Actors at Work

Behrooz Nobakht

October 31, 2015
Version: v1.0

Leiden University

Department of Natural Sciences
Leiden Advanced Institute of Compute Science

Actors at Work

Actors at Work

Behrooz Nobakht

Proefschrift

ter verkrijging van

de graad van doctor aan de Universiteit Leiden

op gezag van de Rector Magnificus prof. dr. C. J. J. M. Stolker,

volgens besluit van het College voor Promoties

te verdedigen op woensdag 18 december 2013

klokke 10.00 uur

door

Behrooz Nobakht

geboren te Tehran, Iran, in 1981

PhD Committee

Promotor:	Prof. Dr. F.S. de Boer	
Co-promotor:	Dr. C. P. T. de Gouw	
Other members:		
	Prof. Dr. F. Arbab	
	Dr. M.M. Bonsangue	
	Prof. Dr. E. B. Johnsen	University of Oslo, Norway
	Prof. Dr. M. Sirjani	Reykjavik University, Iceland
	Dr. P.Y.H. Wong	Travelex, United Kingdom

The work reported in this thesis has been carried out at the Center for Mathematics and Computer Science (CWI) in Amsterdam and Leiden Institute of Advanced Computer Science at Leiden University. This research was supported by the European FP7-231620 project ENVISAGE on Engineering Virtualized Resources.

Copyright © 2016 by Behrooz Nobakht. All rights reserved.

October 31, 2015

Behrooz Nobakht
Actors at Work
Actors at Work, October 31, 2015
Promoter: Prof. Dr. Frank S. de Boer
Cover Design: Ehsan K. Heshmati

Built on `(None)` from `(None)` at
`https://github.com/nobeh/thesis`

Leiden University

Leiden Advanced Institute of Compute Science
Department of Natural Sciences
Niels Bohrweg 1
2333 CA and Leiden

Contents

Part I

Introduction

1 Introduction

Object-oriented programming [**booch1982object**, **meyer1988object**] has been of one the dominant paradigms for software engineering. Object orientation provides principles for abstraction and encapsulation. SIMULA 67 [**Dahl:1968:simula**] introduced notions of classes, subclasses, and virtual procedures. In the main block of a program, objects are created, then their procedures are called. 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. This model of interaction was not the intention of the pioneers of the paradigm: object interactions were meant to be *messages* among objects and objects behaved as autonomous entities possibly on remote locations on a network; Alan Kay clarified later [**alank1**, **alank2**]. On the contrary, almost all the object-oriented languages at hand have followed the blocking and synchronous model of messaging. Object-oriented programming has inspired another paradigm: the actor model.

One of the fundamental elements of the actor model [**actors:agha**, **agha97**] is *asynchronous* message passing. In that approach, interactions between objects are modelled as non-blocking messages. One object, the sender, communicates a message to the other object, the receiver. In contrast with abstraction in object orientation, a message can be any object and is not bound by any interface. At the receiver side, a message is matched with possible patterns and when a match is found, the message is processed by the receiver. Actor model features location transparency; i.e. the physical location of objects is not visible to other objects. A system is composed of objects that communicate through messages.

A considerable amount of research has been carried out to combine object-oriented programming with the actor model [**philippsen2000survey**]. Language extensions, libraries, and even new programming languages are the outcomes of such research. For example, [**actor_frameworks_jvm:agha**] presents a comparative analysis of such research for Java and JVM languages.

The rapid growth of hardware added a new aspect to create a triangle: object orientation, actor model, and *concurrency*. Concurrency motivates utilizing computational power to make programs run faster. One goal has been to combine concurrency with object-oriented programming. However, concurrency makes it harder to verify programs in terms of correctness of runtime behavior [**Herlihy:1990:linear**,

johnsen:history, **agha:predictive:safety**]. Concurrency brings about two aspects in relation to object orientation: *communication* and *execution*.

In a concurrent setting, coroutines [**conway1963design**, **taocp:knuth**] enable interactions with collaborative pre-emption. A coroutine allows multiple entry points for suspending and resuming execution at certain locations. An instance of a coroutine spawns a new process with its own state. A coroutine has no *return* statement, but it may explicitly *yield* to another coroutine (transfer control of the execution). The yield relation is symmetric and not similar to that between caller and callee. This fundamental property makes compositional concurrency of coroutines straightforward. Coroutines are not originally established in the object-oriented paradigm. Object orientation is based on caller-callee interaction, which is asymmetric. The caller invokes a method in the callee and *blocks* until the corresponding *return* occurs in the method of the callee. Object orientation focuses on an object-view with caller-callee interaction whereas coroutines focus on a process-view with transfer of control.

In a concurrent setting with objects, multi-threading has been a common approach to provide an object-oriented concurrency model. A thread holds a single stack of synchronous method calls. Execution in a thread is sequential. The stack has a starting location (i.e. start of invocation) and a finish location (i.e. the return point). In a caller-callee setting, an object calls methods from other objects. If multiple threads execute simultaneously in an object, their instructions are interleaved in an uncontrolled manner. Future values are used to hold the eventual return value of a method at the call site. The unit of interleaving in coroutines is coarse whereas multi-threading is based on implicit scheduling of method invocations. Furthermore, interactions in both coroutines and multi-threading are blocking and synchronous. In contrast, the actor model relies on asynchronous communication (which is non-blocking) as one of its core elements.

In the actor model, all communication is asynchronous. The unit of communication is a *message*. A queue of messages (the inbox) is utilized among actors in the system. The notion of a message is not bound to a specific definition or object interface. When an actor receives a message, it may use pattern matching techniques to extract the content of the message. When the actor completes the processing of a message, it may decide to reply to the message by sending another message. On the other hand, the actor model works based on run-to-completion style of execution. While a message is processed, an actor cannot be pre-empted or intentionally yield to allow other actors in the system to make progress.

A question naturally rises: how to bring coroutine-style execution (co-operative scheduling) to an object-oriented setting with asynchronous communication through

messages bound to object interfaces? This question is the main problem statement of this thesis; cf. Table 1.1.

Paradigm	Abstraction	Communication	Execution
OOP	Class/Object	Sync. Method Call	Blocking + Multi-threading
Actor Model	Actor	Async. Messages	Blocking + RTC
Coroutines	n.a.	*yield*	Co-operative scheduling

Table 1.1: Considering the different aspects, the problem statement of thesis focuses to combine abstraction from OOP with asynchronous messages from Actor model but in a bounded context. In addition, this thesis focuses to bring in co-operative scheduling into the context of object-oriented programming.

There already exist modelling languages that focus on the problem statement. Rebeca [**sirjani2002simulation**, **sirjani2004formal**, **sirjani2007rebeca**] is a modelling language for reactive and concurrent systems. In Rebeca, a number of reactive objects (*rebecs*) interact at runtime. Rebecs interactions is based on asynchronous message passing that leads to a method invocation. In Rebeca, each rebec has its own unique thread of control and an unbounded queue of messages. Rebeca uses run-to-completion execution and does not support future values.

ABS [**johnsen2012abs**, **hahnlehjlssw11**] is a modelling language for concurrent objects and distributed systems. ABS uses asynchronous communication among objects. A message in ABS is bound to a method invocation; this defines the interface of the sent messages. In addition, ABS supports future values in its asynchronous communication. ABS introduces `release` semantics that is based on co-operative scheduling of objects; i.e. similar to that of *yield* in coroutines [**creol:broch:owe**]. The ABS semantics is completely formalized by a structural operational semantics [**plotkin:sos**]. This allows ABS models to take advantage of a wide range of static and dynamic analysis techniques. Co-operative scheduling in ABS has been additionally extended for real-time scheduling with priorities and time constraints [**bjork2013:rtabs**, **johnsen2012modeling**]. All the above characteristics make the ABS language an attractive choice if it can be used as a programming language at industrial and business scale. ABS has mainly developed as a modelling language.

Considering the mainstream programming languages [1], Java [**gosling2000java**] is one of the most commonly-used. Since Java 1.5 and the rise of the concurrency API in Java JSR-166 [**jsr166**], substantial effort focuses on how concurrency can be improved in Java. However, not all development of Java libraries and frameworks proceeds based on a rigorous formal semantics. Therefore, this gives rise to issues and challenges in terms of correctness, semantic preservation, and reason-

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

ing. In addition, the Java language specification (JLS) [**gosling2000java**] clearly reveals that the Java language is not compatible and ready to be extended as a functional language that supports algebraic data types. This has driven research and development that focuses on extending Java with support for functional programming [**odersky1997pizza**, **henkel2003discovering**, **nystrom2003polyglot**, **bracha1998making**]. Scala [**odersky2004scala**] is a result of such research. It is a dynamic, functional, and object-oriented language on JVM.

One core challenge is how to create a mapping of coroutines to multi-threading in Java. Supporting coroutines in Java can be mostly classified in two major categories. One category relies on a *modified* JVM platform (e.g. Da Vinci JVM) in the implementation of thread stack and state such as [**Stadler:2009:LCJ:1596655.1596679**], [**Stadler:2010:ECJ:1852761.1852765**] and [**Liu:2006:II:1111320.1111063**]. The other category involves libraries such as Commons Javaflow[2] or Coroutines[3] that utilize byte-code modification [**dahm1999byte**] at runtime in JVM. In this research, we did not intend to use any of the two above approaches. Custom or modified JVM platform implementations are not mainstream and not officially supported by the Java team. Research and development of a modified JVM requires explicit and periodic maintenance and upgrade effort. Moreover, as byte-code modification changes the byte-code of a running program or inserts new byte-code into JVM at runtime, this complicates reasoning and correctness analysis/verification [**leroy2001java**, **leroy2003java**]. We rely on the mainstream JVM platform released by Java team at Oracle; in addition, we do not use byte-code engineering.

Another, straightforward way to support coroutines in Java multi-threading is that since a thread owns a single stack, we can translate every invocation (entry point) in a coroutine to a new thread in Java [**SchaferP10**, **schafer2010programming**]. This naive approach unsurprisingly leads to a poor performance; the number of threads is not scalable at runtime based on resource limitations in JVM when the number of objects increase (cf. Chapter 2). Therefore, it is reasonable to use a pool of threads (JSR-166 [**jsr166**]) to direct the execution of all method invocations (messages). However, to deploy a message onto a thread for execution, the message is required to be abstracted in a generic way. We utilize Java 8 features (JSR 335 [**jsr335**] and JSR 292 [**jsr292:invokedyn**]) to abstract a message as a data structure expressed in a lambda expression (cf. Chapter 4).

The core contributions of this thesis target the intersection of object orientation, actor model, and concurrency. We choose Java as the main programming language and as one of the mainstream object-oriented languages. We formalize a subset of Java and its concurrency API [**jsr166**] to facilitate formal verification and reasoning about

[2] `http://commons.apache.org/sandbox/commons-javaflow/`
[3] `https://github.com/offbynull/coroutines`

it. We create an abstract mapping from a concurrent-object modelling language, ABS [**johnsen2012abs**], to the programming semantics of concurrent Java (cf. Chapter 3). We provide the formal semantics of the mapping and runtime properties of the concurrency layer including deadlines and scheduling policies (cf. Chapter 2). We provide an implementation of ABS concurrency layer as a Java API library and framework utilizing the latest language additions in Java 8 [**jsr335:lambda:translation**] (cf. Chapter 4). We design and implement a runtime monitoring framework, JMSeq, to verify the the correct ordering of execution of methods through code annotations in JVM (cf. Chapter 5). In addition, we design a large-scale monitoring system as a real-world application; the monitoring system is built with ABS concurrent objects and formal semantics that leverages schedulability analysis to verify correctness of the monitors [**fersman2007task**] (cf. Chapter 6).

1.1 Objectives and Architecture

In this section, we present a high-level overview of design goals that we pursue in this research.

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 aim to deliver ABS semantics and features in a polyglot approach. The programmer develops models with ABS that *partially* take advantage of the target language features (e.g. Java). This approach is also referred to as *Foreign Function Interface* in the context of ABS modelling.

Listing 1: Using Java in ABS

```
1 java.util.List<String> params = new java.util.ArrayList<>();    // Java
2 myObj ! doSomething(params);                                     // ABS
```

Listing 1 shows a snippet of ABS code that uses `java.util.ArrayList` as a data structure. Ideally in ABS, the programmer is able to directly use the libraries and API from Java. This removes the necessity to redundantly repeat definition of common data structures and API at ABS.

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. The goal is to ensure the actor system scales in perfor-

mance with least influence from the number of asynchronous messages delivered in the system.

Modularity We approach ABS modelling and development with a component-based or modular software engineering practices. The scope of the research spans a number of layers around ABS language:

- *Compiling ABS to a target programming language* One first objective is to compile an ABS model to a target programming language. Target languages potentially include mainstream programming languages such as Java, Erlang, Haskell, and Scala. We propose a new architecture for the ABS tool-set and engineering that enables different programming languages to utilize the same architecture.

- *Using ABS concurrency as an API in an existing programming language* The ABS language syntax and semantics are formalized precisely and rigorously by a structural operational semantics [**johnsen2012abs**]. 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 the ABS language semantics. In addition, such an ABS programming API should be *verifiable*. If a mapping from ABS to a programming API is provided, a programmer is able to take advantage of ABS semantics without directly programming in ABS. This enables industry users of mainstream languages to model their systems in ABS semantics using the programming languages and platforms they are already familiar with.

- *Modelling in ABS* ABS language provides rigorous semantics to model concurrent and distributed systems. For practical reasons, it is important that the user (that can be a programmer, an analyst, or a researcher) has 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.

The above objectives and design principles are realized by the modular architecture presented in Figure 1.1.

1.2 Literature Overview

We briefly discuss related work in the context of programming languages, actor model, and concurrency. In the overview, we distinguish two levels; one is at the

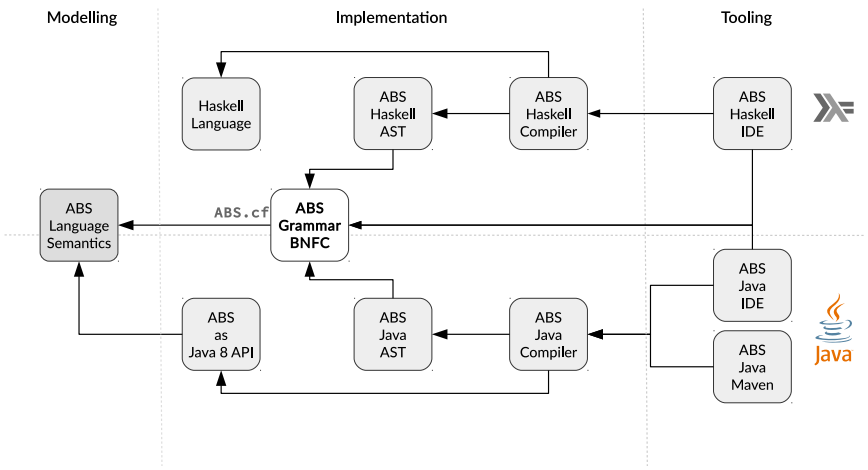


Figure 1.1: General Architecture of ABS API and Java Language Backend

level of the programming languages and the other is for the external (third-party) libraries developed for programming languages.

1.2.1 Programming Languages

In this section, we briefly provide an overview of the programming languages that have targeted similar problem statements. Various programming languages, in the past decade, have emerged to provide an actor-based model of asynchronous message passing [**philippsen2000survey**]. We observe different classes of actor model and concurrent model of programming. Table 1.2 presents a summary.

First-Class Citizen Languages in which the actor model is by-design part of the syntax and semantics of the language. Pony [**ponylang**, **ClebschD13**] targets high-performance computing using actor models and shared memory. When actor model is part of the design of a language, formal verification and correctness of actor semantics at runtime yields naturally from the language semantics at runtime.

Implicit By Design Refers to languages that have no explicit notion of actor models in their syntax or semantics, but do provide fundamental constructs for concurrency and asynchronous message passing. Thus, it becomes an easy task in this kind of programming language 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 provided by an external library.

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

Language	Abstraction	Type
Erlang[**erlang:armstrong**, **erlang:actor**]	Process	Implicit By Design
Elixir[**elixir**, **elixir:actor**]	Agent	Implicit By Design
Haskell[**con_haskell:wiki**]	forkIO & MVars	Implicit By Design
Go[**go:actor**]	Goroutine	Implicit By Design
Rust[**rust:2014**, **rust:actor**]	Send & Sync	Implicit By Design
Scala[**haller09tcs**]	Akka Actors [4]	External Library
Pony[**ponylang**, **ClebschD13**]	actor	First-Class Citizen

Table 1.2: Actor Model Support in Programming Languages

Library	Technique	JVM Language
Killim[**srinivasan2008kilim**, **kilim**]	Byte-Code Modification	Java
Quasar[**quasar**]	Byte-Code Modification, Java 8	Clojure, Java
Akka[**akka**, **scala:actors:ordersky**]	Scala Byte-Code on JVM	Scala, Java

Table 1.3: Actor programming libraries in Java

1.2.2 Frameworks and Libraries

Since programming languages faced challenges to provide the necessary syntax and semantics for actor model and concurrency at the level of the language, many libraries and frameworks aim to fill this gap Table 1.3 presents a summary. 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 enumerate frameworks and libraries for Java [5].

One of the main techniques used in libraries to deliver actor programming in Java is byte-code engineering [**dahm1999byte**, **bruneton2002asm**, **asm**]. 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 [**vallee1999soot**], it introduces other challenges regarding the verification of the running byte-code [**leroy2001java**, **leroy2003java**].

[5] A more comprehensive list can be obtained at [**KarmaniSA09**] and `https://en.wikipedia.org/wiki/Actor_model#Programming_with_Actors`

1.3 Organization and Contributions

Based on the problem statement and approach taken in this research, Table 1.4 summarizes the structure of this text:

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
Monitoring method call sequences using annotations	Application (Part IV)	Chapter 5
Design and implementation of a massive-scale monitoring system based on ABS API in Java	Application (Part IV)	Chapter 6

Table 1.4: Thesis Contributions Summary

In addition, Table 1.5 summarizes the conference and journal publications as a result of this research:

Topic	Proceedings / Journal	Year
Programming and Deployment of Active Objects with Application-Level Scheduling	ACM SAC 2012, Pages 1883–1888	2012
The Future of a Missed Deadline	COORD 2013, Pages 181–195	2013
Monitoring Method Call Sequences using Annotations	Journal of Science of Computer Programming, 2014, Volume 94, Part 3, Pages 362–378	2014
Programming with actors in Java 8	ISoLA 2014, Pages 37–53	2014
Formal verification of service level agreements through distributed monitoring	ESOCC 2015, Pages 125–140	2015

Table 1.5: Actors At Work – Conference and Journal Publications

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.

Conference Publication *Proceedings of the 27th Annual ACM Symposium on Applied Computing – ACM SAC 2012, Pages 1883–1888, DOI 10.1145/2245276.2232086*

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.

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

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 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 [**creol:broch_owe**, **mpd:andrews**]. 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 [**creol:broch_owe**] 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 [**yu06fmoods**]. 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

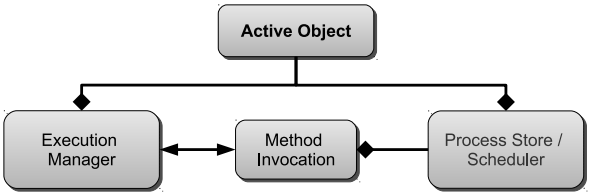


Figure 2.1: $\mathcal{C}$risp Architecture: Structural Overview

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:

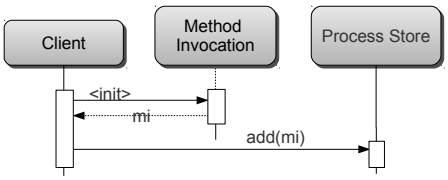


Figure 2.2: Adding the new MethodInvocations are performed on the Client side.

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:

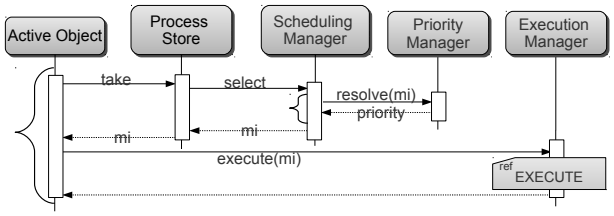


Figure 2.3: An active object selects a method invocation based on its local scheduling policy. After a method finishes execution, the whole scenario is repeated.

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` 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

EXECUTE:

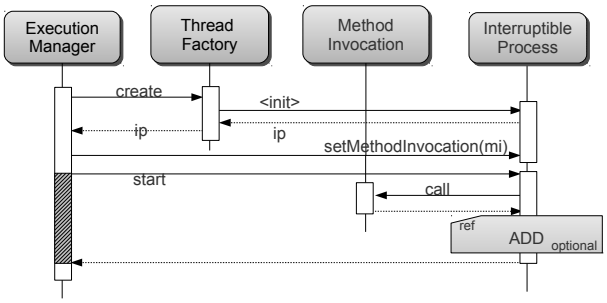


Figure 2.4: A method invocation is executed in an interruptible process. The execution manager thread is blocked while the interruptible process is running.

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

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
```

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 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

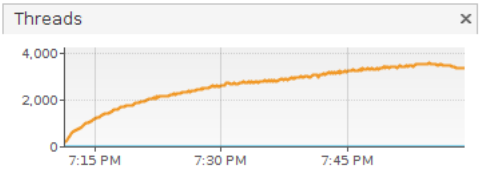


Figure 2.5: Increasing parallelism in $\mathcal{C}$risp for Prime Sieve

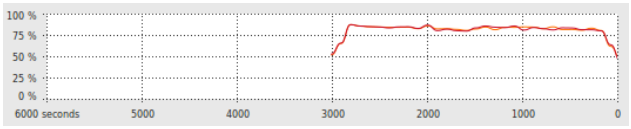


Figure 2.6: Utilizing both CPUs with Prime Sieve in $\mathcal{C}$risp

for every run. To demonstrate this feature of $\mathcal{C}$risp, we collected the minimum stack size needed for every program run in Table 2.1. All $\mathcal{C}$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, $\mathcal{C}$risp provides another interesting feature. It can handle huge number of thread generation if required. Table 2.2 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.

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

The Actor model [**actors:agha**] 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 [**actor_frameworks_jvm:agha**]. 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 [**scala_actors:ordersky**, **coord:ordersky**] 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 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 [**erlang:armstrong**] is a dynamically typed functional language that was developed at Ericsson Computer Science Laboratory with telecommunication purposes [**actors_highly:Correa**]. Recent developments in the deployment of Erlang support the assignment of a scheduler to each processor [**erlang_scheduling**] (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 [**kilim:Srinivasan_Mycroft**], Jetlang [**jetlang**], ActorFoundry [**actor_frameworks_jvm:agha**], and SALSA [**salsa:agha**] among others. In [**actor_frameworks_jvm:agha**], 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

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

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 [**jaghoori_dating**], 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.

2.6 Conclusion

In this paper, we proposed *C*risp as an implementation scheme for application-level scheduling of active objects. *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. *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 *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.

In future, we will focus on thread performance for *C*risp such that thread scalability can be achieved to a certain limit. Additionally, the development of concurrency features on multi-core in *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

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

Table 2.3: Overview of evaluation of challenges

to extend and integrate into our tool set model checking engines such as Modere [**JaghooriMS06**].

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.

Conference Publication *Lecture Notes in Computer Science, Volume 7890, Coordination Models and Languages in 15th conference on Distributed Computing Techniques – COORD 2013, Pages 181–195, DOI 10.1007/978-3-642-38493-6_13*

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) [**jsr1**, **jsr282**] 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 [**fnd:actor:cmpt:agha**] and actor-based programming languages, which have re-emerged in the past few years [**kilim:Srinivasan:Mycroft**, **erlang:armstrong**, **scala:actors:ordersky**, **creol:broch:owe**, **salsa:agha**], provide a different and promising paradigm for concurrency and distributed computing, in which threads are trans-

parently 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 [**BoerCJ07**]. 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 [**plotkin:sos**] 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 5.6. 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 5.6.

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).

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

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.

How to send a message with a deadline? The construct

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

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

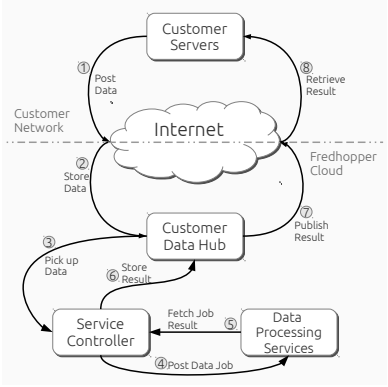


Figure 3.2: Fredhopper's Controller life cycle for remote data processing

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 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 5.6 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 5.6 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 $\mathsf{aborted}$ 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 $\phi(\sigma(f)).\mathsf{aborted} = \mathtt{true}$,
- $\dfrac{(S, \sigma, \phi) \uparrow v}{\mathtt{try}\{S\}\mathtt{catch}(\mathsf{T}\ u)\{S'\} \uparrow v}$ 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:

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

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

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

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

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

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

Timeout is operationally presented by the following transition:

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

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:

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

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 $\tau(\textbf{deadline}) < \sigma(\texttt{time})$. 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 $\phi'(o).\mathsf{aborted} = \texttt{false}$,

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

- $\tau(\texttt{this}) = \mathsf{val}(e_0)(\sigma)$,
- $\tau(x) = \mathsf{val}(e)(\sigma)$, 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.

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

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

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

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

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.

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

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 [**antlr**] 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 5.6, 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 5.6 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.

s_1	s_2
88.71%	94.57%

Table 3.1: Evaluation Results

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). 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 [**crisp-sac**]. This new extension features

- integration of asynchronous messages with deadlines and futures with time-outs;
- 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, [**ACIHHS11**] and [**Nielsen96r**], 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 [**abs:2012**]. 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, [**abs:2012**] does not provide any context or information about the exception or the cause for the failure. Alternatively, [**abs:2012**] describes a way to "compensate" for a failed get operation on future. In [**einar:rt:sched:co**], 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 [**scala:actors:ordersky**, **coord:ordersky**] 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 [**akka:homepage**] extends Scala's actor programming model and as such provides a direct integration with both Java and Scala. Erlang [**erlang:armstrong**] is a dynamically typed functional language that was developed at Ericsson Computer Science Laboratory with telecommunication purposes [**actors_highly:Correa**]. Recent developments in the deployment of Erlang support the assignment of a scheduler to each processor [**erlang_scheduling**] (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 [**kilim:Srinivasan:Mycroft**], Jetlang [**jetlang**], ActorFoundry [**actor_frameworks_jvm:agha**], and SALSA [**salsa:agha**]. In [**actor_frameworks_jvm:agha**], 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 [**maia_rtsj_11**] 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 [**ptolemy:lee**, **actor:lee**] 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 [**jsr1**, **jsr282**]) and Erlang. Specifically in RTSJ, [**zerzel_rtsj**] 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, [**zerzel_rtsj**] 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 [**sched:elena**].

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.

Conference Publication *Lecture Notes in Computer Science, Volume 8803, 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation with Specialized Techniques and Applications – ISoLA 2014, Pages 37–53, DOI 10.1007/978-3-662-45231-8_4*

4.1 Introduction

Java is beyond doubt one of the mainstream object oriented programming languages that supports a *programming to interfaces* discipline [**oop:patterns**, **oop:survey**]. 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

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

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 [**Hewitt69**] is a powerful concept for modeling distributed and concurrent systems [**Agha90**, **agha97**]. Different extensions of actors are proposed in several domains and are claimed to be the most suitable model of computation for many applications [**Hewitt07-commitment**]. Examples of these domains include designing embedded systems [**LeeACtorEmbedded03**, **LeeLN09**], wireless sensor networks [**CheongSensor05**], multi-core programming [**KarmaniSA09**] and delivering cloud services through SaaS or PaaS [**Chang:DAIS07**]. This model of concurrent computation forms the basis of the programming languages Erlang [**Armstrong10Erlang**] and Scala [**haller09tcs**] 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 [**meyer:design**]. 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 [**johnsen2012abs**].

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 [**Armstrong10Erlang**] 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 [**elixir**] 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 [**haller09tcs**] 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 [**haller2012integration**] is a toolkit and runtime for building highly concurrent, distributed, and fault tolerant event-driven applications on the JVM based on actor model.

Kilim [**srinivasan2008kilim**] 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 [**varela2007salsa**, **KarmaniSA09**] 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 [**rxjava**] by Netflix is an implementation of reactive extensions [**rx**] 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 [**lmax**, **lmax:mf**] 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 [**akka**] 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 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 [**johnsen2012abs**]. 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.

Listing 14: main in ABS

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

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.

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

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 [**jsr335**]) 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 [**jsr335**]) 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 [**jsr335:lambda:translation**] 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 [**jsr292:invokedyn**]) 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 [**lambda:perf**]. 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 [**lambda:perf**].

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`.

[3] Such API includes usage of different interfaces and classes in `java.util.concurrent` package [**jsr166**]. The concurrent Java API supports blocking and synchronization features in a high-level that is abstracted from the user.

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 $\texttt{u}_1$, ..., $\texttt{u}_n$ (of the caller) are used to store the values of the Java 8 expressions $\texttt{e}_1$, ..., $\texttt{e}_n$ corresponding to the actual parameters $e_1, \ldots, e_n$. The types $\texttt{T}_i$, $i = 1, \ldots, n$, 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 [**KarmaniSA09**] 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`.

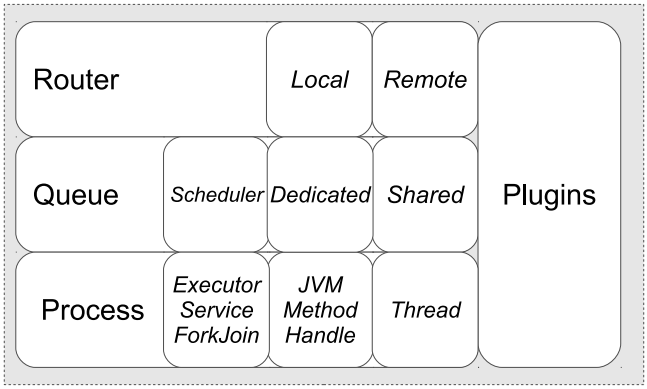


Figure 4.1: Architecture of Actor API in Java 8

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 [**jsr166**] provides different ways to deploy this separation of invocation from execution. We take advantage of Java Method Handles [**jsr292:invokedyn**] 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 [**Nobakht:sched**].

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**, **Agha90**] 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

[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`.

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 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

[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`

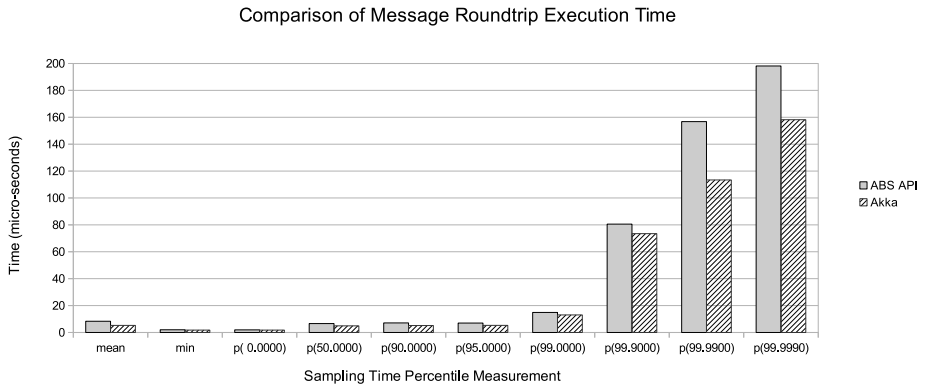


Figure 4.2: Benchmark results of comparing sampling time of message round trips in ABS API and Akka. An example reading of above results is that the time shows for $p(90.0000)$ reads as "message round trips were completed under $10\mu s$ for 90% of the sent messages". The first two columns show the "minimum"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

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 [**akka**] library. The same set of benchmarks are performed in isolation for both of the applications. To perform the benchmarks, we use JMH [**jmh**] 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 [**GiachinoGLLW13**, **JaghooriBCS09**]. 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 [**johnsen2012modeling**] and simulate a distributed environment.

Part IV

Application

5 Monitoring Method Call Sequences using Annotations[1]

Behrooz Nobakht, Frank S. de Boer, Marcello M. Bonsangue, Stijn de Gouw, Mohammad M. Jaghoori

Abstract

In this paper we introduce JMSeq, a Java-based tool for monitoring sequences of method calls. JMSeq provides a simple but expressive language to specify the observables of a Java program in terms of sequences of possibly nested method calls. Similar to many monitoring-oriented environments, verification in JMSeq is done at runtime; unlike all other approaches based on aspect-oriented programming, JMSeq uses code annotation rather than instrumentation, and therefore is suitable for component-based software verification.

Journal Publication *Science of Computer Programming, 2014, Volume 94, Part 3: Selected Best Papers from 7th International Workshop on Formal Aspects of Component Software – FACS 2010, Pages 362–378, DOI 10.1016/j.scico.2013.11.030*

5.1 Introduction

Monitoring refers to the activity of tracking observable information from an execution of a system with the goal of checking it against the constraints imposed by the system specification. The observable information of the monitored system typically includes behavior, input and output, but may also contain quantitative information. A monitoring framework consists of a monitor that extracts the observable behavior from a program execution, and a controller that checks the behavior against a set of specifications. Should an execution violate the constraints imposed by the specification, corrective actions can be taken.

What can be monitored depends, of course, on what can be observed in a system. When the application source code is available, its code can be instrumented to produce informative messages about the execution of an application at run time. New code is inserted for the management of monitoring and verification mechanisms, while preserving the original logic of the application. Bytecode instrumentation is another technique widely used for monitoring and tracing execution of Java programs. Bytecode instrumentation consists of inserting bytecode into classes either at compile-time or at runtime. It is typically preferred to source code instrumentation

[1]This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (`http://www.hats-project.eu/`)

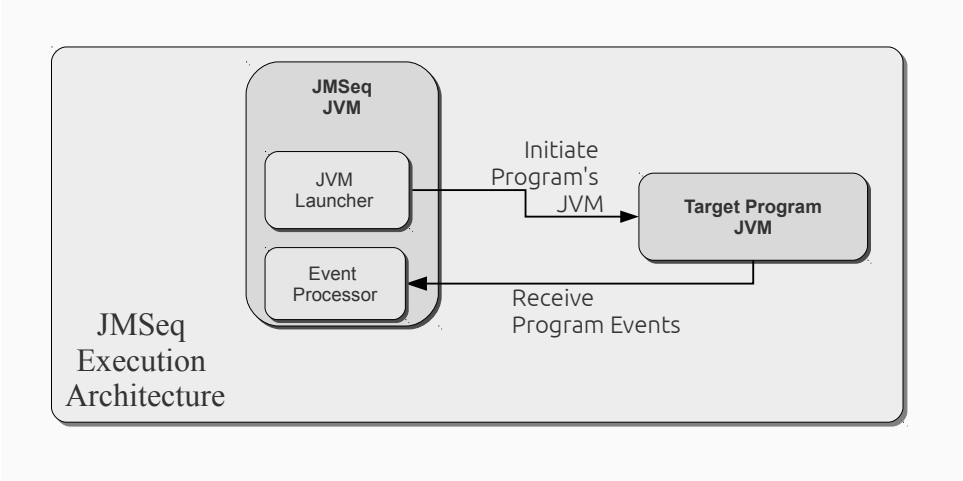


Figure 5.1: JMSeq Execution Architecture

because it can be used to enhance a program temporarily for debugging or analysis purposes, and no source code is required. Main disadvantages however of bytecode instrumentation are the following:

- Although there exist tools for facilitating bytecode instrumentation, building instrumentation in bytecode requires an in-depth knowledge of bytecode; thus, it is *error-prone*.
- Instrumentation by its very nature is an intrusive approach; i.e., the target program is modified either at source level or bytecode level. This makes it unsuitable, for example, for monitoring classes in the Java standard library, which are supposed to remain unchanged. Moreover, if two different bytecode agents simultaneously instrument the same program, they must be compatible; otherwise, the resulting bytecode may contain *conflicts*.
- From the perspective of business security, a verification approach that modifies a program potentially increases its *vulnerability* and as such is of restricted applicability.

In this paper, we introduce JMSeq, a tool for monitoring sequences of method calls in Java programs. JMSeq is based on Java annotations rather than (source code or bytecode) instrumentation. Therefore, it does *not* require access to the source code of the target application. JMSeq detects the annotated methods from the target application at runtime. Furthermore, Java annotations themselves have no direct effect on program execution or logic of the application; they carry metadata that can optionally be used at runtime to perform operations orthogonal to the execution of the main application. The main distinguishing overall characteristic of JMSeq is the integration of the following main features.

Java annotations. JMSeq annotations specify method call sequences in terms of regular expressions of method call signatures. This pure declarative language

is simple and intuitive in specification of different types of sequences of method calls.

JVM decoupling. To ensure that the target program is in *no* way affected or changed during monitoring, JMSeq runs in a separate JVM from the target application's JVM. JMSeq uses Java Platform Debugger API (JPDA) [**JPDA_Home**] to initiate a separate JVM and verify the target application using events from the target JVM. Figure 5.1 displays the high-level execution architecture of JMSeq.

Fail-fast approach. JMSeq terminates execution as soon as a violation of a specification occurs. This approach prevents unsafe behavior, enables the localization of an error, and facilitates a reaction to a failure immediately, thus making the software more robust [**Shore04a**].

An important consequence of these features is that neither the source code (even if available) nor bytecode is modified by JMSeq. Furthermore, JMSeq does not require source code to be present. As such, JMSeq is suited for the specification and verification of both newly built software and legacy systems, including third-party libraries (and even the Java standard library). For example, given the API of a legacy system, its expected behavior can be specified by annotating the methods of a "driver" program which triggers the legacy code. JMSeq then verifies the triggered behavior of the legacy code using these annotations. In the context of an ongoing software development process, JMSeq can also be used as mentioned above by means of a "driver" program but also by directly annotating the "interfaces" of the system API. By annotating the interfaces of a system any service object that implements the interface behavior inherits the annotations and can be verified by JMSeq. In this respect, JMSeq also introduces integration with frameworks such as JUnit (cf. Section 5.5.2).

The rest of the paper is organized as follows. The first three sections describe the usage of JMSeq. Section 5.2 presents the language for specifying sequences of method calls, and Section 5.3 discusses the types of annotations available in JMSeq. These two sections are connected via examples in Section 5.4. Afterwards, implementation of the JMSeq framework is explained in detail in Section 6.6. The practical applicability of JMSeq is demonstrated by means of a case study from industry in Section 5.6 and performance evaluation in Section 5.7. Finally, related work is discussed in Section 5.8. Section 6.7 concludes the paper.

5.2 Method Call Sequence Specification

We consider a component to be a collection of compiled Java classes. The relevant dynamic behavior of a component can be expressed in terms of specific sequences of messages among a small, fixed number of objects. In Figure 5.2, we see two UML message sequence diagrams each describing how, and in which order, the four

objects interact with each other. In this section, we develop a specification language for describing these kinds of interactions.

In addition to specifying the order, a specification language for sequences of method calls needs to distinguish between the following two cases (depicted in Figure 5.2).

Case 1 shows a scenario in which (the call to) `m_c` is nested in `m_b` since `m_c` is called during the activation of `m_b` (i.e., after `m_b` is called and before it returns). Similarly, both `m_b` and `m_c` are nested in `m_a`.

Case 2 represents a method call in which methods from different/same objects are called in a sequential rather than nested manner. In this example, both `m_b` and `m_c` are called by `m_a`.

Typically, a program will need a combination of both cases to specify its dynamic behavior. Specifying only the order of the method calls is not enough, as both cases above have the same order of method calls. It is thus required to have a specification technique that distinguishes between the *method calls* and *method returns*.

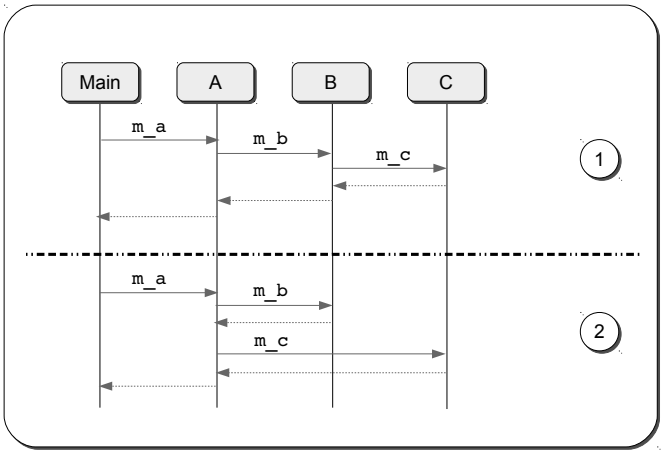


Figure 5.2: Examples of Method Call Sequence Specification

In JMSeq, a specification is written as a post-condition associated with a method. It specifies the set of possible sequences of method calls, or protocol, of an object in the context of the relevant part of its environment. Formally, sequences of method calls can be specified by means of the grammar in Figure 5.3.

A *specification* consists of a sequence of calls that can be repeated an a priori fixed number of times ("$\langle$*Specification*$\rangle^m$"), with $m \geq 0$, one or more times ("$\langle$*Specification*$\rangle$\$"), or zero or more times ("$\langle$*Specification*$\rangle$#"). Although JMSeq does not use aspect-oriented programming in its implementation, we have used the generic method call join points syntax of AspectJ [**kiczales_aspectj**], using for example # to denote the more standard Kleene star operation. Additionally, ($\langle$*Specification*$\rangle$) is a way to

$\langle Specification \rangle ::= \langle Call \rangle \mid \langle Call \rangle\ \langle Specification \rangle \mid$
$\langle Specification \rangle^{\mathrm{m}} \mid \langle Specification \rangle\$ \mid \langle Specification \rangle\# \mid$
$(\langle Specification \rangle)$
$\langle Call \rangle ::= \{\texttt{call}(\langle Signature \rangle)\langle InnerCall \rangle\}\$ \mid$
$\{\texttt{call}(\langle Signature \rangle)\langle InnerCall \rangle\}\# \mid$
$\{\texttt{call}(\langle Signature \rangle)\langle InnerCall \rangle\}^{\mathrm{m}} \mid$
$\{\texttt{call}(\langle Signature \rangle)\langle InnerCall \rangle\} \mid$
$\{\texttt{call}(*)\} \mid$
$< \langle Call \rangle\ ?\ \langle Call \rangle >$
$\langle InnerCall \rangle ::= [\langle Call \rangle]\langle InnerCall \rangle \mid \epsilon$
$\langle Signature \rangle ::= \langle \textit{AspectJ Call Expression Signature} \rangle$

Figure 5.3: Method Sequence Specification Grammar

group specifications to avoid ambiguity. To improve readability of the specifications, grouping is only used when necessary.

A *call* is a call signature followed by a (possibly empty) sequence of inner calls. Each call can be repeated either one or more times, zero or more times, or exactly m-times, for some $m \in \{1, 2, 3, \ldots\}$; i.e., m is a constant. Having the repetition by m does not make the language go beyond the expressiveness of regular expressions. Additionally, the wild card $\{\textbf{call}(*)\}$ denotes a call to an arbitrary method. To support branching, JMSeq also provides $< \langle Call \rangle ? \langle Call \rangle >$ to allow the specification of a choice in a sequence of method executions.

Inner calls are calls that are executed before the outer call returns, i.e., they are nested. We do not have explicit return messages, but rather we specify the scope of a call using brackets. Information about the message call, like the type of the caller and the callee, method name, and the types of the parameters are expressed using a syntax similar to that of AspectJ. Simply put, a call signature is of the form

```
call([ModifiersPattern] TypePattern
        [TypePattern . ] IdPattern (TypePattern | ".." , ... )
        [ throws ThrowsPattern ]
)
```

reflecting the method declarations in Java. The two leftmost patterns specify the method modifiers (like “static” or “public”) and return types. Here `IdPattern` is a pattern specifying the method name. It can possibly be prefixed at the left by a specification of the type of the object called. The method name is followed by a specification of the types of the parameters passed during the call, and possibly by a specification of the exceptions that may be thrown. It implies that JMSeq specification grammar can distinguish the overloaded methods in a class. Patterns may contain wild card symbols “`*`” and “`..`”, used for accepting any value and any number of values, respectively. For example, the call

```
call(* *.A.m_a(..))
```

is denoting a call to method `m_a` of any object of type `A` that is placed in any package, and returning a value of any type. If there is a need to be more specific, a possible restatement of the same specification could be:

```
call(int nl.liacs.jmseq.*.A.m_a(Object, double))
```

Next we give a few examples of correct method sequence specifications. For instance, the specification of the sequence in case 1 of Figure 5.2 is given by:

```
{call(* *.A.m_a(..))[{call(* *.B.m_b(..))[{call(* *.C.m_c(..))}]}]}
```

Here it is important to notice that the call to method `m_b` is internal to `m_a`, and the call to `m_c` is internal to `m_b`. The sequence in case 2 of the same figure would be:

```
{call(* *.A.m_a(..))[{call(* *.B.m_b(..))}][{call(* *.C.m_c(..))}]}
```

where both calls to methods `m_b` and `m_c` are internal to `m_a`.

These two cases depict a fixed sequence of method calls. More interesting are the cases when, for instance, the method `m_a` should be called at least once before any possible method call to `m_b` or `m_c`:

```
{call(* *.A.m_a(..))}$<{call(* *.B.m_b(..))}#?{call(* *.C.m_c(..))}#>
```

Such a specification is used in circumstances where the execution of `m_a` is a prerequisite to the execution of either of `m_b` or `m_c`. It is notable that the actual specifications are regular expressions; for example, they may contain unbounded repetitions and a choice of calls.

5.3 Annotations for method call sequences

Java enables developers to attach extra metadata to their programs using annotations. These metadata do not directly affect the program semantics, but they do affect the way programs are treated by tools and libraries, which can in turn affect the behavior of the running program [**Java_5_Annotations**]. Annotations can be read from source files, class files, or reflectively at runtime. One can define custom annotation types and use them to provide the desired types of metadata for the declarations in his program. In Java, an annotation is a special kind of modifier, and can be used anywhere other modifiers (such as `public`, `static`, or `final`) can be used.

We use annotations as an enabling feature to test components without the need to have their source code. We only use the metadata loaded from the annotations during runtime. These metadata include the specification of the sequences of method

calls based on the grammar discussed in Section 5.2. The metadata is placed on classes or methods in a Java program. Then, they are loaded into JMSeq when it monitors the actual observable behavior of the program at runtime.

JMSeq defines two type of annotations: sequenced object annotations and sequenced method annotations.

5.3.1 Sequenced Object Annotations

To verify the sequence of executions, methods in classes need to be annotated. Thus, JMSeq needs to detect all classes that contain some method carrying an annotation for verification. To improve readability and performance, JMSeq introduces `SequencedObject` annotation. This annotation is a marker for such classes and notifies the annotation metadata loader (cf. Section 5.5.1) that the objects instantiated from the annotated class contain methods that specify a sequence of method executions. The code is given in Listing 27.

Listing 27: SequencedObject Annotation Declaration

```
1 @Retention(RetentionPolicy.RUNTIME)
2 @Target(ElementType.TYPE)
3 public @interface SequencedObject {
4    // we need no properties for this annotation as this is only a marker.
5 }
```

In Listing 27, `@Retention(RetentionPolicy.RUNTIME)` declares that this annotation is only applicable during runtime and may not be used in other scenarios. Moreover, `@Target(ElementType.TYPE)` declares that this annotation can only be used on types including classes, interfaces and enumerated types; in other words, this annotation may not be used to mark a method in a Java class.

5.3.2 Sequenced Method Annotations

A sequenced method annotation is used to specify the sequences of method calls beginning from a given method. The annotation requires a string property declaring the sequence specification discussed in Section 5.2. Listing 28 presents the declaration of this annotation type.

Listing 28: SequencedMethod Annotation Declaration

```
1  @Retention(RetentionPolicy.RUNTIME)
2  @Target(ElementType.METHOD)
3  public @interface SequencedMethod {
4
5     String value();
6
7     boolean allowExceptions() default true;
8
9     Class<?>[] expect() default {};
10
11    Class<VerificationFailureHandler> verificationFailureHandler();
12 }
```

`@Target(ElementType.METHOD)` declares that this annotation is only applicable to *methods*. The string value from `value()` holds the sequence specification (cf. Section 5.2). Additionally, a sequenced method annotation allows verification of exceptions during runtime. The tester can configure whether exceptions are allowed or not using the combination of `allowExceptions()` and `expect()`:

`allowExceptions()=true` implies that exceptions are allowed during execution; this is the default case. If there is no exception configured using `expect()`, *any* exception is acceptable; otherwise, an exception is acceptable only if it is specified in the array of classes in `expect()` or if it is a subclass of one of them. If an exception occurs and it is "expected", JMSeq allows the program to continue execution; the program should nevertheless be able to properly deal with the exception.

`allowExceptions()=false` means that no exceptions are expected during the execution. In this case, the value specified by `expect()` is ignored.

JMSeq additionally provides a general mechanism to specify how to handle situations in which the sequenced execution fails to complete. This can happen because of a method call which was not included in the specification, or an unexpected exception. To this end, JMSeq introduces `VerificationFailureHandler` interface as an extension point where the tester or developer can provide custom behavior to gracefully deal with the verification failure. This interface introduces a single method: `handleFailedExecution(Execution e)`. This method receives information about the execution through a parameter of type `Execution` (cf. Section 6.6). `Execution` is the central data model that carries information about the method execution sequences; the trail of the executions may be useful to the developer to examine the failure causes. If no such handler is provided, verification failure results in termination of program execution with a failure status.

Section 5.4 provides more discussion along with different examples on using JMSeq.

5.4 JMSeq by example

In this section, we go through different usage scenarios of JMSeq by means of illustrative examples.

5.4.1 Sequenced Execution Specification

In Listing 29, we again go through the example introduced in Section 5.2. In this example, we annotate the method `main()` with the sequences of method calls describing the two scenarios given in Figure 5.2. The JMSeq specifications used here are discussed in Section 5.2. The listing shows how to use them in a Java program.

Listing 29: Sample annotated specification

```
1
2 // Case 1
```

```
3 @SequencedObject
4 public class Main {
5
6    @SequencedMethod(
7      "{call(* *.A.m_a(..))[{call(* *.B.m_b(..))" +
8      "[{call(* *.C.m_c(..))}]}]}")
9    public void main() {
10      // ...
11   }
12
13   public void init() {
14      // ...
15   }
16 }
17
18 // Case 2
19 @SequencedObject
20 public class Main {
21
22   @SequencedMethod("{call(* *.A.m_a(..))" +
23     "[{call(* *.B.m_b(..))}][{call(* *.C.m_c(..))}]}")
24   public void main() {
25      // ...
26   }
27
28   public void init() {
29      // ...
30   }
31 }
```

5.4.2 Exception Verification

We discuss how JMSeq can be used to verify occurrence of exceptions during execution of an application using the example given in Figure 5.2 Case 1.

Let's assume that `m_c` may encounter `NullPointerException` and `m_b` may face `IOException`. But, we do not hold any assumption on the location in the code where the exceptions may be handled or even totally ignored. We use the following annotation in Listing 30 to express this situation.

Listing 30: Exception Specification for Case 1

```
1 // Case 1
2 @SequencedObject
3 public class Main {
4
5    @SequencedMethod("{call(* *.A.m_a(..))" +
6      "[{call(* *.B.m_b(..))[{call(* *.C.m_c(..))}]}]}",
7      allowExceptions = true,
8      expect = {NullPointerException.class,
9                IOException.class})
10   public void main() {
11      // ...
12   }
13
14 }
```

During the execution of the application, if any exception occurs, JMSeq receives the event of the exception occurrence. Since exceptions are allowed, there will be no verification failure by JMSeq as long as the exceptions received and verified are either instances of the specified exception classes or any of their subclasses. However, if the underlying application does not handle the exceptions (i.e., proper use of `try/catch`), they are propagated up in the call stack and this may result in JVM crash.

For instance, if an instance of `FileNotFoundException` occurs in `m_c`, JMSeq accepts the exception (because it is a subclass of `IOException`) and allows the application to continue. If the application at some point deals with the exception, the execution continues; otherwise JVM stops, complaining that the exception is not handled. On the other hand, if an `IllegalStateException` happens during the execution of the program, JMSeq does not allow the application to receive the exception since it is not one of the expected exceptions in the specification. Instead, it runs the failure code provided by the tester (if any).

In another situation in Case 1, assume that *no* exceptions are acceptable during the execution. For this to happen, we simply use `allowExceptions = false` in the specification annotation. In this case, no exception should occur during the execution of the application; otherwise JMSeq issues a verification failure. Besides, when no exceptions are allowed, using `expect()` has no effect on JMSeq verification behavior.

JMSeq additionally supports integration with testing frameworks such as JUnit. An example of such integration is discussed in Section 5.5.2.

5.5 The JMSeq framework

In this section, we present the JMSeq monitoring and verification framework and discuss its implementation, which uses Java 5 Annotations [**Java_5_Annotations**] and Java Platform Debugger API [**JPDA_Home**]. More implementation details, including examples and documentation, can be found at `http://github.com/nobeh/jmseq`.

As discussed above, to use JMSeq, it is assumed that Java components are supplied with sequenced object and sequenced method annotations. Multiple annotations are possible within the same class, but not for the same method. This implies that a component specification is in fact scattered across its constituent classes, each of which with a local and partial view of the component behavior. Only the annotated methods will be monitored by JMSeq. The overall verification process in JMSeq is depicted in Figure 5.4.

As a Java program, JMSeq executes naturally inside a Java virtual machine (JVM); JMSeq additionally initiates another JVM (called the *Target JVM*) to be able to control the sequenced execution of the target program. Through initial parameters,

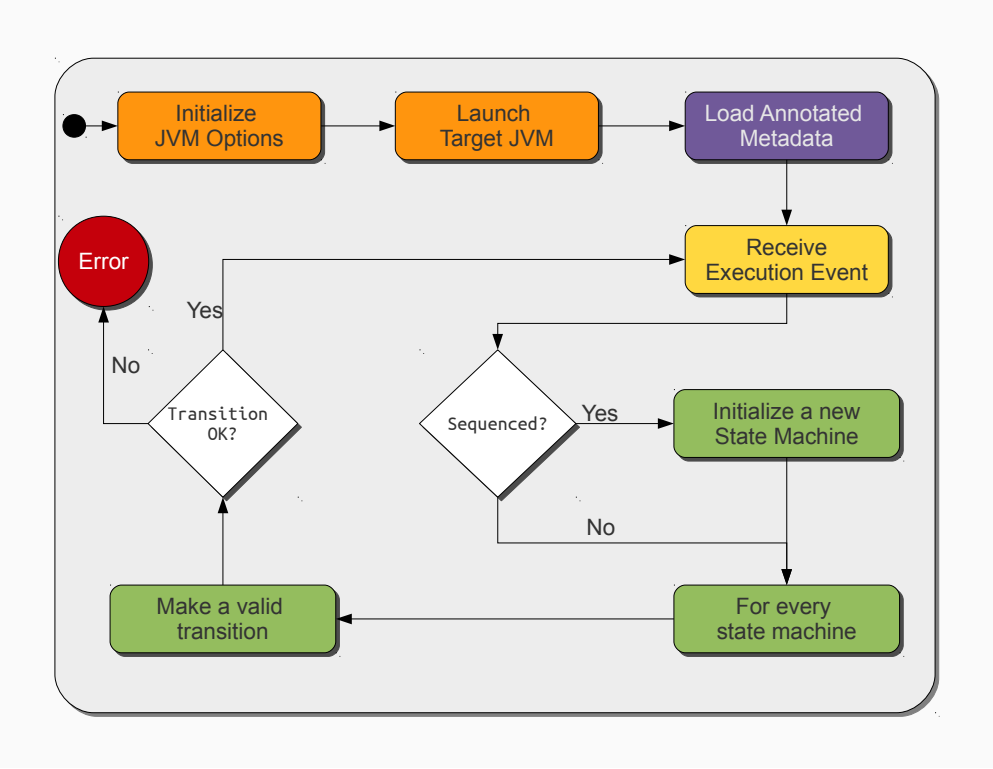


Figure 5.4: Overview of JMSeq's process to verify a program

the target JVM is configured to report back to JMSeq some events such as method entry and method exit. Furthermore, JMSeq does not verify events from all objects but only those specified in these parameters. The target JVM takes advantage of the Java Platform Debugger Architecture (JPDA) to access the needed details on the execution of the application while it is running.

Before verification starts, JMSeq inspects the classes in the execution class path searching for methods that are annotated for sequenced verification; during this process, it collects all of the available method call sequence metadata. This step (denoted as Load Annotated Metadata in Figure 5.4) does not need the source code, but rather uses the annotations metadata available in the compiled code.

The target program subsequently starts its execution and JMSeq reacts to the events it receives. JMSeq uses state machines to verify the method calls occurring in an application during runtime. The state machines are indexed by the pairs of method signature and the callee object; thus JMSeq can uniquely identify the associated state machine. When JMSeq receives a method execution event, it checks if there is a need to initialize a new state machine for that method execution. A state machine is initialized if there is no state machine for the current method execution and the method is annotated `@SequencedMethod`. Then, JMSeq tries to verify the current method execution in the context of all active state machines for other methods. For every state machine, JMSeq tries to make a valid transition in the state machine.

Verification will continue if such a transition can be made for all state machines; if not, JMSeq stops the execution reporting a violation of the specification.

The process of matching method execution events with state machine transitions is done in JMSeq by following these steps:

1. First, every event received from the target JVM, together with the information associated to the event, is transformed to an instance of `Execution` to capture various details about the execution of a method. Then, at verification time, the instance of `Execution` is transformed to an instance of `CallExpression` to hold method signature details used by the state machine to make a transition. (Section 5.5.1)
2. Using the metadata available from the annotations of the methods, then, the next "possible call expressions" of the current state, corresponding to the possible next transitions of the state machine, are built. (Section 5.5.1, 5.5.1)
3. A match making is done between the possible call expressions (state machine transitions) and the current call expression as the candidate. If a match is found, the transition is made and the method is accepted; otherwise, it fails. When a failure occurs, JMSeq will execute the custom verification failure handler that is implemented either as part of the component code by the programmer, or externally by the system tester. (Section 5.5.1)

JMSeq computes the possible call expressions on-the-fly instead of constructing the full state machine in the beginning of the verification process. This way it avoids the construction of states in the state machine that are never used in the execution. To avoid repeated computations of the same states, JMSeq utilizes dynamic programming techniques such as in-memory object caches.

5.5.1 JMSeq Architecture

The overall architecture of JMSeq is given in Figure 5.5. It basically consists of three main modules: one for handling the events raised by the JVM executing the target program, another module for storing the annotation information, and a third one to perform the runtime verification.

Event Handling Module

Java Debugger Interface (JDI) is the interface of Java Platform Debugger Architecture (JPDA) that gives access to different details during the execution of a program. Following the JDI event model, JMSeq takes control over some of the execution events that JVM publishes during a program execution. Therefore, a component was designed to model and hold the execution trace events required for event handling and execution verification.

1. `Execution` is the central data structure that holds all the information about execution that is made available through the JDI event mechanisms. Relevant

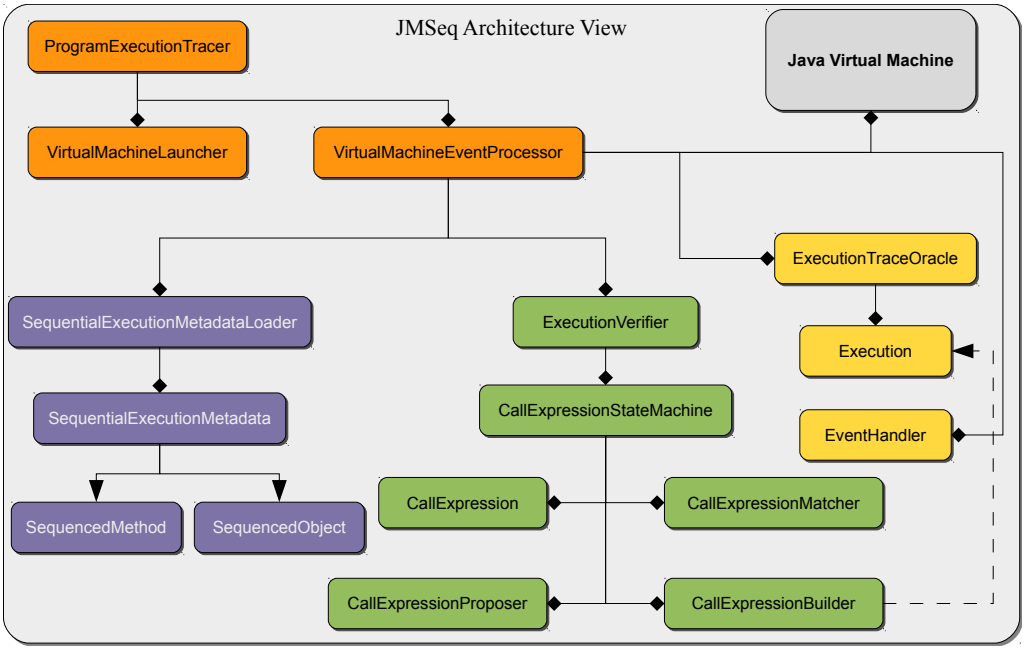


Figure 5.5: Software Architecture for Method Sequence Specification

events include “method entry”, “method exit”, and “exception” occurrences. It provides access to information such as:

- The object that is currently executing (*the callee object*) and its unique identifier in JVM.
- Event details through subclasses such as method return values or *the caller object* reference in case of a method exit event.
- *Parent Execution*: every execution can hold a reference to its parent execution object forming a directed tree of executions. This helps traversing the executions at validation time.

2. `EventHandler` is the event handling interface that is injected into JVM with access to JDI information. The event handler receives the events it is subscribed to and possibly takes an associated action. In particular it creates an instance of `Execution` for each sequence annotation and it stores it in the `ExecutionTraceOracle` registry. In general, all events received and processed by `EventHandler` are stored in this registry for further use by other components, for example, by the verification module.

Annotation Metadata Module

As the execution traces are stored in a repository, they are supposed to be checked and verified against the formal specifications as described on the `@SequencedObject` and `@SequencedMethod` annotations of the compiled classes in the program. It is the task of the annotation metadata module to store this information. A small utility

component in this module is responsible to read and load the metadata of all classes that are annotated. This information is used when there is a need to verify the conformance of an execution event.

Execution Verification Module

During execution of a program, those events that need to be monitored are received and verified against a message sequence specification. For every specification in the metadata repository, a state machine is created on the fly. At each state, JMSeq computes the Brzozowski derivative of the regular expression associated with the current state and memorizes this computation for further use. This avoids the full construction of the state machine at each step of verification.

When a method execution event is received, either the corresponding state machine exists or it should be created. A state machine is initialized if there is no state machine for the current method execution and the method is annotated `@SequencedMethod`. JMSeq then tries to verify the event against all active state machines. For every state machine, an attempt is made to make a valid transition. If the transition is successfully made, JMSeq continues. Otherwise, an "invalid" transition is detected; verification stops. At this point, JMSeq checks if an implementation of the `VerificationFailureHandler` interface is present (cf. Section 5.3). If present, JMSeq tries to instantiate an instance of the verification failure handler and hands over the execution metadata to this handler. Otherwise, JMSeq terminates and reports how the specification is violated.

An event may also be sent due to occurrence of an exception. In this case, the execution is verified based on the exception specifications expressed using the mechanism described in Section 5.3. If the exception is allowed and verified the execution continues; otherwise, it fails and control is delegated to verification failure handler as explained above.

The execution verification component is composed of the following elements:

1. *Call Expression* is a simple component for interacting with each state machine object. Basically, it transforms execution events coming from the JVM into call expressions used by the state machine components.
2. *Call Expression State Machine*: Sequences of executions are translated to "call expressions" that are to be accepted by a state machine. The state machine needs to distinguish the method's context on different calls to the same method. At the end of a successful sequential execution, the state machine associated to that sequenced execution specification should be in a success state.
3. *Call Expression Builder* constructs a call expression out of:
    a) A string which is in the format of the JMSeq specification grammar as in Figure 5.3. This service is used the first time a sequential execution is

detected to build the root of the future possible (candidate) call expressions.

b) An execution event; every time an execution is handed over to the verification module, an equivalent call expression is built for it so that it can be compared and matched against the call expression for the previous event.

4. *Call Expression Proposer* is a service proposing *possible next call expressions* for a given call expression based on the sequences specified in the annotation. As described by the grammar in Figure 5.3, for each current call expression there can be several possible next call expressions that may appear as the next event, but for each of them the state machine associated with the grammar can only make one transition (that is, the specification is deterministic). Note that since more specification sequences are possible involving the same method, only those call expressions that are valid in all specifications will be proposed.
5. *Call Expression Matcher* is another service that tries to match two call expressions. It is used, for example, to validate the current call expression against all those proposed by the previous service. If a match is found, the execution continues; otherwise the verification is regarded as failed. In this case, if a verification failure handler is provided, the failure data is transferred to it for further processing.

5.5.2 JUnit Support

To ease the process of verification, JMSeq provides a seamless integration to JUnit [**JUnit**]. JUnit is a widely accepted unit testing framework for Java programs.

To use JUnit, the programmer is expected to write Java class files that introduce "test" methods. A test method is annotated with `@Test` annotation in JUnit. Each test method is run in isolation and independently of the other test methods. The programmer may provide special methods `setUp` and `tearDown` to, respectively, prepare resources or release them for the unit test. JUnit provides a runner framework that runs all Java classes containing test methods. Different tools such as Eclipse [**eclipse_junit**] or Maven [**maven_junit**] provide easy ways to run unit tests with JUnit.

JMSeq extends the JUnit framework and provides a simple to use mechanism for programmers to write unit tests in order to verify their programs. JMSeq introduces the `JUnit4Support` class that is to be extended by the programmer to direct the verification process. In this unit test class, the programmer uses the `@Test` annotation to identify the methods implementing a test, in the same way as in JUnit. Furthermore, the programmer should override the following methods such that they provide the information that JMSeq requires to start the verification process:

`getClassName()` *should* be overridden to provide the simple name of the class that contains the entry point to the application.

`getPackageBase()` *should* be overridden to provide the qualified package address that contains the class file provided by `getClassName()`.

`getExcludedPatterns()` is a method that may be overridden to provide the patterns that are excluded in matching the events received from the underlying virtual machine started by JMSeq. By default, a set of events received from the standard packages in Java are excluded. The developer may add, remove or modify the patterns based on the expected events.

`setUp()` is a standard base method from JUnit that is executed before each test method execution. It is supposed to prepare the pre-conditions before the actual test. If this method is overridden, it either has to call the parent method first or guarantee that hierarchy objects are ready for execution.

`tearDown()` is another standard base method that is executed after each test method.

`@Test` methods Each test method should carry `@Test` JUnit annotation to be considered a unit test method. Each test method should contain the call to `startJMSeq()`. Other than that, it is the choice of the tester what to add in the test method.

We describe how to use the JUnit support through an example. A `Basket` is a resource holder for apples. Generally, each `Basket` supports two basic features: `put()` and `get()`. The first one inserts an apple (i.e., an instance of a resource) to the basket while the second one retrieves and removes one apple from the basket. There are different resource producers and consumers: producers populate the resource holder (basket) and consumers use the resources available in the resource holder. As an example, `Pat` is a consumer and for now we forget about the producer that co-relates to the mutual `Basket` and `Pat`. `Pat` consumes from the `Basket` and gets apples to `eat()`.

In terms of JMSeq sequenced specifications, each invocation of `eat()` in `Pat` should be followed by an invocation of `get()` in `Basket`. We specify this scenario in Listing 31.

Listing 31: Apples JMSeq Specification

```
1 @SequencedObject
2 public class Apples {
3
4    private Pat pat;
5
6    public Apples(Basket basket) {
7       this.pat = new Pat(basket);
8    }
9
10   @SequencedMethod("{call(public void nl..Pat.eat())" +
11         "[{call(public int nl..Basket.get())}$]}$")
12   public void startLunch() {
13     // generally a sequence of "eat" invocations
14     pat.eat();
15     // still hungry?
```

```
16      pat.eat();
17    }
18
19    public static void main(String[] args) {
20      Basket b = new Basket(10);
21      Apples apples = new Apples(b);
22      apples.startLunch();
23    }
24 }
```

We need now to run the tests for the `Apples` application. This is made very easy by JMSeq's support for JUnit. We develop a JUnit test class for this purpose and provide a set of predefined information to the base test class. The JMSeq test classes should *extend* the `JUnit4Support` that is provided in the library. The code is presented in Listing 32.

Listing 32: Apples JMSeq JUnit Test Unit

```
1 public class ApplesDriver extends JUnit4Support {
2
3    @Test
4    public void startLunch() {
5      startJMSeq();
6    }
7
8    @Override
9    protected String getClassName() {
10     return Apples.class.getSimpleName();
11   }
12
13   @Override
14   protected String getPackageBase() {
15     return Apples.class.getPackage().getName();
16   }
17 }
```

5.6 The Fredhopper Access Server: A Case Study

Fredhopper[2] is a search, merchandising and personalization solution provider, whose products are uniquely tailored to the needs of online businesses. Fredhopper operates behind the scenes of more than 100 of the largest online shops [3]. Fredhopper offers its products and facilities to e-Commerce companies (referred to as customers) as services over the cloud computing infrastructure. In this respect, Fredhopper should tackle various challenges in resource management techniques, the customer cost model, and service level agreements. One of the major components of their software is the Fredhopper Access Server (FAS), which provides access to high quality product catalogs.

To orchestrate different services such as FAS or data processing, Fredhopper takes advantage of a service controller (referred to as Controller). The Controller manages

[2] `http://www.sdl.com/products/fredhopper/`

[3] `http://www.sdl.com/campaign/wcm/gartner-maqic-quadrant-wcm-2013.html?campaignid=70160000000fSXu`

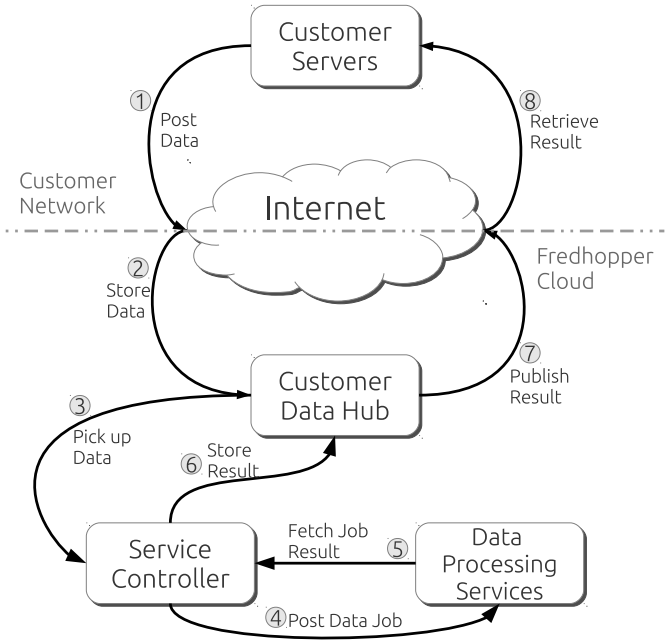


Figure 5.6: Fredhopper's Controller life cycle for data processing requests from customers

different service installations for each customer. For instance, a customer can submit their data along with a processing request to their data hub server. The Controller then picks up this data and initiates a data processing job, usually in the cloud. When the data processing is complete, the result is published to the customer environment and becomes also available through FAS services. Figure 5.6 illustrates this example scenario.

The Controller provides a simple client API for customers to develop applications that use Fredhopper services. Essentially a data job request consists of three major steps:

1. Prepare the data , upload it to the customer data hub and receive a data identifier.
2. Specify the data processing: reindexing, updating or loading data. Such actions are called *triggers*.
3. Collect the results of a trigger and decide how to use them.

The customer is required to have the appropriate data identifier before executing a new trigger. In fact, the customer application is supposed to use the proper sequence of operations from the API.

In this case study, we examined and measured how JMSeq can help customers improve their usage of client API for data processing; i.e. verify data processing triggers and notify customers as soon as they occur. We used "rebuild" and "update"

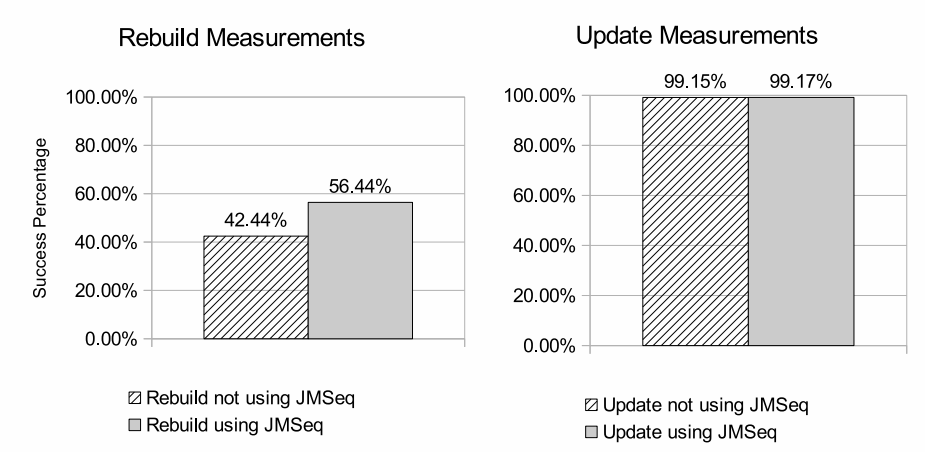


Figure 5.7: Result of experiment measurements without and with using JMSeq on Fredhopper Client API directed with live environment data from customers.

triggers in this experiment based on customer environment data from the period of February 2013 [4]. In this experiment, we measured the following:

1. Using the log files, we replayed the triggers twice with different settings.
2. In one setting, we replayed the original customer data processing trigger.
3. In the other setting, we replayed the original customer data processing that was wrapped with JMSeq verification.
4. In each setting, we measured the number of *successful* and *failed* triggers. A successful trigger is one for which data processing completes and the result is reported back to customer data environment.

Figure 5.7 presents the measurements for each setting without using JMSeq and with using JMSeq for both rebuild and update triggers. The measurements were based on an estimated number of 30300 triggers in the same period. Each figure shows the influence of using JMSeq for the data processing task. On the left, measurements shows that JMSeq verification for rebuilding the catalog helps to decrease failures by 15% if using JMSeq verification. On the right, measurements shows that JMSeq verification has less influence to improve update process as it natually involves less erroneous data because of smaller data size in update triggers.

Next we present the case study in code listings in which JMSeq monitors calls to methods of the Controller API. The runtime environment stops as soon as an incorrect sequence of calls is detected.

Listing 33: Controller Data Dispatcher Interface API

```
1 interface DataDispatcher {
2    Data create(Object object);
3    Future<?> dispatch(String dataId);
4    Future<?> dispatch(Data oid, String dataId);
```

[4]Fredhopper data in this research is confidential due to privacy protection of customers.

```
5 }
```

The relevant part of the API of the Fredhopper Controller is presented in Listing 33. Customer applications use the interface to create a data object reference (using the `create(Object)` method) and then use the created data reference to start a data job (using the overloaded `dispatch` method, which has an optional `Data` parameter). Note that the dispatch method allows for two different uses by the customer, for instance in Listing 34 the data job to be executed performs only a rebuild of the customer's product catalog and as such does not need to have a data reference.

Listing 34: A use case with no data requirement

```
1 class CustomerApp {
2    public void rebuildCatalog() {
3       Future<?> result =
4          dispatcher.dispatch("reindex");
5       // further processing of result
6    }
7 }
```

If the customer needs to update their catalog with new products, then first a data reference is created. This reference is used to initiate a data job, as described in Listing 35.

Listing 35: A use case with data requirement

```
1 class CustomerApp {
2    public void updateCatalog(Catalog catalog) {
3       Data data = dispatcher.create(catalog);
4       Future<?> result =
5          dispatcher.dispatch(data.getId(), "update");
6       // further processing of result
7    }
8 }
```

Note that we cannot expect to have the source code of the above listings. To verify the correct use of the Fredhopper Controller API with JMSeq we can create a `CustomerAppDriver` class which calls the `CustomerApp` methods and monitors the subsequent method calls. The expected usage of the API of the two scenarios above is described by the SequencedMethod annotations in Listing 36. If the `CustomerApp` does not behave as expected, the verifier detects an error.

5.6.1 Discussion

We use JavaMOP [**chen_rosu_jmop**, **chen_rosu_mop**, **MOP**] for the same case study from Fredhopper. Listing 37 shows a JavaMOP script to describe the same verification goal.

Listing 37: Fredhopper case study with JavaMOP

```
1 UpdateCatalog(Catalog catalog) {
2    event create_data before(DataDispatcher d):
3       call(* DataDispatcher.create(..)) && target(d) {}
4
```

Listing 36: How to verify Fredhopper API usage

```
@SequencedObject
class CustomerAppDriver {

   @SequencedMethod(
      "{call(* nl..CustomerApp.rebuildCatalog()" +
      "[{call(* nl..DataDispatcher.dispatch(String))}]}")
   public void verifyRebuildCatalog() {
      // initiate the customer application
      app.rebuildCatalog();
      // finalize verification
   }

   @SequencedMethod(
      "{call(* nl..CustomerApp.updateCatalog(Catalog)" +
      "[{call(* nl..DataDispatcher.create(Object)}" +
      "{call(* nl..DataDispatcher.dispatch(Data, String))}]}")
   public void verifyUpdateCatalog(Catalog catalog) {
      // initiate the customer application
      app.updateCatalog(catalog);
      // finalize verification
   }
}
```

```
   event dispatch_data before(DataDispatcher d):
      call(* DataDispatcher.dispatch(..)) && target(d) &&
      cflow(dispatch_data) {}

   ere: (create_data dispatch_data)

   @fail { __RESET; }
}
```

JavaMOP uses aspect-orientation and Java bytecode instrumentation. Using aspects (e.g., as in AspectJ [**kiczales_aspectj**]) means that the eventual customer application bytecode is instrumented. The question is *how can it be proved that JavaMOP does not have any influence on the application?* If the customer is not willing to cooperate in such a process, the system cannot be monitored with JavaMOP. JMSeq on the other hand does not change the source- or bytecode, because JMSeq runs on a different JVM as the one running the customer application.

5.7 Performance Results

In this section, we present performance measurements of JMSeq and other techniques to verify a program. We choose a problem that has been studied in the context of JavaMOP [**chen_rosu_jmop**] to provide a reference for comparison.

The example verifies the correct use of `java.util.Iterator`. Originally, the interface introduces two major methods: `hasNext()` and `next()`. The first one checks whether there are still elements in the sequence that can be visited and the second one gives out the next element in the sequence and moves forward. An invocation of `next()` presumes that it succeeds an invocation of `hasNext()`. Otherwise, an invocation of `next()` may lead to an exception.

We use JMSeq to verify an example code for this scenario. We use the code snippet displayed in Listing 38 that uses JMSeq annotations to express the correct usage. In the code, we use a variable `count` to control the number of times `hasNext()` is called before there is a call to `next()`. This example always passes the JMSeq verification with success. For performance measurements, we study the "correct" use of the example while verification of wrong usages are also possible.

Listing 38: Iterator usage with JMSeq Annotations

```
1   @SequencedMethod(
2     value = "{call(* java..Vector.iterator())}$"
3         + "{call(* java..Iterator.hasNext())}$"
4         + "{call(* java..Iterator.next())}",
5       allowExceptions = false)
6   public void visitAllElements() {
7      Iterator<Integer> i = v.iterator();
8      for (int k = 0; k < count && i.hasNext(); ++k) {
9         i.hasNext();
10        i.next();
11     }
12  }
```

In the following, we provide certain measurements from executions of JMSeq for the example above for different values of `count` in {10000, 15000, 20000, 30000, 50000, 100000, 200000}. We use VisualVM tool that is a standard profiling and performance measurement tool packaged with standard Java development packages.

We measure the execution time of the following cases:

original. We execute the original program and measure the execution time of the program with the above `count` values.

instrumented. We use a technique that uses instrumentation such as JavaMOP [**chen_rosu_jmop**] and apply it to the program and measure the execution time the same.

original\JMSeq. We execute the original program and monitor it by JMSeq. We measure the execution time of the original program while being verified by JMSeq.

Figure 5.8 displays how different cases compare to each other regarding the execution time. As expected, the original program execution has the lowest execution time. Since techniques based on instrumentation modify the bytecode of the program, this adds to the execution time of the program. The figure shows that compared to instrumentation techniques, JMSeq adds a lower overhead.

We use the performance results here to argue that the technique used by JMSeq is more suitable to verify applications that are mission-critical or operate on a high load. Experience shows that, under high load, issues start to emerge that do not exist under normal operational situations. In such circumstances, the application provider is very likely to look for methods to verify and detect problems. And yet,

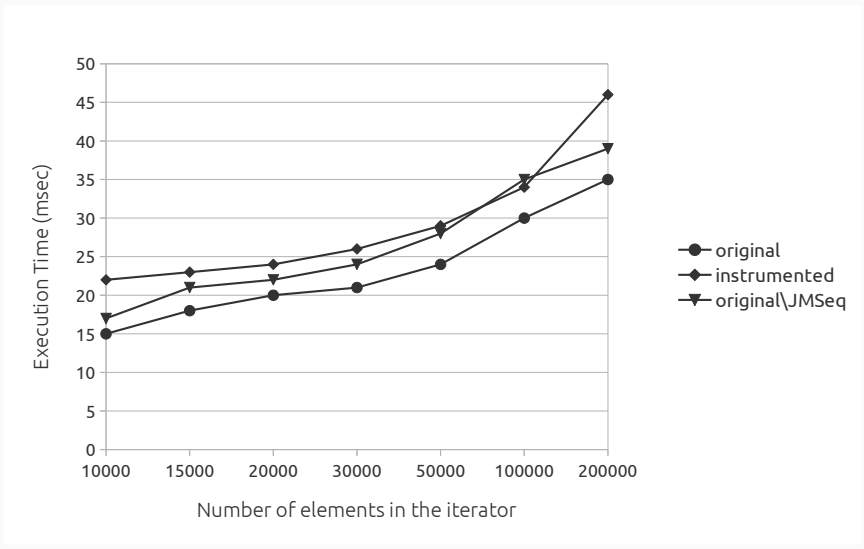


Figure 5.8: Comparison of different techniques applied to the same program. In this figure, "original" denotes the original program that is measured for execution time; "instrumented" denotes using a technique such as JavaMOP that uses instrumentation for verification; and "original\JMSeq" denotes the measurement of the execution time for the program apart from times for JMSeq.

they need to make sure that the application remains at least at the same level of execution and performance. Moreover, the fact that JMSeq runs in a separate JVM increases the safety of our approach, since JMSeq can be used from even a remote machine to verify a program, without the slightest influence on the resources that run the target program.

5.8 Related Work

JML [**leavens_baker_ruby_jml_design**] provides a robust and rigorous grounds for specification of behavioral checks on methods. Although JML covers a wide range of concerns in assertion checking, it does *not* directly address the problem of method call sequence specification as it is more directed towards the reasoning about the state of an object. JML is rather a comprehensive modeling language that provides many improvements to extensions of Design by Contract (DBC) [**dbc**] including jContractor [**jcontractor**] and Jass [**jass**].

In [**yoonsik_ash_mscs**], taking advantage of the concept of *protocols*, an extension to JML is proposed that provides a syntax for method call sequences (protocols) along with JML's functional specification features. Through this extension, the developer can specify the methods' call sequence through a *call sequence clause* in JML-style meta-code. In the proposed method, the state of a program is modeled as a "history" of method calls and return calls using the expressiveness of regular expressions; thus, a program execution is a set of "transitions" on method call histories. The

verification takes place when the execution history is simulated using a finite state machine and checked upon the specified method call sequence clause.

JML features have been equivalently implemented with AspectJ constructs [**jml_aspects**], using aspect-oriented programming [**kiczales_aspectj**]. The authors of this work propose AJMLC (AspectJ JMLC), which integrates AJC and JMLC into a single compiler so that instead of JML-style meta-code specifications, the developer writes *aspects* to specify the requirements.

In [**stijn**] an elegant extension of JML with histories is presented. Attribute Grammars [**DBLP:journals/mst/Knuth68**] are used as a formal modeling language, where the context-free grammar describes properties of the control-flow and the attributes describe properties of the data-flow, providing a powerful separation of concerns. A runtime assertion checker is implemented. Our approach differs in several respects. The implementation of their runtime checker is based on code instrumentation. Additionally, they use local histories of objects, so call-backs can not be modeled. However, the behavior of a stack can be modeled in their approach (and not in ours), since their specifications are not regular expressions but context-free languages.

In the domain of runtime verification, Tracematches [**tracematches**] enables the programmer to specify events in the execution trace of a program that could be specified with "the expressiveness" of a regular pattern. The specification is done with AspectJ pointcuts and upon a match the advised code is run for the pointcut. Along the same line, J-LO [**bodden_jlo**] is a tool that provides temporal assertions in runtime-checking. J-LO shares similar principles as Tracematches with differences in specifications using linear time temporal logic syntax.

Additionally, Martin et al. propose PQL [**martin_livshits_lam_pql**] as a program execution trace query language. It enables the programmer to express queries on the execution events of objects, methods and their parameters. PQL then takes advantage of two "static" and "dynamic" checkers to analyze the application. The dynamic checker instruments the original code to install points of "recovery" and "verification" actions. The dynamic checker also translates the queries into state machine for matching criteria. The set of events PQL can deal with includes method calls and returns, object creations and end of program among others. Accordingly, generic logic-based runtime verification frameworks are proposed as in MaC [**java_mac**], Eagle [**eagle**], and PaX (PathExplorer) [**pax**] in which monitors are instrumented using the specification based on the language specific implementations.

Using runtime verification concepts, Chen and Rosu propose MOP [**chen_rosu_mop**, **MOP**] as a generic runtime framework to verify programs based on monitoring-oriented programming. As an implementation of MOP, JavaMOP [**chen_rosu_jmop**] provides a platform supporting a large part of runtime JML features. Safety properties of a program are specified and inserted into the program with monitors for runtime verification. Basically, the runtime monitoring process in MOP is divided

into two orthogonal mechanisms: “observation” and “verification”. The former stores the desired events specified in the program and the latter handles the actions that are registered for the extracted events. Our approach follows the same idea as MOP, but it does not use AOP to implement it. Another major difference is in the specifications part. MOP specifications are generic in four orthogonal segments: logic, scope, running mode and event handlers. Very briefly, the *scope* section is the fundamental one that defines and specifies the parts of the program under test. It also enables the user to define desired events of the program that need to be verified. The *logic* section helps the user specify the behavioral specification of the events using different notations such as regular expressions or context-free grammars. The *running mode* part lets the user specify what is the running context of the program under test; for instance, if the test needs to be run per thread or in a synchronized way. And, *the event handlers* section is the one to inject customized code of verification or logic when there is a match or fail based on the event expression logic. In Section 5.6.1, we discussed how JMSeq contrasts with JavaMOP and similar approaches.

5.9 Conclusion and future work

We proposed JMSeq, a framework for specifying sequences of possibly nested method calls using Java annotations. The sequences do not only consist of method names, but may contain information such as object caller and callee, and distinguish overloaded methods. JMSeq uses Java Platform Debugger to monitor the execution of a component-based system in Java. Monitoring is divided into two phases: observing the events in the program and verifying them at runtime against the local specifications provided through annotated methods. JMSeq does not use *any* form of instrumentation; neither source level nor bytecode level. Moreover, JMSeq provides a simple way to express exceptions and verify them in the execution of the target application. JMSeq is integrated with JUnit. We presented a case study from Fredhopper distributed cloud services in the domain of e-commerce and marketing. Moreover, we provided a set of performance results for JMSeq in different settings for program execution and runtime verification.

JMSeq is a novel approach to runtime verification of software using code annotations. The approach is especially suitable for runtime component-based verification. We improved the execution data model of JMSeq using optimization techniques such as in-memory object caches with indices. In the context of event processing, we examined different implementations of JPDA and JDI standards (e.g. using Eclipse platform [**eclipse_debug_platform**]) and it showed part of the optimization is out of our reach. We used VisualVM, a standard Java profiling tool, to measure and discuss the performance of JMSeq.

A future line of work consists of porting JMSeq to multi-core platforms. Concurrent and distributed software introduces new challenges for runtime verification of applications. In a concurrent setting, objects interact with each other from different

threads of execution (which may optionally involve synchronization and locking events). JMSeq should be able to verify such objects in different threads and consider how synchronization and locking may affect the runtime or flow of an application. In a distributed context, application objects are located in (physically) remote JVM instances. In such scenarios, JMSeq needs to load or observe a topology of the involved JVMs and their objects to be able to verify the application for correct sequences of method calls.

We plan to extend JMSeq by providing native features such as mock [**frank:mock**] implementations in case parts of the system are unavailable. JMSeq may provide better integration for mocking parts of the system that are not implemented or not available. Such features will be especially useful in the context of integration testing and verification. At the time of integration, not all components of the system may be available and complete; and yet, the application should be verified in the parts that are complete. In comparison with runtime checking, another line of future work can be to extend JMSeq to support static verification of protocols (using approaches as in Copilot [**copilot**]).

Finally, we plan to extend the specification language to a context free language. This is in fact simple as JMSeq already implements a simple form of pushdown automata to recognize the call-return structure of Java. In fact, Java is expressible by a restricted form of deterministic pushdown automata, called visibly pushdown automata [**Alur_abstractvisibly**], where push and pop correspond to call and return, respectively.

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.

Conference Publication *Lecture Notes in Computer Science, Volume 9306, Service Oriented Cloud Computing, 4th European Conference – ESOCC 2015, Pages 125–140, DOI 10.1007/978-3-319-24072-5_9*

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.

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

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.

Runtime monitoring [**Logean_monitoring**, **BratanisDS10**] 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 [**alur:1994:timedautomata**, **fersman2007task**, **jaghoori2010time**]. 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 [**bjork2013:rtabs**, **nobakht2013future**] 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 [**inzinger:monitoring**] 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 [**johnsen2012abs**], an executable modeling language that supports all of these features and offers a wide range of tool-supported analyses [**BubelMH14**, **WongBBGGHMS15**]. The mapping from ABS to timed automata [**alur:1994:timedautomata**] 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 [**fersman2007task**]). 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 [**hogben2013defavail**] 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 [**keller2003wsla**] introduces a framework to define and break down customer agreements into a technical description of SLAs and terms to be monitored. In [**mahbub2011translationsla**], a method is proposed to translate the specification of SLA into a technical domain directed in SLA@SOI EU project. In the same project, [**comuzzi2009defavail**] 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 [**chen2007sladecompose**], 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 [**raimondi2008fsmsla**] to detect violations of SLA and formalize them.

Johnsen [**johnsen2012modeling**] introduce "deployment components" using Real-Time ABS [**bjork2013:rtabs**]. 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 [**coles2011cost**] and [**gilmore2011non**] 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 [**woodcock2014contracts**] 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 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

[2]Quanticol EU project with no. 600708: `http://quanticol.eu/`
[3]COMPASS EU project with no. 287829: `http://www.compass-research.eu/`
[4]`http://en.wikipedia.org/wiki/Full_text_search`
[5]`http://en.wikipedia.org/wiki/Faceted_navigation`

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.
- 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

[6] Figures are indication of complexity and scale. Detailed confidential information may be shared upon official request.

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 [**johnsen2012abs**]. To ensure non-intrusiveness of the monitor with the running service, each monitor is an active object (actor) running on a separate resource from that which runs the service itself, and the components of the monitoring platform communicate through *asynchronous messages* with deadlines [**johnsen2012modeling**].

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 39.

Listing 39: 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 }
```

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

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

a method `getState(Service s)` which returns the number of resources allocated to the given service s at time t_c.

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).

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 $\beta(s,\tau)$. [**jaghoori2010time**] 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 [**uppaal2004**] supporting advanced methods such as state-space reduction [**larsen1997efficient**].

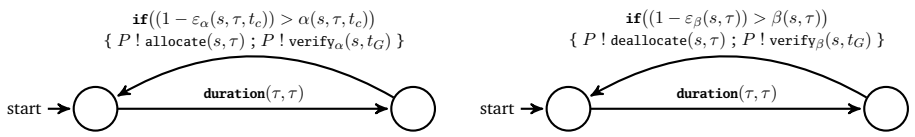
We use task automata as defined in [**fersman2007task**, **jaghoori2012composing**, **jaghoori2010time**]. Task automata are an extension of timed automata [**alur:1994:timedautomata**] In addition, we design the automata for the monitoring platform using the real-time extension of task automata presented in [**jaghoori2010time**] p. 92 in which the author presents a mapping from Real-Time ABS [**johnsen2012modeling**] 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 [**Nobakht:sched**, **nobakht2013future**]. 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 [**jaghoori2010time**] 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 6.1c and 6.1d). 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*.

(c) $M_{\mathtt{V}}^{\alpha}$ to execute $\mathtt{verify}_{\alpha}$ **(d)** $M_{\mathtt{V}}^{\beta}$ to execute $\mathtt{verify}_{\beta}$

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 $\{\mathtt{A}(\mathtt{allocate}), \mathtt{D}(\mathtt{deallocate})\}$. 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:

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

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 [**jaghoori2010time**] p. 92 to design $M_{\mathtt{A}_s}$. The resulting automata is presented in Figure 6.1.

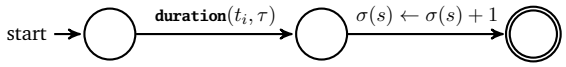


Figure 6.1: $M_{\mathtt{A}_s}$: Timed Automaton to execute task type `allocate` in M_P

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 6.2. Therefore, we also have $M_{\mathtt{D}}$ in M_P as: $M_{\mathtt{D}} = \|_s\ M_{\mathtt{D}_s}$.

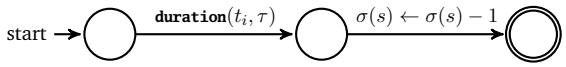


Figure 6.2: $M_{\mathtt{D}_s}$: Timed Automaton to execute task type `deallocate` in M_P

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 6.3. In this environment, in every monitoring, an unspecified constant (c) number of resources fail.

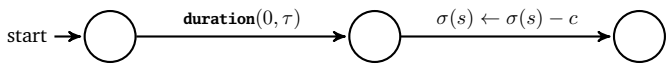


Figure 6.3: An example behavior for M_E

We define system automata [**jaghoori2010time**] (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 $\texttt{verify}_\alpha$ in M_{V}^α 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 [**fersman2007task**] 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 [**jaghoori2010time**], we translate the above task automata into traditional timed automata. This allows to leverage well-established model checking techniques such as UPPAAL [**uppaal2004**] 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 6.4 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.

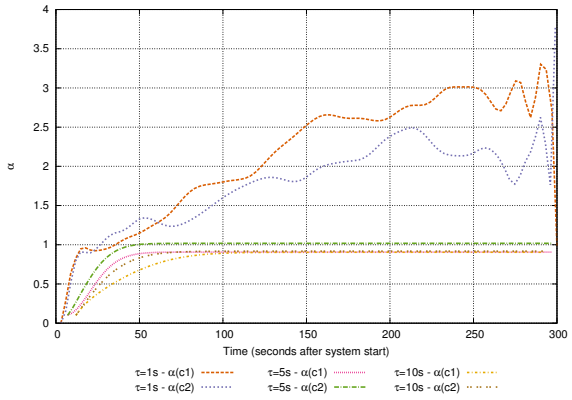


Figure 6.4: Evolving $\alpha(s, \tau, t_c)$ with different τ

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 [**hogben2013defavail**] 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$\star$ [**kearney2010sla**]. Currently, we are integrating our automated monitoring infrastructure into the in-production SDL Fredhopper cloud services (cf. Section 6.3).

Abstract

This thesis contributes to the intersection of object orientation, actor model, and concurrency. We choose Java as the main programming language and as one of the mainstream object-oriented languages. We formalize a subset of Java and its concurrency API to facilitate formal verification and reasoning about it. We create an abstract mapping from a concurrent-object modelling language, ABS, to programming semantics of Java concurrency. We provide the formal semantics of the mapping and runtime properties of the concurrency layer including deadlines and scheduling policies. We provide an implementation of ABS concurrency layer as a Java API library and framework utilizing the latest langauge additions in Java 8. We design and implement a runtime monitoring framework, JMSeq, to verify the sequenced execution of methods through code annotatinon in JVM. In addition, we design a large-scale monitoring system as a real-world application; the monitoring system is built with ABS concurrent objects and formal semantics composed with schedulability analysis.

Samenvatting

[TODO Translate to Dutch. Kindly by Stijn.]

Curriculum Vitae

behrooz.nobakht@gmail.com
Geboren 29-07-1981, Teheran

Opleiding

2011–2015 **PhD Informatica**, *Universiteit Leiden*, Leiden.
Actors At Work©

2009–2011 **MSc cum laude Informatica**, *Universiteit Leiden*, Leiden.
Deployment of active objects onto multi-core

1998–2002 **BSc Informatica**, *Technologie Universiteit Sharif*, Teheran, *7.0*.
Online Shopping

Werkervaring

2011–Nu **Senior Software Ontwikkelaar**, *SDL Fredhopper*, Amsterdam.
Groep leider in cloud

2003–2009 **Medeoprichter en architect**, *ASTA Co.*, Teheran.
- Het beheer van het bedrijf
- Ontwikkelen
- Project onderhandeling

Acknowledgement

Hello, here is some text without a meaning. This text should show, how 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 – Never mind! A blind text like this gives you information about the selected font, how the letters are written and the 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 a special contents, but the length of words should match to the language. Hello, here is some text without a meaning. This text should show, how 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 – Never mind! A blind text like this gives you information about the selected font, how the letters are written and the 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 a special contents, but the length of words should match to the language.

Hello, here is some text without a meaning. This text should show, how 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 – Never mind! A blind text like this gives you information about the selected font, how the letters are written and the 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 a special contents, but the length of words should match to the language. Hello, here is some text without a meaning. This text should show, how 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 – Never mind! A blind text like this gives you information about the selected font, how the letters are written and the 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 a special contents, but the length of words should match to the language.

List of Figures

List of Tables

Colophon

This thesis was typeset with LaTeX 2_ε. 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.

Leiden, October 31, 2015

Behrooz Nobakht