stream<Post>> f =
     p ! request();
   await f?;
   Stream<Post> r = f.get;
   Bool last = False;
   while(last = False) {
      await r? finished
      {
        //probably p left!
        last = true;
      }
      Post post = r.get;
      // consume post
   }
 }

 Stream<Post> request() {
   // accept as a follower
   return r;
 }
}
```

Figure 4.4: Parallel data processing based on publish/subscribe pattern

By **`await`** statement, a *follow* process queries the availability of the next post that is *new* from the perspective of the non-destructive stream variable `r`, denoted by the variable cursor. If the new post is available it is retrieved and consumed. Otherwise the process is suspended so that another enabled process is activated. As such, the member receives posts from all the members it follows, processes the

follow *requests* of other members, and posts new data. The stream corresponding to a followed member can signal the termination. In such case, the `follow` process in the follower object which corresponds to a followed member terminates after retrieving the remaining posts, as the *finished* block of the await statement falsifies the loop condition. The process that instantiates a new member (not mentioned here) also initiates the member by calling the `run` which itself calls the `share` method which returns a new stream and continuously adds new posts to it.

4.2.4 Type System

The ABS type system is presented in [48]. An extension of the type system is specified below using the same notation, which types the streams and the statements that use them (Figure 4.5). A *typing context* Γ is a mapping from names to types, where the names can be variables, constants and method names. A *type lookup* is denoted by $\Gamma(x)$, which returns the type of the name x. By $\Gamma[x \mapsto T]$ we denote the update of Γ such that the type of x is set to T. Then $\Gamma[x \mapsto T](x) = T$ and $\Gamma[x \mapsto T](y) = \Gamma(y)$ if $x \neq y$. An over-lined $\overline{e}$ denotes a sequence of syntactic entities e.

The basic idea underlying the typing rules regarding streams in Figure 4.5 is that the type $\mathsf{stream}\langle T\rangle$ of streams of data items of type T itself can*not* be defined as a *subtype* of $\mathsf{fut}\langle T\rangle$, since for a future variable x, a query $x?$ gives rise to a Boolean guard whereas for a stream variable x, the query $x?$ is not a Boolean guard because it not only checks whether the stream is empty or not but also whether it has terminated. On the other hand, the type $\mathsf{fut}\langle T\rangle$ of futures that refer to return values of type T itself can be defined as a sub-type of $\mathsf{stream}\langle T\rangle$ (as specified by the rule T-FutureStream), where the stream buffer is either empty (denoted by a sentinel $\perp$) or contains an *infinite* sequence of the particular return value. For such streams the **`finished`** statement never executes, as there is no termination token. Note also that for such infinite streams there is no difference between destructive or non-destructive reads.

We proceed with a brief explanation of the typing rules. A streaming method is well-typed by T-StreamMethod, if its body s is well-typed in the typing context extended by the parameters, local variables, and the return stream. The *destiny* variable in ABS is a local variable which holds a reference to the return stream (or future). A regular (non-streaming) method is similarly well-typed by T-Method in core ABS.

By T-AsyncCall, an asynchronous method call to a non-streaming method has type $\mathsf{fut}\langle T\rangle$, if its corresponding synchronous call has type T. Whereas by T-AsyncStream the type of an asynchronous call of a streaming method is of type $\mathsf{stream}\langle T\rangle$, if the interface T' of the callee includes the streaming method. As in ABS, by T-SyncCall, a call to a method m has type T if its actual parameters have types $\overline{T}$ and the signature $\overline{T} \to T$ matches a signature for m in the known interface of the callee (given

by an auxiliary function *match*). The rule does not allow synchronous call on a streaming method (note the difference between how the function *match* is used in T-AsyncStream and T-SyncCall).

The **yield** statement is well-typed in a streaming method by T-Yield, if the type of e is T and the enclosing method is a streaming method of type T. **return** statement without parameter is only used in a streaming method to signal the termination of streaming and the method, and is well-typed by T-ReturnStream.

The T-Return forces that the expression e of the **return** statement in a non-streaming method is of type T, the return type of the enclosing method.

The **await-finished** is well-typed by T-AwaitStream, if a stream of some type T is awaited and if the statement s is also well-typed. It is not difficult to see how the statement **get-finished** is well-typed by T-GetStream.

$$\text{(T-FutureStream)}\ \frac{\Gamma \vdash e : \mathsf{fut}\langle T\rangle \quad T \preccurlyeq T'}{\Gamma \vdash e : \mathsf{stream}\langle T'\rangle}$$

$$\text{(T-Method)}\ \frac{\Gamma' = \Gamma[\overline{x} \mapsto \overline{T}, \overline{x'} \mapsto \overline{T'}] \quad \Gamma'[\mathrm{destiny} \mapsto \mathbf{fut}\langle T''\rangle] \vdash s}{\Gamma \vdash T''\ m\ (\overline{T}\ \overline{x})\{\overline{T'}\ \overline{x'}; s\}}$$

$$\text{(T-StreamMethod)}\ \frac{\Gamma' = \Gamma[\overline{x} \mapsto \overline{T}, \overline{x'} \mapsto \overline{T'}] \quad \Gamma'[\mathrm{destiny} \mapsto \mathsf{stream}\langle T''\rangle] \vdash s}{\Gamma \vdash T''\ \mathsf{stream}\ m(\overline{T}\ \overline{x})\ \{\overline{T'}\ \overline{x'}; s\}}$$

$$\text{(T-ReturnStream)}\ \frac{\Gamma(\mathrm{destiny}) = \mathsf{stream}\langle T\rangle}{\Gamma \vdash \mathbf{return}}$$

$$\text{(T-AsyncCall)}\ \frac{\Gamma \vdash e.m(\overline{e}) : T}{\Gamma \vdash e!m(\overline{e}) : \mathsf{fut}\langle T\rangle}$$

$$\text{(T-SyncCall)}\ \frac{\Gamma \vdash e : T' \quad \Gamma \vdash \overline{e} : \overline{T} \quad \mathrm{match}(m, \overline{T} \to T, T')}{\Gamma \vdash e.m(\overline{e}) : T}$$

$$\text{(T-Return)}\ \frac{\Gamma \vdash e : T \quad \Gamma(\mathrm{destiny}) = \mathsf{fut}\langle T\rangle}{\Gamma \vdash \mathbf{return}\ e}$$

$$\text{(T-AsyncStream)}\ \frac{\Gamma \vdash e : T' \quad \Gamma \vdash \overline{e} : \overline{T} \quad \mathrm{match}(m, \overline{T} \to T\ \mathsf{stream}, T')}{\Gamma \vdash [\mathbf{nd}]\ e!m(\overline{e}) : \mathsf{stream}\langle T\rangle}$$

$$\text{(T-Yield)}\ \frac{\Gamma \vdash e : T \quad \Gamma(\mathit{destiny}) = \mathsf{stream}\langle T\rangle}{\Gamma \vdash \mathbf{yield}\ e}$$

$$\text{(T-AwaitStream)}\ \frac{\Gamma \vdash e : \mathsf{stream}\langle T\rangle \quad \Gamma \vdash s}{\Gamma \vdash \mathbf{await}\ e?\ \mathbf{finished}\ \{s\}}$$

$$\text{(T-GetStream)}\ \frac{\Gamma \vdash e : \mathsf{stream}\langle T\rangle \quad \Gamma \vdash s \quad \Gamma \vdash x : T}{\Gamma \vdash x = e.\mathbf{get\ finished}\ \{s\}}$$

Figure 4.5: Type system

4.2.5 Operational Semantics

The operational semantics of the proposed extension is presented below as a transition system in SOS style [61]. First we extend the ABS run-time configuration and then present those rules in the transition system that involve destructive and non-destructive streams.

Runtime Configuration

The runtime syntax of ABS is extended by the notion of stream is illustrated in Figure 4.6. Configurations *cn* consist of objects (*object*), invocation messages (*invoc*), futures (*fut*), and data streams (*stream*). The commutative and associative composition operator on configurations is denoted by whitespace. The empty configuration is denoted by ϵ.

$$
\begin{array}{rcl@{\qquad}rcl}
cn & ::= & \epsilon \mid fut \mid stream \mid & & & \\
 & & object \mid invoc \mid cn\ cn & & & \\
object & ::= & ob(o, a, p, q) & p & ::= & process \mid \texttt{idle} \\
fut & ::= & fut(f, value) & a & ::= & T\ x\ v \mid a, a \\
stream & ::= & stream(f, u) & value & ::= & v \mid \bot \\
process & ::= & \{a \mid s\} \mid \texttt{error} & u & ::= & u.u \mid v \mid \eta \mid \bot \\
q & ::= & \epsilon \mid process \mid q\ q & v & ::= & o \mid f \mid t \mid (f, n) \\
invoc & ::= & invoc(o, f, m, \overline{v}) & & &
\end{array}
$$

Figure 4.6: Runtime configuration

The term $ob(o, a, p, q)$ represents an object where o is the object identifier, a assigns values to the object's fields, p is an active process (or idle), and q represents a set of suspended processes.

The term $invoc(o, f, m, \overline{v})$ represents an invocation message, where o is the callee object, f is the identifier of a rendezvous for the return value(s) of the method invocation which can be a stream or a future, depending on the invoked method being a *streaming* method or not, m is the name of the invoked method, and $\overline{v}$ are its arguments.

A process $\{a \mid s\}$ consists of an assignment a of values to the local variables, and a statement s. A process results from the activation of a method invocation in a callee with actual parameters, and an associated future or stream. An **`error`** is a process where the binding of such method invocation does not succeed.

A future is represented by $fut(f, value)$, where f is the future identifier and *value* denotes its current value which can either be the actual value returned or $\bot$ which denotes the absence of a return value.

Both destructive and non-destructive streams are semantically represented by $stream(f, u)$, where f is the stream identifier, and u denotes its buffer. Nevertheless, the stream variables referring to destructive streams just hold the stream id, whereas the value of a stream variable referring to a non-destructive stream is a pair (f, n), where n denotes the associated position in the stream.

The buffer u is a FIFO queue, which contains a sequence of values v, and a special symbol, either $\bot$ which is a sentinel denoting end of buffer, or η which denotes termination of streaming. The $\bot$ is replaced by η after adding the last valid

value to the queue when the streaming method terminates. The $u = v.u'$ denotes the head v of the queue u, and its tail u'. In $u' = u.v$, enqueuing the value v to the end of the queue u forms the updated queue u'. The auxiliary function $\text{elem}(u, n)$ returns the content at the position n of the sequence u starting from 0.

A value v can be an object identifier, a future or stream identifier, a term t which is a value of a primitive type, or a pair (f, n) which is a value of a variable referring to a non-destructive stream.

Note that all the identifiers in a configuration are unique and terminal: o is used for object, and f both for future and stream identifiers.

The rules of Figure 4.8 and 4.9 operate on the elementary configurations. To have the rules to apply to full configurations, we need the following rule as well:

$$\frac{cn' \rightarrow cn''}{cn\ cn' \rightarrow cn\ cn''}$$

We also use the reduction system proposed in the ABS formal model to evaluate expressions, e.g., $f = [\![e]\!]^{cn}_{a \circ l}$ in the active process of $ob(o, a, \{l|s\}, q)$ holds if the expression e evaluates to the stream identifier f, in an assignment composed of a and l, where the configuration $cn\ ob(o, a, \{l|s\}, q)$ is given, and cn contains $stream(f, u)$. By definition, $a \circ l(x) = l(x)$ if $x \in dom(l)$ or $a \circ l(x) = a(x)$ otherwise.

The following rule Async-Call represents asynchronous method invocation in core ABS extended with a check that it is not a streaming method:

$$\frac{o' = [\![e]\!]_{(a \circ l)} \quad \overline{v} = [\![\overline{e}]\!]_{(a \circ l)} \quad \text{fresh}(f) \quad \neg\text{streamer}(o'.m(\overline{v}))}{\begin{array}{c} ob(o, a, \{l|x = e!m(\overline{e}); s\}, q) \\ \rightarrow ob(o, a, \{l|x = f; s\}, q) \quad invoc(o', f, m, \overline{v}) \quad fut(f, \bot) \end{array}} \text{(Async-Call)}$$

where it sends an invocation message to object o' with the method name m, the future f and the actual parameters $\overline{v}$. The return value of f is undefined (i.e., $\bot$). Note that, based on Figure 4.6, the definition of v also includes the values f and (f, n) for destructive and non-destructive streams in the extended semantics. Therefore streams can be passed as actual parameters and assigned to formal parameters.

Also for the chapter to be self-contained, the following rules from the core ABS [48] are mentioned. Rules Assign-Local and Assign-Field assign value of expression e to the variable in, respectively, environment l for local variables or environment a for fields of object a. Rule Suspend suspends the active process unconditionally. Rule Activate activates a process p that is *ready* to execute for the idle object o from the set q of its suspended processes.

$$\frac{x \in \mathrm{dom}(l) \quad v = [\![e]\!]_{(a \circ l)}}{ob(o,a,\{l|x = e;s\},q) \rightarrow ob(o,a,\{l[x \mapsto v]|s\},q)} \text{(Assign-Local)}$$

$$\frac{x \in \mathrm{dom}(a) \quad v = [\![e]\!]_{(a \circ l)}}{ob(o,a,\{l|x = e;s\},q) \rightarrow ob(o,a[x \mapsto v],\{l|s\},q)} \text{(Assign-Field)}$$

$$ob(o,a,\{l|\mathbf{suspend};s\},q) \rightarrow ob(o,a,\mathrm{idle},q \cup \{l|s\}) \text{(Suspend)}$$

$$\frac{p = \mathrm{select}(q,a)}{ob(o,a,\mathrm{idle},q) \rightarrow ob(o,a,p,q \backslash p)} \text{(Activate)}$$

The auxiliary method $\mathrm{select}(q,a,l)$ selects the ready process by ensuring the process will not be immediately re-suspended based on the states of a and l. Note that we abstract from the notion of the ABS concurrent object group *cog* in the rule. Also $dom(a)$ denotes the set of variables in the environment a.

In the rest of this section, we present the semantic rules of the extended ABS, where a data stream is involved. Given in Figure 4.7, the rules for the callee side, which only write to the stream, are independent from how the stream is read (i.e., destructively or non-destructively). In the rule Yield, the active process, which is a streaming method, enqueues the value v to the buffer of the stream f, followed by the sentinel $\perp$. The rule ReturnStream enqueues the value η to the buffer of the stream f, which is a token denoting termination of streaming values in the buffer.

$$\frac{v = [\![e]\!]^{cn}_{a \circ l} \quad l(\mathrm{destiny}) = f}{ob(o,a,\{l|\mathbf{yield}\ e;s\},q) \quad stream(f,u.\perp) \rightarrow ob(o,a,\{l|s\},q) \quad stream(f,u.v.\perp)} \text{(Yield)}$$

$$\frac{l(\mathrm{destiny}) = f}{ob(o,a,\{l|\mathbf{return};s\},q) \quad stream(f,u.\perp) \rightarrow ob(o,a,\mathbf{idle},q) \quad stream(f,u.\eta)} \text{(ReturnStream)}$$

Figure 4.7: Operational semantics of streams on the callee side

In the following, the rules for destructive and non-destructive access to the data stream are given. Note that the D and ND are prefixed to the rule names, which stand for the destructive and non-destructive streams, respectively.

Semantics of Destructive Streams

In the rule D-AsyncCall, the object o calls asynchronously a streaming method m with arguments $\overline{v}$ on object o'. The return stream is destructive with the fresh iden-

tifier f as the access mode to the return stream of a streaming method is destructive by default. We also use two auxiliary functions in this rule as follows: the function *streamer*$(o.m(\overline{v}))$ checks if the method $m(\overline{v})$ of the object o is a streaming method. The function *fresh*(f) guarantees that the newly introduced name f is not already used in the system.

(D-AsyncCall)
$$\frac{o' = [\![e]\!]^{cn}_{a \circ l} \quad \overline{v} = [\![\overline{e}]\!]^{cn}_{a \circ l} \quad \text{fresh}(f) \quad \text{streamer}(o'.m(\overline{v}))}{ob(o, a, \{l|x = e!m(\overline{e}); s\}, q) \rightarrow ob(o, a, \{l|x = f; s\}, q) \quad invoc(o', f, m, \overline{v}) \quad stream(f, \bot)}$$

(D-AwaitTrue)
$$\frac{f = [\![e]\!]^{cn}_{a \circ l}}{ob(o, a, \{l|\textbf{await}\ e?\ \textbf{finished}\ \{s_1\}; s_2\}, q) \quad stream(f, v.u) \rightarrow ob(o, a, \{l|s_2\}, q) \quad stream(f, v.u)}$$

(D-AwaitFalse)
$$\frac{f = [\![e]\!]^{cn}_{a \circ l}}{ob(o, a, \{l|\textbf{await}\ e?\ \ \textbf{finished}\ \{s_1\}; s_2\}, q) \quad stream(f, \bot) \rightarrow ob(o, a, \{l|\textbf{suspend}; \textbf{await}\ e?\ \textbf{finished}\ \{s_1\}; s_2\}, q) \quad stream(f, \bot)}$$

(D-AwaitTerminate)
$$\frac{f = [\![e]\!]^{cn}_{a \circ l}}{ob(o, a, \{l|\textbf{await}\ e?\ \ \textbf{finished}\ \{s_1\}; s_2\}, q) \quad stream(f, \eta) \rightarrow ob(o, a, \{l|s_1; s_2\}, q) \quad stream(f, \eta)}$$

(D-GetTrue)
$$\frac{f = [\![e]\!]^{cn}_{a \circ l}}{ob(o, a, \{l|x = e.\textbf{get}\ \ \textbf{finished}\ \{s_1\}; s_2\}, q) \quad stream(f, v.u) \rightarrow ob(o, a, \{l|x = v; s_2\}, q) \quad stream(f, u)}$$

(D-GetTerminate)
$$\frac{f = [\![e]\!]^{cn}_{a \circ l}}{ob(o, a, \{l|x = e.\textbf{get}\ \ \textbf{finished}\ \{s_1\}; s_2\}, q) \quad stream(f, \eta) \rightarrow ob(o, a, \{l|s_1; s_2\}, q) \quad stream(f, \eta)}$$

Figure 4.8: Operational semantics of destructive streams

The await statement in rule D-AwaitTrue is skipped as there exists a data value v in the buffer. By rule D-AwaitFalse, the process querying the empty (but not-yet-terminated) stream f will be suspended. To this aim, the statement **suspend** for unconditional suspension is added to the beginning of the sequence of the statements of the process. According to the standard ABS, the **suspend** then suspends the

active process, namely, it adds the process to q, where the active object is *idle* and ready to activate a suspended process from q. In rule D-AwaitTerminate, the *finished* block s_1 of the statement is selected for execution, since the streaming is terminated, i.e., the head of the buffer of the stream f is equal to the terminating token η.

The rule D-GetTrue assigns the value v from the head of the stream buffer to the variable x destructively, i.e., v is removed from the buffer. By D-GetTerminate, the **`finished`** block s_1 of the statement is executed followed by s_2, as the terminating token is observed at the head of the buffer. Note that the state of x remains the same. There is no rule for the **`get-finished`** statement when the buffer is empty which implies that the active process (and the object) is *blocked* until the buffer contains an element.

Semantics of Non-Destructive Streams

The operational semantics of ABS for those rules that involve non-destructive future-based streams is given in Fig. 4.9. In ND-AsyncCall, an asynchronous call to a streaming method m in o' is given with the actual parameters $\overline{v}$, that results in a reference to a non-destructive stream. The keyword **`nd`** denotes the non-destructive access to the resulting stream. Therefore, the return reference to the newly created stream with identifier f is a pair of f and a cursor which is initialized to 0, denoting the first position in the buffer which is initially $\perp$.

Note that by the before-mentioned Assign-Local and Assign-Field, an assignment $x = y$, where the variable y referring to a non-destructive stream f (i.e., $(f, n) = [\![y]\!]^{cn}_{aol}$), the variable x also refers to the same stream and the cursor of x is initialized to the same position in the buffer as the one of y (i.e., $(f, n) = [\![x]\!]^{cn}_{aol}$ as well).

The await statement in rule ND-AwaitTrue is skipped because there is a value v (which is not η) in the buffer of the stream f at the position determined by the cursor of x. By rule ND-AwaitFalse, the process querying the stream f will be suspended since the cursor of f denotes the empty position in the buffer (denoted by $\perp$). By the semantics, it is not difficult to see that this position will contain either a value v or the termination token η. By the rule ND-AwaitTerminate, the *finished* block s_1 is selected for execution, as the cursor of f points at a position which contains η.

The rule ND-GetTrue assigns to the variable x the value v (which is not η) in the stream buffer from the position determined by the cursor of y, and increments the cursor. By ND-GetTerminate, the **`finished`** block s_1 of the statement is selected for execution followed by s_2, as the cursor of variable y points at the terminating token η in the buffer. Note that the state of x and the cursor are not modified. There is no rule for the **`get-finished`** statement when the cursor denotes the empty position in the buffer (i.e., $\perp$) which implies that the active process (and the object) is *blocked*.

(ND-AsyncCall)

$$\frac{o' = [\![e]\!]_{a \circ l} \quad \overline{v} = [\![\overline{e}]\!]_{a \circ l} \quad \text{fresh}(f) \quad \text{streamer}(o'.m(\overline{v}))}{ob(o, a, \{l | x = \textbf{nd}\ \ e!m(\overline{e}); s\}, q) \rightarrow ob(o, a, \{l | x = (f, 0); s\}, q) \quad invoc(o', f, m, \overline{v}) \quad stream(f, \bot)}$$

(ND-AwaitTrue)

$$\frac{(f, n) = [\![x]\!]^{cn}_{a \circ l} \quad v = \text{elem}(u, n)}{ob(o, a, \{l | \textbf{await}\ \ x?\ \textbf{finished}\ \ \{s_1\}; s_2\}, q) \quad stream(f, u) \rightarrow ob(o, a, \{l | s_2\}, q) \quad stream(f, u)}$$

(ND-AwaitFalse)

$$\frac{(f, n) = [\![x]\!]^{cn}_{a \circ l} \quad \bot = \text{elem}(u, n)}{ob(o, a, \{l | \textbf{await}\ x?\ \ \textbf{finished}\ \ \{s_1\}; s_2\}, q) \quad stream(f, u) \rightarrow ob(o, a, \{l | \textbf{suspend}; \textbf{await}\ \ x?\ \textbf{finished}\ \ \{s_1\}; s_2\}, q) \quad stream(f, u)}$$

(ND-AwaitTerminate)

$$\frac{(f, n) = [\![x]\!]^{cn}_{a \circ l} \quad \eta = \text{elem}(u, n)}{ob(o, a, \{l | \textbf{await}\ x?\ \textbf{finished}\ \{s_1\}; s_2\}, q) \quad stream(f, u) \rightarrow ob(o, a, \{l | s_1; s_2\}, q) \quad stream(f, u)}$$

(ND-GetTrue)

$$\frac{(f, n) = [\![y]\!]^{cn}_{a \circ l} \quad v = \text{elem}(u, n)}{ob(o, a, \{l | x = y.\textbf{get finished}\ \{s_1\}; s_2\}, q) \quad stream(f, u) \rightarrow ob(o, a, \{l | x = v; y = (f, n + 1); s_2\}, q) \quad stream(f, u)}$$

(ND-GetTerminate)

$$\frac{(f, n) = [\![y]\!]^{cn}_{a \circ l} \quad \eta = \text{elem}(u, n)}{ob(o, a, \{l | x = y.\textbf{get finished}\ \ \{s_1\}; s_2\}, q) \quad stream(f, u) \rightarrow ob(o, a, \{l | s_1; s_2\}, q) \quad stream(f, u)}$$

Figure 4.9: Operational semantics of non-destructive streams

$$\frac{f = [\![e]\!]^{cn}_{a \circ l}}{ob(o,a,\{l|\texttt{await}\ \ e?\ \texttt{finished}\ \ \{s_1\};s_2\},q)\quad fut(f,v)\ \rightarrow\ ob(o,a,\{l|s_2\},q)\quad fut(f,v)}\ \text{(F-AwaitTrue)}$$

$$\frac{f = [\![e]\!]^{cn}_{a \circ l}}{ob(o,a,\{l|\texttt{await}\ \ e?\ \ \texttt{finished}\ \ \{s_1\};s_2\},q)\quad fut(f,\bot)\ \rightarrow\ ob(o,a,\{l|\texttt{suspend};\texttt{await}\ \ e?\ \texttt{finished}\ \ \{s_1\};s_2\},q)\quad fut(f,\bot)}\ \text{(F-AwaitFalse)}$$

$$\frac{f = [\![e]\!]^{cn}_{a \circ l}}{ob(o,a,\{l|x = e.\texttt{get}\ \ \texttt{finished}\ \ \{s_1\};s_2\},q)\quad fut(f,v)\ \rightarrow\ ob(o,a,\{l|x = v;s_2\},q)\quad fut(f,v)}\ \text{(F-GetTrue)}$$

Figure 4.10: Semantics of futures as streams

Semantics of Futures as Streams

The type of the value of any expression in ABS at runtime is a subtype of the static type of the expression. The **await** and **get** without **finished** clause can only be applied to futures and Boolean guards, and does exclude the streams. This is guaranteed because **Stream**<T> is not a subtype of **Fut**<T> (discussed in section 4.2.4). Recall that an **await** *without* **finished** clause on a stream is only a syntactic sugar for the one *with* the clause where the following block is empty. Different operational semantics of destructiveness and non-destructiveness does not affect the type system.

In order to support the subtyping relation between $\mathsf{stream}\langle T\rangle$ and $\mathsf{fut}\langle T\rangle$ in the operational semantics, as reflected in the type system, we need an extra set of semantic rules, where a future variable appears as the parameter of **await-finished** and **get-finished** statements. This set is presented in Figure 4.10. In these rules, only the cases are specified where the future contains a value v or not (empty stream). A resolved future is treated as an infinite stream of the same value v. Therefore, termination of future is not defined. The rule names are prefixed with F to denote that future appears as a stream.

We can prove on the basis of the operational semantics in a standard manner that all program executions are type-safe, and in our case this additionally ensures proper use of the data streams. This additionally amounts to ensuring that the **await-finished** and **get-finished** constructs are applied at runtime only to the data streams (and futures) and that the **yield** operation is only applied to the context of a streaming method.

4.2.6 Discussion on Buffer Size and Garbage Collection

The buffer of streams can grow indefinitely according to the above semantics. For practical purposes, however, we must take into consideration the finite nature of computer memory. This can be addressed by a different definition of the buffer which is bounded to a maximum size m. The semantics of a successful write operation to a bounded buffer requires a new premise where the buffer size is strictly less than m. If the buffer is full, on the other hand, different design decisions can be made. For instance writing to a full buffer can be blocking, i.e., the process is blocked until the buffer size is less than m, or it is non-blocking but signals the process about the failure. A successful destructive read operation, on the other side, decrements the buffer size allowing the buffer to shrink. However this is not the case for non-destructive streams, as the non-destructive read cannot change the buffer size since the data will possibly be read by other cursors. Hence, we need a garbage collection mechanism (GC) for non-destructive streams.

By definition, destructive streams do not cause any garbage. However, we can have a definition of garbage for non-destructive streams. In this section, garbage means a data element in the buffer of a non-destructive stream, which is read by *all* the *existing* cursors. In what follows, we define a GC that is executed periodically. It first obtains all the existing cursors in the system and then collects the garbages accordingly. Some of the cursors can be obtained from the immediate value of a variable, while other cursors can be wrapped with an outer future or stream in a nested way. For instance, if the future variable $x : \mathsf{fut}\langle\mathsf{stream}\langle T\rangle\rangle$ is resolved can possibly contain a cursor. To include these cursors in the GC, first we need some definitions. Let type T denote either a primitive type P (a type that is not a future nor a stream) or a non-primitive type N as follows:

$$\begin{aligned} T &::= P \mid N \\ N &::= \mathsf{stream}\langle T\rangle \end{aligned}$$

For notational convenience we rewrite the type $\mathsf{fut}\langle T\rangle$ to $\mathsf{stream}\langle T\rangle$. We also rewrite a run-time future object to a destructive stream with at most one element so that it can be typed as a stream. Therefore a future $fut(f, \bot)$ is rewritten to $stream(f, \bot)$ and $fut(f, v)$ to $stream(f, v\eta)$ in cn. The following algorithm obtains the set of all cursors in the system, based on which it marks the garbage:

1. for each object $(o, a, \{l_0|s_0\}, \{\{l_1|s_1\}, .., \{l_k|s_k\}\})$ in the system, the set of cursors that can be obtained in object o: $cursor_o = \{cursor_o([\![x]\!]^{cn}_{a \circ l} : T) \mid x \in a \cup \bigcup_{i=0}^{k} l_i\}$ where $cursor_o(v : T)$ returns all the existing cursors obtained from value v of type T in object o. The set of all the cursors existing in the system: $cursors = \bigcup cursor_o$. Note that for simplicity we assume there is no name conflict of variable names in the mappings.

2. for each stream identity f in the system, $min_f = min(\{n \mid (f, n) \in cursors\})$, where $min(S)$ returns the smallest number in a set S of numbers.

3. for each $stream(f, u)$ in the system, all the data elements in u with the index less than min_f are garbage and must be collected.

For simplicity we use $v : T$ for typing value v instead of using the formal run-time type system. Below we define $cursor_o^n(v : T)$ inductively, where n is the number of times the term stream appears in the type T of the value v, e.g., n is 0, 1, and 2 for the type P, $\mathsf{stream}\langle P\rangle$ and $\mathsf{stream}\langle\mathsf{stream}\langle P\rangle\rangle$, respectively. Note that n is finite, as the type of a variable is a string with a finite length.

Base case: a cursor can neither be obtained from a value with a primitive type (step 0), nor a value that refers to a destructive stream or a future of a primitive type P (step 1). Whereas one cursor can be obtained from a value that refers to a non-destructive stream of a primitive type P (step 1).

$$
\begin{aligned}
cursor_o^0(v : P) &= \emptyset \\
cursor_o^1(f : \mathsf{stream}\langle P\rangle) &= \emptyset && \text{where } stream(f, u) \in cn \\
cursor_o^1((f, n) : \mathsf{stream}\langle P\rangle) &= \{(f, n)\} && \text{where } stream(f, u) \in cn
\end{aligned}
$$

Inductive step: the induction hypothesis is that $cursor_o^n(v : N)$ returns all the cursors obtained from the value v of type N where $n = k$. Below we show how we obtain the cursors obtained from a value for $n = k + 1$ using the hypothesis:

$$
\begin{aligned}
cursor_o^{k+1}(f : \mathsf{stream}\langle N\rangle) &= \bigcup_{v_i \in s_n(u)} cursor_o^k(v_i : N) \\
&\quad \text{where } stream(f, u) \in cn \\
cursor_o^{k+1}((f, n) : \mathsf{stream}\langle N\rangle) &= \bigcup_{v_i \in s_n(u)} cursor_o^k(v_i : N) \cup \{(f, n)\} \\
&\quad \text{where } stream(f, u) \in cn
\end{aligned}
$$

where $s_n(u)$ denotes a set of elements in the buffer u of $stream(f, u)$ with index greater or equal to n, except special elements $\perp$ and η.

In order for the above algorithm to work, every data element in the buffer u must have an absolute index starting from zero for the first element added to the buffer. Recall that, by the semantics of ABS, every synchronous and asynchronous method call forms a (suspended or active) process in the called object which is denoted by $\{l_i|s_i\}$. Hence, the step 1 covers all the variable assignments in an object ($cursor_o$) and subsequently in the whole system ($cursors$). Also note that there is no need to distinguish the cursors by their variable names or their processes or objects. The

$$\frac{\Delta(f) = \mathsf{stream}\langle T\rangle \quad \forall i \in [1..n].(val_i \notin \{\bot, \eta\} \Rightarrow \Delta(val_i) = T)}{\Delta \vdash_R stream(f, (val_1, ..., val_n))\ \mathbf{ok}}\ (\text{T-Stream})$$

$$\frac{\Delta(val) = (\mathsf{stream}\langle T\rangle, \mathsf{Nat}) \quad \Delta \vdash_R v : \mathsf{stream}\langle T\rangle}{\Delta \vdash_R \mathsf{stream}\langle T\rangle\ v\ val\ \mathbf{ok}}\ (\text{T-StateStream})$$

$$\frac{\Delta(f) = \mathsf{stream}\langle T\rangle \quad \Delta(\overline{v}) = \overline{T} \quad \text{match}(m, \overline{T} \to \mathsf{stream}\langle T\rangle, \Delta(o))}{\Delta \vdash_R invoc(o, f, m, \overline{v})}\ (\text{T-InvocStream})$$

Figure 4.11: The typing rules of streams for run-time configurations

only relevant aspect of the cursor for GC is that there exists at least one cursor that points at a specific index of the buffer.

4.3 Subject Reduction for the Extended ABS

A *run* is a sequence of transitions from an *initial state* based on the rules of the operational semantics, where *initial state* consists of $ob(\text{start}, \epsilon, p, \emptyset)$, an initial object, start, with only one process p that corresponds to the main block of the program. The subject reduction for ABS is already proven in [48], namely, it is shown that a run from a well-typed initial configuration will maintain well-typed configurations, particularly, the assignments preserve well-typedness and method bindings do not give rise to the **`error`** process. In this section, we aim to extend the proof for the ABS subject reduction theorem to also include the notion of stream as specified in this chapter.

The typing context for the run-time configurations Δ extends the static typing context Γ with typing dynamically created values (entities created at run-time), namely, object and future identifiers. Let $\Delta \vdash_R cn\ \mathbf{ok}$ express that the configuration cn is well-typed in the typing context Δ. The typing rules for run-time configurations are defined for ABS and extensively discussed in [48]. The newly added rules for typing streams are shown in Figure 4.11. By T-Stream, the stream f is of type $\mathsf{stream}\langle T\rangle$ if the buffer only contains values of type T or the special tokens η and $\bot$. By T-StateStream, a variable v that refers to a stream val and provide non-destructive access to it is well-typed. Nat denotes the type of natural numbers. The type of val is a pair of the stream type and a Nat that holds the cursor to the stream. The rule T-InvocStream allows the return type of an asynchronous method invocation to be a stream as well.

In [48] (1) it is proven that the initial object corresponding to the main block of a well-typed program is well-typed and also (2) it is shown that the well-typedness of runtime configuration is preserved by reductions (Theorem 1). The proof for (1) also applies here. We only need to extend the proof for (2) with respect to the new transition rules introduced in section 4.2 as follows.

Theorem 1 (Subject Reduction)**.** *If* $\Delta \vdash_R cn$ **ok** *and* $cn \to cn'$*, then there is a* Δ' *such that* $\Delta \subseteq \Delta'$ *and* $\Delta' \vdash_R cn'$ **ok***.*

Proof. The proof is by induction over the defined transition rules in the operational semantics. We assume objects, futures, streams and messages not affected by a transition remain well-typed, and are ignored below. The auxiliary function $match(m, \overline{T} \to \mathsf{stream}\langle T\rangle, T')$ checks if a method m with $\overline{T} \to \mathsf{stream}\langle T\rangle$ is provided by the interface T'.

- *Process Suspension.* It is immediate that the rules D-AwaitTrue, ND-AwaitTrue,F-AwaitTrue, D-AwaitFalse, ND-AwaitFalse, F-AwaitFalse D-AwaitTerminate, ND-AwaitTerminate, D-GetTerminate and ND-GetTerminate preserve the well-typedness.

- Yield. By assumption, we have $\Delta \vdash_R ob(o, a, \{l|\textbf{yield}\ e; s\}, q)$ **ok**, $[\![e]\!]_{a \circ l} = v$ and $\Delta \vdash_R stream(f, u.\bot)$ **ok**. Obviously, $\Delta \vdash_R ob(o, a, \{l|s\}, q)$ **ok**. Since $l(\text{destiny}) = f$ and l is well-typed, we know that $\Delta(\text{destiny}) = \Delta(f)$. Let $\Delta(f) = \mathsf{stream}\langle T\rangle$. By T-Yield, $\Delta \vdash_R e : T$ and subsequently $\Delta(v) = T$, so $\Delta \vdash_R stream(f, u.v.\bot)$ **ok**.

- ReturnStream. By assumption, we have $\Delta \vdash_R ob(o, a, \{l|\textbf{return}; s\}, q)$ **ok**, and $\Delta \vdash_R stream(f, u.\bot)$ **ok**. Obviously, $\Delta \vdash_R ob(o, a, \{l|s\}, q)$ **ok** and $\Delta \vdash_R stream(f, u.\eta)$ **ok**.

- D-AsyncCall. Let $\Delta \vdash_R ob(o, a, \{l|x = e!m(\overline{e}); s\}, q)$ **ok**. We first consider the case $e \neq \textbf{this}$. By T-AsyncStream, we may assume that $\Delta \vdash e!m(\overline{e}) : \mathsf{stream}\langle T\rangle$ and by T-Assign that $\Delta(x) = \mathsf{stream}\langle T\rangle$. Therefore, $\Delta \vdash e : T'$ and $\Delta \vdash \overline{e} : \overline{T}$ such that $match(m, \overline{T} \to T\ \mathsf{stream}, T')$. Assume that $[\![e]\!]_{a \circ l} = o'$ and let $\Delta(o') = C$ for some class C. Based on [48], there is a Δ' such that $\Delta' \vdash_R [\![e]\!]_{a \circ l} : T'$ and $\Delta'(o') = C$, so $C \preceq T'$. By assumption class definitions are well-typed, so for any class C that implements interface T' we have $match(m, \overline{T} \to T\ \mathsf{stream}, C)$. Also $[\![\overline{e}]\!]_{a \circ l}$ similarly preserves the type of $\overline{e}$. Let $\Delta'' = \Delta'[f \mapsto \mathsf{stream}\langle T\rangle]$. Since $fresh(f)$ we know that $f \notin dom(\Delta')$, so if $\Delta' \vdash_R cn$ **ok**, then $\Delta'' \vdash_R cn$ **ok**. Since $\Delta' \vdash e!m(\overline{e}) = \Delta''(f)$, we get $\Delta'' \vdash_R ob(o, a, \{l|x = f; s\}, q)$ **ok**. Furthermore, $\Delta'' \vdash invoc(o', f, m, \overline{v})$ **ok** and $\Delta'' \vdash_R stream(f, \bot)$ **ok**. The case $e = \textbf{this}$ is similar, but uses the class of **this** directly for the match (so internal methods are also visible).

- ND-AsyncCall. Let $\Delta \vdash_R ob(o, a, \{l \mid \textbf{nd}\ x = e!m(\overline{e}); s\}, q)$ **ok**. The argument is similar to the above case, but we get $\Delta'' \vdash_R ob(o, a, \{l|x = (f, 0); s\}, q)$ **ok** as the consequence, in addition to $\Delta'' \vdash invoc(o', f, m, \overline{v})$ **ok** and $\Delta'' \vdash_R stream(f, \bot)$ **ok**.

- D-GetTrue. By assumption, $\Delta \vdash_R ob(o, a, \{l|x = e.\textbf{get finished}\{s_1\}; s_2\}, q)$ **ok**, $\Delta \vdash_R stream(f, v.u)$ **ok**, and $[\![e]\!]_{a \circ l} = f$. Let $\Delta(f) = \mathsf{stream}\langle T\rangle$.

Consequently, $\Delta \vdash_R e.\mathbf{get\ finished}\{s_1\} : T$ and $\Delta(v) = T$, so $\Delta \vdash x = v$, $\Delta \vdash_R ob(o, a, \{l|x = v; s\}, q)$ **ok** and $\Delta \vdash_R stream(f, u)$ **ok**. A similar argument applies for F-GetTrue where f is the identity of a future object $fut(f, v)$.

- ND-GetTrue. By assumption, $\Delta \vdash_R ob(o, a, \{l|x = y.\mathbf{get\ finished}\{s_1\}; s_2\}, q)$ **ok**, $\Delta \vdash_R stream(f, u)$ **ok**, $[\![y]\!]_{a \circ l} = (f, n)$ and $\text{elem}(u, n) = v$. Let $\Delta(f) = \mathsf{stream}\langle T\rangle$. Consequently, $\Delta \vdash_R y.\mathbf{get\ finished}\{s_1\} : T$ and $\Delta(v) = T$, so $\Delta \vdash x = v$, $\Delta \vdash_R ob(o, a, \{l|x = v; s\}, q)$ **ok**, and $\Delta \vdash y = (f, n + 1)$ **ok**.

□

4.4 Data Streams in Distributed Systems

In [10] a scalable distributed implementation of the ABS language is described. In this section we adapt our proposed notion of data streams in ABS to reduce the possible overhead of data steaming in a distributed setting.

To this aim, each streaming method is enabled to package the return values, that is, the method populates its return stream buffer possibly not once per value, but once per sequence of values. The package size can be specified explicitly as a parameter or can be selected based on the underlying deployment, e.g., it can be equal to the packet size of the TCP/IP technology involved. As such the number of packets to be transferred through the network is minimized.

There are two conditions when the package is streamed *before* its size is equal to the pre-specified package size: 1) when the streaming method terminates; 2) when the streaming method cooperatively releases control. The first condition is obvious, while the second prevents a specific kind of deadlock configuration. In general, ABS programs may give rise to deadlocks (see [39] for a discussion of deadlock analysis of ABS programs). However the notion packaging data streams should not give rise to additional deadlock possibilities.

The above second condition prevents the following kind of deadlock situation. Note that *package size* = n means that the number of yielded values needs to be equal to n, so that they are streamed as a package, except for the last package where the size may be less than n. Suppose there are two objects o_1 and o_2 in the run-time configuration where o_1 executes an active process which corresponds to method m_1 given by

$$m1()\{\texttt{r=o2!m2();}\ \mathbf{await}\ \texttt{r?;}\ \texttt{o2!satisfier();}\}$$

and the specification of the streaming method m_2 is an active process p in object o_2 given by

$$m2()\{\mathbf{yield}\ \texttt{x};\ \mathbf{await}\ \texttt{e};\ \mathbf{yield}\ \texttt{y};\}$$

Furthermore, suppose the method *satisfier* in o_2 changes the object state so that the expression e (which is `False` initially) evaluates to `True`. It is not difficult to see that for all $n \geqslant 2$, where n is the package size of the stream, the runtime configuration is deadlocked. The reason is that the first yielded value is not streamed before p releases control, as the package size is smaller than n. The deadlock possibility can be generalized to a category of programs where a streaming method releases control before it communicates the values which are yielded. The solution is that the package with the size smaller than n is streamed, before the process cooperatively releases control or blocks.

4.5 Implementation

In this section, we present a prototype implementation of future-based data streams as an API written in ABS. This API (see Figure 4.12) can be used to simulate the semantics of data streaming in ABS itself. The implementation details of the API can be found online[1].

As discussed in section 4.2, the `Stream<T>` datatype is parametrically polymorphic in its contained values of type T. The original ABS specification, however, offers besides parametric polymorphism also *subtype* polymorphism, through its interface types. In general, when defining and implementing languages with support for subtype-polymorphism, often the issue of *variance* arises: where in the code it is allowed (i.e. type-safe) to upcast to a supertype or downcast to a subtype. For example, given a subtype relation (`T` is subtype of `U`), a structure `S` is called covariant if `S<T>` is safe to "upcast" to `S<U>`; contraviarant if safe to "downcast" `S<U>` to `S<T>`; invariant if none of the above two hold, i.e. subtype polymorphism cannot be used for this structure, but other methods of polymorphism (e.g. parametric) perhaps can. In practice, the "rule of thumb" suggests that structures which are exclusively read-only (i.e. immutable) are allowed to be covariant, structures that are written-only (e.g. log files) contravariant, and structures that are read-write must be invariant.

The extension of ABS with stream that we describe in this chapter, strictly separates at the syntax level the role of the producer of values (write to the stream structure) with the role of the consumer (read from the stream). Since the producer can only append (produce) new values to the stream and not alter (mutate) past values, from the sole point of the consumer the stream structure seems as "immutable" (covariant). In this sense, a consumer holding a variable of type `Stream<T>` should be allowed to upcast it to type `Stream<U>`. Conversely, the producer is allowed to `yield` values of subtype `T`, if the method call's return type is typed as `Stream<U>`. As such, at the surface level (syntax and type system) it is acceptable for the `Stream` structure to be treated as covariant; however, at the implementation level it still re-

[1] `https://github.com/kazadbakht/ABS-Stream/blob/master/lib/Streams.abs`

mains a challenge on how to guarantee type safety at the host language (in our case, Haskell).

The Haskell language has parametric polymorphism but lacks built-in support for subtype polymorphism; for this reason, the ABS-Haskell backend compiler generates dynamic "upcasting" function calls where needed. However, this technique cannot be applied as well with Haskell's builtin *vector* datatype, which is a low-level built-in structure that cannot be made covariant or contravariant since it has been fixed-byte allocated in memory heap upon creation. For this reason, and also the fact that arrays are in general a mutable (read-write) data structure, the vectors in ABS (borrowed from Haskell) are treated as invariant. Since the implementation of streams in ABS relies currently on vectors, there is the practical limitation of having the `Buffer` type to be invariant. Similarly the `Stream` and `Fut` datatypes are treated as invariant, because the ABS-Haskell backend treats each future `Fut<T>` as a pointer to a vector size-1 stored in the heap that holds the value of `T`. Based on this practical limitation, the `Stream<T>` datatype introduced in this extension to ABS is treated as subtype-invariant, with support for parametric polymorphism.

The API is semantically compliant with the semantics of data streams defined in this chapter: The method that yields to a stream is separated from the access mode of readers to the stream (i.e., either destructive or non-destructive). Every reader has access to a stream via an instance of either `Dref` or `NDref` for destructive or non-destructive access mode, respectively. Furthermore a stream variable (that refers to an instance of `Dref` or `NDref`) is only typed by the `Stream` interface, abstracting from the underlying access mode.

The interface `Buffer<T>` is implemented by the class `CBuffer`. The FIFO buffer (an instance of `CBuffer`) is implemented by a vector whose elements are of type `Maybe<T>`, namely, each of which contains either a value (`Just(v)` where `v` is of type `T`) or `Nothing`. A position in the vector can have three different states: It contains `Just(v)` (a value `v` that can be read), `Nil` (the position is empty and will be filled), and `Nothing` (a token of type `Maybe<T>`) that denotes termination of the stream. The interface `Buffer<T>` provides the methods `yield()` and `terminate()` to the streaming method in order to write to the buffer and to explicitly terminate the stream of data values, respectively. The termination enqueues `Nothing` to the buffer and is meant to be the last statement in the definition of the streaming method (to simulate the terminating **`return`**). A stream maintains a global index `wrt` to the buffer which denotes the position where the next yielded value is written. It is incremented by every time calling `yield`. In destructive read, the `CBuffer` maintains a global index (i.e., `rd`) to the buffer for all the readers of the stream, whereas in non-destructive read, every reader (i.e., `NDref` instance) maintains a local index (i.e., `cursor`) to the buffer.

The reader can read from a stream by asynchronously calling `pull()` on the `Stream` object that returns a *future* representing the next data value, whether resolved or not. The operation `pull` is overridden in `Dref` and `NDref` for de-

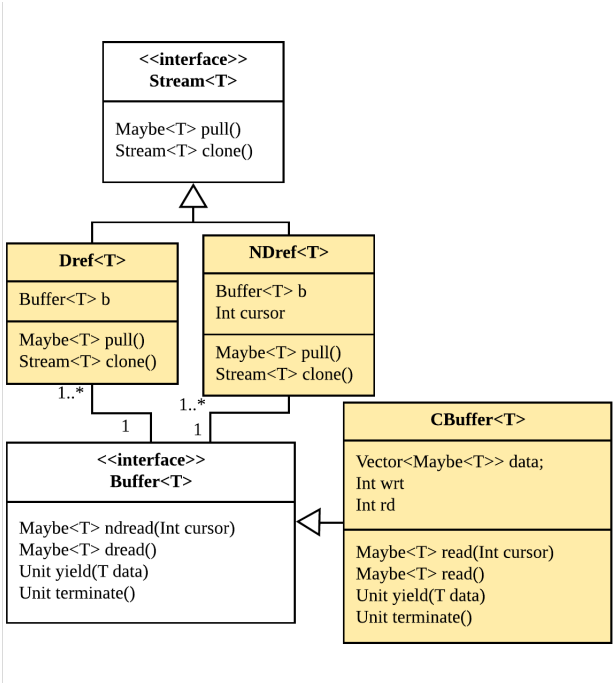


Figure 4.12: Class diagram of ABS Stream library

```
(1) Maybe<T> dread() {
(2) Int temp = rd;
(3) rd = rd + 1;
(4) await (buffer[temp] != Nil);
(5) return buffer[temp]; // is not null
(6) }

(7) Maybe<T> ndread(cursor) {
(8) await (buffer[cursor] != Nil);
(9) return buffer[cursor]; // is not null
(10)}
```

Figure 4.13: Destructive and non-destructive read in `CBuffer`

structive and non-destructive read from the buffer, respectively. The former calls `dread()` method of the `Buffer` which returns the first valid element in the vector, indicated by the index `rd` in buffer, and increments `rd`. Whereas the latter calls `ndread(cursor)` of the `Buffer` where the cursor is a field of the `NDref`, which returns the element indicated by the index `cursor` in the vector. The implementation of `dread()` and `ndread(cursor)` is given in Figure 4.13 where **`await`** at lines (4) and (8) cooperatively release control until the condition (indicating whether the buffer element has been produced) holds.

Also the method `clone` is used to copy a non-destructive stream object, a new instance of `NDref` which has a reference to the same stream but a new cursor which is initialized with the value of the cursor of the original object. For destructive streams, the method only returns the reference to **`this`** which is of type `Dref`.

Awaiting the future resulting from calling `pull()` queries the availability of next data value. Therefore, statement **`await`** `r?` **`finished`** {`S`} is expressed in the library as follows:

```
(1) f = r!pull(); // a future f to the target data value
(2) await f?; // awaits if the future f is not resolved yet
(3) m = f.get; // gets the resolved data value
(4) if (m == Nothing) // Nothing is the special token denoting the termination
(5) {S}
```

where `r` is a reference to a stream object and `S` is a block of statements. This can either give rise to the release of control in case the data is not available (line 2) or to skip otherwise. The variable `m` is of type `Maybe<T>` which contains either the value `v`, denoted by `Just(v)`, where `v` is of type `T`, or `Nothing`.

Similarly, statement `x = r.get finished` {`S`} can be expressed using the library as follows:

```
(1) f = r!pull(); // a future to the target data value
(2) m = f.get; // gets the resolved data value
(3) case (m) {
(4) Nothing => {S} // "Nothing" is the special token denoting the termination
(5) Just(v) => {x = v} // the value v is assigned to x
(6) }
```

In line 2, the object running this process blocks until the the data value is written to the future `f`.

The keyword **`nd`** is implemented in the API by a Boolean argument passed to the called streaming method. The argument specifies whether the return object of the streaming method to be an object of class `Dref` or class `NDref`.

The following snippet shows how the library is used to stream *integer* data values. The streaming method `m` instantiates a stream, delegates yielding values to the stream asynchronously to an auxiliary method `m2`, and returns the stream to the caller. Note that `m` sends the same list of parameters it receives to `m2`.

```
// caller :
// False means the return stream is non-destructive
Fut<Stream<Int>> f = o!m(False, ...);
Stream<Int> r = f.get; // r is a reference to the stream
// reading from the non-destructive stream r
...

// callee
Stream<Int> m(Bool isDestr, ...){
      Buffer<Int> b = new Cbuffer();
      // b is the return stream which is filled by m2
      this!m2(b, ...);
      // isDestr determines the access mode to the stream
      // i.e., destructively or non-destructively
      if (isDestr)
            return new Dref<Int>(b);
      else
            return new NDref<Int>(b, 0);
}

// the implementation of the callee
Unit m2(Buffer<Int> b, ...) {
```

```
      Int x ;
      ...
      b!yield(x); // yield a value
      ...
      b!terminate(); // termination token
}
```

Remark. In the above-mentioned API, having multiple readers for one stream may result in a performance bottleneck, as the buffer object itself queries the availability of data item to be returned for every `pull` request (via `dread` or `ndread`). Alternatively, such availability check can be delegated to the reader's `pull` method itself. However in the current design, doing the checks of line (4) and (8) in Figure 4.13 in their corresponding `pull` definitions give rise to busy-wait polling. The key feature that enables delegating the check without busy-wait polling is the data type `Promise<T>`. A promise is of this type is either contains a data value of type `T` (resolved) or not. An unresolved promise p can be resolved by *p.give(v)* with some data value v. Similar to futures, *get* and *await* operations can also be applied to promises.

By converting the type of the buffer vector from `Maybe<T>` to `Promise<Maybe<T>>`, the methods `dread` and `ndread` can immediately return to the reader's `pull` request the promise object in the vector that holds the expected value, which is either already resolved and can be retrieved or will be resolved in the future. In this new design, the `ndread` in `CBuffer` only returns the promise without availability check as follows:

```
(1) Promise<Maybe<T>> ndread(cursor) {
(2) return buffer[cursor];
(3) }
```

And `pull` method in `NDref` which checks the availability is given as follows:

```
(1) Maybe<T> pull() {
(2) Fut<Promise<Maybe<T>>> f = b!ndread(cursor); // cursor is a field in NDref
(3) Promise<Maybe<T>> p = f.get;
(4) await p?; // availability check is moved to pull
(5) Maybe<T> result = p.get;
(6) case (result) {Just(v) => cursor = cursor + 1;}
(7) return result;
(8) }
```

In line (6), if `result` is equal to `Nothing` then `cursor` is not incremented, such that the next `pull` requests to this object result in `Nothing`. Similar changes apply to `pull` in `Dref` and `dread()` in `CBuffer` for destructive read, except the index `rd` which is updated in `dread`.

An implementation similar to the example in Figure 4.2 using the above API is provided online[2]. We also ran the implementation on a PC which was an Intel Core i7-5600U 2.60GHz × 4 with 12GB RAM, and 64-bit Ubuntu 16.04 LTS as the

[2] `https://github.com/kazadbakht/ABS-Stream/blob/master/Examples/map\_reduce.abs`

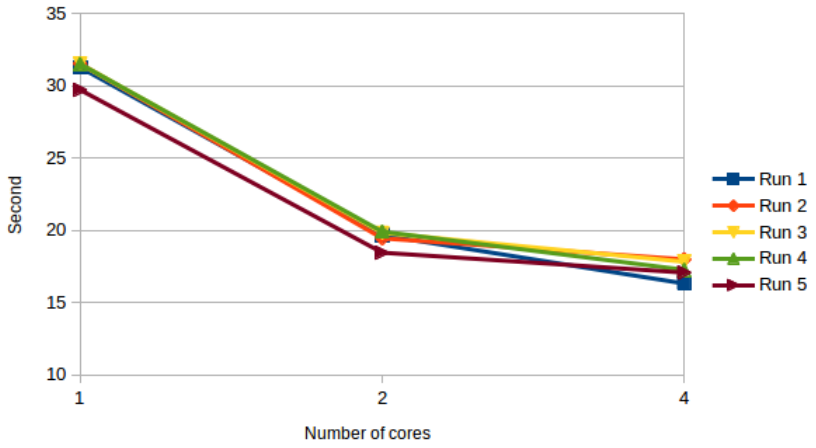


Figure 4.14: Execution time of parallel map-reduce program for 10^6 data values

operating system. In Figure 4.14, we represent the time measured for execution of the program for different number of parallel processors in 5 runs. On average for this specific implementation, we observed that for two cores we achieve 1.59× speedup compared to one core, and for four cores 1.13× speedup compared to two cores.

Lower speedup achieved for higher number of processors, among other reasons, stems from the fact that a stream is a shared resource where parallel access and yielding values to it in a safe manner is limited. This is confirmed experimentally, by adding more workload on the reader per data value that it reads from the stream. As such, the access rate to the stream becomes low enough such that the stream is not a performance bottleneck. With this modification, we could achieve up to 1.41 speedup for four cores compared to two cores.

The current implementation does not feature garbage collection of streams: produced data values are stored in a vector which dynamically (at-runtime) grows indefinitely (until memory exhaustion). This choice was made for a separation of concerns. This orthogonal issue of garbage collection can be trivially solved in the case of destructive streams: all produced values before the global index can be considered as garbage. A future implementation should automatically reclaim the space for such values and appropriately resize (shrink) the vector. In the case, however, of non-destructive streams, some extra bookkeeping and communication is involved to have safe, distributed garbage collection of streams. One possible implementation would require storing at the producer's side a global (system-wide) minimum of all the readers' local cursors. Besides this bookkeeping of the producer, once a reader forwards a `Stream<T>` to another ABS process (local or remote), it involves notifying the producer about the local minimum of the new reader process. Furthermore, in case of a real distributed system, the producer should monitor the quality of the network connection to every reader, otherwise it runs the risk of memory leaking from a dropped connection to a reader.

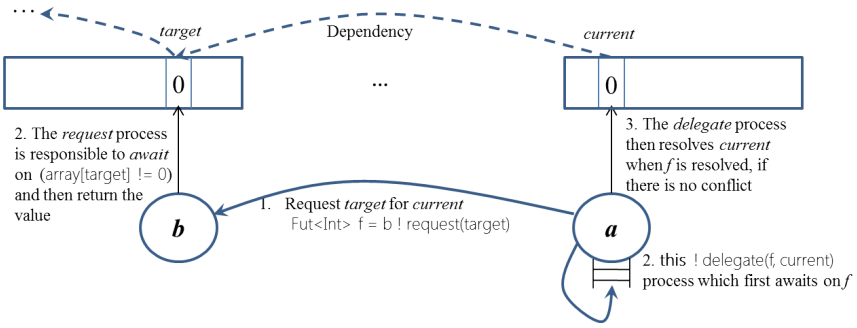


Figure 4.15: Resolving dependencies in Distributed PA using cooperative scheduling

4.6 Case Study

In this section, we use data streams in the ABS model of the distributed PA proposed in chapter 3. Figure 4.15 illustrates the high-level scheme for the unresolved dependencies proposed in chapter 3, using the notion of an active object, future, and cooperative scheduling in ABS.

In above scheme, the current, unresolved slot which belongs to the active object a requires the value of the unresolved target slot which belongs to the the active object b. To this aim, the object a asynchronously calls the *request* method of the object b, and delegates the resulting future as a suspended process in its queue, so that the active process continues with the rest of its partition. On the other side, the request awaits on a Boolean condition which checks if the target is resolved and returns the value. Finally, the *delegate* method which awaited on the future, gets the future value and processes it.

4.6.1 Incorporating Data Streams

The generation of distributed PA-based graphs as described above is fairly high-level and intuitive at the modeling level. However, the number of messages and return values communicated among the active objects poses a considerable overhead. Packaging the requests and the corresponding return values can considerably improve the performance of the run-time system.

In the distributed scheme in Figure 4.15, the request is sent per each required target slot, which is too fine-grained. Instead, we propose a modification of the algorithm so that the requests for the target values located on the same active object are sent together as a package of requests via one message, and the returning values are received via a stream with packaging capability. An experimental validation of a scalable distributed implementation of our model that utilizes streams is presented in [11] which is based on Haskell. It shows a significant performance improvement for the model compared the one presented in [10] (chapter 3).
Resolving the dependencies in the modified approach is shown in Figure 4.16. For

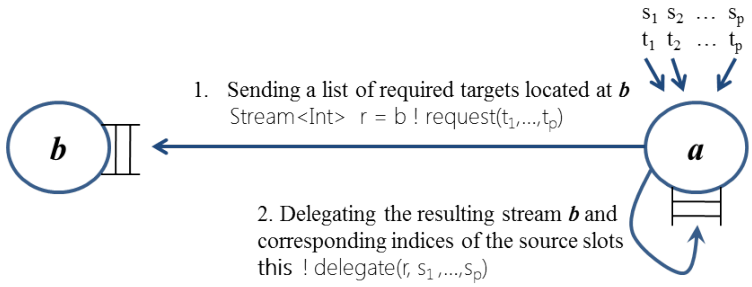


Figure 4.16: The modified approach using destructive streams

all $i \in [1, p]$, each pair (s_i, t_i) represents a request from object a to object b, where s_i represents the index of an unresolved slot belonging to the partition hosted by a, and t_i represents its corresponding slot belonging to the partition hosted by b. The value obtained from each t_i is used to resolve the unresolved slot s_i. Assuming p is the package size, the list $[t_1, .., t_p]$ is sent to b as a package of requests, and the requests process returns corresponding values per each t_i via a stream r (e.g., ***yield*** *array*$[t_i]$). Figure 4.17 illustrates abstract ABS code for *requests* which streams the values, and for *delegates* which receives them.

4.7 Related Work

There already exists a variety of different programming constructs for streaming data in different programming languages like Java, and software frameworks for processing big data like Apache Hadoop and Spark.

Asynchronous generators specified in [66] enable the streaming of data in an asynchronous method invocation. This includes, on the callee side, yielding the data, and on the caller side receiving them as an asynchronous iterator or raising an exception if there is no further yielded data. These generators are defined in the context of the multi-threaded model of concurrency, where asynchrony is provided by spawning a thread for a method call.

Akka Streams [70] provides an API to specify a streaming setup between actors which allows to adapt behavior to the underlying resources in terms of both memory and speed.

There are also languages which utilize the notion of *channel* as a means of communication, inspired by the model of *Communicating Sequential Processes* (CSP). For instance, Go language and *JCSP* [76], which is a library in Java, provide CSP-like elements, e.g., processes (referred to as Goroutines in Go) that communicate via channels by means of read and write primitives. Buffered channels in Go provide asynchronous read (cf. write) when the buffer is not empty (cf. not full). Otherwise the primitives are blocking.

Similarly to asynchronous generators, streaming data as proposed in this chapter

```
1: Each actor o executes the following in parallel
2: Unit run(...)
3: while the whole partition is not yet processed do
4:     /*
         Resolve the slots. Next pack of unresolved sources = [s_1, .., s_p] from
         the partition belonging to this object, and its corresponding targets =
         [t_1, .., t_p] whose owning partition hosted by some object w are calculated
5:     */
6:     Stream<Int> f = w ! requests(targets);
                                        ▷ The stream f is destructive by default
7:     this ! delegates(f, sources);
8:
9: Int stream requests(List<Int> targets)
10: while targets is not nil do
11:     Int tar = head(targets);
12:     targets = tail(targets);
13:     await (arr[tar] ≠ 0);
                                        ▷ At this point the target is resolved
14:     yield arr[tar];
15:
16: Unit delegate(Stream<Int> r, Int sources)
17: while True do
18:     Int val;
19:     await r? finished {break;}
                                        ▷ Quit the while if r is terminated
20:     val = r.get;
21:     Int src = head(sources);
22:     sources = tail(sources);
23:     // Use val to resolve arr[src]
```

Figure 4.17: The sketch of the data streaming in the modified approach

is fully integrated with asynchronous method invocation, i.e., it is not a separate orthogonal concept like channels are. But its integration with the ABS language allows for an additional loose coupling between the producer and consumer of data streams: by means of cooperative scheduling of tasks the consumption of data can be interleaved with other tasks on demand.

The distributed shared memory (DSM) paradigm [34, 57, 72] enables access to a common shared space across disjoint address spaces, where communication and synchronization between processes are enforced through operations on shared data. The notion of *tuple space* was originally integrated at the language level in Linda [37]. The processes communicate via insertion/read/removal of tuples into/from the tuple: `out()` to write a tuple into a tuple space, `in()` to retrieve (and remove),

and `read()` to read (without removing) a tuple from it.

Similarly to tuple spaces, the interaction model of streams proposed in this chapter provides time and space decoupling, namely, data producers and consumers can remain anonymous with respect to each other, and the sender of a data needs no knowledge about the future use of that data or its destination (the reference to a stream can be passed around). Also producer-side synchronization decoupling is guaranteed, whereas, the consumer-side decoupling is not provided in tuple-spaces, as the consumers synchronously pull the data. In ABS streams, however, the decoupling is provided at the consumer object level, thanks to the notion of cooperative scheduling. Similarly to tuple space *in*-based and *read*-based communication, the destructive and non-destructive data streams, respectively, can be naturally used to implement *one-of-n* semantics (only one consumer reads a given data), and *one-to-n* message delivery (a given data can be read by all such consumers).

4.8 Future work

We focused on extending the main asynchronous core of ABS with data streams. Other main features of the ABS like *concurrent object groups* (cogs) and deployment components are orthogonal and compatible with this extension. As an example, ABS features *cog* that, in principle, shares a thread of control among its constituent objects, which enables internal synchronous calls. By the nature of data streaming, it is natural to restrict the streaming to the asynchronous method calls.

Another interesting line of work consists of investigating a more efficient GC for non-destructive streams. The approach proposed in this chapter involves a periodic execution of GC that requires gathering information from all actors in the system synchronously which can in practice give rise to a bottleneck. Alternatively, a more efficient GC can be investigated where each stream maintains a table counting the number of cursors to each data element in the buffer.

The ABS with Haskell backend supports real-time programming techniques which allows for specifying deadlines with method invocations. This provides an interesting basis to extend ABS with real-time data streaming which may, as an example, involve timeout on read operations. We also need to extend the various formal analysis techniques (e.g., deadlock detection, general functional analysis based on method contracts) currently supported by the ABS to the ABS model of streaming data discussed in this chapter.

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

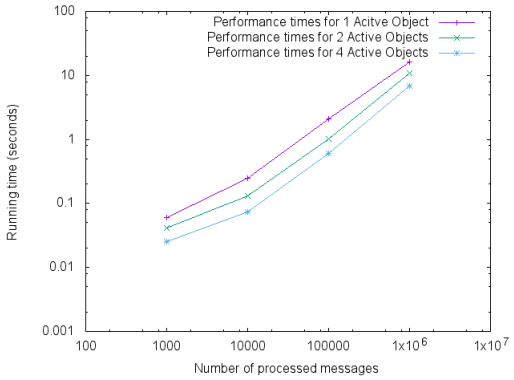


Figure 5.5: Performance Times for processing 100-1M messages

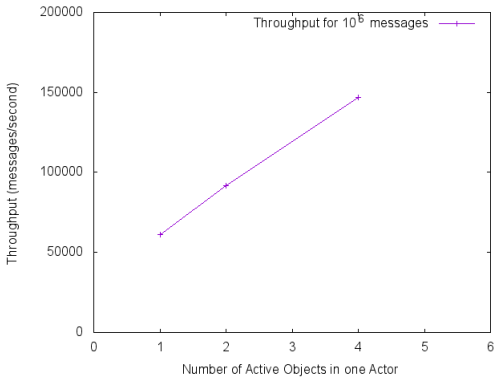


Figure 5.6: Throughput for processing 1M messages

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
    }
}
```

Part IV

Deadlock Analysis

This part consists of the following chapter:

Chapter 6 In this chapter, we introduce an approach for detecting deadlocks in an actor-based program in ABS. The underlying language features active objects, that communicate asynchronously, and cooperative scheduling of the tasks belonging to an object. To this aim, we model the system as a well-structured transition system based on predicate abstraction and prove the decidability of the deadlock detection.

Chapter 6

Deadlock Detection for Actor-Based Coroutines

6.1 Introduction

Actors [2][47] provide an event-driven concurrency model for the analysis and construction of distributed, large-scale parallel systems. In actor-based modeling languages, like Rebeca [69], Creol [51], and ABS [48], the events are generated by *asynchronous calls* to methods provided by the actors. The resulting integration with object-orientation allows for new object-oriented models of concurrency, better suited for the analysis and construction of distributed systems than the standard model of *multi-threading* in languages like Java.

The new object-oriented models of concurrency arise from the combination of different synchronization mechanisms. By design, the basic *run-to-completion* mode of execution of asynchronously called methods as for example provided by the language Rebeca does not provide any synchronization between actors. Consequently, the resulting concurrent systems of actors do not give rise to undesirable consequences of synchronization like *deadlock*. The languages Creol and ABS extend the basic model with synchronization on the values returned by a method. So-called *futures* [27] provide a general mechanism for actors to synchronize on return values. Creol and ABS further integrate a model of execution of methods based on and inspired by *coroutines*, attributed by D. Knuth to M. Conway [24]. This model allows for controlled suspension and resumption of the executing method invocation and so-called cooperative scheduling of another method invocation of the actor.

Both the synchronization mechanisms of futures and coroutines may give rise to deadlock. Futures may give rise to *global* deadlock in a system of actors. Such a global deadlock consists of a circular dependency between different method invocations of possibly different actors which are suspended on the generation of the return value. On the other hand, coroutines may give rise to a *local* deadlock which occurs when all method invocations of a single actor are suspended on a Boolean

condition. In this chapter we provide the formal foundations of a novel method for the analysis of such local deadlocks.

To the best of our knowledge, our work provides a first method for deciding local deadlocks in actor-based languages with coroutines. The method itself is based on a new technique for predicate abstraction of actor-based programs with coroutines, which aims at the construction of a well-structured transition system. In contrast, the usual techniques of predicate abstraction [17] aim at the construction of a *finite* abstraction, which allows model checking of properties in temporal logic. In [29], a restricted class of actor-based programs is modeled as a well-structured transition system. This class does not support coroutines and actors do not have a global state specifying the values of the global variables.

Methods that utilize different techniques aiming at detection of global deadlocks in various actor settings include the following. The work in [53] uses ownership to organize CoJava active objects into hierarchies in order to prevent circular relationships where two or more active objects wait indefinitely for one another. Also data-races and data-based deadlocks are avoided in CoJava by the type system that prevents threads from sharing mutable data. In [26], a sound technique is proposed that translates a system of asynchronously communicating active objects into a Petri net and applies Petri net reachability analysis for deadlock detection. The work that is introduced in [38] and extended in [45] defines a technique for analyzing deadlocks of stateful active objects that is based on behavioural type systems. The context is the actor model with wait-by-necessity synchronizations where futures are not given an explicit "Future" type. Also, a framework is proposed in [52] to statically verify communication correctness in a concurrency model using futures, with the aim that the type system ensures that interactions among objects are deadlock-free.

A deadlock detection framework for ABS is proposed in [39] which mainly focuses on deadlocks regarding future variables, i.e., await and get operations on futures. It also proposes a naive annotation-based approach for detection of local deadlocks (await on Boolean guards), namely, letting programmers annotate the statement with the dependencies it creates. However, a comprehensive approach to investigate local deadlocks is not addressed. Our approach, and corresponding structure of the chapter, consists of the following. First, we introduce the basic programming concepts of asynchronous method calls, futures and coroutines in Section 6.2. In Section 6.3 we introduce a new operational semantics for the description of the local behavior of a single actor. The only external dependencies stem from method calls generated by other actors and the basic operations on futures corresponding to calls of methods of other actors. Both kinds of external dependencies are modeled by *non-determinism*. Method calls generated by other actors are modeled by the non-deterministic scheduling of method invocations. The basic operations on futures are modeled by the corresponding non-deterministic evaluation of the availability of the return value and random generation of the return value itself. Next, we introduce in Section 6.4 a *predicate abstraction* [17, 40] of the value assignments to

the global variables ("fields") of an actor as well as the local variables of the method invocations. The resulting abstraction still gives rise to an *infinite* transition system because of the generation of *self*-calls, that is, calls of methods of the actor by the actor itself, and the corresponding generation of "fresh" names of the local variables.

Our main contribution consists of the following technical results.

- a proof of the *correctness* of the predicate abstraction, in Section 6.5, and
- *decidability* of checking for the occurrence of a local deadlock in the abstract transition system in Section 6.7.

Correctness of the predicate abstraction is established by a *simulation* relation between the concrete and the abstract transition system. Decidability is established by showing that the abstract system is a so-called well-structured transition system, cf. [36]. Since the concrete operational semantics of the local behavior of a single actor is an *over-approximation* of the local behavior in the context of an arbitrary system of actors, these technical results together comprise a general method for proving *absence* of local deadlock of an actor. A short discussion follow-up in Section 6.8 concludes the chapter.

6.2 The Programming Language

In this section we present, in the context of a class-based language (with a subset of ABS features), the basic statements which describe asynchronous method invocation and cooperative scheduling.

A class introduces its global variables, also referred to as "fields", and methods. We use $x, y, z, \ldots$ to denote both the fields of a class and the local variables of the methods (including the formal parameters). Method bodies are defined as sequential control structures, including the usual conditional and iteration constructs, over the basic statements listed below.

Dynamic instantiation For x a so-called future variable or a class variable of type C, for some class name C, the assignment

$$x = \texttt{new}$$

creates a new future or a unique reference to a new instance of class C.

Side effect-free assignment In the assignment

$$x = e$$

the expression e denotes a side effect-free expression. The evaluation of such an expression does not affect the values of any global or local variable and also does

not affect the status of the executing process. We do not detail the syntactical structure of side effect-free expressions.

Asynchronous method invocation A method is called asynchronously by an assignment of the form

$$x = e_0!m(e_1, \ldots, e_n)$$

Here, x is a future variable which is used as a unique reference to the return value of the invocation of method m with actual parameters $e_1, \ldots, e_n$. The called actor is denoted by the expression e_0. Without loss of generality we restrict the actual parameters and the expression e_0 to side effect-free expressions. Since e_0 denotes an actor, this implies that e_0 is a global or local variable.

The get operation The execution of an assignment

$$x = y.\texttt{get}$$

blocks till the future variable y holds the value that is returned by its corresponding method invocation.

Awaiting a future The statement

$$\texttt{await}\ x?$$

releases control and schedules another process in case the future variable x does not yet hold a value, that is to be returned by its corresponding method invocation. Otherwise, it proceeds with the execution of the remaining statements of the executing method invocation.

Awaiting a Boolean condition Similarly, the statement

$$\texttt{await}\ e$$

where e denotes a side effect-free Boolean condition, releases control and schedules another process in case the Boolean condition is false. Otherwise, it proceeds with the execution of the remaining statements of the executing method invocation.

We describe the possible deadlock behavior of a system of dynamically generated actors in terms of *processes*, where a process is a method invocation. A process is either *active* (executing), *blocked* on a get operation, or *suspended* by a future or Boolean condition. At run-time, an actor consists of an active process and a set of suspended processes (when the active method invocation blocks on a get operation it blocks the entire actor). Actors execute their active processes in parallel and only interact via asynchronous method calls and futures. When an active process

awaits a future or Boolean condition, the actor can cooperatively schedule another process instead. A *global* deadlock involves a circular dependency between processes which are awaiting a future. On the other hand, a *local* deadlock appears when all the processes of an actor are awaiting a Boolean condition to become true. In the following sections we present a method for showing if an initial set of processes of an individual actor does *not* give rise to a local deadlock.

6.3 The Concrete System

In order to formally define local deadlock we introduce a formal operational semantics of a single actor. Throughout this chapter we assume a definition of a class C to be given. A typical element of its set of methods is denoted by m. We assume the definition of a class C to consist of the usual declarations of global variables and method definitions. Let $Var(C)$ denote all the global and local variables declared in C. Without loss of generality we assume that there are no name clashes between the global and local variables appearing in C, and no name clashes between the local variables of different methods. To resolve in the semantics name clashes of the local variables of the different invocations of a method, we assume a given infinite set Var such that $Var(C) \subseteq Var$. The set $Var \setminus Var(C)$ is used to generate "fresh" local variables. Further, for each method m, we introduce an infinite set $\Sigma(m)$ of renamings σ such that for every local variable x of m, $\sigma(x)$ is a fresh variable in Var, i.e. not appearing in $Var(C)$. We assume that any two distinct $\sigma, \sigma' \in \bigcup_m \Sigma(m)$ are *disjoint* (Here m ranges over the method names introduced by class C.) Renamings σ and σ' are disjoint if their ranges are disjoint. Note that by the above assumption the domains of renamings of *different* methods are also disjoint. By auxiliary function $fresh(\sigma')$ we check that the renaming $\sigma' \in \Sigma(m)$ is different from all the existing renamings in Q.

A process p arising from an invocation of a method m is described formally as a pair (σ, S), where $\sigma \in \Sigma(m)$ and S is the sequence of remaining statements to be executed, also known as *continuation*. An actor configuration then is a triple (Γ, p, Q), where Γ is an assignment of values to the variables in Var, p denotes the active process, and Q denotes a set of suspended processes. A configuration is *consistent* if for every renaming σ there exists at most one statement S such that $(\sigma, S) \in \{p\} \cup Q$.

A computation step of a single actor is formalized by a transition relation between consistent actor configurations. A structural operational semantics for the derivation of such transitions is given in Table 6.1. Here, we assume a given set Val of values of built-in data types (like Integer and Boolean), and an infinite set R of references or "pointers". Further, we assume a global variable *refs* such that $\Gamma(refs) \subseteq R$ records locally stored references.

We proceed with the explanation of the rules of Table 6.1. The rule <ASSIGN>

$$\frac{}{(\Gamma, (\sigma, x = e; S), Q) \to (\Gamma[x\sigma = \Gamma(e\sigma)], (\sigma, S), Q)}\ \texttt{<ASSIGN>}$$

$$\frac{r \in R \backslash \Gamma(\mathit{refs})}{(\Gamma, (\sigma, x = \texttt{new}; S), Q) \to (\Gamma[\mathit{refs} = \Gamma[\mathit{refs}] \cup \{r\}], (\sigma, x = r; S), Q)}\ \texttt{<NEW>}$$

$$\frac{v \in \mathit{Val}}{(\Gamma, (\sigma, x = \ y.\texttt{get}; S), Q) \to (\Gamma[x\sigma = v], (\sigma, S), Q)}\ \texttt{<GET-VALUE>}$$

$$\frac{r \in R}{(\Gamma, (\sigma, x = \ y.\texttt{get}; S), Q) \to (\Gamma[\mathit{refs} = \Gamma(\mathit{refs}) \cup \{r\}], (\sigma, x = r; S), Q)}\ \texttt{<GET-REF>}$$

$$\frac{\Gamma(y\sigma) \neq \Gamma(\texttt{this})}{(\Gamma, (\sigma, x = y!m(\bar{e}); S), Q) \to (\Gamma, (\sigma, x = \texttt{new}; S), Q)}\ \texttt{<REMOTE-CALL >}$$

$$\frac{\Gamma(y\sigma) = \Gamma(\texttt{this}) \quad \mathit{fresh}(\sigma')}{(\Gamma, (\sigma, x = y!m(\bar{e}); S), Q) \to (\Gamma[\bar{z}\sigma' = \Gamma(\bar{e}\sigma)], (\sigma, x = \texttt{new}; S), Q \cup \{(\sigma', S')\})}\ \texttt{<LOCAL-CALL>}$$

$$\frac{\Gamma(e\sigma) = \mathit{true}}{(\Gamma, (\sigma, \texttt{if}\ e\ \{S'\}\ \texttt{else}\ \{S''\}; S), Q) \to (\Gamma, (\sigma, S'; S), Q)}\ \texttt{<IF-THEN>}$$

$$\frac{\Gamma(e\sigma) = \mathit{false}}{(\Gamma, (\sigma, \texttt{if}\ e\ \{S'\}\ \texttt{else}\ \{S''\}; S), Q) \to (\Gamma, (\sigma, S''; S), Q)}\ \texttt{<IF-ELSE>}$$

$$\frac{\Gamma(e\sigma) = \mathit{true}}{(\Gamma, (\sigma, \texttt{while}\ e\ \{S'\}; S), Q) \to (\Gamma, (\sigma, S'; \texttt{while}\ e\ \texttt{do}\ \{S'\}; S), Q)}\ \texttt{<WHILE-TRUE>}$$

$$\frac{\Gamma(e\sigma) = \mathit{false}}{(\Gamma, (\sigma, \texttt{while}\ e\ \{S'\}; S), Q) \to (\Gamma, (\sigma, S), Q)}\ \texttt{<WHILE-FALSE>}$$

$$\frac{\Gamma(e\sigma) = \mathit{true}}{(\Gamma, (\sigma, \texttt{await}\ e; S), Q) \to (\Gamma, (\sigma, S), Q)}\ \texttt{<AWAITB-TRUE>}$$

$$\frac{\Gamma(e\sigma) = \mathit{false} \qquad (\sigma', S') \in Q}{(\Gamma, (\sigma, \texttt{await}\ e; S), Q) \to (\Gamma, (\sigma', S'), (Q \cup \{(\sigma, \texttt{await}\ e; S)\}) \backslash \{(\sigma', S')\})}\ \texttt{<AWAITB-FALSE>}$$

$$\frac{}{(\Gamma, (\sigma, \texttt{await}\ x?; S), Q) \to (\Gamma, (\sigma, S), Q)}\ \texttt{<AWAITF-SKIP>}$$

$$\frac{(\sigma', S') \in Q}{(\Gamma, (\sigma, \texttt{await}\ x?; S), Q) \to (\Gamma, (\sigma', S'), (Q \cup \{(\sigma, \texttt{await}\ \mathit{true}; S)\}) \backslash \{(\sigma', S')\})}\ \texttt{<AWAITF-SCHED>}$$

$$\frac{(\sigma', S') \in Q}{(\Gamma, (\sigma, \texttt{return}\ e), Q) \to (\Gamma, (\sigma', S'), Q \backslash \{(\sigma', S')\})}\ \texttt{<RETURN>}$$

Figure 6.1: Concrete transition relation

describes a side effect-free assignment. Here, and in the sequel, $e\sigma$ denotes the result of replacing any local variable x in e by $\sigma(x)$. By $\Gamma(e)$ we denote the extension of the variable assignment Γ to the evaluation of the expression e. By $\Gamma[x = v]$, for some value v, we denote the result of updating the value of x in Γ by v.

The rule <NEW> describes the non-deterministic selection of a fresh reference not appearing in the set $\Gamma(\mathit{refs})$. The rule <GET-VALUE> models an assignment involving a get operation on a future variable y which holds a value of some built-in data type by an assignment of a random value $v \in \mathit{Val}$ (of the appropriate type). The rule <GET-REF> models an assignment involving a get operation on a future variable y which holds a reference by first adding a random value $r \in R$ to the set $\Gamma(\mathit{refs})$ and then assign it to the variable x (note that we do *not* exclude that $r \in \Gamma(\mathit{refs})$).

It should be observed that we model the local behavior of an actor. The absence of information about the return values in the semantics of a get operation is accounted for by a non-deterministic selection of an arbitrary return value. Further, since we restrict to the analysis of local deadlocks, we also abstract from the possibility that the get operation blocks and assume that the return value is generated.

The rules regarding choice and iteration statements are standard. The rule <REMOTE-CALL> describes an assignment involving an external call ($\Gamma(y\sigma) \neq \Gamma(\mathbf{this})$, where $y\sigma$ denotes y, if y is a global variable, otherwise it denotes the variable $\sigma(y)$). It is modeled by the creation and storage of a new future reference uniquely identifying the method invocation. On the other hand, according to the rule <LOCAL-CALL> a local call ($\Gamma(y\sigma) = \Gamma(\mathbf{this})$) generates a new process and future corresponding to the method invocation. Also it is checked that the renaming σ' is fresh. Further, by $\Gamma[\bar{z}\sigma' = \Gamma(\bar{e}\sigma)]$ we denote the simultaneous update of Γ which assigns to each local variable $\sigma'(z_i)$ (i.e., the renamed formal parameter z_i) the value of the corresponding actual parameter e_i with its local variables renamed by σ, i.e., the local context of the calling method invocation. For technical convenience we omitted the initialization of the local variables that are not formal parameters. The body of method m is denoted by S'.

The rule <AWAITB-TRUE> describes that when the Boolean condition of the await statement is true, the active process proceeds with the continuation, and <AWAITB-FALSE> describes that when the Boolean condition of the await statement is false, a process is selected for execution. This can give rise to the activation of a disabled process, which is clearly not optimal. The transition system can be extended to only allow the activation of enabled processes. However, this does not affect the results of this chapter and therefore is omitted for notational convenience.

The rule <AWAITF-SKIP> formalizes the assumption that the return value referred to by x has been generated. On the other hand, <AWAITF-SCHED> formalizes the assumption that the return value has not (yet) been generated. Note that we transform the initial await statement into an await on the Boolean condition "true". Availability of the return value then is modeled by selecting the process for

execution. Finally, in the rule RETURN we assume that the return statement is the last statement to be executed. Note that here we do not store the generated return value (see also the discussion in section 6.8).

In view of the above, we have the following definition of a local deadlock.

Definition 6.3.1. *A local configuration* (Γ, p, Q) deadlocks *if*

> *for all* $(\sigma, S) \in \{p\} \cup Q$ *we have that the initial statement of* S *is an await statement* `await` e *such that* $\Gamma(e\sigma) = \mathit{false}$.

In the sequel we describe a method for establishing that an initial configuration does not give rise to a local deadlock configuration. Here it is worthwhile to observe that the above description of the local behavior of a single actor provides an over-approximation of its actual local behavior as part of *any* system of actors. Consequently, absence of a local deadlock of this over-approximation implies absence of a local deadlock in *any* system of actors.

6.4 The Abstract System

Our method of deadlock detection is based on predicate abstraction. This boils down to using predicates instead of concrete value assignments. For the class C, the set $\mathit{Pred}(m)$ includes all (the negations of) the Boolean conditions appearing in the body of m. Further, $\mathit{Pred}(m)$ includes all (negations of) equations $x = y$ between reference variables x and y, where both x and y are global variables of the class C (including `this`) or local variables of m (a reference variable is either a future variable or used to refer to an actor.) In addition to these conditions, the set $\mathit{Pred}(m)$ can also include user-defined predicates that possibly increases the precision of the analysis.

An abstract configuration α is of the form (T, p, Q), where, as in the previous section, p is the active process and Q is a set of suspended processes. The set T provides for each invocation of a method m a logical description of the relation between its local variables and the global variables. Formally, T is a set of pairs (σ, u), where $u \subseteq \mathit{Pred}(m)$, for some method m, is a set of predicates of m with fresh local variables as specified by σ. We assume that for each process $(\sigma, S) \in \{p\} \cup Q$ there exists a corresponding pair $(\sigma, u) \in T$. If for some $(\sigma, u) \in T$ there does not exist a corresponding process $(\sigma, S) \in \{p\} \cup Q$ then the process has terminated. Further, we assume that for any σ there is at most one $(\sigma, u) \in T$ and at most one $(\sigma, S) \in \{p\} \cup Q$.

We next define a transition relation on abstract configurations in terms of a strongest postcondition calculus. To describe this calculus, we first introduce the following notation. Let $L(T)$ denote the set $\{\, u\sigma \mid (\sigma, u) \in T \,\}$, where $u\sigma = \{\, \varphi\sigma \mid \varphi \in u \,\}$, and $\varphi\sigma$ denotes the result of replacing every local variable x in φ with $\sigma(x)$.

Logically, we view each element of $L(T)$ as a conjunction of its predicates. Therefore, when we write $L(T) \vdash \varphi$, i.e., φ is a logical consequence (in first-order logic) of $L(T)$, the sets of predicates in $L(T)$ are interpreted as conjunctions. (It is worthwhile to note that in practice the notion of logical consequence will also involve the first-order theories of the underlying data structures.) The strongest postcondition, defined below, describes for each basic assignment a and local context $\sigma \in \Sigma(m)$, the set $sp_\sigma(L(T), a)$ of predicates $\varphi \in \mathit{Pred}(m)$ such that $\varphi\sigma$ holds after the assignment, assuming that all predicates in $L(T)$ hold initially.

For an assignment $x = e$ we define the strongest postcondition by

$$sp_\sigma(L(T), x = e) = \{\, \varphi \mid L(T) \vdash \varphi\sigma[e/x],\, \varphi \in \mathit{Pred}(m)\}$$

where $[e/x]$ denotes the substitution which replaces occurrences of the variable x by the side effect-free expression e. For an assignment $x = \texttt{new}$ we define the strongest postcondition by

$$sp_\sigma(L(T), x = \texttt{new}) = \{\, \varphi \mid L(T) \vdash \varphi\sigma[\texttt{new}/x],\, \varphi \in \mathit{Pred}(m)\,\}$$

The substitution $[\texttt{new}/x]$ replaces every equation $x = y$, with y distinct from x, by *false*, $x = x$ by *true*. It is worthwhile to note that for every future variable and variable denoting an actor, these are the only possible logical contexts consistent with the programming language. (Since the language does not support de-referencing, actors encapsulate their local state.)

For an assignment $x = y.\texttt{get}$ we define the strongest postcondition by

$$sp_\sigma(L(T), x = y.\texttt{get}) = \{\, \varphi \mid L(T) \vdash \forall x.\varphi\sigma,\, \varphi \in \mathit{Pred}(m)\,\}$$

The universal quantification of the variable x models a non-deterministic choice for the value of x.

Table 6.2 presents the structural operational semantics of the transition relation for abstract configurations. In the <ASSIGN> rule the set of predicates u for each $(\sigma', u) \in T$, is updated by the strongest postcondition $sp_{\sigma'}(L(T), (x = e)\sigma)$. Note that by the substitution theorem of predicate logic, we have for each predicate φ of this strongest postcondition that $\varphi\sigma'$ will hold after the assignment $(x = e)\sigma$ (i.e., $x\sigma = e\sigma$) because $L(T) \vdash \varphi\sigma[e/x]$. Similarly, the rules <GET> and <NEW> update T of the initial configuration by their corresponding strongest postcondition as defined above.

In the rule <REMOTE-CALL> we identify a remote call by checking whether the information $\texttt{this} \neq y\sigma$ can be added consistently to $L(T)$. By $T \cup \{(\sigma, \varphi)\}$ we denote the set $\{\, (\sigma', u) \in T \mid \sigma' \neq \sigma \,\} \cup \{\, (\sigma, u \cup \{\varphi\}) \mid (\sigma, u) \in T \,\}$. In the rule <LOCAL-CALL> the set of predicates u of the generated invocation of method m consists of all those predicates $\varphi \in \mathit{Pred}(m)$ such that $L(T) \vdash \varphi[\bar{e}\sigma/\bar{z}]$, where $\bar{z}$ denotes the formal parameters of m. By the substitution theorem of predicate

<ASSIGN>
$$\frac{T' = \{\, (\sigma', sp_{\sigma'}(L(T), (x = e)\sigma)) \mid (\sigma', u) \in T \,\}}{(T, (\sigma, x = e; S), Q) \to (T', (\sigma, S), Q)}$$

<GET>
$$\frac{T' = \{\, (\sigma', sp_{\sigma'}(L(T), (x = y.\texttt{get})\sigma)) \mid (\sigma', u) \in T \,\}}{(T, (\sigma, x = y.\texttt{get}; S), Q) \to (T', (\sigma, S), Q)}$$

<NEW>
$$\frac{T' = \{\, (\sigma', sp_{\sigma'}(L(T), (x = \texttt{new})\sigma)) \mid (\sigma', u) \in T \,\}}{(T, (\sigma, x = \texttt{new}; S), Q) \to (T', (\sigma, S), Q)}$$

<REMOTE-CALL>
$$\frac{L(T) \cup \{\texttt{this} \neq y\sigma\} \not\vdash \mathit{false}}{(T, (\sigma, x = y!m(\bar{e}); S), Q) \to (T \cup \{(\sigma, \texttt{this} \neq y)\}\}, (\sigma, x = \texttt{new}; S), Q)}$$

<LOCAL-CALL>
$$\frac{\begin{array}{c} L(T) \cup \{\texttt{this} = y\sigma\} \not\vdash \mathit{false} \\ u = \{\, \varphi \mid L(T) \vdash \varphi[\bar{e}\sigma/\bar{z}],\ \varphi \in \mathit{Pred}(m) \,\} \quad \mathit{fresh}(\sigma') \end{array}}{\begin{array}{l} (T, (\sigma, x = y!m(\bar{e}); S), Q) \to \\ (T \cup \{(\sigma', u)\} \cup \{(\sigma, \texttt{this} = y)\}, (\sigma, x = \texttt{new}; S), Q \cup \{(\sigma', S')\}) \end{array}}$$

<IF-THEN>
$$\frac{L(T) \cup \{e\sigma\} \not\vdash \mathit{false}}{\begin{array}{c} (T, (\sigma, \texttt{if}\ e\ \{S'\}\ \texttt{else}\ \{S''\}; S), Q) \\ \to (T \cup \{(\sigma, e)\}, (\sigma, S'; S), Q) \end{array}}$$

<IF-ELSE>
$$\frac{L(T) \cup \{\neg e\sigma\} \not\vdash \mathit{false}}{\begin{array}{c} (T, (\sigma, \texttt{if}\ e\ \{S'\}\ \texttt{else}\ \{S''\}; S), Q) \\ \to (T \cup \{(\sigma, \neg e)\}, (\sigma, S''; S), Q) \end{array}}$$

<WHILE-TRUE>
$$\frac{L(T) \cup \{e\sigma\} \not\vdash \mathit{false}}{\begin{array}{c} (T, (\sigma, \texttt{while}\ e\ \texttt{do}\ \{S\}'; S), Q) \\ \to (T \cup \{(\sigma, e)\}, (\sigma, S'; \texttt{while}\ e\ \texttt{do}\ \{S'\}; S), Q) \end{array}}$$

<WHILE-FALSE>
$$\frac{L(T) \cup \{\neg e\sigma\} \not\vdash \mathit{false}}{\begin{array}{c} (T, (\sigma, \texttt{while}\ e\ \texttt{do}\ \{S'\}; S), Q) \\ \to (T \cup \{(\sigma, \neg e)\}, (\sigma, S), Q) \end{array}}$$

<AWAIT-TRUE>
$$\frac{L(T) \cup \{e\sigma\} \not\vdash \mathit{false}}{(T, (\sigma, \texttt{await}\ e; S), Q) \to (T \cup \{(\sigma, e)\}, (\sigma, S), Q)}$$

<AWAIT-FALSE>
$$\frac{L(T) \cup \{\neg e\sigma\} \not\vdash \mathit{false} \qquad (\sigma', S') \in Q}{(T, (\sigma, \texttt{await}\ e; S), Q) \to (T \cup \{(\sigma, \neg e)\}, (\sigma', S'), (Q \cup \{(\sigma, \texttt{await}\ e; S)\})\backslash\{(\sigma', S')\})}$$

<AWAITF-SKIP>
$$(T, (\sigma, \texttt{await}\ x?; S), Q) \to (T, (\sigma, S), Q)$$

<AWAITF-SCHED>
$$\frac{(\sigma', S') \in Q}{\begin{array}{l} (T, (\sigma, \texttt{await}\ x?; S), Q) \to \\ (T, (\sigma', S'), (Q \cup \{(\sigma, \texttt{await}\ \mathit{true}; S)\})\backslash\{(\sigma', S')\}) \end{array}}$$

<RETURN>
$$\frac{(\sigma', S') \in Q}{(T, (\sigma, \texttt{return}\ e), Q) \to (T, (\sigma', S'), Q\backslash\{(\sigma', S')\})}$$

Figure 6.2: Abstract transition system

logic, the (simultaneous) substitution $[\bar{e}\sigma/\bar{z}]$ ensures that φ holds for the generated invocation of method m. Note that by definition, $L(T)$ only refers to fresh local variables, i.e., the local variables of m do not appear in $L(T)$ because for any $(\sigma, u) \in T$ we have that $\sigma(x)$ is a fresh variable not appearing in the given class C. For technical convenience we omitted the substitution of the local variables that are not formal parameters. The renaming σ', which is assumed not to appear in T, introduces fresh local variable names for the generated method invocation. The continuation S' of the new process is the body of method m. The generation of a new future in both the rules <REMOTE-CALL> and <LOCAL-CALL> is simply modeled by the $x = \texttt{new}$ statement.

By <IF-THEN>, the active process transforms to the "then" block, i.e. S', followed by S, if the predicate set $L(T)$ is consistent with the guard e of the if-statement. (Note that as $L(T)$ is in general not complete, it can be consistent with e as well as with $\neg e$.) The other rules regarding choice and iteration statements are defined similarly. By <RETURN> the active process terminates, and is removed from the configuration. A process is selected from Q for execution. Note that the pair $(\sigma, u) \in T$ is not affected by this removal.

The rules <AWAIT-TRUE> and <AWAIT-FALSE> specify transitions assuming the predicate set $L(T)$ is consistent with the guard e and with $\neg e$, respectively. In the former case, the `await` statement is skipped and the active process continues, whereas in the latter, the active process releases control and a process from Q is activated. Similar to the concrete semantics in the previous section, in <AWAITF-SKIP> and <AWAITF-SCHED>, the active process non-deterministically continues or cooperatively releases the control. In the latter, a process from Q is activated.

We conclude this section with the counterpart of Definition 6.3.1 for the abstract setting.

Definition 6.4.1. *A local configuration (T, p, Q) is a (local)* deadlock *if*

> *for all $(\sigma, S) \in \{p\} \cup Q$ we have that the initial statement of S is an await statement* `await` e *such that $L(T) \cup \{\neg e\sigma\} \not\vdash$ false.*

6.5 Correctness of Predicate Abstraction

In this section we prove that the concrete system is simulated by the abstract system. To this end we introduce a simulation relation $\sim$ between concrete and abstract configurations:

$$(\Gamma, p, Q) \sim (T, p, Q), \text{ if } \Gamma \models L(T)$$

where $\Gamma \models L(T)$ denotes that Γ satisfies the formulas of $L(T)$.

Theorem 3. *The abstract system is a simulation of the concrete system.*

Proof. Given $(\Gamma, p, Q) \sim (T, p, Q)$ and a transition $(\Gamma, p, Q) \to (\Gamma', p', Q')$, we need to prove that there exists a transition $(T, p, Q) \to (T', p', Q')$ such that $(\Gamma', p', Q') \sim (T', p', Q')$.

For all the rules that involve the evaluation of a guard e, it suffices to observe that $\Gamma \models L(T)$ and $\Gamma \models e$ implies $L(T) \cup \{e\} \not\vdash \mathit{false}$.

We treat the case $x = e$ where e is a side effect-free expression (the others cases are treated similarly). If $p = (\sigma, x = e; S)$, where e is a side effect-free expression, then $\Gamma' = \Gamma[(x = e)\sigma]$. We put $T' = \{\, (\sigma', sp_{\sigma'}(L(T), (x = e)\sigma)) \mid (\sigma', u) \in T \,\}$. Then it follows that $(T, p, Q) \to (T', p', Q')$. To prove $\Gamma' \models L(T')$ it remains to show for $(\sigma, u) \in T$ and $\varphi \in sp_{\sigma'}(L(T), (x = e)\sigma)$ that $\Gamma' \models \varphi\sigma'$: Let $(\sigma', u) \in T$ and $\varphi \in sp_{\sigma'}(L(T), (x = e)\sigma)$. By definition of the strongest postcondition, we have $L(T) \vdash \varphi\sigma'[(x = e)\sigma]$. Since $\Gamma \models L(T)$, we have $\Gamma \models \varphi\sigma'[(x = e)\sigma]$. Since $\Gamma' = \Gamma[x\sigma = \Gamma(e\sigma)]$, we obtain from the substitution theorem of predicate logic that

$$\Gamma' \models \varphi\sigma' \iff \Gamma \models \varphi\sigma'[(x = e)\sigma]$$

and hence we are done. □

We conclude this section with the following observation: if the initial abstract configuration (T, p, Q) does not give rise to a local deadlock then also the configuration (Γ, p, Q) does not give rise to a local deadlock, when $\Gamma \models L(T)$. To see this, by the above theorem it suffices to note that if (Γ', p', Q') is a local deadlock and $\Gamma' \models L(T')$ then (T', p', Q') is also a local deadlock because for any $(\sigma, \texttt{await}\ e; S) \in \{p'\} \cup Q'$ we have that $\Gamma' \not\models e\sigma$ implies $L(T') \cup \{\neg e\sigma\} \not\vdash \mathit{false}$.

6.6 Example

We represent the proposed method by means of an example. Given partial definition of the class C as follows:

```
class C {
  Int a = 0;
  void m() {
    a= a + 1; await a < 5;
  }
  ...
}
```

We want to check the program for the absence of the local deadlock, where Q contains only one process $(\sigma, \texttt{a=a+1;await a<5})$, which is an invocation of the method m. The $\mathit{Pred}(m) = \{a < 5, \neg(a < 5), a = 3, \neg(a = 3)\}$ is the set of predicates of the method m, and a user-defined predicate $a = 3$ and its negation. We try to check the system for the absence of deadlock for the initial $u = \{a < 5\}$,

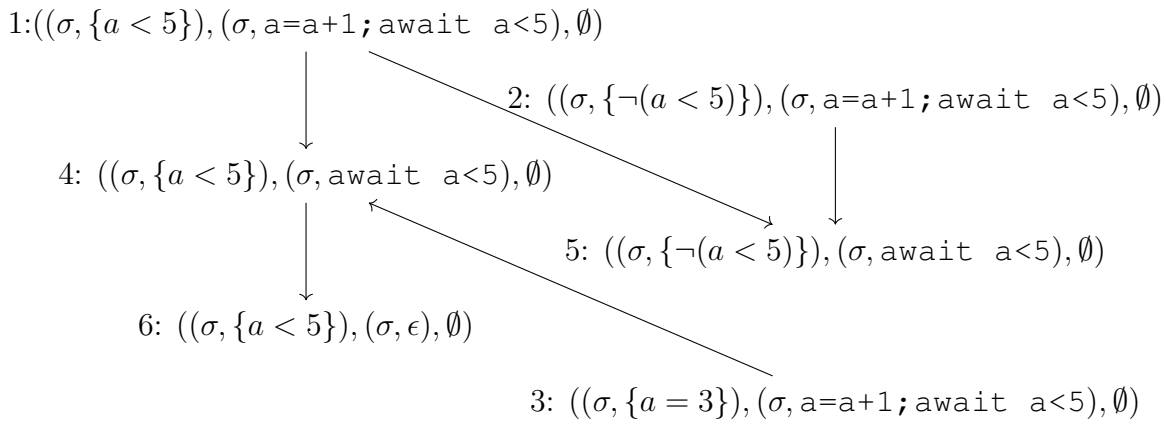


Figure 6.3: Example of the abstract system

$u = \{\neg(a < 5)\}$ and $u = \{a = 3\}$. The abstract systems for three different initial states form a finite transition system which is shown in Figure 6.3.

The states 1,2 and 3 denote the three initial states. State 5 denotes a local deadlock configuration. State 6 denotes a normal termination configuration. The resulting transition system shows that the execution of method m, where initially $a \geqslant 4$, possibly causes deadlock.

6.7 Decidability of Deadlock Detection

The abstract local behavior of a single actor, as defined in the previous section, gives rise, for a given initial configuration, to an infinite transition system because of dynamic generation of local calls and the corresponding introduction of fresh local variables. In this section we show how we can model an abstract system for which the transition relation is computable as well-structured transition system and obtain the decidability of deadlock detection for such abstract systems. To this end, we first provide a canonical representation of an abstract configuration which abstracts from renamings of the local variables by means of *multisets* of *closures*. A closure of a method m is a pair (u, S), where S is a *continuation* of the body of m and $u \subseteq \mathit{Pred}(m)$. (Here $\mathit{Pred}(m)$ denotes the set of predicates associated with m as defined in Section 6.3). The set of continuations of a statement S is the smallest set $\mathit{Cont}(S)$ such that $S \in \mathit{Cont}(S)$ and $\epsilon \in \mathit{Cont}(S)$, where the "empty" statement ϵ denotes termination, and which is closed under the following conditions

- $S'; S'' \in \mathit{Cont}(S)$ implies $S'' \in \mathit{Cont}(S)$
- `if` e $\{S_1\}$ `else` $\{S_2\}$; $S' \in \mathit{Cont}(S)$ implies $S_1; S' \in \mathit{Cont}(S)$ and $S_2; S' \in \mathit{Cont}(S)$
- `while` e $\{S'\}$; $S'' \in \mathit{Cont}(S)$ implies S'; `while` e $\{S'\}$; $S'' \in \mathit{Cont}(S)$.

Note that for a given method the set of all possible closures is finite. We formally represent a multiset of closures as a function which assigns a natural number $f(c)$ to each closure c which indicates the number of occurrences of c. For notational convenience we write $c \in f$ in case $f(c) > 0$.

In preparation of the notion of canonical representation of abstract configurations, we introduce for every abstract configuration $\alpha = (T, p, Q)$ the set $\bar{\alpha}$ of triples (σ, u, S) for which $(\sigma, u) \in T$ and either $(\sigma, S) \in \{p\} \cup Q$ or $S = \epsilon$.

Definition 6.7.1. *An abstract configuration (T, p, Q) is canonically represented by a multiset of closures f, if for every method m and closure (u, S) of m we have*

$$f((u, S)) = |\{\, \sigma \mid (\sigma, u, S) \in \bar{\alpha} \,\}|$$

(where $|V|$ denotes the cardinality of the set V).

Note that each abstract configuration has a unique multiset representation. For any multiset f of closures, let $T(f)$ denote the set of predicates $\{\exists v \mid (v, S)^n \in f\}$, where $\exists v$ denotes v with all the local variables appearing in the conjunction of the predicates of v existentially quantified.

The following lemma states the equivalence of a set of closures and its canonical representation.

Lemma 6.7.1. *Let the abstract configuration (T, p, Q) be canonically represented by the multiset of closures f. Further, let $(\sigma, u) \in T$, where $\sigma \in \Sigma(m)$, and $\varphi \in Pred(m)$. It holds that*

$$L(T) \vdash \varphi\sigma \text{ iff } \{u\} \cup T(f) \vdash \varphi$$

Proof. Proof-theoretically we reason, in first-order logic, as follows. For notational convenience we view a set of predicates as the conjunction over its elements. By the Deduction Theorem we have

$$L(T) \vdash \varphi\sigma \text{ iff } \vdash L(T) \to \varphi\sigma$$

From the laws of universal quantification we obtain

$$\vdash L(T) \to \varphi\sigma \text{ iff } \vdash \forall X(L(T) \to \varphi\sigma)$$

and

$$\vdash \forall X(L(T) \to \varphi\sigma) \text{ iff } \vdash \exists X L(T) \to \varphi\sigma$$

where X denotes the set of local variables appearing in $L(T) \setminus \{u\sigma\}$. Note that no local variable of X appears in $\varphi\sigma$ or $u\sigma$.

Since any two distinct $v, v' \in L(T)$ have no local variables in common, we can

push the quantification of $\exists X L(T)$ inside. That is,

$$\vdash \exists X L(T) \to \varphi\sigma \text{ iff } \vdash \{\, \exists X v \mid v \in L(T) \,\} \to \varphi\sigma$$

No local variable of X appears in $u\sigma$, therefore we have

$$\vdash \{\, \exists X v \mid v \in L(T) \,\} \to \varphi\sigma \text{ iff } \vdash u\sigma \wedge \{\, \exists X v \mid v \in L(T) \,\} \to \varphi\sigma$$

Again by the Deduction Theorem we then have

$$\vdash u\sigma \wedge \{\, \exists X v \mid v \in L(T) \,\} \to \varphi\sigma \text{ iff } \{u\sigma\} \vdash \{\, \exists X v \mid v \in L(T) \,\} \to \varphi\sigma$$

Clearly $u\sigma \vdash \exists u$ and $\exists X v$ is logically equivalent to $\exists v$, for any $v \in L(T) \setminus \{u\sigma\}$. So, we have

$$\{u\sigma\} \vdash \{\, \exists X v \mid v \in L(T) \,\} \to \varphi\sigma \text{ iff } \{u\sigma\} \vdash \{\, \exists v \mid v \in L(T) \,\} \to \varphi\sigma$$

Since f represents (T, p, Q) we have that $T(f) = \{\, \exists v \mid v \in L(T) \,\}$. Renaming the local variables of $u\sigma$ and $\varphi\sigma$ then finally gives us

$$\{u\sigma\} \vdash \{\, \exists v \mid v \in L(T) \,\} \to \varphi\sigma \text{ iff } \{u\} \vdash T(f) \to \varphi$$

which proves the lemma. □

We next define an ordering on multisets of closures.

Definition 6.7.2. *By $f \preccurlyeq f'$ we denote that $f(c) \leqslant f'(c)$ and $f'(c) = 0$ if $f(c) = 0$.*

In other words, $f \preccurlyeq f'$ if all occurrences of f belong to f' and f' does not add occurrences of closures which do not already occur in f. The following result states that this relation is a well-quasi-ordering.

Lemma 6.7.2. *The relation $f \preccurlyeq f'$ is a quasi-ordering such that for any infinite sequence $(f_n)_n$ there exist indices $i < j$ such that $f_i \preccurlyeq f_j$.*

Proof. First observe that for a given class there is only a finite number of closures. We show that the proof for the standard subset relation for multisets also holds for this variation. Assume that for some set X of closures we have constructed an infinite subsequence $(f'_n)_n$ of $(f_n)_n$ such that $f'_i(c) \leqslant f'_j(c)$, for every $c \in X$ and $i < j$. Suppose that for every $c \notin X$ the set $\{\, k \mid f'_j(c) = k,\, j \in \mathbb{N} \,\}$ is bounded. It follows that there exists an f'_k which appears infinitely often in $(f'_n)_n$, since there exists only a finite number of combinations of occurrences of closures in $\bar{X} = \{\, c \mid c \notin X \,\}$. On the other hand, if there exists a $d \notin X$ such that set $\{\, k \mid f'_j(d) = k,\, j \in \mathbb{N} \,\}$ has no upperbound then we can obtain a subsequence $(f''_n)_n$ of $(f'_n)_n$ such that $f''_i(c) \leqslant f''_j(c)$ for every $c \in X \cup \{d\}$ and $i < j$. Thus, both cases lead to the existence of indices $i < j$ such that $f_i \preccurlyeq f_j$. □

From the above lemma it follows immediately that the following induced ordering on abstract configurations is also a well-quasi-ordering.

Definition 6.7.3. *We put* $(T, (\sigma, S), Q) \preccurlyeq (T', (\sigma', S), Q')$ *iff* $f \preccurlyeq f'$*, for multi-sets of closures* f *and* f' *(uniquely) representing* $(T, (\sigma, S), Q)$ *and* $(T', (\sigma', S), Q')$*, respectively.*

We can now formulate and prove the following theorem which states that this well-quasi-ordering is preserved by the transition relation of the abstract system.

Theorem 4. *For abstract configurations* α*,* α'*, and* β*, if* $\alpha \to \alpha'$ *and* $\alpha \preccurlyeq \beta$ *then* $\beta \to \beta'$*, for some abstract configuration* β' *such that* $\alpha' \preccurlyeq \beta'$*.*

Proof. The proof proceeds by a case analysis of the transition $\alpha \to \alpha'$. Crucial in this analysis is the observation that $\alpha \preccurlyeq \beta$ implies that $\alpha = (T, p, Q)$ and $\beta = (T', p, Q)$, for some T and T' such that

$$L(T) \vdash \varphi\sigma \iff L(T') \vdash \varphi\sigma'$$

for renamings $\sigma, \sigma' \in \Sigma(m)$, where m is a method defined by the given class C, such that $(\sigma, u, S) \in \bar{\alpha}$ and $(\sigma', u, S) \in \bar{\beta}$, for some closure (u, S) and predicate φ of the method m. This follows from Lemma 6.7.1 and that $f \preccurlyeq f'$ implies $T(f) = T(f')$, where f and f' represent α and β, respectively. Note that by definition, f' does not add occurrences of closures which do not already occur in f. □

It follows that abstract systems for which the transition relation is computable are well-structured transition systems (see [36] for an excellent explanation and overview of well-structured transition systems). For such systems the *covering* problem is decidable. That is, for any two abstract configurations α and β it is decidable whether starting from α it is possible to cover β, meaning, whether there exists a computation $\alpha \to^* \alpha'$ such that $\beta \preccurlyeq \alpha'$. To show that this implies decidability of absence of deadlock, let α be a *basic* (abstract) deadlock configuration if α is a deadlock configuration according to Definition 6.4.1 and for any closure (u, S) there exists *at most one renaming* σ such that $(\sigma, u, S) \in \bar{\alpha}$. Note that thus $f(c) = 1$, for any closure c, where f represents α. Let Δ denote the set of all basic deadlock configurations. Note that this is a finite set. Further, for every (abstract) deadlock configuration α there exists a basic deadlock configuration $\alpha' \in \Delta$ such that $f \preccurlyeq f'$, where f and f' represent α and α', respectively. This is because the different renamings of the same closure do not affect the definition of a deadlock. Given an initial abstract configuration α, we now can phrase presence of deadlock as the covering problem of deciding whether there exists a computation starting from α reaching a configuration β that covers a deadlock configuration in Δ.

Summarizing the above, we have the following the main technical result of this chapter.

Theorem 5. *Given an abstract system with a computable transition relation and an abstract configuration α, it is decidable whether*

$$\{\, \beta \mid \alpha \rightarrow^* \beta \,\} \cap \{\, \beta \mid \exists \beta' \in \Delta \colon \beta' \preccurlyeq \beta \,\} = \emptyset \tag{6.1}$$

Given this result and the correctness of predicate abstraction, to show that an initial concrete configuration (Γ, p, Q) does *not* give rise to a local deadlock, it suffices to construct an abstract configuration $\alpha = (T, p, Q)$ such that $\Gamma \models L(T)$ and for which Equation (6.1) holds. Note that we can construct T by the constructing pairs (σ, u), where $u = \{\phi \in \mathit{Pred}(m) \mid \Gamma \models \phi\sigma\}$ (assuming that $\sigma \in \Sigma(m)$).

6.8 Conclusion

For future work we first have to validate our method for detecting local deadlock in tool-supported case studies. For this we envisage the use of the theorem-prover KeY [3] for the construction of the abstract transition relation, and its integration with *on-the-fly* reachability analysis of the abstract transition system.

Another major challenge is the extension of our method to (predicate) abstraction of *local* futures, that is, futures generated by self calls. Note that in the method described in the present chapter, we do not distinguish between these futures and those generated by external calls. The main problem is to extend the abstraction method to describe and reason about local futures which preserves the properties of a well-structured transition system.

Of further interest, in line with the above, is the integration of the method of predicate abstraction in the theorem-prover KeY for reasoning *compositionally* about general safety properties of actor-based programs. For reasoning about programs in the ABS language this requires an extension of our method to *synchronous* method calls and *concurrent object groups*.

Epilogue

In the following, we present a few major research directions concerning the new modeling and analysis techniques introduced in this thesis.

Regarding the preferential attachment case study discussed in parts II and III, the proposed models are implemented in the ABS with the Haskell backend, and [10–12] that represent the performance results for the generation of the social networks. A further challenge is to investigate the practical limits of the network size (i.e., the number of nodes in the resulting network) that can be generated in the parallel and distributed implementations. To this aim, the models need to be further investigated for improvements in both time and memory complexities, that possibly enable generation of larger networks. Also the Haskell backend can be further improved such that ABS leverages efficient underlying Haskell data structures.

The ABS with Haskell backend supports real-time programming techniques which allows for specifying deadlines with method invocations. This provides an interesting basis to extend ABS with real-time data streaming which may, as an example, involve timeouts on read operations. Another interesting direction is to extend the various formal analysis techniques (e.g., deadlock detection, general functional analysis based on method contracts) currently supported by the ABS to the ABS model of streaming data discussed in part III.

A major new research direction, in line with the deadlock analysis technique introduced in part IV, is to extend the predicate abstraction technique to the full ABS language. This requires the development of abstraction techniques which capture in a finite model an unbounded number of actors and their interactions.

This line of research is related to the development of a theory for proving correctness of ABS models. An open problem in this area is a proof theory for an actor-based language like ABS which integrates asynchronous method invocations, futures and cooperative scheduling, which is both sound and complete. Further proof-theoretical challenges concern the asynchronous programming techniques of data streaming and multi-threaded actors introduced in this thesis.

Acknowledgements

Foremost, I would like to express my gratitude to all those who helped me along the course of my PhD and made it possible to accomplish the journey. I would like to thank my colleagues in the Formal Methods group at CWI: Frank, Farhad, Jan, Vlad, Nikolaos, Kasper, Sung, Benjamin, Hans, Jana, and others, for creating a pleasant working environment. With some of you I went on trips for different scientific occasions which I have great memories from. We had many fun, exciting and instructive discussions, especially over lunch and coffee times.

I would like to thank Marcello Bonsangue for his great support during my career as lecturer at LIACS. Teaching Logic and Theory of Languages and Automata to many enthusiastic bachelor students was a great experience.

My sincere thanks goes to my family for all the kind and generous support remotely throughout the years of study. I proudly dedicate this book to Azita, my lovely sister and my hero, who is fighting for her health with a big smile on her face.

I would like to give a warm thankful message to my friends: Hosein, Mojtaba, Hadi, Reza and Mehran. I am very grateful to have all of you in my life at once, you genuine people! I would like to thank my great friend Masoud for his wonderful support during my PhD.

I would also like to mention the friends whom I had the chance to know during my PhD studies, alphabetically listed: Ali, Amal, Amin, Amir, Amirhossein, Andrej, Arshia, Benjamin, Behnaz, Bita, Dena, Elli, Emke, Erfan, Fereshteh, Hoda, Irene, Iris, Jo, Julia, Kamran, Kasper, Leila, Maayke, Mahtab, Masood, Maziar, Melissa, Mehdi, Mehri, Miad, Mohammadreza, Mostafa, Mozhdeh, Nasrin, Niloofar, Parisa, Pieta, Rahul, Rozita, Samira, Sara, Saskia, Sepideh, Tannaz, Vlad, Yaser and Ylva. Thank you all for making this PhD a wonderful journey for me.

Summary

The mainstream object-oriented languages use multi-threading as the model of concurrency and parallelism. However, reasoning about the correctness of multi-threaded programs is notoriously difficult. Also due to the complexity of balancing work evenly across cores, the thread model is of little benefit for efficient processor use or horizontal scalability. On the other hand, chip manufacturers are rapidly moving towards so-called manycore chips with thousands of independent processors on the same silicon real estate. Current programming languages can only leverage the potential power by inserting code with low level concurrency constructs, sacrificing clarity. Alternatively, a programming language can integrate a thread of execution with a stable notion of identity, e.g., in active objects.

Abstract Behavioural Specification (ABS) is a language for designing *executable* models of parallel and distributed object-oriented systems based on active objects, and is defined in terms of a formal operational semantics which enables a variety of static and dynamic analysis techniques for the ABS models.

The overall goal of this thesis is to extend the asynchronous programming model and the corresponding analysis techniques in ABS. Based on the different results, the thesis is structured as follows: Part I gives a preliminary overview of the ABS. In part II, we apply an extension of ABS with a restricted notion of shared memory to provide a parallel and distributed model of *preferential attachment* which is used to simulate large-scale social networks with certain mathematical properties. In Part III, we formally extend ABS to enhance both asynchronous programming by data streaming between processes, and parallelism by multi-threading within an actor. Finally in part IV, a new technique based on predicate abstraction is introduced to analyze the ABS models for the absence of deadlock within an actor.

Samenvatting

Veelvoorkomende objectgeoriënteerde programmeertalen maken gebruik van "multithreading" als het gaat om het modelleren van parallelle berekeningen: er zijn meerdere leidraden aan de hand waarvan meerdere aan elkaar parallellopende berekeningen worden uitgevoerd. Echter is het redeneren over de correctheid van programma's met meerdere leidraden erg lastig. Bovendien is het evenredig verdelen van rekenwerk over rekenkernen complex: het leidraadmodel benut mogelijk niet alle beschikbare rekencapaciteit van parallelle processoren en is lastig horizontaal op te schalen. Computerchipfabrikanten maken tegenwoordig chips met meer rekenkernen dan voorheen. Zo zijn chips, met duizenden onafhankelijke rekenkernen geëtst op dezelfde siliciumplaat, niet meer ondenkbaar. Huidige programmeertechnieken gebaseerd op het leidraadmodel, kunnen niet op eenvoudige wijze gebruikmaken van de toename in rekencapaciteit van dergelijke chips, tenzij programma's hiervoor specifiek zijn ontworpen: dit vergt veel bijzondere operaties, op laag niveau in de architectuur, om toename in parallellisme te bevangen. Het alternatief is een andere programmeertechniek, waarbij de ideeën van een leidraad van een berekening en van een zekere identiteit van een object zijn verenigd. Deze programmeertechniek wordt ook wel programmeren met "actieve objecten" genoemd.

De ABS-taal (ABS staat voor "Abstract Behavioural Specification") is bedoeld voor het ontwerpen van *uitvoerbare* wiskundige modellen van parallelle en gedistribueerde, objectgeoriënteerde computersystemen. In ABS maakt men gebruik van actieve objecten. De taal is gedefinieerd in termen van een formele operationele semantiek, die een verscheidenheid aan statische- en dynamische analysetechnieken mogelijk maakt: o.a. het detecteren van wederzijdse uitsluiting ("deadlock").

Het uiteindelijke doel van dit proefschrift is het uitbreiden van het programmeermodel en de bijbehorende analysetechnieken in ABS. Dit proefschrift is opgedeeld in vier delen op basis van verschillende resultaten: deel I beschrijft vliegensvlug de ABS-taal, benodigd voor de andere delen. In deel II passen we een uitbreiding, viz. een beperkte vorm van gemeenschappelijke geheugencellen, toe om een parallel en gedistribueerd model van het "preferential attachment"-algoritme te geven, dat men gebruikt voor simulatie van grootschalige sociale netwerken met bepaalde wiskundige eigenschappen. In deel III formaliseren we uitbreidingen van ABS voor programmeren met gegevensstromen tussen processen, en meerdere leidraden binnen objecten. Tenslotte wordt in deel IV een nieuwe techniek geïntroduceerd gebaseerd

op predicatenabstractie, om ABS modellen te analyseren op vrijheid van wederzijdse uitsluiting.

Bibliography

[1] Gul Agha. Actors: A model of concurrent computation in distributed systems. Technical report, DTIC Document, 1985.

[2] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, 1986.

[3] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book – From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.

[4] Maksudul Alam, Maleq Khan, and Madhav V Marathe. Distributed-memory parallel algorithms for generating massive scale-free networks using preferential attachment model. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 91. ACM, 2013.

[5] Elvira Albert, Puri Arenas, Jesús Correas, Samir Genaim, Miguel Gómez-Zamalloa, Germán Puebla, and Guillermo Román-Díez. Object-sensitive cost analysis for concurrent objects. *Software Testing, Verification and Reliability*, 25(3):218–271, 2015.

[6] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Mikroelektronik och informationsteknik, 2003.

[7] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. Concurrent programming in erlang. 1993.

[8] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[9] James Atwood, Bruno Ribeiro, and Don Towsley. Efficient network generation under general preferential attachment. *Computational Social Networks*, 2(1):1, 2015.

[10] Keyvan Azadbakht, Nikolaos Bezirgiannis, and Frank S de Boer. Distributed network generation based on preferential attachment in abs. In *International Conference on Current Trends in Theory and Practice of Informatics*, pages 103–115. Springer, 2017.

[11] Keyvan Azadbakht, Nikolaos Bezirgiannis, and Frank S de Boer. On futures for streaming data in ABS. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 67–73. Springer, 2017.

[12] Keyvan Azadbakht, Nikolaos Bezirgiannis, Frank S de Boer, and Sadegh Aliakbary. A high-level and scalable approach for generating scale-free graphs using active objects. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1244–1250. ACM, 2016.

[13] Keyvan Azadbakht, Frank S de Boer, Nikolaos Bezirgiannis, and Erik de Vink. A formal actor-based model for streaming the future. *Science of Computer Programming*, 186:102341, 2019.

[14] Keyvan Azadbakht, Frank S. de Boer, and Erik de Vink. Deadlock detection for actor-based coroutines. In Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik de Vink, editors, *Formal Methods*, pages 39–54, Cham, 2018. Springer International Publishing.

[15] Keyvan Azadbakht, Frank S. de Boer, and Vlad Serbanescu. Multi-threaded actors. In *Proceedings 9th Interaction and Concurrency Experience, ICE 2016*, volume 223 of *EPTCS*, pages 51–66, 2016.

[16] David Bader, Kamesh Madduri, et al. Parallel algorithms for evaluating centrality indices in real-world networks. In *Parallel Processing, 2006. ICPP 2006. International Conference on*, pages 539–550. IEEE, 2006.

[17] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[18] Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[19] Vladimir Batagelj and Ulrik Brandes. Efficient generation of large random networks. *Physical Review E*, 71(3):036113, 2005.

[20] Nikolaos Bezirgiannis and Frank de Boer. ABS: a high-level modeling language for cloud-aware programming. In *International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, pages 433–444. Springer, 2016.

[21] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I Pun, S Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. Parallel objects for multicores: A glimpse at the parallel language encore. In *Formal Methods for Multicore Programming*, pages 1–56. Springer, 2015.

[22] Denis Caromel. Keynote 1-strong programming model for strong weak mobility: The proactive parallel suite. In *Mobile Data Management, 2008. MDM'08. 9th International Conference on*, pages xvi–xvi. IEEE, 2008.

[23] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, volume 4, pages 442–446. SIAM, 2004.

[24] Melvin E. Conway. Design of a separable transition-diagram compiler. *Commununications of the ACM*, 6(7):396–408, 1963.

[25] Leonardo Dagum and Rameshm Enon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[26] Frank S. de Boer, Mario Bravetti, Immo Grabe, Matias David Lee, Martin Steffen, and Gianluigi Zavattaro. A Petri net based analysis of deadlocks for active objects and futures. In *FACS*, volume 7684, pages 110–127. Springer, 2012.

[27] Frank S De Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *Programming Languages and Systems*, pages 316–330. Springer, 2007.

[28] Frank S. de Boer and Stijn de Gouw. Being and change: Reasoning about invariance. In *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. Proceedings*, pages 191–204, 2015.

[29] Frank S. de Boer, Mohammad Mahdi Jaghoori, Cosimo Laneve, and Gianluigi Zavattaro. Decidability problems for actor systems. In *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings*, pages 562–577, 2012.

[30] Crystal Chang Din, Richard Bubel, and Reiner Hähnle. Key-abs: A deductive verification tool for the concurrent modelling language abs. In *International Conference on Automated Deduction*, pages 517–526. Springer International Publishing, 2015.

[31] Johan Dovland, Einar Broch Johnsen, and Olaf Owe. Verification of concurrent objects with asynchronous method calls. In *2005 IEEE International Conference on Software - Science, Technology and Engineering (SwSTE 2005), 22-23 February 2005, Herzelia, Israel*, pages 141–150, 2005.

[32] Jeff Epstein, Andrew P Black, and Simon Peyton-Jones. Towards haskell in the cloud. In *ACM SIGPLAN Notices*, volume 46, pages 118–129. ACM, 2011.

[33] Paul Erdös and Alfréd Rényi. On the central limit theorem for samples from a finite population. *Publ. Math. Inst. Hungar. Acad. Sci*, 4:49–61, 1959.

[34] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, 35(2):114–131, 2003.

[35] Kiko Fernandez-Reyes, Dave Clarke, and Daniel S. McCain. Part: An asynchronous parallel abstraction for speculative pipeline computations. In *COORDINATION*. To appear, 2016.

[36] Alain Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1):63–92, 2001.

[37] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[38] Elena Giachino, Ludovic Henrio, Cosimo Laneve, and Vincenzo Mastandrea. Actors may synchronize, safely! In *18th International Symposium on Principles and Practice of Declarative Programming*, pages 118–131, 2016.

[39] Elena Giachino, Cosimo Laneve, and Michael Lienhardt. A framework for deadlock detection in core ABS. *Software & Systems Modeling*, 15(4):1013–1048, 2016.

[40] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *International Conference on Computer Aided Verification*, pages 72–83. Springer, 1997.

[41] Douglas Gregor and Andrew Lumsdaine. The parallel bgl: A generic library for distributed graph computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, page 2, 2005.

[42] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2):202–220, 2009.

[43] Max Haustein and Klaus-Peter Löhr. Jac: declarative java concurrency. *Concurrency and Computation: Practice and Experience*, 18(5):519–546, 2006.

[44] Ludovic Henrio, Fabrice Huet, and Zsolt István. Multi-threaded active objects. In *COORDINATION*, pages 90–104. Springer, 2013.

[45] Ludovic Henrio, Cosimo Laneve, and Vincenzo Mastandrea. Analysis of synchronisations in stateful active objects. In *International Conference on Integrated Formal Methods*, pages 195–210, 2017.

[46] Ludovic Henrio and Justine Rochas. From modelling to systematic deployment of distributed active objects. In *International Conference on Coordination Languages and Models*, pages 208–226. Springer, 2016.

[47] Carl Hewitt. Description and theoretical analysis (using schemata) of planner: A language for proving theorems and manipulating models in a robot. Technical report, Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab, 1972.

[48] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In *International Symposium on Formal Methods for Components and Objects*, pages 142–164. Springer, 2010.

[49] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software & Systems Modeling*, 6(1):39–58, 2007.

[50] Einar Broch Johnsen, Olaf Owe, Dave Clarke, and Joakim Bjørk. A formal model of service-oriented dynamic object groups. *Science of Computer Programming*, 115:3–22, 2016.

[51] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1-2):23–66, 2006.

[52] Eduard Kamburjan, Crystal Chang Din, and Tzu-Chun Chen. Session-based compositional analysis for actor-based languages using futures. In *International Conference on Formal Engineering Methods*, pages 296–312, 2016.

[53] Eric Kerfoot, Steve McKeever, and Faraz Torshizi. Deadlock freedom through object ownership. In *5th International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming*, 2009.

[54] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, D Sivakumar, Andrew Tomkins, and Eli Upfal. Stochastic models for the web graph. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 57–65. IEEE, 2000.

[55] R Greg Lavender and Douglas C Schmidt. Active object–an object behavioral pattern for concurrent programming. 1995.

[56] Jurij Leskovec. *Dynamics of large networks*. ProQuest, 2008.

[57] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[58] Yi-Chen Lo, Hung-Che Lai, Cheng-Te Li, and Shou-De Lin. Mining and generating large-scaled social networks via mapreduce. *Social Network Analysis and Mining*, 3(4):1449–1469, 2013.

[59] Yi-Chen Lo, Cheng-Te Li, and Shou-De Lin. Parallelizing preferential attachment models for generating large-scale social networks that cannot fit into memory. In *Privacy, Security, Risk and Trust (PASSAT), 2012 International Conference on and 2012 International Confernece on Social Computing (SocialCom)*, pages 229–238. IEEE, 2012.

[60] Sadegh Nobari, Xuesong Lu, Panagiotis Karras, and Stéphane Bressan. Fast random graph generation. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 331–342. ACM, 2011.

[61] Gordon D Plotkin. A structural approach to operational semantics. 1981.

[62] John C Reynolds. The discoveries of continuations. *Lisp and symbolic computation*, 6(3-4):233–247, 1993.

[63] Jan Schäfer and Arnd Poetzsch-Heffter. Jcobox: Generalizing active objects to concurrent components. In *ECOOP 2010–Object-Oriented Programming*. Springer, 2010.

[64] Jan Schäfer and Arnd Poetzsch-Heffter. Jcobox: Generalizing active objects to concurrent components. In *ECOOP 2010–Object-Oriented Programming*, pages 275–299. Springer, 2010.

[65] Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. Parallel actor monitors: Disentangling task-level parallelism from data partitioning in the actor model. *Science of Computer Programming*, 80:52–64, 2014.

[66] Yury Selivanov. Asynchronous generators. `https://www.python.org/dev/peps/pep-0525/`, 2016.

[67] Koushik Sen and Mahesh Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *International Conference on Computer Aided Verification*, pages 300–314. Springer, 2006.

[68] Vlad Şerbănescu, Keyvan Azadbakht, and Frank de Boer. A java-based distributed approach for generating large-scale social network graphs. In *Resource Management for Big Data Platforms*, pages 401–417. Springer, 2016.

[69] Marjan Sirjani. Rebeca: Theory, applications, and tools. In *Formal Methods for Components and Objects*, pages 102–126. Springer, 2007.

[70] Streams. Version 2.4.17. `http://doc.akka.io/docs/akka/2.4/scala/stream/index.html`, 2017.

[71] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005.

[72] Ming-Chit Tam, Jonathan M Smith, and David J Farber. A taxonomy-based comparison of several distributed shared memory systems. *ACM SIGOPS Operating Systems Review*, 24(3):40–67, 1990.

[73] Roberto Tonelli, Giulio Concas, and Mario Locci. Three efficient algorithms for implementing the preferential attachment mechanism in yule-simon stochastic process. *WSEAS Transactions on Information Science and Applications*, 7(2):176–185, 2010.

[74] Thierry Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In *Chilean Society of Computer Science, 2007. SCCC'07. XXVI International Conference of the*, pages 3–12. IEEE, 2007.

[75] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442, 1998.

[76] Peter Welch and Neil Brown. Communicating sequential processes for java^{tm} (jcsp). `https://www.cs.kent.ac.uk/projects/ofa/jcsp/`, 2014.

[77] Antonty S Williams, Alexander A Mitchell, Robert G Atkinson, C Douglas Hodges, Johann Posch, and Craig H Wittenberg. Method and system for multithreaded processing, January 30 2001. US Patent 6,182,108.

[78] Akinori Yonezawa and Mario Tokoro. Object-oriented concurrent programming. 1986.

[79] Andy Yoo and Keith Henderson. Parallel generation of massive scale-free graphs. *arXiv preprint arXiv:1003.3684*, 2010.