Actors at Work

Behrooz Nobakht

October 31, 2015
Version: v1.0

Clean Thesis Style University

CleanThesis

Department of Clean Thesis Style
Institut for Clean Thesis Dev
Clean Thesis Group (CTG)

Actors at Work

Actors at Work

Behrooz Nobakht

1. Reviewer Frank S. de Boer
Leiden Advanced Institute of Compute Science
Leiden University

2. Reviewer John Doe
Department of Clean Thesis Style
Clean Thesis Style University

Supervisors Jane Doe and John Smith

October 31, 2015

Behrooz Nobakht
Actors at Work
Actors at Work, October 31, 2015
Reviewers: Frank S. de Boer and John Doe
Supervisors: Jane Doe and John Smith

Clean Thesis Style University
Clean Thesis Group (CTG)
Institut for Clean Thesis Dev
Department of Clean Thesis Style
Street address
Postal Code and City

Abstract

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Abstract (different language)

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Acknowledgement

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

This is the second paragraph. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Contents

Part I

Introduction

1 Introduction

1.1 From actor *modelling* to actor *programming*

Object-oriented programming [15, 86] has been of one the dominant paradigms for software engineering in the past three decades. One object interacts with another object using the notion of a method. Method invocations are *blocking*; i.e. the caller object waits until it receives the result of the method call from the callee object. Surprisingly, such model of interaction was *never* the intention of the creator of the paradigm. Originally, object interactions were meant to be *messaging* among objects and objects behaved as autonomous entities possibly on remote locations on a network; Alan Kay clarified later [71, 72]. On the contrary, almost all the object-oriented languages at hand have followed the blocking and synchronous model of messaging based on a common misunderstanding. The original definition of object-oriented is close to another model of computation: actor model.

One of the fundamental elements of actor model [3, 2] is *asynchronous* message passing. In this approach, interactions between objects are modelled as *non-blocking* messages. One object, the sender, communicates a message to the other object, the receiver. At the receiver side, a message eventually leads to a method invocation (in object-oriented paradigm) or a function call (in functional paradigm). In actor model, the locality of the objects is transparent from the messages. A system is composed of objects with communicating messages.

Considerable amount of research have been carried out to mix object-oriented programming languages with actor model. Language extensions, libraries, and even new programming languages are the naturual outcomes of such research. The more mainstream the language, the more research and development are available in the intersection of object orientation and actor model. For example, [69] presents a comparative analysis of such research on Java and JVM languages.

With rapid increase of computation power, a new aspect added to create a triangle: object-oriented, actor model, and *concurrency*. Concurrency makes it harder to verify programs in terms of correctness of runtime behavior. Scheduling of object messages in threads of executions is one of main challenges. In addition, in the presence of scheduling and thread interleaving, it is important that objects are able to protect their state from concurrent modifications for correctness properties. In this research,

we take advantage of a modelling language that aims to mix the three aspects in addition to the capabilities of formal semantics and verification.

ABS [65, 47] is a modelling language for concurrent objects and distributed systems. ABS provides the semantics of actor model [3] as a modelling language. ABS language provides a set of core features including algebraic data types (ADT), functional layer, and concurrency layer. ABS concurrency layer lays its basis on co-operative scheduling of active objects [63]. ABS semantics is completely expressed in structural operational semantics [98]. This allows ABS models to take advantage of various verification methods and static analysis techniques. Co-operative scheduling in ABS has been additionally extended for real-time scheduling considering priorities and time constraints [12, 67]. Thus, in general, ABS concurrency layer is a perfect fit if it can be integrated with industry-level real-world programming languages.

Considering the mainstream programming languages [1], Java [45] is one of the most commonly-used. Additionally, since Java 1.5 and the rise of concurrency API in Java [68], a remarkable effort by the community focus on how concurrency and distribution can be improved in Java as in provided by ABS semantics and actor model. However, not all the development as Java libraries and frameworks form their basis on formal semantics. Therefore, this gives rise to issues and challenges in terms of correctness, semantic preservation, and reasoning.

Java language specification (JLS) [45] clearly concludes that Java language is not comptabile and ready to be extended as a functional language with the support of algebraic data types. This has been the focus of research and development to extend Java to a functional language [95, 52, 93, 16]. Scala language [96] is a result of the research that is a dynamic, functional, and object-oriented language on JVM. Although such extensions over Java towards a functional language exist, they lack formal semantics in the context of concurrency and actor model.

We primarily focus on ABS concurrency layer and deliver its semantics as a Java API. We argue that the advantage of a correct semantic translation from ABS concurrency layer to its Java equivalent constructs overcomes the additional support of functional layer with *partial* formal semantics. Our approach enables programmers to model their Java systems in terms of ABS concurrency semantics and then create a correct mapping and implementation from ABS to Java using an API. In addition, we aim to facilitate ways to actually program in ABS models as if the programmer is able to take advantage of standard Java libraries.

[1] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

1.2 Architecture

In this section, we present a high-level overview of design goals that we pursue in this research. Equivalently, the effectivness and success of this research is measured upon the achievement of the discussed goals and objectives.

Polyglot Programming With the rise of distributed computing challenges, software engineering practice has turned to methods that combine multiple programming languages and models to complete a task. In this approach, different languages with different focus and abstractions contribute to the same problem statement in different layers. Polyglot programming essentially enables software practice to apply the *right* language in the appropriate layer of the problem statement. In this research, we follow the question in which we try to deliver ABS semantics and features in a polyglot approach. The programmer develops models with ABS that *partically* take advantage of the target language features (e.g. from Java). This approach is also referred to as *Foreign Function Interface* in the context of ABS modelling. Listing 1 shows an ABS code that uses `java.util.ArrayList` as a data strucuture. With Polyglot programming in ABS, the programmer does *not* need to define the abstractions necessary for the list construct.

Listing 1: Using Java in ABS

```
1 List<String> params = new ArrayList<>();
2 myObj ! doSomething(params);
```

Scalability Asynchronous message passing in ABS is a fit for distributed systems. In such systems, the number of messsages delivered among actors in the environment is not predictable at runtime. Therefore, distribution challenges performance efficiency or scalability. The goal is to ensure the actor system scales in performance with least influence from the number of asynchronous messages delivered in the system.

Modularity The scope of the research spans to a number of layers revolving around ABS language:

- *Compiling ABS to a target programming language* One first objective is to compile an ABS model to a target programmig language. The compilation can be at *source-to-source* or *source-to-bytecode* level. Target languages potentially include mainstream programming languages such as Java, Erlang, Haskell, and Scala. Thus, the ABS programmer should be able to use an ABS compiler to compile their ABS models to the target languages.

- *Using ABS semantics as an API in a programming language* ABS language is precisely expressed with structural operational semantics [65] in addition

to its syntax definition. Therefore, the semantics of ABS can be delivered in a programming language as Application Programming Interface (API) as long as the programming language provides sufficient constructs to respect ABS language semantics. In addition, such ABS programming API should be *verifiable*. If such ABS mapping to a programming API is provided, a programmer is able to take advantage of ABS semantics without directly programming in ABS. Such capability from ABS enables industry users of mainstraim languages to model their systems in ABS semantics using their programming languages and platforms.

- *Modelling in ABS* ABS language provides solid structure and semantics to model concurrent and distributed systems. The user (that can be a programmer, an analysist, or a researcher) rightfully demands to have access to a tool-set and IDE that allows working with ABS models in a user-friendly way. The ABS IDE and tool-set developers should be able to easily reuse and compose over existing modules and components.

Extensibility Delivering ABS language features as an API brings about a new requirement: to configure or extend the API. It is a common practice in software engineering that components are developed with a mixture of *configuration*, *extension* and *composition*. Therefore, the ABS programmer should be able to extend such API to custom requirements. The extensibility is provided either through configuration of the API or API extension.

Behrooz: *It's worth to include the figure but needs more work to justify its presence and how it relates to this thesis.*

Figure 1.1 presents the general architecture of the implementation.

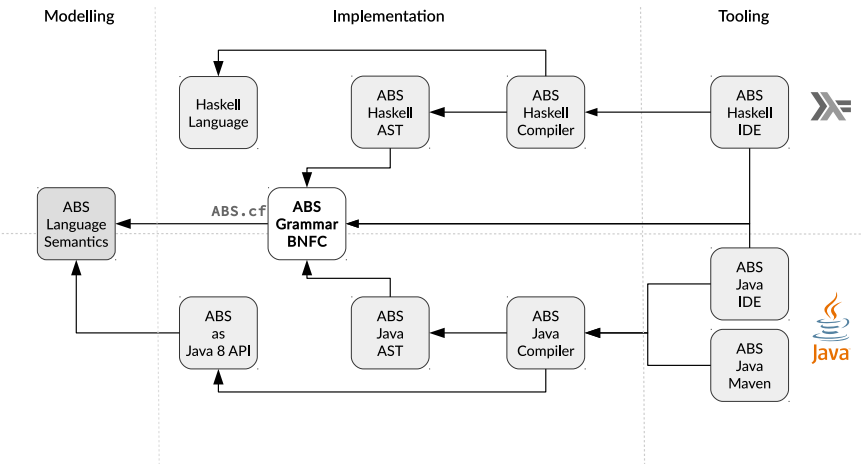


Figure 1.1: General Architecture of ABS API and Java Language Backend

1.3 Literature Overview

We discuss a brief overview of related work in the context of programming languages, actor model, and concurrency. We devide the overview in two levels; one is at the level of the programming languages and the other is for the external (third-party) libraries developed for programming languages.

1.3.1 Programming Languages

In this section, we briefly provide an overview of the programming languages that have targetted the similar problem statements. It is important to note the timeline and the rapid evolution of programming languages trying to provide an actor-based model of asynchronous message passing. In addition, we identify different types of research and work related to actor model and concurrency programming:

First-Class Citizen is the type of programming languages that deliver actor model programming support as a first-class citizen of the language. In such languages, actor model is by-design part of the syntax and semantics of the language.

Implicit By Design refers to the type that there is no explicit notion of actor models in the language syntax or semantics. However, the programming language provides fundamental constructs for concurrenty and asynchronous message passing. Thus, it becomes a trivial task in such type of programming languages to create an abstraction to support the actor model by coding.

External Library refers to the type of programming languages for which actor model support is possible through using an external library developed for the language.

Table 1.1 presents a summary.

Language	Abstraction	Type
Erlang[8, 34]	Process	Implicit By Design
Elixir[109, 33]	Agent	Implicit By Design
Haskell[27]	forkIO & MVars	Implicit By Design
Go[41]	Goroutine	Implicit By Design
Rust[85, 28]	Send & Sync	Implicit By Design
Scala[49]	Akka Actors [2]	External Library

Table 1.1: Actor Model Support in Programming Languages

[2]Scala 2.11.0 adopts Akka as default actor model implementation: `http://docs.scala-lang.org/overviews/core/actors-migration-guide.html`

1.3.2 Frameworks and Libraries

Since many programming languages faced different challenges to provide the necessary syntax and semantics for actor model and concurrency at the level of the language, many libraries and frameworks have been born with a motivation to fill this gap. We observe that the more the language itself is close to the actor model semantics, the less external libraries and frameworks target this gap. In the following, we briefly provide a few frameworks [3] and libraries for Java as one of the main focus areas of this research is to study this gap for a mainstream language such as Java.

Library	Technique	JVM Language
Killim[104, 103]	Byte-Code Modification	Java
Quasar[24]	Byte-Code Modification, Java 8	Clojure, Java
Akka[107, 50]	Scala Byte-Code on JVM	Scala, Java

Table 1.2: Actor programming libraries in Java

One of the main techniques used in libraries to deliver actor programming in Java is byte-code engineering [31, 18, 94]. Byte-code engineering modifies the generated byte-code for compiled classes in Java either during compilation or at runtime. Although, this technique is commonly used and argued to provide better performance optimization [110], it introduces deep challenges regarding the verification of the running byte-code [80, 79].

1.4 Oraganization and Contributions

In this thesis, we focus on the concurrency layer of ABS. We formalize a subset of Java language in regards with ABS. We provide the operaional semantics of the concurrency layer for scheduling of actors with an extension for deadlines and priorities. We deliver ABS concurrency as a Java API. We utilize Java 8 and Java Concurrency API to preserve the semantics of ABS concurrency layer. We sketch a general architecture on how ABS language can be modularly translated into different programming languages. We present how ABS API library can be used in the translation of ABS models into Java.

Table 1.3 summarizes the structure of the research and contributions in the chapters of this text:

[3] A more comprehensive list can be obtained at [70] and `https://en.wikipedia.org/wiki/Actor_model#Programming_with_Actors`

Topic	Part	Chapter/Section
Formalization of the mapping from ABS to Java including the operational semantics and ABS co-operative scheduling in Java	Programming Model (Part II)	Chapter 2 and 3
Design and implementation of ABS concurrency layer in Java	Implementation (Part III)	Chapter 4 and 5
Monitoring method call sequences using annotations	Application (Part IV)	Chapter X
Design and implementation of a massive-scale monitoring system based on ABS API in Java	Application (Part IV)	Chapter 6

Table 1.3: Thesis Contributions Summary

Part II

Programming Model

2 Programming and Deployment of Active Objects with Application-Level Scheduling[1]

Behrooz Nobakht, Frank S. de Boer, Mohammad Mahdi Jaghoori, Rudolph Schlatte

Abstract

We extend and implement a modeling language based on concurrent active objects with application-level scheduling policies. The language allows a programmer to assign priorities at the application level, for example, to method definitions and method invocations, and assign corresponding policies to the individual active objects for scheduling the messages. Thus, we leverage scheduling and performance related issues, which are becoming increasingly important in multi-core and cloud applications, from the underlying operating system to the application level. We describe a tool-set to transform models of active objects extended with application-level scheduling policies into Java. This tool-set allows a direct use of Java class libraries; thus, we obtain a full-fledged programming language based on active objects which allows for high-level control of deployment related issues.

2.1 Introduction

One of the major challenges in the design of programming languages is to provide high-level support for multi-core and cloud applications which are becoming increasingly important. Both multi-core and cloud applications require an explicit and precise treatment of non-functional properties, e.g., resource requirements. On the cloud, services execute in the context of virtual resources, and the amount of resources actually available to a service is subject to change. Multi-core applications require techniques to help the programmer optimally use potentially many cores. At the operating system level, resource management is greatly affected by scheduling which is largely beyond the control of most existing high-level programming languages. Therefore, for optimal use of the available resources, we cannot avoid

[1]This work is partially supported by the EU FP7-231620 project: HATS.

leveraging scheduling and performance related issues from the underlying operating system to the application level. However, the very nature of high-level languages is to provide suitable abstractions that hide implementation details from the programmer. The main challenge in designing programming languages for multi-core and cloud applications is to find a balance between these two conflicting requirements.

We investigate in this paper how concurrent active objects in a high-level object-oriented language can be used for high-level scheduling of resources. We use the notion of concurrent objects in Creol [64, 6]. A concurrent object in Creol has control over one processor; i.e. it has a single thread of execution that is controlled by the object itself. Creol processes never leave the enclosing object; method invocations result in a new process inside the target object. Thus, a concurrent object provides a natural basis for a deployment scheme where each object virtually possesses one processor. Creol further provides high-level mechanisms for synchronization of the method invocations in an object; however, the scheduling of the method invocations are left unspecified. Therefore, for the deployment of concurrent objects in Creol, we must, in the very first place, resolve the basic scheduling issue; i.e. which *method* in which *object* to select for execution. We show how to introduce priority-based scheduling of the messages of the individual objects at the application-level itself.

In this paper we also propose a tool architecture to deploy Creol applications. To prototype the tool architecture, we choose Java as it provides low-level concurrency features, i.e., threads, futures, etc., required for multi-core deployment of object-oriented applications. The tool architecture prototype transforms Creol's constructs for concurrency to their equivalent Java constructs available in the `java.util.-concurrent` package. As such, Creol provides a high-level structured programming discipline based on active objects on top of Java. Every active object in Creol is transformed to an object in Java that uses a priority manager and scheduler to respond to the incoming messages from other objects. Besides, through this transformation, we allow the programmer to seamlessly use, in the original Creol program, Java's standard library including all the data types. Thus, our approach converts Creol from a modeling language to a full-fledged "programming" language.

Section 2.2 first provides an overview of the Creol language with application-level scheduling. In Section 2.3, we elaborate on the design of the tool-set and the prototype. The use of the tool-set is exemplified by a case study in Section 2.4. Section 3.5 summarizes the related work. Finally, we conclude in Section 6.7.

Listing 2: Exclusive Resource in Creol

```
1 interface Resource begin
2
3     op request()
4     op release()
5 end
6
7 class ExclusiveResource implements Resource begin
8   var taken := false;
9
10  op request () ==
11    await ~taken;
12    taken := true;
13  op release () ==
14    taken := false
15 end
```

2.2 Application-Level Scheduling

Creol [64] is a full-fledged object-oriented language with formal semantics for modeling and analysis of systems of concurrent objects. Some Creol features include interface and class inheritance and being strongly typed such that safety of dynamic class upgrades can be statically ensured [116]. In this section, we explain the concurrency model of Creol using a toy example: an exclusive resource, i.e., a resource that can be exclusively allocated to one object at a time, behaving as a mutual exclusion token. Further, we extend Creol with priority-based application-level scheduling.

The state of an object in Creol is initialized using the `init` method. Each object then starts its active behavior by executing its `run` method if defined. When receiving a method call, a new process is created to execute the method. Creol promotes *cooperative* non-preemptive scheduling for each active object. It means that a method runs to completion unless it explicitly releases the processor. As a result, there is no race condition between different processes accessing object variables. Release points can be conditional, e.g., **`await`** `~taken`. If the guard at a release point evaluates to true, the process keeps the control, otherwise, it releases the processor and becomes disabled as long as the guard is not true. Whenever the processor is free, an enabled process is *nondeterministically* selected for execution, i.e., scheduling is left unspecified in standard Creol in favor of more abstract modeling.

To explain extending Creol with priority specification and scheduling, we take a *client/server* perspective. Each caller object is viewed as a client for the callee object who behaves as a server. We define priorities at the level of language constructs like method invocation or definition rather than low-level concepts like processes.

On the server side, an interface may introduce a *priority range* that is available to all clients. For instance, in Line 2 of `Resource` in Listing 2, we can define a priority range:

```
priority range 0..9
```

On the client side, method calls may be given priorities within the range specified in the server interface. For example, calling the `request` method of the `mutex` object:

```
mutex ! request() priority(7);
```

Scheduling *only* on the basis of client-side priority requirements is too restrictive. For example, if there are many `request` messages with high priorities and a low priority `release`, the server might block as it would fail to schedule the `release`. In this particular example, we can solve this problem by allowing the servers to prioritize their methods. This involves a declaration of a priority range generally depending on the structure of the class. In our example, assuming a range 0..2 added in Line 9, this requires changing the method signatures in the `ExclusiveResource` class:

```
op request() priority(2) == ...
op release() priority(0) == ...
```

This gives `release` a higher priority over `request`, because by default, the smaller values indicate higher priorities. Furthermore, the server may also introduce priorities on certain characteristics of a method invocation such as the kind of "release statement" being executed. For example, a process waiting at Line 11 could be given a higher priority over new requests by assigning a smaller value:

```
await ~taken priority(1);
```

The priority can be specified in general as an expression; we used here only constant values. Evaluation of this expression at runtime should be within the given range. If no priority is specified for a method invocation or definition, a default priority value will be assigned.

We discussed different levels of application-level priorities. Note that now each method invocation in the queue of the object involves a tuple of priorities. We define a general function δ as an "abstract priority manager":

$$\delta : P_1 \times P_2 \times P_3 \times \ldots \times P_n \longrightarrow P$$

The function δ maps the different levels of priority in the object ($\{P_1, \ldots, P_n\}$) to a single priority value in P that is internally used by the object to order all the messages that are queued for execution. Each method in the object may have a δ

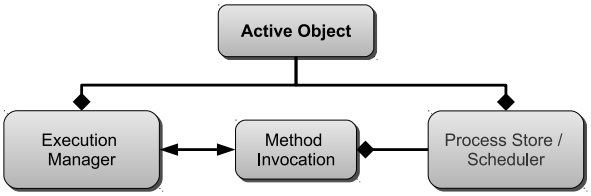


Figure 2.1: $\mathcal{C}$risp Architecture: Structural Overview

assigned to it. In an extended version of δ, it can also involve the internal state of the object, e.g., the fields of the object. In this case, we have dynamic priorities.

For example, in `ExclusiveResource`, we have two different levels of priorities, namely the client-side and server-side priorities, which range over $P_1 = \{0, \ldots, 9\}$ and $P_2 = \{0, 1, 2\}$, respectively. So, we define $\delta : P_1 \times P_2 \to P$ as:

$$\delta(p_1, p_2) = p_1 + p_2 \times |P_1|$$

To see how it works, consider a `release` and a `request` message, both sent with the client side priority of 5. Considering the above method priorities, we have $\delta(5, request) = \delta(5, 2) = 25$ and $\delta(5, release) = \delta(5, 0) = 5$. It is obvious that the range of the final priority value is $P = \{0, \ldots, 29\}$.

Note that the abstract priority manager in general does *not* completely fix the "choice" of which method invocation to execute. In our tool-set, we include an extensible library of predefined scheduling policies such as strong or weak fairness that further refine the application-specific multi-level priority scheduling. The policies provided by the library are available to the user to annotate the classes. We may declare a scheduling policy for the `ExclusiveResource` class by adding at Line 9 in Listing 2:

```
scheduling policy StronglyFairScheduler;
```

A scheduling policy may use runtime information to re-compute the dynamic priorities and ensure properties such as fairness of the selected messages; for instance, it may take advantage of the "aging" technique to avoid starvation.

2.3 Tool Architecture

We have implemented a tool to translate Creol programs into Java programs for execution, called *$\mathcal{C}$risp* ($\mathcal{C}$reolized Interacting Scheduling Policies). $\mathcal{C}$risp provides a one-to-one mapping from Creol classes and methods to their equivalent Java con-

ADD:

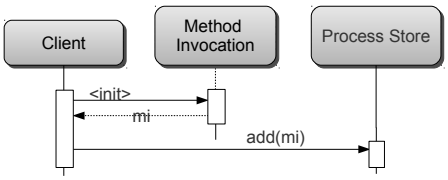


Figure 2.2: Adding the new MethodInvocations are performed on the Client side.

structs. In order to implement active objects in Creol, we use the `java.util.concurrent` package (see Figure 2.1). Each active object consists of an instance of a process store and an execution manager to store and execute the method invocations.

Incoming messages to the active object are modeled as instances of `MethodInvocation`, a subclass of `java.util.concurrent.FutureTask` that wraps around the original method call. Therefore the caller can later get the result of the call. Additionally, `MethodInvocation` encapsulates information such as priorities assigned to the message.

The `ProcessStore` itself uses an implementation of the `BlockingQueue` interface in `java.util.concurrent` package. Implementations of `BlockingQueue` are thread-safe, i.e., all methods in this interface operate atomically using internal locks encapsulating the implementation details from the user.

The `ExecutionManager` component is responsible for selecting and executing a pending method invocation. It makes sure that only one method runs at a time, and takes care of processor release points.

In the following, we explain how the active object behaves in different scenarios, from a "client/server" perspective.

2.3.1 A New Method Invocation

A method call needs to be sent from the client to the server in an asynchronous way. To implement this in Java, the client first constructs an instance of `MethodInvocation` that wraps around the original method call for the server. Then, there are two implementation options how to add it to the server's process store:

1. The client calls a method on the server to store the instance.

2. The client directly adds the method invocation into the process store of the server.

TAKE:

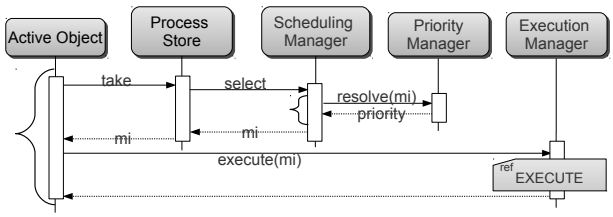


Figure 2.3: An active object selects a method invocation based on its local scheduling policy. After a method finishes execution, the whole scenario is repeated.

In option 1, the server may be busy doing something else. Therefore, in this case the client must wait until the server is free, which is against the asynchronous nature of communication. In Option 2, the Java implementation of each active object exposes its process store as an immutable public property. Thus, the actual code for adding the new method invocation instance is run in the execution thread of the client. We adopt the second approach as depicted in Figure 2.2. At any time, there can be concurrent clients that are storing instances of `MethodInvocation` into the server's process store, but since the process store implementation encapsulates the mechanisms for concurrency and data safety, the clients have no concern on data synchronization and concurrency issues such as mutual exclusion. The method or policy used to store the method invocation in the process store of the server is totally up to the server's process store implementation details.

2.3.2 Scheduling the Next Method Invocation

On the server side of the story, an active object repeatedly fetches an instance of method invocation from its process store for execution (cf. Fig 2.3). The process store uses its instance of `SchedulingManager` to choose one of the method invocations. *C*risp has some predefined scheduling policies that can be used as scheduling managers; nevertheless, new scheduling policies can be easily developed and customized based on the requirements by the user.

`SchedulingManager` is an interface the implementations of which introduce a function to *select* a method invocation based on different possible criteria (such as time or data) that is either predefined or customized by the user. The scheduler manager is a component used by process store when asked to remove and provide an instance of method invocation to be executed. Thus, the implementation of the scheduling manager is responsible how to choose one method invocation out of the ones currently stored in the process store of the active object. Different flavors of the scheduling manager may include time-based, data-centric, or a mixture.

Every method invocation may carry different levels of priority information, e.g., a server side priority assigned to the method or a client side priority. The `PriorityManager`

EXECUTE:

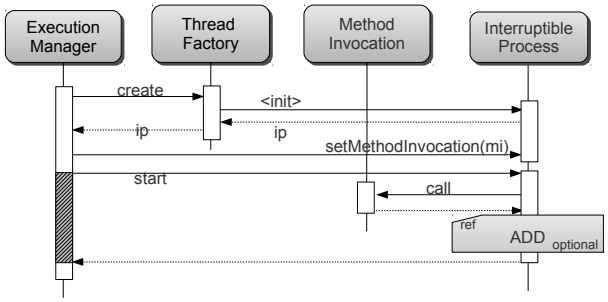


Figure 2.4: A method invocation is executed in an interruptible process. The execution manager thread is blocked while the interruptible process is running.

provides a function to determine and resolve a final priority value in case there are different levels of priorities specified for a method invocation. Postponing the act of resolving priorities to this point rather than when inserting new processes to the store enables us to handle dynamic priorities.

2.3.3 Executing a Method Invocation

To handle processor release points, Creol processes should preserve their state through the time of awaiting. This is solved by assigning an instance of a Java thread to each method invocation. An `ExecutionManager` instance, therefore, employs a "thread pool" for execution of its method invocations. To create threads, it makes use of the factory pattern: `ThreadFactory` is an interface used by the execution manager to initiate a new thread when new resources are required. We cache and reuse the threads so that we can control and tune the performance of resource allocation.

When a method invocation has to release the processor, its thread must be suspended and, additionally, its continuation must be added to the process store. To have the continuation, the thread used for the method invocation should be preserved to hold the current state; otherwise the thread may be taken away and the continuation is lost. The original `wait` in Java does not provide a way to achieve this requirement. Therefore, we introduce `InterruptibleProcess` as an extension of `java.lang.Thread` to preserve the relation.

As shown in Figure 2.4, the thread factory creates threads of type `InterruptibleProcess`. The execution manager thread blocks as soon as it starts the interruptible process which executes the associated method invocation. If the method releases the processor before completion, it will be added back to the process store as explained in Section 2.3.1. When a suspended method invocation is resumed, the execution manager skips the creation of a new thread and reuses the one that was assigned to the method invocation before.

2.3.4 Extension Points

Besides the methods `add` and `take` for adding and removing method invocations, `ProcessStore` provides methods such as `preAdd` and `postAdd` along with `preTake` and `postTake` respectively to enable further customization of the behavior before/after adding or taking a method invocation to/from the store. These extension points enable the customization of priority or scheduling management of the method invocations.

*C*risp provides two generic interfaces for priority specification and scheduling management: `PriorityManager` and `SchedulingManager` respectively. These two interface can be freely developed by the programmer to replace the generated code for priorities and scheduling of the messages. It will be the task of the programmer to configure the generated code to use the custom developed classes.

2.4 Case Study

In this section, we demonstrate the use of application-level scheduling and *C*risp with a more complicated example: we program the "Sieve of Eratosthenes" to generate the prime numbers. To implement this algorithm, the `Sieve` is initialized by creating an instance of the `Prime` object representing the first prime number, i.e., `two`. The active behavior of `Sieve` consists of generating all natural numbers up to the given limit (100000 in our example) and passing them to the object `two`. A `Prime` object that cannot divide its input number passes it on to the next `Prime` object; if there is no next object, then the input number is the next prime and therefore a new object is created.

We parallelize this algorithm by creating active objects that run in parallel. The numbers are passed asynchronously as a parameter to the `divide` message. Correctness of the parallel algorithm essentially depends on the numbers being processed in increasing order. For example, if object `two` processes 9 before 3, it will erroneously treat 9 as a prime, because 3 is not there yet to eliminate 9. To avoid erroneous behavior, we use the actual parameter n in `divide` method to define its priority level, too (see line 15). As a result, every invocation of this method generates a process with a priority equal to its parameter. The default scheduling policy for objects always selects a process for execution that has the smallest priority value. This guarantees that the numbers sent to a `Prime` object are processed exactly in increasing order.

We used two different setups to execute the prime sieve program and compare the results. In one setting, we ran the parallel prime sieve compiled by *C*risp; in the

Listing 3: Prime Sieve in Creol

```
1  interface IPrime begin
2    op divide(n:Int)
3  end
4
5  class Sieve begin
6    var n: Int, two: IPrime
7    op init == two := new Prime(2); n := 3
8    op run ==
9      !two.divide(n);
10     if n < 100000 then n := n + 1; !run() end
11 end
12
13 class Prime(p: Int) implements IPrime begin
14   var next: IPrime
15   op divide(n: Int) priority (n) ==
16     if (n % p) ≠ 0 then
17       if next ≠ null then
18         !next.divide(n)
19       else
20         next := new Prime(n)
21     end end
22 end
```

other, we executed a sequential program developed based on the same algorithm that uses a single thread of execution in JVM. We performed the experiments on a hardware with 2 CPU's of each 2GHz with a main memory of size 2GB. We ran both programs for $max \in \{10000, 20000, 30000, 50000, 100000\}$.

The first interesting observation was that *C*risp prime sieve utilizes all the CPU's on the underlying hardware as much as possible during execution. This can be seen in Figure 2.6 which shows the CPU usage. Both CPUs are fairly in use while running this program. Figure 2.5 depicts the results of the monitoring of the parallel prime sieve in *C*risp using Visual VM tool. It depicts the number of threads generated for the program. This shows that *C*risp can handle a massive number of concurrent tasks.

One interesting feature of *C*risp is that the execution of any program under *C*risp can constantly utilize the *minimum* memory that can be allocated for each thread in JVM (thread stack). In JVM, the size of thread stack can be configured using `-Xss` option for every run. To demonstrate this feature of *C*risp, we collected the minimum stack size needed for every program run in Table 2.1. All *C*risp runs use the minimum thread stack size of `64k` that is possible for the JVM. On the contrary, the stack size required for the sequential version of the sieve program increases by the number of primes detected. This is also expected because of the long chain of method calls in the sequential sieve.

Having the constant thread stack size feature, *C*risp provides another interesting feature. It can handle huge number of thread generation if required. Table 2.2

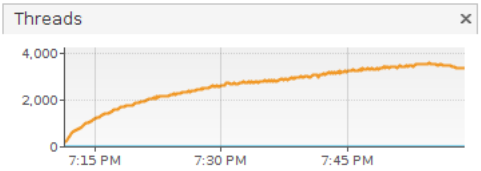


Figure 2.5: Increasing parallelism in $\mathcal{C}$risp for Prime Sieve

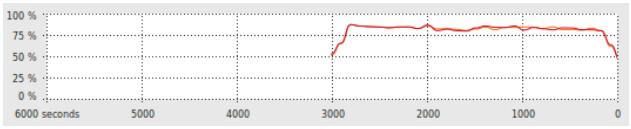


Figure 2.6: Utilizing both CPUs with Prime Sieve in $\mathcal{C}$risp

summarizes the thread generation data for parallel prime sieve in $\mathcal{C}$risp. It shows scalability of $\mathcal{C}$risp as p rises for parallel prime sieve.

As the results show the use of Java threads is costly; first, $\mathcal{C}$risp does not need much of the memory allocated to each thread and, second, the context switch cost is higher for larger memory allocation. In line with this, JVM uses a *one-to-one* mapping from an application-level Java thread to a native OS-level thread. In the current setting, the context switch of the threads are in the OS level. When the context switch is taken to the application level, we leverage the performance issue from the OS level to the application level. We further discuss this in Section 6.7.

2.5 Related Work

The concurrency model of Creol objects, used in this paper, is derived from the Actor model enriched by synchronization mechanisms and coupled with strong typing. The Actor model [3] is a suitable ground for multi-core and distributed programming, as objects (actors) are inherently concurrent and autonomous entities with a single thread of execution which makes them a natural fit for distributed deployment [69]. Two successful examples of actor-based languages are Erlang and Scala.

Scala is a hybrid object-oriented and functional programming language inspired by Java. The most important concept introduced in [51, 29] is that Scala Actors unify *thread-based* and *event-based* programming model to fill the gap for concurrency programming. Through the event-based model, Scala also provides the notion of

max	10000	20000	30000	50000	100000
Sequential	64k	72k	96k	160k	190k
$\mathcal{C}$risp	64k	64k	64k	64k	64k

Table 2.1: Thread stack allocated for different executions

continutations. Scala provides quite the same features of scheduling of tasks as in concurrent Java; i.e. it does not provide a direct and customizable platform to manage and schedule priorities on messages corresponded among actors.

Erlang [8] is a dynamically typed functional language that was developed at Ericsson Computer Science Laboratory with telecommunication purposes [30]. Recent developments in the deployment of Erlang support the assignment of a scheduler to each processor [82] (instead of one global scheduler for the entire application). This is a crucial improvement in Erlang, because the massive number of light-weight processes in the asynchronous setting of Erlang turns scheduling into a serious bottleneck. However, the scheduling policies are not yet controllable by the application.

There are well-known efforts in Java to bring in the functionality of asynchronous message passing onto multicore including Killim [105], Jetlang [100], ActorFoundry [69], and SALSA [111] among others. In [69], the authors present a comparative analysis of actor-based frameworks for JVM platform. However, pertaining to the domain of priority scheduling of asynchronous messages, all provide a predetermined approach or a limited control over how message priority scheduling may be at the hand of the programmer.

In general, existing high-level languages provide the programmer with little control over scheduling. The state of the art allows specifying priorities for threads or processes that are then used by the operating system to order them, e.g. Real-Time Specification for Java (RTSJ) and Erlang. In Crisp, we provide a fine-grain mechanism which allows for assigning priorities to high-level constructs, e.g., messages and methods.

Finally, we have considered, in previous work [14], local scheduling policies for Creol objects, with the purpose of schedulability analysis of real-time models. First of all, this paper is different as it investigates different levels of priorities that provide a high-level flexible mechanism to control scheduling. Secondly, we describe at present work how to compile Creol code to concurrent Java, and by allowing the use of class libraries in the underlying framework of Java, we can use Creol as a full-fledged programming language.

max	Live Peak	Total
10000	817	540591
20000	1468	1854067
30000	2204	4054814
50000	3707	11852985

Table 2.2: Number of live threads and total threads created for different runs of parallel prime sieve

2.6 Conclusion

In this paper, we proposed $\mathcal{C}$risp as an implementation scheme for application-level scheduling of active objects. $\mathcal{C}$risp first introduces asynchronous message passing with fine-grain priority management and scheduling of messages. Additionally, it introduces a Creol to Java compiler that translates the active objects in Creol into an equivalent Java application. $\mathcal{C}$risp compiler seamlessly integrates Java class libraries into Creol including data types that turns Creol from a modeling language to a fully-fledged one in the hands of the programmer.

The `java.util.concurrent` package provides useful API for concurrent programming. Java futures facilitate modeling asynchronous message passing. However, for processor release points, we had to preserve threads (using `InterruptibleProcess`) to allow continuations which leads to their OS-level context switching that is costly. Moreover, we were tightly directed to use the out-of-the-box `ExecutorService` which is limitedly extensible. We had no control over the scheduling mechanisms of the internal queue used in the service implementations. Thus, we needed to re-implement some of the concepts. Through prototyping $\mathcal{C}$risp, we learned that there are two major challenges ahead. Firstly, we need to integrate continuations into Java using a many-messages-to-one-thread mapping model. Secondly, we need complete control over scheduling of messages and threads in `ExecutorService`'s internal queue. Table 3.1 summarizes this discussion.

	Asynchronous Communication	Processor Release Point	Scheduling
Modeling	✓	✓	×
Performance	✓	×	✓
	Java Futures	Interruptible Process	Executor Service

Table 2.3: Overview of evaluation of challenges

In future, we will focus on thread performance for $\mathcal{C}$risp such that thread scalability can be achieved to a certain limit. Additionally, the development of concurrency features on multi-core in $\mathcal{C}$risp is one of the major future concentrations. Moreover, another line of future work involves profiling and monitoring objects at runtime to be used for optimization and performance improvement. In addition, we intend to extend and integrate into our tool set model checking engines such as Modere [59].

3 The Future of a Missed Deadline

Behrooz Nobakht, Frank S. de Boer, Mohammad Mahdi Jaghoori

Abstract

In this paper, we introduce a real-time actor-based programming language and provide a formal but intuitive operational semantics for it. The language supports a general mechanism for handling exceptions raised by missed deadlines and the specification of application-level scheduling policies. We discuss the implementation of the language and illustrate the use of its constructs with an industrial case study from distributed e-commerce and marketing domain.

3.1 Introduction

In real-time applications, rigid deadlines necessitate stringent scheduling strategies. Therefore, the developer must ideally be able to program the scheduling of different tasks inside the application. Real-Time Specification for Java (RTSJ) [61, 62] is a major extension of Java, as a mainstream programming language, aiming at enabling real-time application development. Although RTSJ extensively enriches Java with a framework for the specification of real-time applications, it yet remains at the level of conventional *multithreading*. The drawback of multithreading is that it involves the programmer with OS-related concepts like threads, whereas a real-time Java developer should only be concerned about high-level entities, i.e., objects and method invocations, also with respect to real-time requirements.

The actor model [46] and actor-based programming languages, which have re-emerged in the past few years [106, 8, 50, 63, 111], provide a different and promising paradigm for concurrency and distributed computing, in which threads are transparently encapsulated inside actors. As we will argue in this paper, this paradigm is much more suitable for real-time programming because it enables the programmer to obtain the appropriate high-level view which allows the management of complex real-time requirements.

In this paper, we introduce an actor-based programming language Crisp for real-time applications. Basic real-time requirements include deadlines and timeouts. In Crisp, deadlines are associated with asynchronous messages and timeouts with futures [13]. Crisp further supports a general actor-based mechanism for handling exceptions raised by missed deadlines. By the integration of these basic real-time control mechanisms with the application-level policies supported by Crisp for scheduling of the messages inside an actor, more complex real-time requirements of the application can be met with more flexibility and finer granularity.

We formalize the design of Crisp by means of structural operational semantics [98] and describe its implementation as a full-fledged programming language. This implementation uses both the Java and Scala language with extensions of Akka library. We illustrate the use of the programming language with an industrial case study from SDL Fredhopper that provides enterprise-scale distributed e-commerce solutions on the cloud.

The paper continues as follows: Section 3.2 introduces the language constructs and provides informal semantics of the language with a case study in Section 3.2.1. Section 3.3 presents the operational semantics of Crisp. Section 3.4 follows to provide a detailed discussion on the implementation. The case study continues in this section with further details and code examples. Section 3.5 discusses related work of research and finally Section 6.7 concludes the paper and proposes future line of research.

3.2 Programming with deadlines

In this section, we introduce the basic concepts underlying the notion of "deadlines" for asynchronous messages between actors. The main new constructs specify how a message can be sent with a deadline, how the message response can be processed, and what happens when a deadline is missed. We discuss the informal semantics of these concepts and illustrate them using a case study in Section 3.2.1.

Listing 3.1 introduces a minimal version of the real-time actor-based language Crisp. Below we discuss the two main new language constructs presented at lines (7) and (8).

How to send a message with a deadline? The construct

$$f = e_0 \,!\, m(\overline{e})\ \texttt{deadline}(e_1)$$

$$C ::= \textbf{class}\ \mathsf{N}\ \textbf{begin}\ V^{?}\ \{M\}^{*}\ \textbf{end} \tag{3.1}$$
$$M_{sig} ::= \mathsf{N}(\overline{T\ x}) \tag{3.2}$$
$$M ::= \{M_{sig} == \{V\ ;\}^{?} S\} \tag{3.3}$$
$$V ::= \textbf{var}\ \{\{x\},^{+}\ :\ \mathsf{T}\ \{= e\}^{?}\},^{+} \tag{3.4}$$
$$S ::= x := e\ | \tag{3.5}$$
$$::= x := \textbf{new}\ \mathsf{T}(e^{?})\ | \tag{3.6}$$
$$:= f = e\ !\ m(\overline{e})\ \textbf{deadline}(e)\ | \tag{3.7}$$
$$::= x := f.\textbf{get}(e^{?})\ | \tag{3.8}$$
$$::= \textbf{return}\ e\ | \tag{3.9}$$
$$::= S\ ;\ S\ | \tag{3.10}$$
$$::= \textbf{if}\ (b)\ \textbf{then}\ S\ \textbf{else}\ S\ \textbf{end}\ | \tag{3.11}$$
$$::= \textbf{while}\ (b)\ \{\ S\ \}\ | \tag{3.12}$$
$$::= \textbf{try}\ \{S\}\ \textbf{catch}(\mathsf{T}_{\texttt{Exception}}\ x)\ \{\ S\ \} \tag{3.13}$$

Figure 3.1: A kernel version of the real-time programming language. The bold scripted **keywords** denote the reserved words in the language. The over-lined $\overline{v}$ denotes a sequence of syntactic entities v. Both local and instance variables are denoted by x. We assume distinguished local variables `this`, `myfuture`, and `deadline` which denote the actor itself, the unique future corresponding to the process, and its deadline, respectively. A distinguished instance variable `time` denotes the current time. Any subscripted type $\mathsf{T}_{specialized}$ denotes a *specialized* type of general type T; e.g. $\mathsf{T}_{\texttt{Exception}}$ denotes all "exception" types. A variable f is in T_{future}. N is a name (identifier) used for classes and method names. C denotes a class definition which consists of a definition of its instance variables and its methods; M_{sig} is a method signature; M is a method definition; S denotes a statement. We abstract from the syntax the side-effect free expressions e and boolean expressions b.

describes an asynchronous message with a deadline specified by e_1 (of type T_{time}). Deadlines can be specified using a notion of time unit such as millisecond, second, minute or other units of time. The caller expects the callee (denoted by e_0) to process the message within the units of time specified by e_1. Here processing a message means starting the execution of the process generated by the message. A deadline is missed if and only if the callee does not start processing the message within the specified units of time.

What happens when a deadline is missed? Messages received by an actor generate processes. Each actor contains one active process and all its other processes are queued. Newly generated processes are *inserted* in the queue according to an application-specific policy. When a *queued* process misses its deadline it is *removed* from the queue and a corresponding *exception* is recorded by its future (as described below). When the currently active process is terminated the process at the head of the queue is activated (and as such dequeued). The active process cannot be *preempted*

and is forced to *run to completion*. In Section 3.4 we discuss the implementation details of this design choice.

How to process the response of a message with a deadline? In the above example of an asynchronous message, the *future* result of processing the message is denoted by the variable f which has the type of **Future**. Given a future variable f, the programmer can query the availability of the result by the construct

$$v = f.\mathbf{get}(e)$$

The execution of the **get** operation terminates successfully when the future variable f contains the result value. In case the future variable f records an exception, e.g. in case the corresponding process has missed its deadline, the **get** operation is *aborted* and the exception is *propagated*. Exceptions can be caught by `try-catch` blocks.

Listing 4: Using try-catch for processing future values

```
1 try {
2   x = f.get(e)
3   S_1
4 } catch(Exception x) {
5   S_2
6 }
```

For example, in Listing 4, if the **get** operation raises an exception control, is transferred to line (5); otherwise, the execution continues in line (3). In the `catch` block, the programmer has also access to the occurred exception that can be any kind of exception including an exception that is caused by a missed deadline. In general, any uncaught exception gives rise to abortion of the active process and is recorded by its future. Exceptions in our actor-based model thus are propagated by futures.

The additional parameter e of the get operation is of type T_{time} and specifies a *timeout*; i.e., the **get** operation will timeout after the specified units of time.

3.2.1 Case Study: Fredhopper Distributed Data Processing

Fredhopper is an SDL company since 2008 and a leading search, merchandising and personalization solution provider, whose products are uniquely tailored to the needs of online business. Fredhopper operates behind the scenes of more than 100 of the largest online sellers. The Fredhopper Access Server (FAS) provides access to high quality product catalogs. Typically deployments have about 10 explicit attribute values associated with a product over thousands of attribute dimensions. This challenging task involves working on difficult issues, such as the performance of information retrieval algorithms, the scalability of dealing with huge amounts of data and in satisfying large amounts of user requests per unit of time, the fault tolerance of complex

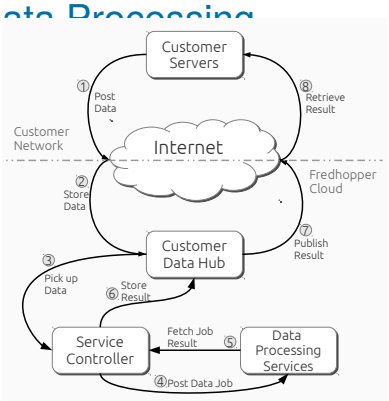


Figure 3.2: Fredhopper's Controller life cycle for remote data processing

distributed systems, and the executive monitoring and management of large-scale information retrieval operations. Fredhopper offers its services and facilities to e-Commerce companies (customers) as services (SaaS) over the cloud computing infrastructure (IaaS); which gives rise to different challenges in regards with resources management techniques and the customer cost model and service level agreements (SLA).

To orchestrate different services such as FAS or data processing, Fredhopper takes advantage of a service controller (a.k.a. Controller). Controller is responsible to passively manage different service installations for each customer. For instance, in one scenario, a customer submits their data along with a processing request to their data hub server. Controller, then picks up the data and initiates a data processing job (usually an ETL job) in a data processing service. When the data processing is complete, the result is again published to customer environment and additionally becomes available through FAS services. Figure 3.2 illustrates an example scenario that is described above.

In the current implementation of Controller, at Step 4, a data job instance is submitted to a remote data processing service. Afterwards, the future response of the data job is determined by a periodic remote check on the data service (Step 4). When the job is finished, Controller continues to retrieve the data job results (Step 5) and eventually publishes it to customer environment (Step 6).

In terms of system responsiveness, Step 4 may never complete. Step 4 failure can have different causes. For instance, at any moment of time, there are different customers' data jobs running on one data service node; i.e. there is a chance that a data service becomes overloaded with data jobs preventing the periodic data job check to return. If Step 4 fails, it leads the customer into an *unbounded waiting* situation. According to SLA agreements, this is *not* acceptable. It is strongly required that for any data job, the customer should be notified of the result: either a completed job with *success/failed* status, a job that is not completed, or a job with an unknown state. In other words, Controller should be able to guarantee that any data job request terminates.

To illustrate the contribution of this paper, we extract a closed-world simplified version of the scenario in Figure 3.2 from Controller. In Section 3.4, we provide an implementation-level usage of our work applied to this case study.

3.3 Operational Semantics

We describe the semantics of the language by means of a two-tiered labeled transition system: a local transition system describes the behavior of a single actor and a global transition system describes the overall behavior of a system of interacting actors. We define an actor state as a pair $\langle p, q \rangle$, where

- p denotes the current active process of the actor, and

- q denotes a queue of pending processes.

Each pending process is a pair (S, τ) consisting of the current executing statement S and the assignment τ of values to the *local* variables (e.g., formal parameters). The active process consists of a pair (S, σ), where σ assigns values to the local variables and additionally assigns values to the instance variables of the actor.

3.3.1 Local transition system

The local transition system defines transitions among actor configurations of the form $\langle p, q, \phi \rangle$, where (p, q) is an actor state and for any object o identifying a created future, ϕ denotes the shared heap of the created future objects, i.e., $\phi(o)$, for any future object o existing in ϕ, denotes a record with a field val which represents the return value and a boolean field aborted which indicates abortion of the process identified by o.

In the local transition system we make use of the following axiomatization of the occurrence of exceptions. Here $(S, \sigma, \phi) \uparrow v$ indicates that S raises an exception v:

- $(x = f.\mathbf{get}(), \sigma, \phi) \uparrow \sigma(f)$ where $\phi(\sigma(f))$.aborted $= \texttt{true}$,
- $\dfrac{(S, \sigma, \phi) \uparrow v}{\texttt{try}\{S\}\texttt{catch}(\texttt{T}\ u)\{S'\} \uparrow v}$ where v is not of type T, and,
- $\dfrac{(S, \sigma, \phi) \uparrow v}{(S; S, \sigma, \phi)' \uparrow v}$.

We present here the following transitions describing internal computation steps (we denote by $\mathsf{val}(e)(\sigma)$ the value of the expression e in σ and by $f[u \mapsto v]$ the result of assigning the value v to u in the function f).

Assignment statement is used to assign a value to a variable:

$$\langle (x = e; S, \sigma), q, \phi \rangle \rightarrow \langle (S, \sigma[x \mapsto \mathsf{val}(e)(\sigma)]), q, \phi \rangle$$

Returning a result consists of setting the field val of the future of the process:

$$\langle (\mathbf{return}\ e; S, \sigma), q, \phi \rangle \rightarrow \langle (S, \sigma), q, \phi[\sigma(\texttt{myfuture}).\mathsf{val} \mapsto \mathsf{val}(e)(\sigma)] \rangle$$

Initialization of `timeout` in `get` operation assigns to a distinguished (local) variable `timeout` its initial *absolute* value:

$$\langle (x = f.\mathbf{get}(e); S, \sigma), q, \phi \rangle \rightarrow \langle (x = f.\mathbf{get}(e); S, \sigma[\texttt{timeout} \mapsto \mathsf{val}(e + \mathsf{time})(\sigma), q, \phi \rangle$$

The `get` operation is used to assign the value of a future to a variable:

$$\langle (x = f.\mathbf{get}(); S, \sigma), q, \phi \rangle \to \langle (S, \sigma[x \mapsto \phi(\sigma(f)).\mathsf{val}]), q, \phi \rangle$$

where $\phi(\sigma(f)).\mathsf{val} \neq \bot$.

Timeout is operationally presented by the following transition:

$$\langle (x = f.\mathbf{get}(); S, \sigma), q, \phi \rangle \to \langle (S, \sigma), q, \phi \rangle$$

where $\sigma(\texttt{time}) < \sigma(\texttt{timeout})$.

The `try-catch` block semantics is presented by:

$$\frac{\langle (S, \sigma), q, \phi \rangle \to \langle (S', \sigma'), q', \phi' \rangle}{\langle (\texttt{try}\{S\}\texttt{catch}(\texttt{T}\ x)\{S''\}; S''', \sigma), q, \phi \rangle \to \langle (\texttt{try}\{S'\}\texttt{catch}(\texttt{T}\ x)\{S''\}; S''', \sigma), q', \phi' \rangle}$$

Exception handling. We provide the operational semantics of exception handling in a general way in the following:

$$\frac{(S, \sigma, \phi) \uparrow v}{\langle (\texttt{try}\{S\}\texttt{catch}(\texttt{T}\ x)\{S''\}; S''', \sigma), q, \phi \rangle \to \langle (S''; S''', \sigma[x \mapsto v]), q, \phi \rangle}$$

where the exception v is of type `T`.

Abnormal termination of the active process is generated by an uncaught exception:

$$\frac{(S, \sigma, \phi) \uparrow v}{\langle (S; S', \sigma), q, \phi \rangle \to \langle (S'', \sigma'), q', \phi' \rangle}$$

where $q = (S'', \tau) \cdot q'$ and σ' is obtained from restoring the values of the local variables as specified by τ (formally, $\sigma'(x) = \sigma(x)$, for every instance variable x, and $\sigma'(x) = \tau(x)$, for every local variable x), and $\phi'(\sigma(\texttt{myfuture})).\texttt{aborted} = \texttt{true}$ ($\phi'(o) = \phi(o)$, for every $o \neq \sigma(\texttt{myfuture})$).

Normal termination is presented by:

$$\langle (E, \sigma), q, \phi \rangle \to \langle (S, \sigma'), q', \phi \rangle$$

where $q = (S, \tau) \cdot q'$ and σ' is obtained from restoring the values of the local variables as specified by τ (see above). We denote by E termination (identifying $S; E$ with S).

Deadline missed. Let (S', τ) be some pending process in q such that $\tau(\texttt{deadline}) < \sigma(\texttt{time})$. Then

$$\langle (S, \sigma), q, \phi \rangle \rightarrow \langle p, q', \phi' \rangle$$

where q' results from q by removing (S', τ) and $\phi'(\tau(\texttt{myfuture})).\texttt{aborted} = \texttt{true}$ ($\phi'(o) = \phi(o)$, for every $o \neq \tau(\texttt{myfuture})$).

A message $m(\tau)$ specifies for the method m the initial assignment τ of its local variables (i.e., the formal parameters and the variables `this`, `myfuture`, and `deadline`). To model locally incoming and outgoing messages we introduce the following labeled transitions.

Incoming message. Let the active process p belong to the actor $\tau(\texttt{this})$ (i.e., $\sigma(\texttt{this}) = \tau(\texttt{this})$ for the assignment σ in p):

$$\langle p, q, \phi \rangle \xrightarrow{m(\tau)} \langle p, \mathsf{insert}(q, m(\overline{v}, d)), \phi \rangle$$

where $\mathsf{insert}(q, m(\tau))$ defines the result of inserting the process (S, τ), where S denotes the body of method m, in q, according to some application-specific policy (described below in Section 3.4).

Outgoing message. We model an outgoing message by:

$$\langle (f = e_0 \,!\, m(\bar{e})\ \textbf{deadline}(e_1); S, \sigma), q, \phi \rangle \xrightarrow{m(\tau)} \langle (S, \sigma[f \mapsto o]), q, \phi' \rangle$$

where

- ϕ' results from ϕ by extending its domain with a new future object o such that $\phi'(o).\mathsf{val} = \perp$[1] and $\phi'(o).\mathsf{aborted} = \texttt{false}$,
- $\tau(\texttt{this}) = \mathsf{val}(e_0)(\sigma)$,
- $\tau(x) = \mathsf{val}(e)(\sigma)$, for every formal parameter x and corresponding actual parameter e,
- $\tau(\texttt{deadline}) = \sigma(\mathsf{time}) + \mathsf{val}(e_1)(\sigma)$,
- $\tau(\texttt{myfuture}) = o$.

3.3.2 Global transition system

A (global) system configuration S is a pair (Σ, ϕ) consisting of a set Σ of actor states and a global heap ϕ which stores the created future objects. We denote actor states by s, s', s'', etc.

[1] $\perp$ stands for "uninitialized"

Local computation step. The interleaving of local computation steps of the individual actors is modeled by the rule:

$$\frac{(s, \phi) \rightarrow (s', \phi')}{(\{s\} \cup \Sigma, \phi) \rightarrow (\{s'\} \cup \Sigma, \phi')}$$

Communication. Matching a message sent by one actor with its reception by the specified callee is described by the rule:

$$\frac{(s_1, \phi) \xrightarrow{m(\tau)} (s_1', \phi') \quad (s_2, \phi) \xrightarrow{m(\tau)} (s_2', \phi)}{(\{s_1, s_2\} \cup \Sigma, \phi) \rightarrow (\{s_1', s_2'\} \cup \Sigma, \phi')}$$

Note that only an outgoing message affects the shared heap ϕ of futures.

Progress of Time. The following transition uniformly updates the local clocks (represented by the instance variable `time`) of the actors.

$$(\Sigma, \phi) \rightarrow (\Sigma', \phi)$$

where

$$\Sigma' = \{\langle (S, \sigma'), q, \phi \rangle \mid \langle (S, \sigma), q, \phi \rangle \in \Sigma,\ \sigma' = \sigma[\texttt{time} \mapsto \sigma(\texttt{time}) + \delta]\}$$

for some positive δ.

3.4 Implementation

We base our implementation on Java's concurrent package: `java.util.concurrent`. The implementation consists of the following major components:

1. An extensible language API that owns the core abstractions, architecture, and implementation. For instance, the programmer may extend the concept of a scheduler to take full control of how, i.e., in what order, the processes of the individual actors are queued (and as such scheduled for execution). We illustrate the scheduler extensibility with an example in the case study below.
2. Language Compiler that translates the modeling-level programs into Java source. We use ANTLR [97] parser generator framework to compile modeling-level programs to actual implementation-level source code of Java.
3. The language is seamlessly integrated with Java. At the time of programming, language abstractions such as data types and third-party libraries from either Crisp or Java are equally usable by the programmer.

We next discuss the underlying deployment of actors and the implementation of real-time processes with deadlines.

Deploying actors onto JVM threads. In the implementation, each actor owns a main thread of execution, that is, the implementation does *not* allocate *one* thread per process because threads are *costly* resources and allocating to each process one thread in general leads to a poor performance: there can be an arbitrary number of actors in the application and each may receive numerous messages which thus give rise to a number of threads that goes beyond the limits of memory and resources. Additionally, when processes go into pending mode, their correspondent thread may be reused for other processes. Thus, for better performance and optimization of resource utilization, the implementation assigns a single thread for all processes inside each actor.

Consequently, at any moment in time, there is only one process that is executed inside each actor. On the other hand, the actors share a thread which is used for the execution of a watchdog for the deadlines of the queued processes (described below) because allocation of such a thread to each actor in general slows down the performance. Further this sharing allows the implementation to decide, based on the underlying resources and hardware, to optimize the allocation of the watchdog thread to actors. For instance, as long as the resources on the underlying hardware are abundant, the implementation decides to share as less as possible the watchdog thread. This gives each actor a better opportunity with higher precision to detect missed deadlines.

Implementation of processes with deadlines. A process itself is represented in the implementation by a data structure which encapsulates the values of its local variables and the method to be executed. Given a relative deadline d as specified by a call we compute at run-time its absolute deadline (i.e. the expected starting time of the process) by

$$\texttt{TimeUnit.toMillis}(d) + \texttt{System.currentTimeMillis()}$$

which is a *soft* real-time requirement. As in the operational semantics, in the real-time implementation always the head of the process queue is scheduled for execution. This allows the implementation of a *default* earliest deadline first (EDF) scheduling policy by maintaining a queue ordered by the above absolute time values for the deadlines.

The important consequence of our non-preemptive mode of execution for the implementation is the resulting simplicity of thread management because preemption requires additional thread interrupts that facilitates the abortion of a process in the middle of execution. As stated above, a single thread in the implementation detects if a process has missed its deadline. This task runs periodically and to the end of all actors' life span. To check for a missed deadline it suffices to simply check for a process that the above absolute time value of its deadline is *smaller* than

`System.currentTimeMillis()`. When a process misses its deadline, the actions as specified by the corresponding transition of the operational semantics are subsequently performed. The language API provides extension points which allow for each actor the definition of a customized watchdog process and scheduling policy (i.e., policy for enqueuing processes). The customized watchdog processes are still executed by a single thread.

Fredhopper case study. As introduced in Section 3.2.1, we extract a closed-world simplified version from Fredhopper Controller. We apply the approach discussed in this paper to use deadlines for asynchronous messages.

Listing 5 and 6 present the difference in the previous Controller and the approach in Crisp. The left code snippet shows the Controller that uses polling to retrieve data processing results. The right code snippet shows the one that uses messages with deadlines.

Listing 5: With polling

```
1  class DataProcessor begin
2    op process(d: Data) ==
3      var p := allocDataProcessor(d)
4      p ! process (d)
5      do {
6        s := p ! getStatus (d)
7        if (s <> nil)
8          var r := p ! getResults(d)
9          publishResult(r)
10       wait(TimeUnit.toSecond(1))
11     } while (true)
12 end
```

Listing 6: With deadlines

```
1  class DataProcessor begin
2    op process(d: Data) ==
3      var p := allocDataProcessor(d)
4      var D := estimateDeadline(d)
5      var f :=
6        p ! process (d) deadline (D)
7      try {
8        publishResult(f.get())
9      } catch (Exception x) {
10       if (f.isAborted)
11         notifyFailure(d)
12     }
13 end
```

When the approach in Crisp in the right snippet is applied to Controller, it is guaranteed that all data job requests are terminated in a *finite* amount of time. Therefore, there cannot be complains about never receiving a response for a specific data job request. Many of Fredhopper's customers rely on data jobs to eventually deliver an e-commerce service to their end users. Thus, to provide a guarantee to them that their job result is always published to their environment is critical to them. As shown in the code snippet, if the data job request is failed or aborted based on a deadline miss, the customer is still eventually informed about the situation and may further decide about it. However, in the previous version, the customer may never be able to react to a data job request because its results are never published.

In comparison to the Controller using polling, there is a way to express timeouts for future values. However, it does not provide language constructs to specify a deadline for a message that is sent to data processing service. A deadline may be simulated using a combination of timeout and periodic polling approaches (Listing 5). Though, this approach cannot guarantee eventual termination in all cases; as discussed before that Step 4 in Figure 3.2 may never complete. Controller is required to meet certain customer expectations based on an SLA. Thus, Controller needs to

take advantage of a language/library solution that can provide a higher level of abstraction for real-time scheduling of concurrent messages. When messages in Crisp carry a deadline specification, Controller is able to guarantee that it can provide a response to the customer. This termination guarantee is crucial to the business of the customer.

Additionally, on the data processing service node, the new implementation takes advantage of the extensibility of schedulers in Crisp. As discussed above, the default scheduling policy used for each actor is EDF based on the deadlines carried by incoming messages to the actor. However, this behavior may be extended and replaced by a custom implementation from the programmer. In this case study, the priority of processes may differ if they the job request comes from specific customer; i.e. apart from deadlines, some customers have priority over others because they require a more real-time action on their job requests while others run a more relaxed business model. To model and implement this custom behavior, a custom scheduler is developed for the data processing node.

Listing 7: Data Processor class

```
1 class DataProcessor begin
2   var scheduler := new
        DataScheduler()
3   op process(d: Data) ==
4     // do process
5 end
```

Listing 8: Custom scheduler

```
1 class DataScheduler extends
      DefaultSchedulingManager {
2   boolean isPrior(Process p1,
        Process p2) {
3     if (p1.getCustomer().equals("A
          ")) {
4       return true;
5     }
6     return super.isPrior(p1, p2);
7   }
8 }
```

In the above listings, Listing 8 defines a custom scheduler that determines the priority of two processes with custom logic for specific customer. To use the custom scheduler, the only requirement is that the class `DataProcessor` defines a specific class variable called `scheduler` in Listing 7. The *custom scheduler* is picked up by Crisp core architecture and is used to schedule the queued processes. Thus, all processes from customer `A` have priority over processes from other customers no matter what their deadlines are.

We use Controller's logs for the period of February and March 2013 to examine the evaluation of Crisp approach. We define *customer satisfaction* as a property that represents the effectiveness of futures with deadline. For a customer c, the satisfaction can be denoted by $s = \frac{r_c^F}{r_c}$; in which r_c^F is the number of finished data processing jobs and r_c is the total number of requested data processing jobs from customer c. We extracted statistics for completed and never-ended data processing jobs from Controller logs (s_1).

s_1	s_2
88.71%	94.57%

Table 3.1: Evaluation Results

We replayed the logs with Crisp approach and measured the same property (s_2). We measured the same property for 180 customers that Fredhopper manages on the cloud. In this evaluation, a total number of about 25000 data processing requests

were included. The results show 6% improvement in Table 3.1 (that amounts to around 1600 better data processing requests). Because of data issues or wrong parameters in the data processing requests, there are requests that still fail or never end and should be handled by a human resource.

You may find more information including documentation and source code of Crisp at `http://nobeh.github.com/crisp`.

3.5 Related Work

The programming language presented in this paper is a real-time extension of the language introduced in [90]. This new extension features

- integration of asynchronous messages with deadlines and futures with timeouts;
- a general mechanism for handling exceptions raised by missed deadlines;
- high-level specification of application-level scheduling policies; and
- a formal operational semantics.

To the best of our knowledge the resulting language is the first implemented real-time actor-based programming language which formally integrates the above features.

In several works, e.g, [1] and [89], asynchronous messages in actor-based languages are extended with deadlines. However these languages do not feature futures with timeouts, a general mechanism for handling exceptions raised by missed deadlines or support the specification of application-level scheduling policies. Futures and fault handling are considered in the ABS language [66]. This work describes recovery mechanisms for failed get operations on a future. However, the language does not support the specification of real-time requirements, i.e., no deadlines for asynchronous messages are considered and no timeouts on futures. Further, when a get operation on a future fails, [66] does not provide any context or information about the exception or the cause for the failure. Alternatively, [66] describes a way to "compensate" for a failed get operation on future. In [11], a real-time extension of ABS with scheduling policies to model distributed systems is introduced. In contrast to Crisp, Real-Time ABS is an executable *modeling* language which supports the explicit specification of the progress of time by means of duration statements for the *analysis* of real-time requirements. The language does not support however asynchronous messages with deadlines and futures with timeouts.

Two successful examples of actor-based programming languages are Scala and Erlang. Scala [50, 29] is a hybrid object-oriented and functional programming language inspired by Java. Through the event-based model, Scala also provides the notion of continuations. Scala further provides mechanisms for scheduling of tasks similar to those provided by concurrent Java: it does not provide a direct and customizable platform to manage and schedule messages received by an individual actor. Additionally, Akka [108] extends Scala's actor programming model and

as such provides a direct integration with both Java and Scala. Erlang [8] is a dynamically typed functional language that was developed at Ericsson Computer Science Laboratory with telecommunication purposes [30]. Recent developments in the deployment of Erlang support the assignment of a scheduler to each processor [82] (instead of one global scheduler for the entire application) but it does not, for example, support application-level scheduling policies. In general, none these languages provide a formally defined real-time extension which integrates the above features.

There are well-known efforts in Java to bring in the functionality of asynchronous message passing onto multicore including Killim [106], Jetlang [100], ActorFoundry [69], and SALSA [111]. In [69], the authors present a comparative analysis of actor-based frameworks for JVM platform. Most of these frameworks support futures with timeouts but do not provide asynchronous messages with deadlines, or a general mechanism for handling exceptions raised by missed deadlines. Further, pertaining to the domain of priority scheduling of asynchronous messages, these efforts in general provide a predetermined approach or a limited control over message priority scheduling. As another example, in [84] the use of Java Fork/Join is described to optimize mulicore applications. This work is also based on a *fixed priority* model. Additionally, from embedded hardware-software research domain, Ptolemy [32, 76] is an actor-oriented open architecture and platform that is used to design, model and simulate embedded software. Their approach is hardware software co-design. It provides a platform framework along with a set of tools.

In general, existing high-level programming languages provide the programmer with little real-time control over scheduling. The state of the art allows specifying priorities for threads or processes that are used by the operating system, e.g., Real-Time Specification for Java (RTSJ [61, 62]) and Erlang. Specifically in RTSJ, [117] extensively introduces and discusses a framework for application-level scheduling in RTSJ. It presents a flexible framework to allow scheduling policies to be used in RTSJ. However, [117] addresses the problem mainly in the context of the standard multithreading approach to concurrency which in general does not provide the most suitable approach to distributed applications. In contrast, in this paper we have shown that an actor-based programming language provides a suitable formal basis for a fully integrated real-time control in distributed applications.

3.6 Conclusion and future work

In this paper, we presented both a formal semantics and an implementation of a real-time actor-based programming language. We presented how asynchronous messages with deadline can be used to control application-level scheduling with higher abstractions. We illustrated the language usage with a real-world case study from SDL Fredhopper along the discussion for the implementation. Currently we are investigating further optimization of the implementation of Crisp and the formal

verification of real-time properties of Crisp applications using schedulability analysis [35].

Part III

Implementation

4 Programming with actors in Java 8[1]

Behrooz Nobakht, Frank S. de Boer

Abstract

There exist numerous languages and frameworks that support an implementation of a variety of actor-based programming models in Java using concurrency utilities and threads. Java 8 is released with fundamental new features: lambda expressions and further dynamic invocation support. We show in this paper that such features in Java 8 allow for a high-level actor-based methodology for programming distributed systems which supports the programming to interfaces discipline. The embedding of our actor-based Java API is shallow in the sense that it abstracts from the actual thread-based deployment models. We further discuss different concurrent execution and thread-based deployment models and an extension of the API for its actual parallel and distributed implementation. We present briefly the results of a set of experiments which provide evidence of the potential impact of lambda expressions in Java 8 regarding the adoption of the actor concurrency model in large-scale distributed applications.

4.1 Introduction

Java is beyond doubt one of the mainstream object oriented programming languages that supports a *programming to interfaces* discipline [38, 113]. Through the years, Java has evolved from a mere programming language to a huge platform to drive and envision standards for mission-critical business applications. Moreover, the Java language itself has evolved in these years to support its community with new language features and standards. One of the noticeable domains of focus in the past decade has been distribution and concurrency in research and application. This has led to valuable research results and numerous libraries and frameworks with an attempt to provide distribution and concurrency at the level of Java language. However, it is widely recognized that the thread-based model of concurrency in Java that is a well-known approach is not appropriate for realizing distributed systems because of its inherent synchronous communication model. On the other hand, the event-driven actor model of concurrency introduced by Hewitt [53] is a powerful concept for modeling distributed and concurrent systems [**Agha97**, 4]. Different extensions of actors are proposed in several domains and are claimed to

[1]This paper is funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services, `http://www.envisage-project.eu`.

be the most suitable model of computation for many applications [54]. Examples of these domains include designing embedded systems [77, 78], wireless sensor networks [22], multi-core programming [70] and delivering cloud services through SaaS or PaaS [20]. This model of concurrent computation forms the basis of the programming languages Erlang [7] and Scala [49] that have recently gained in popularity, in part due to their support for scalable concurrency. Moreover, based on the Java language itself, there are numerous libraries that provide an implementation of an actor-based programming model.

The main problem addressed in this paper is that in general existing actor-based programming techniques are based on an explicit encoding of mechanisms at the application level for message passing and handling, and as such overwrite the general object-oriented approach of method look-ups that forms the basis of programming to interfaces and the design-by-contract discipline [87]. The entanglement of event-driven (or asynchronous messaging) and object-oriented method look-up makes actor-based programs developed using such techniques extremely difficult to reason about and formalize. This clearly hampers the promotion of actor-based programming in mainstream industry that heavily practices object-oriented software engineering.

The main result of this paper is a Java 8 API for programming distributed systems using asynchronous message passing and a corresponding actor programming methodology which abstracts invocation from execution (e.g. thread-based deployment) and fully supports programming to interfaces discipline. We discuss the API architecture, its properties, and different concurrent execution models for the actual implementation.

Our main approach consists of the explicit description of an actor in terms of its *interface*, the use of the recently introduced lambda expressions in Java 8 in the implementation of asynchronous message passing, and the formalization of a corresponding high-level actor programming methodology in terms of an executable modeling language which lends itself to formal analysis, ABS [65].

The paper continues as follows: in Section 6.2, we briefly discuss a set of related works on actors and concurrent models especially on JVM platform. Section 4.3 presents an example that we use throughout the paper, we start to model the example using a library. Section 4.4 briefly introduces a concurrent modeling language and implements the example. Section 4.5 briefly discusses Java 8 features that this works uses for implementation. Section 4.6 presents how an actor model maps into programming in Java 8. Section 4.7 discusses in detail the implementation architecture of the actor API. Section 4.8 discusses how a number of benchmarks were performed for the implementation of the API and how they compare with current related works. Section 6.7 concludes the paper and discusses the future work.

4.2 Related Work

There are numerous works of research and development in the domain of actor modeling and implementation in different languages. We discuss a subset of the related work in the level of modeling and implementation with more focus on Java and JVM-based efforts in this section.

Erlang [7] is a programming language used to build massively scalable soft real-time systems with requirements on high availability. Some of its uses are in telecoms, banking, e-commerce, computer telephony and instant messaging. Erlang's runtime system has built-in support for concurrency, distribution and fault tolerance. While threads require external library support in most languages, Erlang provides language-level features for creating and managing processes with the aim of simplifying concurrent programming. Though all concurrency is explicit in Erlang, processes communicate using message passing instead of shared variables, which removes the need for locks. Elixir [109] is a functional meta-programming aware language built on top of the Erlang VM. It is a dynamic language with flexible syntax with macros support that leverages Erlang's abilities to build concurrent, distributed, fault-tolerant applications with hot code upgrades.

Scala is a hybrid object-oriented and functional programming language inspired by Java. The most important concept introduced in [49] is that Scala actors unify *thread-based* and *event-based* programming model to fill the gap for concurrency programming. Through the event-based model, Scala also provides the notion of continuations. Scala provides quite the same features of scheduling of tasks as in concurrent Java; i.e., it does not provide a direct and customizable platform to manage and schedule priorities on messages sent to other actors. Akka [48] is a toolkit and runtime for building highly concurrent, distributed, and fault tolerant event-driven applications on the JVM based on actor model.

Kilim [104] is a message-passing framework for Java that provides ultra-lightweight threads and facilities for fast, safe, zero-copy messaging between these threads. It consists of a bytecode postprocessor (a "weaver"), a run time library with buffered mailboxes (multi-producer, single consumer queues) and a user-level scheduler and a type system that puts certain constraints on pointer aliasing within messages to ensure interference-freedom between threads. The SALSA [112, 70] programming language (Simple Actor Language System and Architecture) is an active object-oriented programming language that uses concurrency primitives beyond asynchronous message passing, including token-passing, join, and first-class continuations.

RxJava [23] by Netflix is an implementation of reactive extensions [88] from Microsoft. Reactive extensions try to provide a solution for composing asynchronous and event-based software using observable pattern and scheduling. An interesting direction of this library is that it uses reactive programming to avoid a phenomenon known as "callback hell"; a situation that is a natural consequence of composing

`Future` abstractions in Java specifically when they wait for one another. However, RxJava advocates the use of asynchronous functions that are triggered in response to the other functions. In the same direction, LMAX Disruptor [9, 37] is a highly concurrent event processing framework that takes the approach of event-driven programming towards provision of concurrency and asynchronous event handling. The system is built on the JVM platform and centers on a Business Logic Processor that can handle 6 million events per second on a single thread. The Business Logic Processor runs entirely in-memory using event sourcing. The Business Logic Processor is surrounded by Disruptors - a concurrency component that implements a network of queues that operate without needing locks.

4.3 State of the Art: An example

In the following, we illustrate the state of the art in actor programming by means of a simple example using the Akka [107] library which features asynchronous messaging and which is used to program actors in both Scala and Java. We want to model in Akka an "asynchronous ping-pong match" between two actors represented by the two interfaces `IPing` and `IPong` which are depicted in Listings 9 and 10. An asynchronous call by the actor implementing the `IPong` interface of the `ping` method of the actor implementing the `IPing` interface should generate an asynchronous call of the `pong` method of the callee, and vice versa. We intentionally design `ping` and `pong` methods to take arguments in order to demonstrate how method arguments may affect the use of an actor model in an object-oriented style.

Listing 9: Ping as an interface

```
1 public interface IPing {
2   void ping(String msg);
3 }
```

Listing 10: Pong as an interface

```
1 public interface IPong {
2   void pong(String msg);
3 }
```

To model an actor in Akka by a class, say `Ping`, with interface `IPing`, this class is required *both* to *extend* a given pre-defined class `UntypedActor` and *implement* the interface `IPing`, as depicted in Listings 11 and 12. The class `UntypedActor` provides two Akka framework methods `tell` and `onReceive` which are used to enqueue and dequeue asynchronous messages. An asynchronous call to, for example, the method `ping` then can be modeled by passing a user-defined encoding of this call, in this case by prefixing the string argument with the string "pinged", to a (synchronous) call of the `tell` method which results in enqueuing the message. In case this message is dequeued the implementation of the `onReceive` method as provided by the `Ping` class then calls the `ping` method.

Listing 11: Ping actor in Akka

```
1 public class Ping(ActorRef pong)
2   extends UntypedActor
3   implements IPing {
4
5   public void ping(String msg) {
6     pong.tell("ponged," + msg)
7   }
8
9   public void onReceive(Object m)
        {
10    if (!(m instanceof String)) {
11      // Message not understood.
12    } else
13    if (((String) m).startsWith("
          pinged") {
14      // Explicit cast needed.
15      ping((String) m);
16    }
17   }
18 }
```

Listing 12: Pong class in Akka

```
1 public class Pong
2   extends UntypedActor
3   implements IPong {
4
5   public void pong(String msg) {
6     sender().tell(
7       "pinged," + msg);
8   }
9
10  public void onReceive(Object m)
        {
11    if (!(m instanceof String)) {
12      // Message not understood.
13    } else
14    if (m.startsWith("ponged") {
15      // Explicit cast needed.
16      ping((String) m);
17    }
18   }
19 }
```

Access to the sender of the message in Akka is provided by `sender()`. In the main method as described in Listing 13 we show how the initialize and start the ping/pong match. Note that a reference to the "pong" actor is passed to the "ping" actor.

Listing 13: main in Akka

```
1 ActorSystem s = ActorSystem.create
      ();
2 ActorRef pong = s.actorOf(Props.
      create(Pong.class));
3 ActorRef ping = s.actorOf(Props.
      create(Ping.class, pong));
4 ping.tell(""); // To get a Future
```

Further, both the `onReceive` methods are invoked by Akka `ActorSystem` itself. In general, Akka actors are of type `ActorRef` which is an abstraction provided by Akka to allow actors send asynchronous messages to one another. An immediate consequence of the above use of inheritance is that the class `Ping` is now exposing a public behavior that is *not* specified by its *interface*. Furthermore, a "ping" object refers to a "pong" object by the type `ActorRef` . This means that the interface `IPong` is not directly visible to the "ping" actor. Additionally, the implementation details of receiving a message should be "hand coded" by the programmer into the special method `onReceive` to define the responses to the received messages. In our case, this implementation consists of a decoding of the message (using type-checking) in order to *look up* the method that subsequently should be invoked. This fundamentally interferes with the general object-oriented mechanism for method look-up which forms the basis of the programming to interfaces discipline. In the next section, we continue the same example and discuss an actor API for directly calling asynchronously methods using the general object-oriented mechanism for method look-up. Akka has recently released a new version that supports Java 8 features [2]. However, the new features can be categorized as

[2]Documentation available at `http://doc.akka.io/docs/akka/2.3.2/java/lambda-index-actors.html`

syntax sugar on how incoming messages are filtered through object/class matchers to find the proper type.

4.4 Actor Programming in Java

We first describe informally the actor programming model assumed in this paper. This model is based on the Abstract Behavioral Specification language (ABS) introduced in [65]. ABS uses asynchronous method calls, futures, interfaces for encapsulation, and cooperative scheduling of method invocations inside concurrent (active) objects. This feature combination results in a concurrent object-oriented model which is inherently compositional. More specifically, actors in ABS have an identity and behave as active objects with encapsulated data and methods which represent their state and behavior, respectively. Actors are the units of concurrency: conceptually an actor has a dedicated processor. Actors can only send asynchronous messages and have queues for receiving messages. An actor progresses by taking a message out of its queue and processing it by executing its corresponding method. A method is a piece of sequential code that may send messages. Asynchronous method calls use futures as dynamically generated references to return values. The execution of a method can be (temporarily) suspended by release statements which give rise to a form of cooperative scheduling of method invocations inside concurrent (active) objects. Release statements can be conditional (e.g., checking a future for the return value). Listings 15, 16 and 14 present an implementation of ping-pong example in ABS. By means of the statement on line 6 of Listing 15 a "ping" object directly calls asynchronously the `pong` method of its "pong" object, and vice versa. Such a call is stored in the message queue and the called method is executed when the message is dequeued. Note that variables in ABS are declared by interfaces. In ABS, `Unit` is similar to `void` in Java.

Listing 14: main in ABS

```
1 ABSIPong pong;
2 pong = new ABSPong;
3 ping = new ABSPing(pong);
4 ping ! ping("");
```

Listing 15: Ping in ABS

```
1 interface ABSIPing {
2   Unit ping(String msg);
3 }
4 class ABSPing(ABSIPong pong)
      implements ABSIPing {
5   Unit ping(String msg) {
6   pong ! pong("ponged," + msg);
7   }
8 }
```

Listing 16: Pong in ABS

```
1 interface ABSIPong {
2   Unit pong(String msg);
3 }
4 class ABSPong implements ABSIPong
      {
5   Unit pong(String msg) {
6     sender ! ping( "pinged," + msg
          );
7   }
8 }
```

4.5 Java 8 Features

In the next section, we describe how ABS actors are implemented in Java 8 as API. In this section we provide an overview of the features in Java 8 that facilitate an efficient, expressive, and precise implementation of an actor model in ABS.

Java Defender Methods Java *defender* methods (JSR 335 [42]) use the new keyword **`default`**. Defender methods are declared for **`interface`**s in Java. In contrast to the other methods of an interface, a default method is not an abstract method but must have an implementation. From the perspective of a client of the interface, defender methods are no different from ordinary interface methods. From the perspective of a hierarchy descendant, an implementing class can optionally *override* a default method and change the behavior. It is left as a decision to any class implementing the interface whether or not to override the default implementation. For instance, in Java 8 `java.util.Comparator` provides a default method `reversed()` that creates a reversed-order comparator of the original one. Such default method eliminates the need for any implementing class to provide such behavior by inheritance.

Java Functional Interfaces Functional interfaces and lambda expressions (JSR 335 [42]) are fundamental changes in Java 8. A **`@FunctionalInterface`** is an annotation that can be used for interfaces in Java. Conceptually, any class or interface is a functional interface if it consists of exactly one *abstract* method. A lambda expression in Java 8, is a runtime translation [44] of any type that is replaceable by a functional interface. Many of Java's classic interfaces are functional interfaces from the perspective of Java 8 and can be turned into lambda expressions; e.g. `java.lang.Runnable` or `java.util.Comparator`. For instance,

```
(s1, s2) → return s1.compareTo(s2);
```

is a lambda expression that can be statically cast to an instance of a `Comparator<String>`; because it can be replaced with a functional interface that has a method with two strings and returning one integer. Lambda expressions in Java 8 *do not* have an intrinsic type. Their type is bound to the context that they are used in but their type is always a functional interface. For instance, the above definition of a lambda expression can be used as:

```
Comparator<String> cmp1 = (s1, s2) → return s1.compareTo(s2);
```

in one context while in the other:

```
Function<String> cmp2 = (s1, s2) → return s1.compareTo(s2);
```

given that `Function<T>` is defined as:

```
interface Function<T> { int apply(T t1, T t2); }
```

In the above examples, the same lambda expression is statically cast to a different matching functional interface based on the context. This is a fundamental new feature in Java 8 that facilitates application of functional programming paradigm in an object-oriented language.

This work of research extensively uses this feature of Java 8. Java 8 marks many of its own APIs as functional interfaces most important of which in this context are `java.lang.Runnable` and `java.util.concurrent.Callable`. This means that a lambda expression can replace an instance of `Runnable` or `Callable` at runtime by JVM. We will discuss later how we utilize this feature to allow us model an asynchronous message into an instance of a `Runnable` or `Callable` as a form of a lambda expression. A lambda expression equivalent of a `Runnable` or a `Callable` can be treated as a queued message of an actor and executed.

Java Dynamic Invocation Dynamic invocation and execution with method handles (JSR 292 [101]) enables JVM to support efficient and flexible execution of method invocations in the absence of static type information. JSR 292 introduces a new byte code instruction `invokedynamic` for JVM that is available as an API through `java.lang.invoke.MethodHandles`. This API allows translation of lambda expression in Java 8 at runtime to be executed by JVM. In Java 8, use of lambda expression are favored over anonymous inner classes mainly because of their performance issues [43]. The abstractions introduced in JSR 292 perform better that Java Reflection API using the new byte code instruction. Thus, lambda expressions are compiled and translated into method handle invocations rather reflective code or anonymous inner classes. This feature of Java 8 is indirectly use in ABS API through the extensive use of lambda expressions. Moreover, in terms of performance, it has been revealed that invoke dynamic is much better than using anonymous inner classes [43].

4.6 Modeling actors in Java 8

In this section, we discuss how we model ABS actors using Java 8 features. In this mapping, we demonstrate how new features of Java 8 are used.

The `Actor` Interface We introduce an interface to model actors using Java 8 features discussed in Section 4.5. Implementing an interface in Java means that the object exposes public APIs specified by the interface that is considered the behavior of the object. Interface implementation is opposed to inheritance extension in which the object is possibly forced to expose behavior that may not be part of its intended interface. Using an interface for an actor allows an object to preserve its own interfaces, and second, it allows for multiple interfaces to be implemented and composed.

A Java API for the implementation of ABS models should have the following main three features. First, an object should be able to send asynchronously an arbitrary message in terms of a method invocation to a receiver actor object. Second, sending a message can optionally generate a so-called future which is used to refer to the return value. Third, an object during the processing of a message should be able to access the "sender" of a message such that it can reply to the message by another message. All the above must co-exist with the fundamental requirement that for an object to act like an actor (in an object-oriented context) should *not* require a modification of its intended interface.

The `Actor` interface (Listings 17 and 18) provides a set of **`default`** methods, namely the `run` and `send` methods, which the implementing classes do not need to re-implement. This interface further encapsulates a queue of messages that supports concurrent features of Java API [3]. We distinguish two types of messages: messages that are not expected to generate any result and messages that are expected to generate a result captured by a future value; i.e. an instance of `Future` in Java 8. The first kind of messages are modeled as instances of `Runnable` and the second kind are modeled instances of `Callable`. The default `run` method then takes a message from the queue, checks its type and executes the message correspondingly. On the other hand, the default (overloaded) `send` method stores the sent message and creates a future which is returned to the caller, in case of an instance of `Callable`.

Listing 17: Actor interface (1)

```
1 public interface Actor {
2   public void run() {
3     Object m = queue.take();
4
5     if (m instanceof Runnable) {
6       ((Runnable) m).run();
7     } else
8
9     if (m instanceof Callable) {
10      ((Callable) m).call();
11    }
12  }
13
14  // continue to the right
```

Listing 18: Actor interface (2)

```
1
2   public void send(Runnable m) {
3     queue.offer(m);
4   }
5
6   public <T> Future<T>
7     send(Callable<T> m) {
8       Future<T> f =
9         new FutureTask(m);
10      queue.offer(f);
11      return f;
12  }
13 }
```

Modeling Asynchronous Messages We model an asynchronous call

$$\texttt{Future<V>}\ f = e_0\ !\ m(e_1, \ldots, e_n)$$

to a method in ABS by the Java 8 code snippet of Listing 19. The final local variables $\texttt{u}_1, \ldots, \texttt{u}_n$ (of the caller) are used to store the values of the Java 8 expressions $\texttt{e}_1, \ldots,$

[3] Such API includes usage of different interfaces and classes in `java.util.concurrent` package [68]. The concurrent Java API supports blocking and synchronization features in a high-level that is abstracted from the user.

e_n corresponding to the actual parameters $e_1, \ldots, e_n$. The types T_i, $i = 1, \ldots, n$, are the corresponding Java 8 types of e_i, $i = 1, \ldots, n$.

Listing 19: Async messages with futures

```
1 final T1 u1 = e1;
2 . . .
3 final Tn un = en;
4 Future<V> v = e0.send(
5   () → { return m(u1,...,un); }
6 );
```

Listing 20: Async messages w/o futures

```
1 final T1 u1 = e1;
2 . . .
3 final Tn un = en;
4 e0.send(
5   { () → m(u1,...,un); }
6 );
```

The lambda expression which encloses the above method invocation is an instance of the functional interface; e.g. `Callable`. Note that the generated object which represents the lambda expression will contain the local context of the caller of the method "m" (including the local variables storing the values of the expressions $e_1, \ldots, e_n$), which will be restored upon execution of the lambda expression. Listing 20 models an asynchronous call to a method without a return value.

As an example, Listings 21 and 22 present the running ping/pong example, using the above API. The main program to use ping and pong implementation is presented in Listing 23.

Listing 21: Ping as an Actor

```
1 public class Ping(IPong pong)
      implements IPing, Actor {
2  public void ping(String msg) {
3   pong.send( () → { pong.("ponged
        ," + msg) } );
4   }
5 }
```

Listing 22: Pong as an Actor

```
1 public class Pong implements IPong
      , Actor {
2   public void pong(String msg) {
3     sender().send(( ) → { ping.("
          pinged," + msg) } );
4   }
5 }
```

As demonstrated in the above examples, the "ping" and "pong" objects *preserve* their own *interfaces* contrary to the example depicted in Section 4.3 in which the objects *extend* a specific "universal actor abstraction" to inherit methods and behaviors to become an actor. Further, messages are processed *generically* by the `run` method described in Listing 17. Although, in the first place, sending an asynchronous may look like to be able to change the recipient actor's state, this is not correct. The variables that can be used in a lambda expression are *effectively* final. In other words, in the context of a lambda expression, the recipient actor only provides a snapshot view of its state that cannot be changed. This prevents abuse of lambda expressions to change the receiver's state.

Modeling Cooperative Scheduling The ABS statement **`await`** g, where g is a boolean guard, allows an active object to preempt the current method and schedule another one. We model cooperative scheduling by means of a call to the `await` method in

Listing 24. Note that the preempted process is thus passed as an additional parameter and as such queued in case the guard is false, otherwise it is executed. Moreover, the generation of the continuation of the process is an optimization task for the code generation process to prevent code duplication.

Listing 23: main in ABS API

```
1 IPong pong = new Pong();
2 IPing ping = new Ping(pong);
3 ping.send(
4   () -> ping.ping("")
5 );
```

Listing 24: Java 8 await implementation

```
1 void await(final Boolean guard,
2            final Runnable cont) {
3   if (!guard) {
4     this.send(() →
5       { this.await(guard, cont) })
6   } else { cont.run() }
7 }
```

4.7 Implementation Architecture

Figure 4.1 presents the general layered architecture of the actor API in Java 8. It consists of three layers: the routing layer which forms the foundation for the support of distribution and location transparency [70] of actors, the queuing layer which allows for different implementations of the message queues, and finally, the processing layer which implements the actual execution of the messages. Each layer allows for further customization by means of plugins. The implementation is available at `https://github.com/CrispOSS/abs-api`.

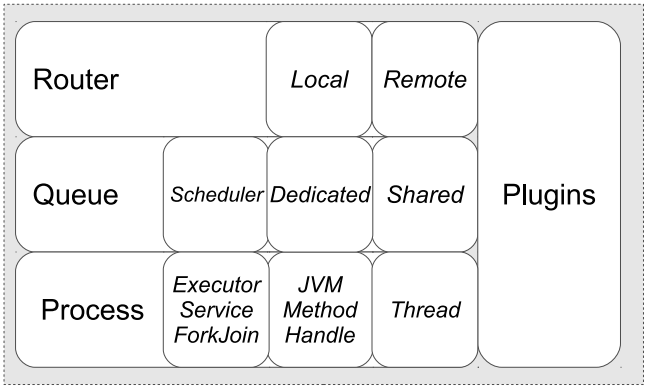


Figure 4.1: Architecture of Actor API in Java 8

We discuss the architecture from bottom layer to top. The implementation of actor API preserves a faithful mapping of message processing in ABS modeling language. An actor is an active object in the sense that it controls how the next message is executed and may release any resources to allow for co-operative scheduling. Thus, the implementation is required to optimally utilize JVM threads. Clearly, allocating a dedicated thread to each message or actor is not scalable. Therefore, actors need to *share* threads for message execution and yet be in full control of resources when required. The implementation fundamentally separates *invocation* from *execution*.

An asynchronous message is a reference to a method invocation until it starts its execution. This allows to minimize the allocation of threads to the messages and facilitates sharing threads for executing messages. Java concurrent API [68] provides different ways to deploy this separation of invocation from execution. We take advantage of Java Method Handles [101] to encapsulate invocations. Further we utilize different forms of `ExecutorService` and `ForkJoinPool` to deploy concurrent invocations of messages in different actors.

In the next layer, the actor API allows for different implementations of a queue for an actor. A dedicated queue for each actor simplifies the process of queuing messages for execution but consumes more resources. However, a shared queue for a set of actors allows for memory and storage optimization. This latter approach of deployment, first, provides a way to utilize the computing power of multi-core; for instance, it allows to use work-stealing to maximize the usage of thread pools. Second, it enables application-level scheduling of messages. The different implementations cater for a variety of plugins, like one that releases computation as long as there is no item in the queue and becomes active as soon as an item is placed into the queue; e.g. `java.util.concurrent.BlockingQueue`. Further, different plugins can be injected to allow for scheduling of messages extended with deadlines and priorities [91].

We discuss next the distribution of actors in this architecture. In the architecture presented in Figure 4.1, each layer can be *distributed* independently of another layer in a transparent way. Not only the routing layer can provide distribution, the queue layer of the architecture may also be remote to take advantage of cluster storage for actor messages. A remote routing layer can provide access to actors transparently through standard naming or addresses. We exploit the main properties of actor model [**Agha97**, 4] to distribute actors based on our implementation. From a distributed perspective, the following are the main requirements for distributing actors:

Reference Location Transparency Actors communicate to one another using references. In an actor model, there is no in-memory object reference; however, every actor reference denotes a location by means of which the actor is accessible. The reference location may be local to the calling actor or remote. The reference location is *physically* transparent for the calling actor.

Communication Transparency A message m from actor A to actor B may possibly lead to transferring m over a network such that B can process the message. Thus, an actor model that supports distribution must provide a layer of remote communication among its actors that is transparent, i.e., when actor A sends message m, the message is transparently transferred over the network to reach actor B. For instance, actors existing in an HTTP container that transparently allows such communication. Further, the API implementation is required to provide a mechanism for serialization of messages. By default, every object in JVM cannot be assumed to be an instance of `java.io.Serializable`. However, the

API may enforce that any remote actor should have the required actor classes in its JVM during runtime which allows the use of the JVM's general object serialization [4] to send messages to remote actors and receive their responses. Additionally, we model asynchronous messages with lambda expressions for which Java 8 supports serialization by specification [5].

Actor Provisioning During a life time of an actor, it may need to create new actors. Creating actors in a local memory setting is straightforward. However, the local setting *does* have a capacity of number of actors it can hold. When an actor creates a new one, the new actor may actually be initialized in a remote resource. When the resource is not available, it should be first provisioned. However, this resource provisioning should be transparent to the actor and only the eventual result (the newly created actor) is visible.

We extend the ABS API to ABS Remote API[6] that provides the above properties for actors in a seamless way. A complete example of using the remote API has been developed[7]. Expanding our ping-pong example in this paper, Listing 25 and 26 present how a remote server of actors is created for the ping and pong actors. In the following listings, `java.util.Properties` is used provide input parameters of the actor server; namely, the address and the port that the actor server responds to.

Listing 25: Remote ping actor main

```
1 Properties p =  new Properties();
2 p.put("host", "localhost");
3 p.put("port", "7777");
4 ActorServer s = new ActorServer(p)
      ;
5 IPong pong =
6  s.newRemote("abs://pong@http://
       localhost:8888",
7   IPong.class);
8 Ping ping = new Ping(pong);
9 ping.send(
10   () -> ping.ping("")
11 );
```

Listing 26: Remote pong actor main

```
1 Properties p = new Properties();
2 p.put("host", "localhost");
3 p.put("port", "8888");
4 ActorServer s = new ActorServer(p)
      ;
5 Pong pong = new Pong();
```

In Listing 25, a remote reference to a pong actor is created that exposes the `IPong` interface. This interface is proxied [8] by the implementation to handle the remote communication with the actual pong actor in the other actor server. This mechanism hides the communication details from the ping actor and as such allows the ping

[4] Java Object Serialization Specification: `http://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html`

[5] Serialized Lambdas: `http://docs.oracle.com/javase/8/docs/api/java/lang/invoke/SerializedLambda.html`

[6] The implementation is available at `https://github.com/CrispOSS/abs-api-remote`.

[7] An example of ABS Remote API is available at `https://github.com/CrispOSS/abs-api-remote-sample`.

[8] Java Proxy: `http://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html`

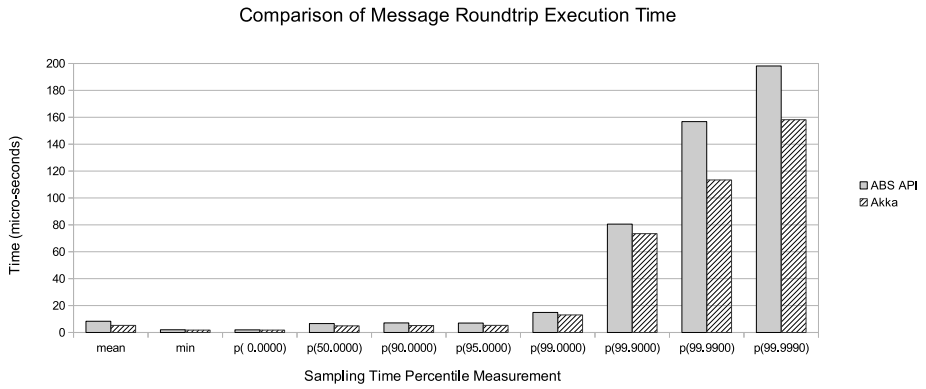


Figure 4.2: Benchmark results of comparing sampling time of message round trips in ABS API and Akka. An example reading of above results is that the time shows for $p(90.0000)$ reads as "message round trips were completed under $10\mu s$ for 90% of the sent messages". The first two columns show the "minimum" and "mean" message round trip times in both implementations.

actor to use the same API to send a message to the pong actor (without even knowing that the pong actor is actually remote). When an actor is initialized in a distributed setting it transparently identifies its actor server and registers with it. The above two listings are aligned with the similar main program presented in Listing 23 that presents the same in a local setting. The above two listings run in separate JVM instances and therefore do not share any objects. In each JVM instance, it is required that both interfaces `IPing` and `IPong` are visible to the classpath; however, the ping actor server only needs to see `Ping` class in its classpath and similarly the pong actor server only needs to see `Pong` class in its classpath.

4.8 Experiments

In this section, we explain how a series of benchmarks were directed to evaluate the performance and functionality of actor API in Java 8. For this benchmark, we use a simple Java application that uses the "Ping-Pong" actor example discussed previously. An application consists of one instance of `Ping` actor and one instance of `Pong` actor. The application sends a `ping` message to the ping actor and waits for the result. The ping message depends on a `pong` message to the pong actor. When the result from the pong actor is ready, the ping actor completes the message; this completes a round trip of a message in the application. To be able to make comparison of how actor API in Java 8 performs, the example is also implemented using Akka [107] library. The same set of benchmarks are performed in isolation for both of the applications. To perform the benchmarks, we use JMH [102] that is a Java microbenchmarking harness developed by OpenJDK community and used to perform benchmarks for the Java language itself.

The benchmark is performed on the round trip of a message in the application. The benchmark starts with a warm-up phase followed by the running phase. The benchmark composes of a number of iterations in each phase and specific time period for each iteration specified for each phase. Every iteration of the benchmark triggers a new message in the application and waits for the result. The measurement used is *sampling time* of the round trip of a message. A specific number of samples are collected. Based on the samples in different phases, different *percentile* measurements are summarized. An example percentile measurement $p(99.9900) = 10\ \mu s$ is read as 99.9900% of messages in the benchmark took 10 micro-seconds to complete.

Each benchmark starts with 500 iterations of warm-up with each iteration for 1 micro-second. Each benchmark runs for 5000 iterations with each iteration for 50 micro-seconds. In each iteration, a maximum number of 50K samples are collected. Each benchmark is executed in an isolated JVM environment with Java 8 version b127. Each benchmark is executed on a hardware with 8 cores of CPU and a maximum memory of 8GB for JVM.

The results are presented in Figure 4.2. The performance difference observed in the measurements can be explained as follows. An actor in Akka is expected to expose a certain behavior as discussed in Section 4.3 (i.e. `onReceive`). This means that every message leads to an eventual invocation of this method inside actor. However, in case of an actor in Java 8, there is a need to make a look-up for the actual method to be executed with expected arguments. This means that for every method, although in the presence of caching, there is a need to find the proper method that is expected to be invoked. A constant overhead for the method look-up in order to adhere to the object-oriented principles is naturally to be expected. Thus, this is the minimal performance cost that the actor API in Java 8 pays to support programming to interfaces.

4.9 Conclusion

In this paper, we discussed an implementation of the actor-based ABS modeling language in Java 8 which supports the basic object-oriented mechanisms and principles of method look-up and programming to interfaces. In the full version of this paper we have developed an operational semantics of Java 8 features including lambda expressions and have proved formally the correctness of the embedding in terms of a bisimulation relation.

The underlying modeling language has an executable semantics and supports a variety of formal analysis techniques, including deadlock and schedulability analysis [39, 60]. Further it supports a formal behavioral specification of interfaces [**HahnleHJLSSW11**], to be used as contracts.

We intend to expand this work in different ways. We aim to automatically *generate* ABS models from Java code which follows the ABS design methodology. Model extraction allows industry level applications be abstracted into models and analyzed

for different goals such as deadlock analysis and concurrency optimization. This approach of model extraction we believe will greatly enhance industrial uptake of formal methods. We aim to further extend the implementation of API to support different features especially regarding distribution of actors especially in the queue layer, and scheduling of messages using application-level policies or real-time properties of a concurrent system. Furthermore, the current implementation of ABS API in a distributed setting allows for instantiation of remote actors. We intend to use the implementation to model ABS deployment components [67] and simulate a distributed environment.

Design Pattern

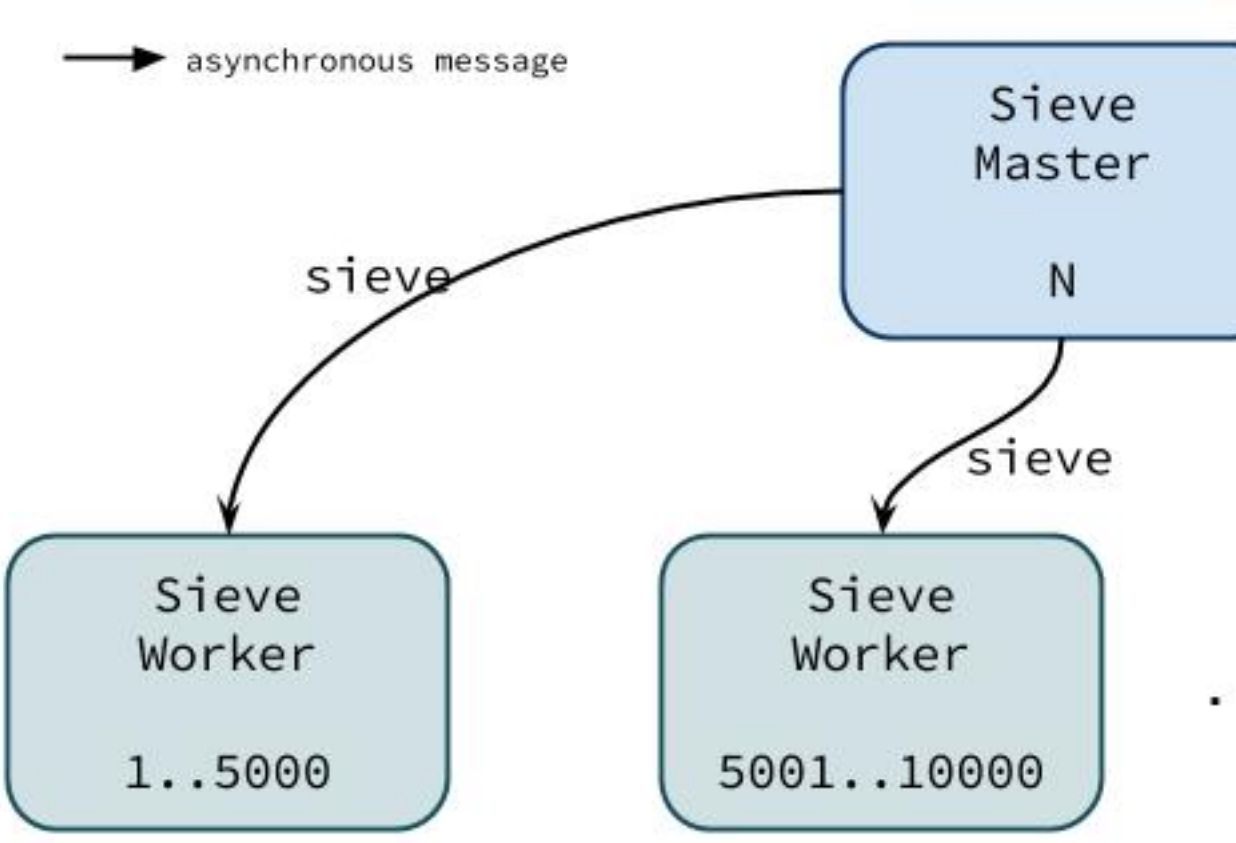
5

We design and implement a prime sieve generator using ABS API. In this example, we strictly follow object-oriented design principles and eventually mix in the actor model with the implementation.

The following summarizes the prime sieve generation:

1. The implementation works with a fixed input number `N`.
2. The result is a sequence of prime numbers less than or equal to `N`.
3. The implementation is required to be *concurrent* and *asynchronous*.

The following diagram depicts a general structure of the prime sieve generator in this example:



- `SieveMaster` creates a number of buckets; i.e. `SieveWorker`, each of which are responsible for a specific range.

- The bucket size is chosen as `5000` as a fixed number is this example.
- `SieveMaster` sends a message `sieve` to all `SieveWorkers`.
- When a `SieveWorker` finishes its computation it responds to the master by sending a message `done`.
- When the last worker is done, the prime generation completes.

Next, we design a Java interface for the prime sieves. Both `SieveMaster` and `SieveWorker` are different implementations of `Sieve`:

Listing 27: Sieve Actor

```
1 interface Sieve {
2   void sieve();
3   void done(List<Long> primes);
4   Long last();
5 }
```

The above interface allows `SieveMaster` to:

1. break the input `N` into a number of buckets
2. create a `SieveWorker` for every bucket
3. initiate the algorithm with the first worker and moving on one by one when each worker is done
4. return `last()` when the last worker finishes the computation

The following presents the above steps in `SieveWorker`'s `sieve()` method:

```
1 SieveWorker w = getNextWorker() or finish
2 (M) send a message to ‘w‘ to ‘sieve‘
```

and equivalently in `SieveWorker`'s `done(List<Long> primes)` method:

```
1 Collect generated ‘primes‘
2 ‘sieve‘ again
```

When `sieve()` in the master is done, the collected primes are the generated prime numbers up to input `N`.

The interesting part of the problem above in master's `sieve()` method is:

```
1 (M) send a message to ‘w‘ to ‘sieve‘
```

We model *a message* by using

1. `java.lang.Runnable` to present a message with no interest in its return value
2. `java.util.concurrent.Callable<V>` to present a message with a specific type of return value `<V>`

Through this model, we **separate** *message delivery* from *message execution*. Essentially, a message execution is eventually running the method invocation encapsulated in `Runnable` or `Callable`.

For example:

1. a message can be delivered in the same thread but executed in a different thread

2. a message can be delivered in a separate thread but executed in the same delivery thread

3. a message can be delivered in a separate thread and executed in a thread again different from the delivery thread

The above models of message delivery and execution build the foundation for *synchronous* and *asynchronous* message passing in ABS API. By default, ABS API uses the 3rd approach in which all objects in the runtime of the system share the same pool of threads.

We present `(M)` in the following as:

```
1 Runnable msg = new Runnable() {
2   void run() {
3     w.sieve();
4   }
5 }
```

In Java 8, we can simply use Lambda expressions, so:

```
1 Runnable msg = () -> w.sieve();
```

If we want to rewrite `(M)` from the above:

```
1 Runnable msg = () -> w.sieve();
2 send(w, msg);
```

in which `send(Object, Object)` is expected to be a method that semantically provides Approach 2 explained above to deliver the message and execute it. Now the question is where `send(Object, Object)` can come from?

Next, let us zoom at `SieveWorker`. A `SieveWorker` is also a `Sieve` implementation a sketch of which can be presented as:

```
1 Sieve my range for prime numbers
2 (W) Reply to the "sender" with generated prime numbers
```

When a sieve worker has generated the prime numbers in its own range, it is expected to reply back to the master. We designed `void done(List<Long>)` for this purpose in `Sieve`. We can rewrite `(W)` from above as:

```
1 List<Long> primes = // compute primes in my range
2 (W) Reply to master (sender) with message with invoking ‘done(primes)‘
```

which means as before, we need to *send a message* but the difference here is that this is a *reply message*. We can rewrite then as:

```
1 List<Long> primes = // compute primes in my range
2 Runnable replyMsg = () -> {
3   (S) Sieve sender = sender();
4   s.done(primes);
5 };
6 (R) reply(replyMsg);
```

We can *functionally* define `reply` in `(R)` as:

```
1 reply = send(sender(), Object)
```

in which `sender()` in `(S)` refers to the *sender* object of the current message being processed by **`this`** object.

Naturally, we need to sketch the implementation of `done(List<Long>)` in `SieveMaster` as it will be the main receive of done messages:

```
1 void done(List<Long> primes) {
2   Store the primes
3   sieve();
4 }
```

Simply, after each receipt of `done`, the master stores the newly generated primes and sieve again to see if there are still sieve workers to do their computation.

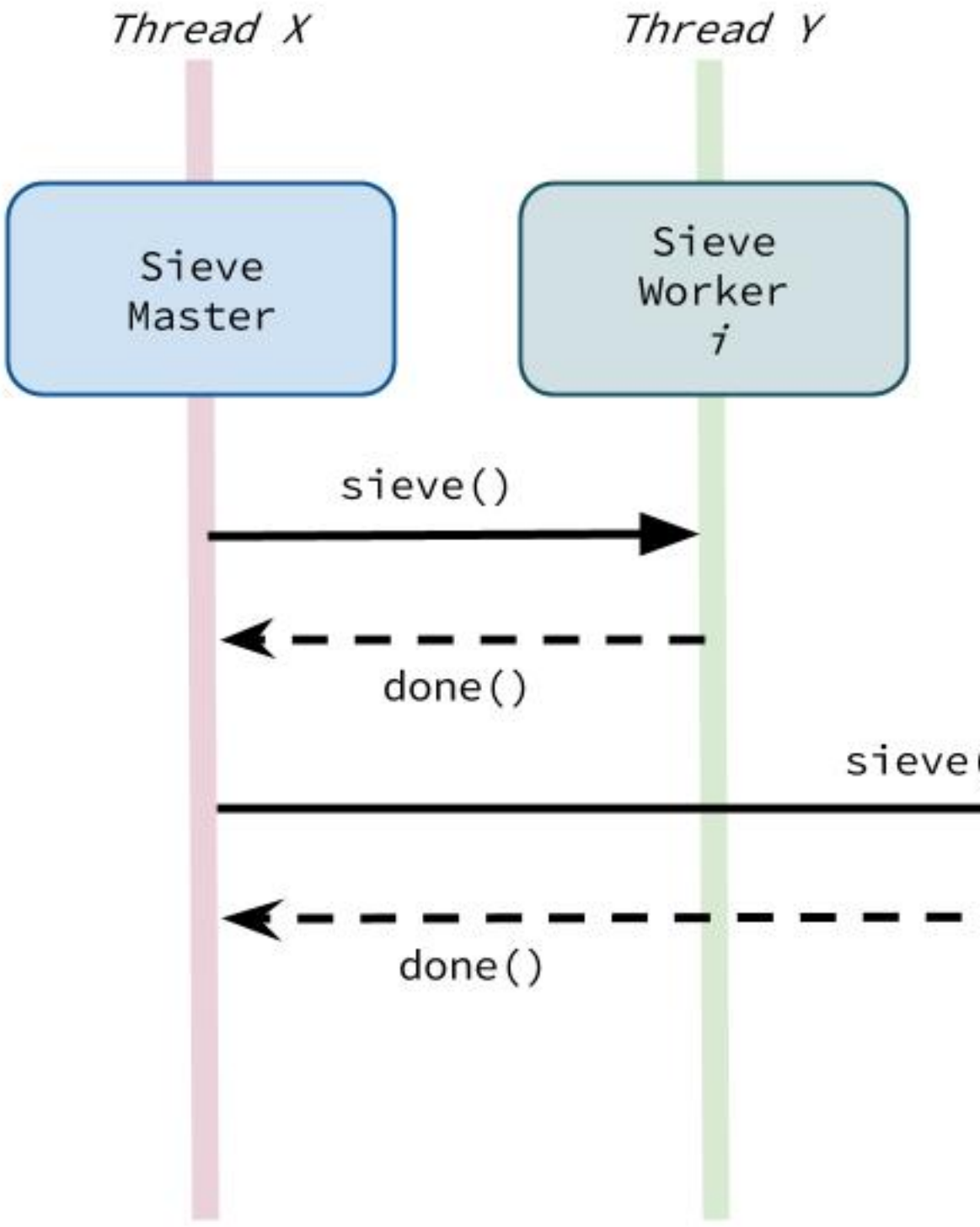
We summarize the requirements that we have to be able to design this example following both object-oriented design and asynchronous computation:

- `send(Object o, Object msg)` sends a message `msg` to an object `o` in an asynchronous way
- `sender()` refers to the sender object of the currently being processed message in **`this`** object
- `reply(Object msg)` send a message `msg` as a reply to the `sender()` of the processed message in **`this`** object

This is where we mix object-oriented principles with Actor model brought by ABS API. To add the above capabilities to our example, we simple change `Sieve` interface as:

```
1 interface Sieve extends Actor {
2 }
```

to extend `abs.api.Actor` interface. Now, all implementations of `Sieve` interface have access to the methods we draw as expectations for this example. Applying the actor model API to this sieve example as explained above can be depicted conceptually as:



TODO: Generalize how we mixed the original object-oriented principles and Actor model principles implemented in ABS API through Java 8.

Part IV

Application

6 Formal verification of service level agreements through distributed monitoring[1]

Behrooz Nobakht, Stijn de Gouw, Frank S. de Boer

Abstract

In this paper, we introduce a formal model of the availability, budget compliance and sustainability of distributed services, where service sustainability is a new concept which arises as the composition of service availability and budget compliance. The model formalizes a distributed platform for monitoring the above service characteristics in terms of a parallel composition of task automata, where dynamically generated tasks model asynchronous events with deadlines. The main result of this paper is a formal model to optimize and reason about service characteristics through monitoring. In particular, we use schedulability analysis of the underlying timed automata to optimize and guarantee service sustainability.

6.1 Introduction

Cloud computing provides the elastic technologies for virtualization. Through virtualization, software itself can be offered as a service (Software as a Service, SaaS). One of the aims of SaaS is to allow service providers to offer reliable software services while scaling up and down allocated resources based on their availability, budget, service throughput and the Service Level Agreements (SLA). Thus, it becomes essential that virtualization technologies facilitate elasticity in a way that enables business owners to *rapidly* evolve their systems to meet their customer requirements and expectations.

The fundamental technical challenge to a SaaS offering is maintaining the quality of service (QoS) promised by its SLA. In SaaS, providers must ensure a consistent QoS in a dynamic virtualized environment with variable usage patterns. Specifically, virtualized environments such as the cloud provide elasticity in resource allocation, but they often do not offer an SLA that can guarantee constant resource availability. As a result, SaaS providers are required to react to resource availability at runtime. Furthermore, by offering a $24/7$ software service, SaaS providers must be able to react to certain service usage patterns, such as an increase in throughput to ensure the SLA is maintained.

[1]This paper is funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services, `http://www.envisage-project.eu`.

Runtime monitoring [81, 17] is a dynamic analysis approach based on extracting relevant information about the execution. Runtime monitoring may be employed to collect statistics about the service usage over time, and to detect and react to service behavior. This latter ability is fundamental in the SaaS approach to guarantee the SLA of a service and is the focus of this paper.

The monitoring model that is presented in this paper is designed to *observe* in real-time certain service characteristics and *react* to them to ensure the evolution of the system towards its SLA. Asynchronous communication is an essential feature of a monitoring model in a distributed context. Asynchronous communication accomplishes non-intrusive observations of the service runtime. Further, the monitoring model is expected to operate according to certain real-time constraints specified by the SLA of the service. Satisfying the real-time constraints is the main challenge in a distributed monitoring model.

In this paper, we formalize service availability and budget compliance in a distributed deployment environment. This formalization is based on high-level task automata models [5, 36, 58]. The automata capture the real-time evolution of the resources provided by a distributed deployment platform and the above two main service characteristics. These task automata represent the real-time generation of the asynchronous events extended with deadlines [12, 92] by the monitoring platform for managing resources (i.e. allocation or deallocation). The main result of this paper is a formal model to optimize and reason about the above service characteristics through monitoring. In particular, the *schedulability* of the underlying timed automata implies service availability and budget compliance. Furthermore, we introduce a composition of service availability and budget compliance which captures service sustainability. We show that service sustainability presents a multi-objective optimization problem.

6.2 Related Work

Vast research work present different aspects of runtime monitoring. We focus on those that present a line of research for distributed deployment of services.

MONINA [56] is a DSL with a monitoring architecture which supports certain mathematical optimization techniques. A prototype implementation is available. Accurately capturing the behavior of an in-production legacy system coded in a conventional language seems challenging: it requires developing MONINA components, which generate events at a specified fixed rate, there are no control structures (if-else, loops), the data types that can be used in events are pre-defined, and there are no OO-features. We use ABS [65], an executable modeling language that supports all of these features and offers a wide range of tool-supported analyses [19, 114]. The mapping from ABS to timed automata [5] allows to exploit the state-of-the-art tools for timed automata, in particular for reasoning about real-time properties (and, as we show, SLAs using schedulability analysis [36]). MONINA offers *two pre-defined*

parameters that can be used in monitoring to adapt the system: cost and capacity. Our service metric function generalizes this to *arbitrary user-defined* parameters, including cost and capacity.

Hogben and Pannetrat examine in [55] the challenges of defining and measuring availability to support real-world service comparison and dispute resolution through SLAs. They show how two examples of real-world SLAs would lead one service provider to report 0% availability while another would report 100% for the same system state history but using a different period of time. The transparency that the authors attempt to reach is addressed in our work by the concept of monitoring window and expectation tolerance in Section 6.4. Additionally, the authors take a continuous time approach contrasted with ours that uses discrete time advancements. Similarly, they model the property of availability using a two-state model.

The following research works provide a language or a framework that allows to formalize service level agreements (SLA). However, they do not study how such SLAs can be used to monitor the service and evolve it as necessary. WSLA [74] introduces a framework to define and break down customer agreements into a technical description of SLAs and terms to be monitored. In [83], a method is proposed to translate the specification of SLA into a technical domain directed in SLA@SOI EU project. In the same project, [26] defines terms such as availability, accessibility and throughput as notions of SLA, however, the formal semantics and properties of the notions are not investigated. In [21], authors describe how they introduce a function how to decompose SLA terms into measurable factors and how to profile them. Timed automata is used in [99] to detect violations of SLA and formalize them.

Johnsen [67] introduce "deployment components" using Real-Time ABS [12]. A deployment component enables an application to acquire and release resources on-demand based on a QoS specification of the application. A deployment component is a high level abstraction of a resource that promotes an application to a resource-aware level of programming. Our work is distinguished by the fact that we separate the monitors from the application (service) themselves. We argue that we aim to design the monitoring model to be as *non-intrusive* as possible to the service runtime. Thus, we do not deploy the monitors inside the service runtime.

In Quanticol EU project[2], authors in [25] and [40] use statistical approaches to observe and guarantee service level agreements for public transportation. We also present that service characteristics can be composed together. This means that evolving a system based on SLAs turns into a multi-object optimization problem. In addition, in COMPASS EU project[3], CML [115] defines a formal language to model systems of systems and the contracts between them. CML studies certain properties of the model and their applications. CML is used in the context of a

[2]Quanticol EU project with no. 600708: `http://quanticol.eu/`
[3]COMPASS EU project with no. 287829: `http://www.compass-research.eu/`

Robotics technology to model and ensure how emergency sensors should react and behave according to the SLAs defined for them. Our approach is similar to provide a generic model for service characteristics definition, however, we utilize timed and task automata.

6.3 SDL Fredhopper Cloud Services

In this section, we introduce a running example in the context of SDL Fredhopper. We use the example in different parts of the paper and also in the experiments.

SDL Fredhopper develops the Fredhopper Cloud Services to offer search and targeting facilities on a large product database to e-Commerce companies as services (SaaS) over the cloud computing infrastructure (IaaS). Fredhopper Cloud Services provides several SaaS offerings on the cloud. These services are exposed at endpoints. In practice these endpoints typically are implemented to accept connections over HTTP. For example, one of the services offered by these endpoints is the Fredhopper Query API, which allows users to query over their product catalog via full text search[4] and faceted navigation[5].

A customer of SDL Fredhopper using Query API owns a *single* HTTP endpoint to use for search and other operations. However, internally, a number of resources (virtual machines) are used to deliver Query API for the customer. The resources used for a customer are managed by a load balancer. In this model of deployment, each resource is launched to serve *one* instance of Query API; i.e. resources are *not* shared among customers.

When a customer signs a contract with SDL Fredhopper, there is a clause in the contract that describes the minimal QoS levels of the Query API. For example, we have a notion of query per second (QPS) that defines the number of completed queries per second for a customer. An agreement is a bound on the expected QPS and forms the basis of many decisions (technical or legal) thereafter. The agreement is used by the operations team to set up an environment for the customer which includes the necessary resources described above. The agreement is additionally used by the support team to manage communications with the customer during the lifetime of the service for the customer.

Maintaining the services for more than 250 customers on more than 1000 servers is not an easy operation task [6]. Thus, to ensure the agreements in a customer's contract:

- The operation team maintains a monitoring platform to get notifications on the current metrics.

[4] `http://en.wikipedia.org/wiki/Full_text_search`
[5] `http://en.wikipedia.org/wiki/Faceted_navigation`
[6] Figures are indication of complexity and scale. Detailed confidential information may be shared upon official request.

- The operation team performs *manual* intervention to ensure that sufficient resources are available for a customer (launching or terminating).
- The monitoring platform depends on *human* reaction.
- The cost that is spent for a customer on the basis of safety can be *optimized*.

In this paper, we use the notion of QPS as an example in the concepts that are presented in this research. We use the example here to demonstrate how the model that is proposed in this research can address the issues above and alleviate the *manual* work with *automation*. The manual life cycle depends on the domain-specific and contextual knowledge of the operations team for every customer service that is maintained in the deployment environment. This is labor-intensive as the operations team stands by 24×7. In such a manual approach, the business is forced to over-spend to ensure service level agreements for customers.

6.4 Distributed Monitoring Model

We introduce a distributed monitoring platform and its components and discuss some underlying assumptions and definitions. Further, we define the notion of service availability and service budget compliance. In the deployment environment (e.g., "the cloud"), every server from the IaaS provider is used for a *single* service of a customer, such as the Query Service API for a customer of SDL Fredhopper (c.f. Section 6.3). Typically, multiple servers are allocated to a single customer. The number of servers allocated for a customer is not visible to the customer. The customer uses a single endpoint - in the load balancer layer - to access all their services.

The ultimate goal is to maintain the environment in such a way that customers and their end users experience the delivered services up to their expectations while minimizing the cost of the system. The first objective can be addressed by adding resources; however, this conflicts with the second goal since it increases the cost of the environment for the customer. In this section, we formalize the above intuitive notions as *service availability* and *service budget compliance*.

We then develop a distributed monitoring platform that aims to optimize these service characteristics in a deployment environment. The monitoring platform works in two cyclic phases: *observation* and *reaction*. The observation phase takes measurements on services in the deployment environment. Subsequently, the corresponding levels of the service characteristics are calculated. In the reaction phase, if needed, a platform API is utilized to make the necessary changes to the deployment environment (e.g. adjust the number of allocated resources) to optimize the service characteristics. The monitoring platform builds on top of a real-time extension of the actor-based language ABS [65]. To ensure non-intrusiveness of the monitor with the running service, each monitor is an active object (actor) running on a separate re-

source from that which runs the service itself, and the components of the monitoring platform communicate through *asynchronous messages* with deadlines [67].

Below, we discuss assumptions and basic oncepts that will be used in the analysis of the formal properties of the monitoring platform and corresponding theorems. We assume that the external infrastructure provider has an *unlimited* number of resources. Further, we assume that all resources are of the *same type*; i.e. they have the same computing power, memory, and IO capacity. Finally, we assume that every resource is initialized within at most t_i amount of time.

In our framework time T is a universally shared clock based on the NTP [7] that is used by all elements of the system in the same way. T is discrete. We fix that the unit of time is *milliseconds*. This level of granularity of time unit means that between two consecutive milliseconds, the system is not observable. For example, we use the UTC time standard for all services, monitors and platform API. We refer to the current time by t_c.

We denote by r a resource which provides computational power and storage and by s a general abstraction of a service in the deployment environment. A service exposes an API that is accessible through a delivery layer, such as HTTP. In our example, a service is the Query API (c.f. Section 6.3) that is accessible through a single HTTP endpoint.

In our framework, *monitoring platform* P is responsible for (de-)allocation of resources for computation or storage. We abstract from a specific implementation of the monitoring platform P through an API in Listing 28. There is only *one* instance of P available. In this paper, P internally uses an external infrastructure provisioning API to provide resources (e.g. AWS EC2). The term "platform" is interchangeably used for monitoring in this paper. The platform provides a method `getState(Service s)` which returns the number of resources allocated to the given service s at time t_c.

Listing 28: Platform API

```
1 interface Platform {
2   void    allocate(Service s);
3   void    deallocate(Service s);
4   Number  getState(Service s);
5   boolean verify_α(Service s);
6   boolean verify_β(Service s);
7 }
```

We use monitoring to observe the external behavior of a service. We formalize the external behavior of a service with its service-level agreement (SLA). An SLA is a contract between the customer (service consumer) and the service provider which defines (among other things) the minimal quality of the offered service, and the compensation if this minimal level is not reached. To formally analyze an SLA, we introduce the notion of a service metric function. We make basic measurements of the service externally in a given monitoring window (a duration). The service metric function aggregates the basic measurements into a single number that indicates the quality of a certain service characteristic (higher numbers are better).

[7] `https://tools.ietf.org/html/rfc1305`

Basic measurement $\mu(s, r, t)$ is a function that produces a real number of a *single* monitoring check on a resource r allocated to service s at some time t. For example, for SDL Fredhopper cloud services, a basic measurement is the number of completed queries at the current time.

Service Metric f_s is a function that aggregates a sequence of basic non-negative measurements to a single non-negative real value: $f_s : \bigcup_n \mathbb{R}^n \to \mathbb{R}$. For example, for SDL Fredhopper cloud services, the service metric function f_s calculates the average number of queries per second (QPS) given a list of basic measurements.

Monitoring Window is a duration of time τ throughout which basic measurements for a service are taken.

Monitoring Measurement is a function that aggregates the basic measurements for a service over its resources in the last monitoring window. The last monitoring window is defined as $[t_c - \tau, t_c]$. To produce the monitoring measurement, f_s is applied. Formally:

$$\mu(s, r, \tau) = f_s(\langle \mu_i(s, r, t) \rangle_{i=0}^{\infty}) \text{ where } t \in [t_c - \tau, t_c]$$

in which $\mu_i(s, r, t)$ is the i-th basic measurement of services s on resource r at time t where $t \in [t_c - \tau, t_c]$.

Definition 1 (Service Availability $\alpha(s, \tau, t_c)$)**.** First, we need a few auxiliary definitions before we can define service availability.

Service Capacity $\kappa_\sigma(s, \tau) = \sum_{r \in \sigma(s)} \mu(s, r, \tau)$ denotes the capability of service s that is the aggregated monitoring measurements of its resources over the monitoring window τ and $\sigma(s)$ is the number of allocated resources to service s.

Agreement Expectation $E(s, \tau, t_c)$ is the minimum number of requests that a customer expects to complete in a monitoring window τ. The agreement expectation depends on the current time t_c because the expectation may change over time. For example, SDL Fredhopper customers expect a different QPS during Christmas.

We define the availability of a service $\alpha(s, \tau, t_c)$ in every monitoring window τ as:

$$\alpha(s, \tau, t_c) = \frac{\kappa_\sigma(s, \tau)}{E(s, \tau, t_c)}$$

Capacity Tolerance $\varepsilon_\alpha(s, \tau)) \in [0, 1]$ defines how much $\kappa_\sigma(s, \tau)$ can deviate from $E(s, \tau, t_c)$ in every time span of duration τ.

Service Guarantee Time t_G is the duration within which a customer expects service availability reaches an acceptable value after a violation. Typically, t_G is an input parameter from the customer's contract.

Example 1. Intuitively, $\alpha(s, \tau, t_c)$ presents the actual capability of a service s over a time period τ compared to the expectation on the service $E(s, \tau)$. For values $\alpha(s, \tau, t_c) \ll 1 - \varepsilon_\alpha(s, \tau))$, the resource for service s are at "under-capacity" while for values $\alpha(s, \tau, t_c) \gg 1 + \varepsilon_\alpha(s, \tau))$, there is "over-capacity". The goal is optimize $\alpha(s, \tau, t_c)$ towards a value of 1.

For example, we expect a query service to be able to complete 10 queries per second. We define the monitoring window $\tau = 5$ minutes; thus, $E(s, \tau, t_c) = 10 \times 60 \times 5 = 3000$. Suppose we allocate only one resource to the service, measure the service during a single monitoring window τ and find $\mu(s, r, \tau) = 2900$. Then $\alpha(s, \tau, t_c) = \frac{2900}{3000} = 0.966$. If we have $\varepsilon_\alpha(s, \tau)) = 0.03$, this means that service s is under-capacity because $\alpha(s, \tau, t_c) < 1 - \varepsilon_\alpha$.

Definition 2 (Budget Compliance $\beta(s, \tau)$)**.** We first provide a few auxiliary definitions.

Resource Cost $€(r, \tau) \in \mathbb{R}^+$ is the cost of resource r in a monitoring window τ which is determined by a fixed resource cost per time unit.

Service Cost $€_\sigma(s, \tau) \in \mathbb{R}^+$ is the cost of a service s in a monitoring window τ and defined as $€_\sigma(s, \tau) = \sum_{r \in \sigma(s)} €(r, \tau)$.

Service Budget $B(s, \tau)$ specifies an upper bound of the expected cost of a service in the time span τ. Intuitively $B(s, \tau)$ is the allowed budget that can be spent for service s over the time span τ. The service budget is typically chosen to be fixed over any time span τ.

We are now ready to define service budget compliance $\beta(s, \tau)$ that, intuitively, represents how a service complies with its allocated budget:

$$\beta(s, \tau) = \frac{€_\sigma(s, \tau)}{B(s, \tau)}$$

Budget Tolerance $\varepsilon_\beta(s, \tau) \in [0, 1]$ specifies how much the service cost $€(s, \tau)$ can deviate from $B(s, \tau)$ in every time span of duration τ.

Service Guarantee Time t_G is similar to that defined for service availability.

Example 2. Assume every resource on the environment costs 1 (e.g. €) per hour. Suppose we set a budget of 1.5 per hour for every service, allocate *one* resource to the service and define a monitoring window of $\tau = 5$ minutes. Every hour has 12 monitoring windows. This means that each resource costs $€(r, \tau) = \frac{1}{12} \approx 0.08$ per monitoring window. Since there is only one resource, the service cost is $€(s, \tau) = \sum_{r \in \sigma(s)} €(s, \tau) \approx 0.08$ per monitoring window. On the other hand, if we calculate the budget for one monitoring window, we have $B(s, \tau) = \frac{1.5}{12} = 0.125$ per monitoring window. This yields budget compliance as $\beta(s, \tau) = \frac{0.08}{0.125} = 0.64$.

The formal definitions of service availability and budget compliance provide a rigorous basis for automatic deployment of resource-aware services with an appropriate quality of service, taking costs into account. This in particular includes automated scaling up or down of the service with the help of monitoring checks that are installed for the service. The fundamental challenge in ensuring service availability and budget compliance is that they have *conflicting* objectives:

$$\alpha(s, \tau, t_c) \uparrow \iff \beta(s, \tau) \downarrow$$

Intuitively, if more resources are used to ensure the availability of a service; then $\alpha(s, \tau, t_c)$ increases. However, at the same time, the service costs more; i.e. budget compliance $\beta(s, \tau)$ decreases.

6.5 Service Characteristics Verification

In this section, we use timed automata and task automata to model the behavior of a monitoring platform P, the deployment environment E, and the monitoring components for service availability $\alpha(s, \tau, t_c)$ and budget compliance $\beta(s, \tau)$. [58] defines a task automata as an extension of timed automata in which each task is a piece of executable program with (b, w, d): best/worst time and deadline of the task. A task automata uses a scheduler for the tasks to schedule each task with a location on a queue.

Modeling the elements of the monitoring platform is necessary to be able to study certain properties of the system. The most important goal of a monitoring platform is to enable the autonomous operation of a set of services according to their SLA. Thus, it is essential how to analyze that the monitoring platform can provide certain guarantees about the service and its SLA. In addition, it is important be able to verify the monitoring platform through model checking and schedulability analysis. Using timed automata and task automata facilitates model checking and verification through formal method tools such as UPPAAL [10] supporting advanced methods such as state-space reduction [75].

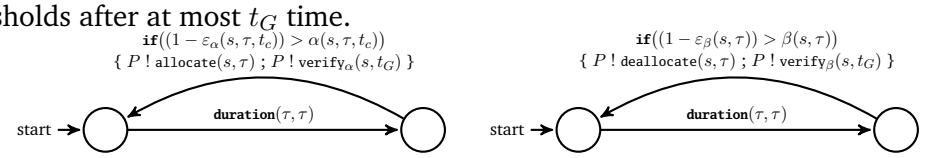
We use task automata as defined in [36, 57, 58]. Task automata are an extension of timed automata [5]. In addition, we design the automata for the monitoring platform using the real-time extension of task automata presented in [58] p. 92 in which the author presents a mapping from Real-Time ABS [67] to the equivalent task automata.

A task type is a piece of executable program/code represented by a tuple (b, w, d), where b and w respectively are the best-case and worst-case execution times and d is the deadline. In a task automata, there are two types of transitions: *delay* and *discrete*. A delay transition models the execution of a running task by idling for other tasks. A discrete transition corresponds to the arrival of a new task. When a new task is triggered, it is placed into a certain position in the queue based on a scheduling policy [91, 92]. Examples of a scheduling policy are FIFO or EDF (earliest deadline first). The scheduling policy is modeled as a timed automaton Sch. Every task has its own stop watch. The scheduler also maintains a separate stop watch for each task to determine if a task misses its deadline. All stop watches work at the same clock speed specified by T.

We design separate automata for each service s characteristic: service availability $\alpha(s, \tau, t_c)$ by an automata M_{α_s} and service budget compliance $\beta(s, \tau)$, by an automata M_{β_s}. Each automaton is responsible for one goal: to optimize the service characteristic. M_{α_s} aims to improve $\alpha(s, \tau, t_c)$ whereas M_{β_s} aims to improve $\beta(s, \tau)$.

M_{α_s} uses `allocate` to launch a new resource in the environment and improve the service s. In contrast, M_{β_s} uses `deallocate` to terminate a resource to decrease the cost of the service.

We use task automata to design M_{α_s}. Periodically, M_{α_s} checks whether the service availability is within the thresholds, taking tolerance into account (Definition 1). If the condition fails, M_{α_s} generates a task for monitoring platform P to allocate a new resource to service s with a deadline of τ. We define the period to be τ. We use the semantics of a task automata in [58] p. 92 in the transitions of the task automata. Figure 6.1a and 6.1b present M_{α_s} and M_{β_s}. Both M_{α_s} and M_{β_s} share state with the monitoring platform P. The state keeps the current number of resources for a service s that is denoted by $\sigma(s)$. All timed automata and task automata in the monitoring platform have shared access to $\sigma(s)$. In the automata, we use a conditional statement to check the service characteristics $\alpha(s,\tau,t_c)$ or $\beta(s,\tau)$. If the condition fails, M_{α_s} requests P to `allocate` a new resource to s and M_{β_s} requests P to `deallocate` a resource. In addition, M_{α_s} triggers a new task $\texttt{verify}_\alpha$ with deadline t_G. Intuitively, this means the service characteristic $\alpha(s,\tau,t_c)$ is verified to be within the expected thresholds after at most t_G time.



(a) M_{α_s} task automata for $\alpha(s,\tau,t_c)$ **(b)** M_{β_s} task automata for $\beta(s,\tau)$

We use a separate task automaton for each service characteristic to verify the SLA of the service after t_G time. Respectively, M_{V}^{α} and M_{V}^{β} execute tasks $\texttt{verify}_\alpha$ and $\texttt{verify}_\beta$ (Figures 6.2a and 6.2b). M_{V}^{α} uses **`await`** to ensure the condition of the SLA. In addition, the task is controlled by the scheduler using a deadline that is specified as t_G in the generated task $\texttt{verify}_\alpha(s,t_G)$ in M_{α_s}. If t_G passes before the guard statement in **`await`** statement holds, it leads to a *missed deadline*.

(a) M_{V}^{α} to execute $\texttt{verify}_\alpha$ **(b)** M_{V}^{β} to execute $\texttt{verify}_\beta$

Both M_{α_s} and M_{β_s} are specific to one particular service s. A generalized automaton for all services is obtained as their parallel composition: $M_\alpha = (\|_s\ M_{\alpha_s})$ and $M_\beta = (\|_s\ M_{\beta_s})$. The tasks generated by M_α and M_β (triggered by the calls to `allocate` and `deallocate`) are executed by the task automata for platform M_P.

We model monitoring platform P by a task automata M_P. The task types are $\{\texttt{A}(\texttt{allocate}), \texttt{D}(\texttt{deallocate})\}$. For task type `A` in M_P, we use $(b,w,d) = (t_i,\tau,\tau)$; i.e. the best-case execution time of a task is the resource initialization time, the worst-case is the length of the monitoring window, and the deadline is the length of the monitoring window. For task type `D` in M_P, we use $(b,w,d) = (0,\tau,\tau)$. We do not fix the scheduling policy Sch. The error state q_{err} in M_P is defined when either a deadline is missed or when the platform fails to provision a resource. Thus the monitoring platform P contains the following ingredients:

$$M_P = \langle M_{\text{A}} \parallel M_{\text{D}} \parallel M_{\text{V}}^{\alpha} \parallel M_{\text{V}}^{\beta}, \text{Sch}, \tau \rangle$$

We define $M_{\mathtt{A}_s}$ as the timed automata to execute the tasks of type `allocate` in M_P. We use the model semantics presented in [58] p. 92 to design $M_{\mathtt{A}_s}$. The resulting automata is presented in Figure 6.3.

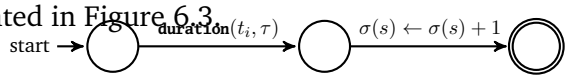


Figure 6.3: $M_{\mathtt{A}_s}$: Timed Automaton to execute task type `allocate` in M_P

Then, we define $M_{\mathtt{A}}$ in M_P as: $M_{\mathtt{A}} = \|_s \, M_{\mathtt{A}_s}$; i.e. the composition of all timed automata to execute a task `allocate` for some service s. Similarly, we design $M_{\mathtt{D}_s}$ to execute task type `deallocate` in Figure 6.4. Therefore, we also have $M_{\mathtt{D}}$ in M_P as: $M_{\mathtt{D}} = \|_s \, M_{\mathtt{D}_s}$.

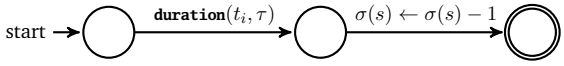


Figure 6.4: $M_{\mathtt{D}_s}$: Timed Automaton to execute task type `deallocate` in M_P

For a particular service s, its automaton M_{α_s} regularly measures the service characteristics and calculates $\alpha(s, \tau, t_c)$. When s is under-capacity, M_{α_s} requests to `allocate` a new resource for s through monitoring platform P. This generates a new task in M_P that is executed by $M_{\mathtt{A}_s}$. When the task completes, the state of the service $\sigma(s)$ is updated; strictly increased. Thus, in isolation, the combination of M_{α_s} and $M_{\mathtt{A}_s}$ increase the value of service availability $\alpha(s, \tau, t_c)$ for service s over time. Similarly, in isolation, the combination of M_{β_s} and $M_{\mathtt{D}_s}$ increase the value of service budget compliance $\beta(s, \tau)$ for service s over time. Because in the latter, `deallocate` is used to decrease the cost of the service and as such increases $\beta(s, \tau)$.

In reality, resources might fail in the environment. The failure of a resource is not and cannot be controlled by the monitoring platform P. However, the failure of a resource affects the state of a service and its characteristics. Thus, we model the environment, including failures, as an additional timed automata, M_E. In M_E, in every monitoring window, there is a probability that some resources fail. For example, we present a particular instance of M_E in Figure 6.5. In this environment, in every monitoring, an unspecified constant (c) number of resources fail.

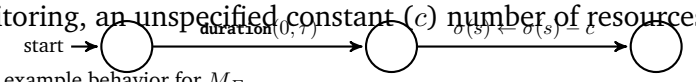


Figure 6.5: An example behavior for M_E

We define system automata [58] (p. 33, Definition 3.2.7) for each service characteristic; $\mathcal{S}_\alpha$ for $\alpha(s, \tau, t_c)$ and $\mathcal{S}_\beta$ for $\beta(s, \tau)$:

$$\mathcal{S}_\alpha = M_\alpha \parallel M_E \parallel M_P \qquad \textit{and} \qquad \mathcal{S}_\beta = M_\beta \parallel M_E \parallel M_P$$

With the above automata that we designed for $\alpha(s, \tau, t_c)$ and $\beta(s, \tau)$, we are now ready to present the main results.

Theorem 1. If the SLA for service s on $\alpha(s, \tau, t_c)$ is violated, either:

- $\mathcal{S}_\alpha$ re-establishes the condition $\alpha(s, \tau, t_c) \geq 1 - \varepsilon_\alpha(s, \tau)$ (thereby satisfying the SLA) within t_G time, or,
- there exists at least one task $\mathtt{verify}_\alpha$ in $M_{\mathtt{V}}^{\alpha}$ with a missed deadline.

Proof. At any given time in T:

- If $\alpha(s, \tau, t_c) \geq 1 - \varepsilon_\alpha(s, \tau)$, then the SLA for service availability α is satisfied.

- If the above condition does not hold, on every monitoring window τ, M_α generates a new task `allocate` in M_{A}. In addition, a new task $\texttt{verify}_\alpha$ is generated with a deadline t_G. After a duration of t_G, the **`await`** statement allows M_{V}^{α} to complete the task $\texttt{verify}_\alpha$ only if the condition $\alpha(s,\tau,t_c) \geq 1 - \varepsilon_\alpha(s,\tau)$ holds. If this is not the case, since t_G has passed, the scheduler generates a missed deadline (moving to its error state).

□

Theorem 2. If the SLA for service s on $\beta(s,\tau)$ is violated, either:

- $\mathcal{S}_\beta$ re-establishes the condition $\beta(s,\tau) \geq 1 - \varepsilon_\beta(s,\tau)$ (thereby satisfying the SLA) within t_G time, or,
- there exists at least one task $\texttt{verify}_\beta$ in M_{V}^{β} with a missed deadline.

Proof. Similar to the proof of Theorem 1. □

In practice, the guarantee of $\mathcal{S}_\alpha$ and $\mathcal{S}_\beta$ in isolation to eventually evolve the system to satisfy the SLA is not enough. In reality, a service provider tries ensure both simultaneously to reduce their cost of service delivery while ensuring the delivered service is of the expectations agreed upon with the customer. However, these goals conflict. When $\alpha(s,\tau,t_c)$ increases because of adding a new resource, it means that service s costs more, hence $\beta(s,\tau)$ decreases. The same applies in the other direction: increasing $\beta(s,\tau)$ negatively affects $\alpha(s,\tau,t_c)$.

To capture the combined behavior of service availability and budget compliance, we compose them. We define *service sustainability* $\gamma(s,\tau)$ as the composition of $\alpha(s,\tau,t_c)$ and $\beta(s,\tau)$. We present the composition by system automata $\mathcal{S}_\gamma$ as:

$$\mathcal{S}_\gamma = \mathcal{S}_\alpha \parallel \mathcal{S}_\beta$$

Authors in [36] define that a task automata is *schedulable* if there exists no task on the queue that misses its deadline. The next theorem presents the relationship between schedulability analysis of service sustainability and satisfying its SLA.

Theorem 3. If $\mathcal{S}_\gamma$ is *schedulable* given input parameters (τ, t_i, t_G), then the SLA for both service characteristics $\alpha(s,\tau,t_c)$ and $\beta(s,\tau)$ is satisfied within t_G time after a violation.

Proof. When a violation of the SLA occurs in $\mathcal{S}_\gamma$, either $\mathcal{S}_\alpha$ or $\mathcal{S}_\beta$ (or both) start to evolve the service based on Theorems 1 and 2. Therefore, there exists at least one task of $\texttt{verify}_\alpha$ or $\texttt{verify}_\beta$ with a deadline t_G. Hence, if $\mathcal{S}_\gamma$ is schedulable, then neither $\texttt{verify}_\alpha$ nor $\texttt{verify}_\beta$ miss their deadline. Thus, both $\mathcal{S}_\alpha$ and $\mathcal{S}_\beta$ are schedulable. This means that both $\texttt{verify}_\alpha$ and $\texttt{verify}_\beta$ complete successfully. Therefore, the SLA of the service is guaranteed within t_G after a violation in $\mathcal{S}_\gamma$. □

Using the algorithm presented in Chapter 6 [58], we translate the above task automata into traditional timed automata. This allows to leverage well-established model checking techniques such as UPPAAL [10] to determine the schedulability of $\mathcal{S}_\gamma$. Moreover, the results of the schedulability analysis serves as a method to optimize the input parameters of the monitoring model including τ and t_G.

6.6 Evaluation of the monitoring model

In this section, we evaluate the implementation of the monitoring model.

We set up an environment to evaluate how the monitoring evolves a service according to its SLA. In the environment, a single instance of monitoring platform is present to provide new resources as necessary. Every resource hosts only one service. We define two customers in the environment. For both customers, we deploy the same service, Fredhopper Query API. For every resource that hosts a service, we set up a monitor that measures QPS and reports it to the platform. Both customers run with the same SLA: the QPS expectation is $E(s, \tau, t_c) = 10$ and $\varepsilon_\alpha(s, \tau, t_c) = 0.1$. We launch every customer service with only one resource. Monitors observe the customer service and calculate the service availability of every customer service $\alpha(s, \tau, t_c)$.

We run the environment setup for different monitoring windows $\tau \in \{1, 5, 10\}$ (seconds). We fix the initialization time of a resource to $t_i = 2.5$ seconds. We set $t_G = 300$ seconds; i.e. we verify the service after this time and evaluate if the service is guaranteed based on its SLA.

Figure 6.6 plots the service availability $\alpha(s, \tau, t_c)$ over time with the different monitoring windows. The following summarizes the behavior:

- As the monitoring window τ increases, the system converges with a slower pace towards the expected $\alpha(s, \tau, t_c)$.
- When the monitoring window is chosen such that $\tau < t_i$, the evolution of the system becomes *non-deterministic*.
- The setting $\tau < t_i$ causes a missed deadline in $\texttt{verify}_\alpha$ because after a duration of t_G the service availability has not yet reached the expected value.

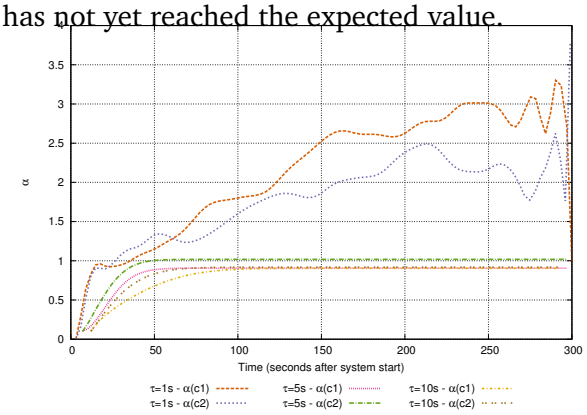


Figure 6.6: Evolving $\alpha(s, \tau, t_c)$ with different τ

Every monitoring measurement is performed in a monitoring window τ. Monitoring measurements are aggregated and calculated in every window and form the basis of reactions necessary to evolve the service to meet their SLA. Thus, selection of an appropriate monitoring window length τ is crucial, as we also discussed how schedulability analysis can be used to optimize it. The authors in [55] present that for the same setup and

deployment of services, measurements using different monitoring windows yield to very different understanding of service properties such as service availability. Therefore, it is essential to choose the value of τ such that monitoring measurements do not lead to *unrealistic* understanding and inappropriate reactions.

If $\tau < t_i$, Theorem 1 does not hold because every task `allocate` in M_{A} misses its deadline. Thus, it is essential that $\tau \geq t_i$. Analogously, choosing monitoring window as $\tau \gg 2 \times t_i$ also has a counter-productive effect on the service deployments. In a real setting, different services may use different types of resources. In such a setting, the monitoring window should be chosen as the largest t_i of any resource type that is available in the platform: $\tau \geq \mathsf{max}(t_i)\ \forall r \in P$.

6.7 Future work

We continue to generalize the notion of the distributed service characteristics and investigate how the composition of an arbitrary number of such properties can be formalized and reasoned about. In the context of the ENVISAGE project, industry partners define their service characteristics in this framework and monitor the service evolution. Moreover, the work will be extended to generate parts of the monitoring platform based on an input of different SLA formalizations such as SLA* [73]. Currently, we are integrating our automated monitoring infrastructure into the in-production SDL Fredhopper cloud services (cf. Section 6.3).

Bibliography

[1] L. Aceto, M. Cimini, A. Ingólfsdóttir, et al. „Modelling and Simulation of Asynchronous Real-Time Systems using Timed Rebeca“. In: *FOCLASA*. 2011, pp. 1–19 (cit. on p. 39).

[2] G. Agha, I. Mason, S. Smith, and C. Talcott. „A Foundation for Actor Computation“. In: *Journal of Functional Programming* 7 (1997), pp. 1–72 (cit. on p. 3).

[3] Gul Agha. „Actors: a model of concurrent computation in distributed systems“. PhD thesis. MIT, 1986 (cit. on pp. 3, 4, 23).

[4] Gul Agha. „The Structure and Semantics of Actor Languages“. In: *Proc. the REX Workshop*. 1990, pp. 1–59 (cit. on pp. 45, 56).

[5] Rajeev Alur and David L Dill. „A theory of timed automata“. In: *Theoretical computer science* 126.2 (1994), pp. 183–235 (cit. on pp. 70, 77).

[6] G.R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000 (cit. on p. 14).

[7] Joe Armstrong. „Erlang“. In: *Communications of ACM* 53.9 (2010), pp. 68–75 (cit. on pp. 46, 47).

[8] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007 (cit. on pp. 7, 24, 27, 40).

[9] Michael Baker and Martin Thompson. *LMAX Disruptor*. `http://github.com/LMAX-Exchange/disruptor`. LMAX Exchange (cit. on p. 48).

[10] Gerd Behrmann, Alexandre David, and Kim G Larsen. „A tutorial on uppaal“. In: *Formal methods for the design of real-time systems*. Springer, 2004, pp. 200–236 (cit. on pp. 77, 81).

[11] Joakim Bjørk, Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. „User-defined Schedulers for Real-Time Concurrent Objects“. In: *Innovations in Systems and Software Engineering* (2012) (cit. on p. 39).

[12] Joakim Bjørk, Frank S de Boer, Einar Broch Johnsen, Rudolf Schlatte, and S Lizeth Tapia Tarifa. „User-defined schedulers for real-time concurrent objects“. In: *Innovations in Systems and Software Engineering* 9.1 (2013), pp. 29–43 (cit. on pp. 4, 70, 71).

[13] Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. „A Complete Guide to the Future“. In: *ESOP*. 2007, pp. 316–330 (cit. on p. 28).

[14] Frank S. de Boer, Mohammad Mahdi Jaghoori, and Einar Broch Johnsen. „Dating Concurrent Objects: Real-Time Modeling and Schedulability Analysis“. In: *CONCUR 2010*. Vol. 6269. 2010, pp. 1–18 (cit. on p. 24).

[15] Grady Booch. „Object-oriented design“. In: *ACM SIGAda Ada Letters* 1.3 (1982), pp. 64–76 (cit. on p. 3).

[16] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. „Making the future safe for the past: Adding genericity to the Java programming language“. In: *Acm sigplan notices* 33.10 (1998), pp. 183–200 (cit. on p. 4).

[17] Konstantinos Bratanis, Dimitris Dranidis, and Anthony J. H. Simons. „Towards Run-Time Monitoring of Web Services Conformance to Business-Level Agreements“. In: ed. by Leonardo Bottaci and Gordon Fraser. Vol. 6303. Springer, 2010, pp. 203–206 (cit. on p. 70).

[18] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. „ASM: a code manipulation tool to implement adaptable systems“. In: *Adaptable and extensible component systems* 30 (2002) (cit. on p. 8).

[19] Richard Bubel, Antonio Flores-Montoya, and Reiner Hähnle. „Analysis of Executable Software Models“. In: *SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures*. 2014, pp. 1–25 (cit. on p. 70).

[20] Po-Hao Chang and Gul Agha. „Towards Context-Aware Web Applications“. In: *7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*. 2007, pp. 239–252 (cit. on p. 46).

[21] Yuan Chen, Subu Iyer, Xue Liu, Dejan Milojicic, and Akhil Sahai. „SLA decomposition: Translating service level objectives to system level thresholds“. In: *Autonomic Computing, 2007. ICAC'07. Fourth International Conference on*. IEEE. 2007, pp. 3–3 (cit. on p. 71).

[22] Elaine Cheong, Edward A. Lee, and Yang Zhao. „Viptos: a graphical development and simulation environment for tinyOS-based wireless sensor networks“. In: *Proc. Embedded net. sensor sys., SenSys 2005*. 2005, pp. 302–302 (cit. on p. 46).

[23] Ben Christensen. *RxJava: Reactive Functional Progamming in Java*. `http://github.com/Netflix/RxJava/wiki`. Netflix (cit. on p. 47).

[24] Parallel Universe Co. *Quasar*. `https://github.com/puniverse/quasar` (cit. on p. 8).

[25] Andrew Coles, Amanda Jane Coles, Allan Clark, and Stephen Gilmore. „Cost-Sensitive Concurrent Planning Under Duration Uncertainty for Service-Level Agreements.“ In: *ICAPS*. 2011 (cit. on p. 71).

[26] Marco Comuzzi, Constantinos Kotsokalis, George Spanoudakis, and Ramin Yahyapour. „Establishing and monitoring SLAs in complex service based systems“. In: *Web Services, 2009. ICWS 2009. IEEE International Conference on*. IEEE. 2009, pp. 783–790 (cit. on p. 71).

[27] *Concurrent Haskell*. `http://en.wikipedia.org/wiki/Concurrent_Haskell` (cit. on p. 7).

[28] *Concurrenty in Rust Language*. `https://doc.rust-lang.org/book/concurrency.html` (cit. on p. 7).

[29] „Coordination Models and Languages“. In: vol. 4467. Springer Berlin / Heidelberg, 2007. Chap. Actors That Unify Threads and Events, pp. 171–190 (cit. on pp. 23, 39).

[30] Fábio Corrêa. „Actors in a new “highly parallel" world“. In: *Proc. Warm Up Workshop for ACM/IEEE ICSE 2010*. WUP ’09. ACM, 2009, pp. 21–24 (cit. on pp. 24, 40).

[31] Markus Dahm. „Byte code engineering“. In: *JIT’99*. Springer, 1999, pp. 267–277 (cit. on p. 8).

[32] J. Eker, J.W. Janneck, E.A. Lee, et al. „Taming heterogeneity - the Ptolemy approach“. In: *Proceedings of the IEEE* 91.1 (), pp. 127–144 (cit. on p. 40).

[33] *Elixir Agents*. `http://elixir-lang.org/getting-started/mix-otp/agent.html` (cit. on p. 7).

[34] *Erlang Concurrent Programming*. `http://www.erlang.org/course/concurrent_programming.html` (cit. on p. 7).

[35] Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. „Schedulability Analysis Using Two Clocks“. In: vol. 2619. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 224–239 (cit. on p. 41).

[36] Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. „Task automata: Schedulability, decidability and undecidability“. In: *Information and Computation* 205.8 (2007), pp. 1149–1172 (cit. on pp. 70, 77, 80).

[37] Martin Fowler. *LMAX Architecture*. `http://martinfowler.com/articles/lmax.html`. Martin Fowler. 2011 (cit. on p. 48).

[38] Gamma, Erich and Helm, Richard and Johnson, Ralph and Vlissides, John. „Design Patterns: Abstraction and Reuse of Object-Oriented Design“. In: *ECOOP ’93 – Object-Oriented Programming*. Vol. 707. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1993, pp. 406–431 (cit. on p. 45).

[39] Elena Giachino, Carlo A. Grazia, Cosimo Laneve, Michael Lienhardt, and Peter Y. H. Wong. „Deadlock Analysis of Concurrent Objects: Theory and Practice“. In: *IFM*. 2013, pp. 394–411 (cit. on p. 59).

[40] Stephen Gilmore, László Gönczy, Nora Koch, et al. „Non-functional properties in the model-driven development of service-oriented systems“. In: *Software & Systems Modeling* 10.3 (2011), pp. 287–311 (cit. on p. 71).

[41] *Go Statement Specification*. `https://golang.org/ref/spec#Go_statements` (cit. on p. 7).

[42] Brian Goetz. *JSR 335, Lambda Expressions for the Java Programming Language*. `http://jcp.org/en/jsr/detail?id=335`. Oracle. 2013 (cit. on p. 51).

[43] Brian Goetz. *Lambda: A Peek Under The Hood*. JAX London. Oracle. 2012 (cit. on p. 52).

[44] Brian Goetz. *Lambda Expression Translation in Java 8*. `http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html`. Oracle. 2013 (cit. on p. 51).

[45] James Gosling. *The Java language specification*. Addison-Wesley Professional, 2000 (cit. on p. 4).

[46] Scott F. Smith Gul A. Agha Ian A. Mason and Carolyn L. Talcott. „A foundation for actor computation“. In: *Journal of Functional Programming* 7 (1997), pp. 1–72 (cit. on p. 27).

[47] Reiner Hähnle, Michiel Helvensteijn, Einar Broch Johnsen, et al. „HATS Abstract Behavioral Specification: The Architectural View“. In: *FMCO*. 2011, pp. 109–132 (cit. on p. 4).

[48] Philipp Haller. „On the integration of the actor model in mainstream technologies: the Scala perspective“. In: *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*. ACM. 2012, pp. 1–6 (cit. on p. 47).

[49] Philipp Haller and Martin Odersky. „Scala Actors: Unifying thread-based and event-based programming“. In: *Theoretical Computer Science* 410.2-3 (2009), pp. 202–220 (cit. on pp. 7, 46, 47).

[50] Philipp Haller and Martin Odersky. „Scala Actors: Unifying thread-based and event-based programming“. In: *Theoretical Computer Science* 410.2-3 (2009), pp. 202–220 (cit. on pp. 8, 27, 39).

[51] Philipp Haller and Martin Odersky. „Scala Actors: Unifying thread-based and event-based programming“. In: *Theoretical Computer Science* 410.2–3 (2009), pp. 202 –220 (cit. on p. 23).

[52] Johannes Henkel and Amer Diwan. „Discovering algebraic specifications from Java classes“. In: *ECOOP 2003–Object-Oriented Programming*. Springer, 2003, pp. 431–456 (cit. on p. 4).

[53] Carl Hewitt. „Procedural Embedding of knowledge in Planner“. In: *Proc. the 2nd International Joint Conference on Artificial Intelligence*. 1971, pp. 167–184 (cit. on p. 45).

[54] Carl Hewitt. „What Is Commitment? Physical, Organizational, and Social (Revised)“. In: *Proc. Coordination, Organizations, Institutions, and Norms in Agent Systems II*. LNCS Series. Springer, 2007, pp. 293–307 (cit. on p. 46).

[55] Giles Hogben and Alain Pannetrat. „Mutant Apples: A Critical Examination of Cloud SLA Availability Definitions“. In: *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*. Vol. 1. IEEE. 2013, pp. 379–386 (cit. on pp. 71, 81).

[56] Inzinger, Christian and Hummer, Waldemar and Satzger, Benjamin and Leitner, Philipp and Dustdar, Shahram. „Generic event-based monitoring and adaptation methodology for heterogeneous distributed systems“. In: *Software – Practice and Experience* (2014) (cit. on p. 70).

[57] Mohammad Mahdi Jaghoori. „Composing Real-Time Concurrent Objects Refinement, Compatibility and Schedulability“. In: *Fundamentals of Software Engineering*. Springer Berlin Heidelberg, 2012, pp. 96–111 (cit. on p. 77).

[58] Mohammad Mahdi Jaghoori. „Time at your service: schedulability analysis of real-time and distributed services“. PhD thesis. Leiden University, 2010 (cit. on pp. 70, 77–79, 81).

[59] Mohammad Mahdi Jaghoori, Ali Movaghar, and Marjan Sirjani. „Modere: the model-checking engine of Rebeca“. In: *Proc. 21st ACM Symposium on Applied Computing*. 2006, pp. 1810–1815 (cit. on p. 25).

[60] Mohammad Mahdi Jaghoori, Frank S. de Boer, Tom Chothia, and Marjan Sirjani. „Schedulability of asynchronous real-time concurrent objects“. In: *J. Log. Algebr. Program.* 78.5 (2009), pp. 402–416 (cit. on p. 59).

[61] JCP. *RTSJ v1 JSR 1*. http://jcp.org/en/jsr/detail?id=1. 1998 (cit. on pp. 27, 40).

[62] JCP. *RTSJ v1.1 JSR 282*. http://jcp.org/en/jsr/detail?id=282. 2005 (cit. on pp. 27, 40).

[63] Einar Broch Johnsen and Olaf Owe. „An Asynchronous Communication Model for Distributed Concurrent Objects“. In: *Software and Systems Modeling* 6.1 (2007), pp. 39–58 (cit. on pp. 4, 27).

[64] Einar Broch Johnsen and Olaf Owe. „An Asynchronous Communication Model for Distributed Concurrent Objects“. In: *Software and Systems Modeling* 6.1 (2007), pp. 39–58 (cit. on pp. 14, 15).

[65] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. „ABS: A core language for abstract behavioral specification“. In: *Formal Methods for Components and Objects*. Springer. 2012, pp. 142–164 (cit. on pp. 4, 5, 46, 50, 70, 73).

[66] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. „ABS: A Core Language for Abstract Behavioral Specification“. In: *Formal Methods for Components and Objects*. Vol. 6957. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 142–164 (cit. on p. 39).

[67] Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa. „Modeling resource-aware virtualized applications for the cloud in Real-Time ABS“. In: *Formal Methods and Software Engineering*. Springer, 2012, pp. 71–86 (cit. on pp. 4, 60, 71, 74, 77).

[68] *JSR 166: Java concurrency utilities*. http://www.jcp.org/jsr/detail/166.jsp (cit. on pp. 4, 53, 56).

[69] Rajesh K. Karmani, Amin Shali, and Gul Agha. „Actor frameworks for the JVM platform: a comparative analysis“. In: *Proc. 7th International Conference on Principles and Practice of Programming in Java*. PPPJ '09. ACM, 2009, pp. 11–20 (cit. on pp. 3, 23, 24, 40).

[70] Rajesh K. Karmani, Amin Shali, and Gul Agha. „Actor frameworks for the JVM platform: a comparative analysis“. In: *Proc. Principles and Practice of Prog. in Java (PPPJ'09)*. ACM, 2009, pp. 11–20 (cit. on pp. 8, 46, 47, 55).

[71] Alan Kay. *Alan Kay on Messaging*. http://c2.com/cgi/wiki?AlanKayOnMessaging (cit. on p. 3).

[72] Alan Kay. *Dr. Alan Kay on the Meaning of "Object-Oriented Programming"*. http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en (cit. on p. 3).

[73] Keven T Kearney, Francesco Torelli, and Constantinos Kotsokalis. „SLA*: An abstract syntax for Service Level Agreements“. In: *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on*. IEEE. 2010, pp. 217–224 (cit. on p. 82).

[74] Alexander Keller and Heiko Ludwig. „The WSLA framework: Specifying and monitoring service level agreements for web services“. In: *Journal of Network and Systems Management* 11.1 (2003), pp. 57–81 (cit. on p. 71).

[75] Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. „Efficient verification of real-time systems: compact data structure and state-space reduction“. In: *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*. IEEE. 1997, pp. 14–24 (cit. on p. 77).

[76] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. „Actor-Oriented Design of Embedded Hardware and Software Systems“. In: *Journal of Circuits, Systems and Computers* 12.03 (2003), pp. 231–260 (cit. on p. 40).

[77] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. „Actor-Oriented Design of Embedded Hardware and Software Systems“. In: *Journal of Circuits, Systems, and Computers* 12.3 (2003), pp. 231–260 (cit. on p. 46).

[78] Edward A. Lee, Xiaojun Liu, and Stephen Neuendorffer. „Classes and inheritance in actor-oriented design“. In: *ACM Transactions in Embedded Computing Systems* 8.4 (2009) (cit. on p. 46).

[79] Xavier Leroy. „Java bytecode verification: algorithms and formalizations“. In: *Journal of Automated Reasoning* 30.3-4 (2003), pp. 235–269 (cit. on p. 8).

[80] Xavier Leroy. „Java bytecode verification: an overview“. In: *Computer aided verification*. Springer. 2001, pp. 265–285 (cit. on p. 8).

[81] Xavier Logean, Falk Dietrich, Hayk Karamyan, and Shawn Koppenhöfer. „Run-time monitoring of distributed applications“. In: *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*. Middleware '98. 1998, pp. 459–474 (cit. on p. 70).

[82] K Lundin. „Inside the Erlang VM, focusing on SMP“. 2008 (cit. on pp. 24, 40).

[83] Khaled Mahbub, George Spanoudakis, and Theocharis Tsigkritis. „Translation of SLAs into monitoring specifications“. In: *Service Level Agreements for Cloud Computing*. Springer, 2011, pp. 79–101 (cit. on p. 71).

[84] Cláudio Maia, Luís Nogueira, and Luís Miguel Pinho. „Combining RTSJ with Fork/Join: a priority-based model“. In: *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*. JTRES '11. York, United Kingdom: ACM, 2011, pp. 82–86 (cit. on p. 40).

[85] Nicholas D Matsakis and Felix S Klock II. „The rust language“. In: *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology*. ACM. 2014, pp. 103–104 (cit. on p. 7).

[86] Bertrand Meyer. „Object oriented software construction“. In: (1988) (cit. on p. 3).

[87] Meyer, B. „Applying “design by contract”“. In: *Computer* 25.10 (1992), pp. 40–51 (cit. on p. 46).

[88] Microsoft. *Reactive Extensions*. `https://rx.codeplex.com/`. Microsoft (cit. on p. 47).

[89] Brian Nielsen and Gul Agha. „Semantics for an Actor-Based Real-Time Language“. In: *In Fourth International Workshop on Parallel and Distributed Real-Time Systems*. 1996 (cit. on p. 39).

[90] Behrooz Nobakht, Frank S. de Boer, Mohammad Mahdi Jaghoori, and Rudolf Schlatte. „Programming and Deployment of Active Objects with application-Level Scheduling“. In: (2012). ACM SAC (cit. on p. 39).

[91] Behrooz Nobakht, Frank S. de Boer, Mohammad Mahdi Jaghoori, and Rudolf Schlatte. „Programming and deployment of active objects with application-level scheduling“. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. SAC '12. ACM, 2012, pp. 1883–1888 (cit. on pp. 56, 77).

[92] Behrooz Nobakht, Frank S de Boer, and Mohammad Mahdi Jaghoori. „The Future of a Missed Deadline“. In: *Coordination Models and Languages*. Springer. 2013, pp. 181–195 (cit. on pp. 70, 77).

[93] Nathaniel Nystrom, Michael R Clarkson, and Andrew C Myers. „Polyglot: An extensible compiler framework for Java“. In: *Compiler Construction*. Springer. 2003, pp. 138–152 (cit. on p. 4).

[94] ObjectWeb. *ASM*. `http://asm.ow2.org/`. ObjectWeb (cit. on p. 8).

[95] Martin Odersky and Philip Wadler. „Pizza into Java: Translating theory into practice“. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1997, pp. 146–159 (cit. on p. 4).

[96] Martin Odersky, Philippe Altherr, Vincent Cremet, et al. *The Scala language specification*. 2004 (cit. on p. 4).

[97] Terence Parr. *ANTLR*. `http://antlr.org/` (cit. on p. 35).

[98] Gordon D Plotkin. „The origins of structural operational semantics“. In: *The Journal of Logic and Algebraic Programming* 60-61.0 (2004), pp. 3–15 (cit. on pp. 4, 28).

[99] Franco Raimondi, James Skene, and Wolfgang Emmerich. „Efficient online monitoring of web-service SLAs“. In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM. 2008, pp. 170–180 (cit. on p. 71).

[100] Mike Rettig. *Jetlang Library*. `http://code.google.com/p/jetlang/`. 2008 (cit. on pp. 24, 40).

[101] John Rose. *JSR 292: Supporting Dynamically Typed Languages on the Java Platform*. `http://jcp.org/en/jsr/detail?id=292`. Oracle. 2011 (cit. on pp. 52, 56).

[102] Aleksey Shipilev. *JMH: Java Microbenchmark Harness*. `http://openjdk.java.net/projects/code-tools/jmh/`. Oracle (cit. on p. 58).

[103] Sriram Srinivasan. *Kilim*. `http://www.malhar.net/sriram/kilim/` (cit. on p. 8).

[104] Sriram Srinivasan and Alan Mycroft. „Kilim: Isolation-typed actors for Java“. In: *ECOOP 2008–Object-Oriented Programming*. Springer, 2008, pp. 104–128 (cit. on pp. 8, 47).

[105] Sriram Srinivasan and Alan Mycroft. „Kilim: Isolation-Typed Actors for Java“. In: *ECOOP 2008 – Object-Oriented Programming*. Vol. 5142. Springer Berlin / Heidelberg, 2008, pp. 104–128 (cit. on p. 24).

[106] Sriram Srinivasan and Alan Mycroft. „Kilim: Isolation-Typed Actors for Java“. In: *ECOOP 2008 - Object-Oriented Programming*. Vol. 5142. Springer Berlin / Heidelberg, 2008, pp. 104–128 (cit. on pp. 27, 40).

[107] Typesafe. *Akka*. `http://akka.io/`. Typesafe (cit. on pp. 8, 48, 58).

[108] TypeSafe. *Akka*. `http://akka.io/`. 2010 (cit. on p. 39).

[109] Jose Valim. *Elixir*. `http://elixir-lang.org/`. Elixir (cit. on pp. 7, 47).

[110] Raja Vallée-Rai, Phong Co, Etienne Gagnon, et al. „Soot-a Java bytecode optimization framework“. In: *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press. 1999, p. 13 (cit. on p. 8).

[111] Carlos Varela and Gul Agha. „Programming dynamically reconfigurable open systems with SALSA“. In: *SIGPLAN Not.* 36 (12 2001), pp. 20–34 (cit. on pp. 24, 27, 40).

[112] Carlos A Varela, Gul Agha, Wei-Jen Wang, et al. „The SALSA Programming Language 1.1.2 Release Tutorial“. In: *Dept. of Computer Science, RPI, Tech. Rep* (2007), pp. 07–12 (cit. on p. 47).

[113] Wirfs-Brock, Rebecca J. and Johnson, Ralph E. „Surveying Current Research in Object-oriented Design“. In: *Commun. ACM* 33.9 (1990), pp. 104–124 (cit. on p. 45).

[114] Peter Y. H. Wong, Richard Bubel, Frank S. de Boer, et al. „Testing abstract behavioral specifications“. In: *STTT* 17.1 (2015), pp. 107–119 (cit. on p. 70).

[115] Jim Woodcock, Ana Cavalcanti, John Fitzgerald, Simon Foster, and Peter Gorm Larsen. „Contracts in CML“. In: *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*. Springer, 2014, pp. 54–73 (cit. on p. 71).

[116] Ingrid Chieh Yu, Einar Broch Johnsen, and Olaf Owe. „Type-Safe Runtime Class Upgrades in Creol“. In: *Proc. FMOODS'06*. Vol. 4037. LNCS. Springer-Verlag, June 2006, pp. 202–217 (cit. on p. 15).

[117] Alexandros Zerzelidis and Andy Wellings. „A framework for flexible scheduling in the RTSJ“. In: *ACM Trans. Embed. Comput. Syst.* 10.1 (2010), 3:1–3:44 (cit. on p. 40).

List of Figures

List of Tables

Colophon

This thesis was typeset with LaTeX 2$_\varepsilon$. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.

Declaration

You can put your declaration here, to declare that you have completed your work solely and only with the help of the references you mentioned.

City, October 31, 2015

Behrooz Nobakht