FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

An Actor Constraint Prototype

Verifying Event Order

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Rudolf Mildner
Matrikelnummer 0426776

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr. Franz Puntigam

Wien, 21. August 2015

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

An Actor Constraint Prototype

Verifying Event Order

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Rudolf Mildner
Registration Number 0426776

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr. Franz Puntigam

Vienna, August 21, 2015 ______________________ ______________________
(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Rudolf Mildner
Tigergasse 6/21 A1080 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Danksagung

Meiner Mutter und meiner Großmutter. Danke für Liebe, Geduld und Weisheit. Ich habe viel von Euch gelernt.

Der kleinen Anna Michaela Margaretha Alles Gute zur Geburt und für Ihren Lebensweg.

Professor Puntigam vielen herzlichen Dank für die großartige Betreuung.

Abstract

This thesis describes design, implementation and evaluation of a prototype that allows us to define synchronisation protocols for the verification of message orders between actors. The prototype extends *AKKA* [66], an existing *actor concurrency* implementation, provides a verification layer on top of it and a domain-specific language to write synchronization rules. Rules can be defined on a per-actor basis. The prototype makes sure the actor system complies with them and detects unwanted message orders. Two verification algorithms were implemented for the prototype. Both use the same basic verification logic, but differ in how they log interactions between actors and how they check firing of rules. *History logging verification* keeps a log of interactions for each actor and uses it to check if a rule fires. Because this approach proved to be memory intensive and not performant enough, a second algorithm was implemented which circumvents these issues. *Automaton verification* creates an automaton for each rule. Interactions with the actor are logged by changing the automaton's active state and the active state of an automaton is used to determine if a rule fires.

Evaluation of the prototype consisted of worst-case runtime determination, tests with real world applications and benchmarking. For both implemented verifiction algorithms the worst-case runtime of the verification itself was determined. For the *automaton verification* also the worst-case runtime of automaton creation and -optimization was determined. To test the prototype under real world conditions, two sample applications were written: One which implements the *Chameneos Concurrency Game* [45] and another one which implements a *token ring*. The latter also was used to perform benchmark tests, by running the application in different configurations without the prototype, with the *history logging verification* and with the *automaton verification* and the resulting overall runtimes were measured and graphically evaluated.

Kurzfassung

Diese Diplomarbeit beschreibt Design, Umsetzung und Analyse eines Prototyps, welcher die Definition eines Synchronisations-Protokolls zur Verifizierung der Reihenfolge von Nachrichten zwischen Aktoren erlaubt. Der Prototyp erweitert AKKA [66], ein etabliertes Aktoren-System und erlaubt uns, mittels einer Domain-Specific-Language Regeln zu definieren. Der Prototyp stellt sicher, dass das das Aktoren-System diese Regeln einhält, indem er alle Nachrichten findet, die eine Regel verletzen. Zwei Versionen der Verifikation wurden implementiert. Beide verwenden dieselbe Verifikationslogik, unterscheiden sich aber in der Art und Weise, in der sie Interaktionen loggen und überprüfen, ob eine Regel feuert. Die *History Logging Verification* hält ein Log aller bisherigen Interaktionen mit einem Aktor und verwendet diese History, um zu überprüfen, ob eine Regel feuert. Dieser Ansatz stellte sich als speicher- und zeitintensiv heraus, weswegen eine zweite Variante implementiert wurde. Die *Automaton Verification* erzeugt für jede Regel einen Automaten. Interaktionen mit einem Aktor werden als State-Changes in diesem Automaten abgebildet, und der aktuelle Zustand des Automaten wird verwendet um zu überprüfen, ob eine Regel feuert.

Zur Evaluation des Prototyps wurde die Worst-Case Laufzeit der Verifikations-Algorithmen bestimmt, es wurden Tests mit Real-World Applikationen durchgeführt und ein Benchmark wurde durchgeführt. Für beide Verifikations-Algorithmen wurde die Worst-Case Laufzeit der Verifikation bestimmt. Für die *Automaton Verification* wurden zusätzlich die Worst-Case Laufzeiten für das Erzeugen sowie das Optimieren eines Automaten für eine Regel bestimmt. Um den Prototyp unter realen Bedingungen zu testen wurden zwei Applikationen implementiert: Die Erste realisiert das *Chameneos Concurrency Game* [45], die Zweite simuliert einen *Token-Ring*. Diese zweite Applikation wurde auch verwendet, um Benchmark-Tests mit dem Prototyp durchzuführen. Dazu wurde die *Token-Ring*-Applikation in verschiedenen Konfigurationen ohne Prototyp, mit der *History Logging Verification* und mit der *Automaton Verification* ausgeführt und die ermittelten Laufzeiten zusammengefasst sowie grafisch aufbereitet.

Contents

CHAPTER 1

Introduction

Actor concurrency [40] is a concurrency model introduced by *Carl Hewitt*. It was heavily influenced by advances in modern physics, especially by *special relativity*. Currently this concurrency model is regaining much interest, because it elegantly brings together concurrency and object orientation.

This thesis describes design, implementation and analysis of a prototype which extends the *AKKA* [66] actor system. The prototype provides a mechanism for establishing synchronisation protocols between actors for the detection of unexpected or unwanted message orders. Developers build the synchronization protocol for each actor by defining rules in a domain-specific language. The prototype verifies communication between actors and makes sure, that message orders comply with the established synchronzation protocol.

1.1 Motivation, Objective and Methodology

Actor concurrency is a very powerful concurrency mechanism. Actors are self-contained entities similar to objects in the object-oriented programming paradigm. Different actors can be executed concurrently and multiple instances of an actor can be executed concurrently as well [40]. Actors run completely independent of each other [46] and do not share any state. If used correctly, no mechanism for dealing with mutual exclusion is therefore required.

Actors may receive messages in orders not compatible with their behavior. These messages may influence the actor's behavior and also the interactions of multiple actors in unexpected ways. Unwanted and possibly faulty behavior can be the result.

The goal of this thesis is to provide developers with an easy-to-use framework that can be used to establish a synchronisation protocol between actors, so that they never receive messages in unexpected or unwanted orders. The prototype proposed in this thesis uses the *AKKA* actor system [66] as underlying actor implementation and extends it with a verification layer to check the actors' message orders and a domain-specific language for easy creation of synchronization protocol rules. *AspectJ* [26] aspects tie the verification layer into *AKKA*: Whenever an actor sends or receives a message, the operation gets intercepted. As the code gets weaved into the actor, runs in the actor's execution context, and only requires local data stored directly at the actor, the verification can be seen as a local operation of the actor itself, that enhances the actor's behavior.

The approach taken for this thesis is as follows. First, several concurrency models were examined, to bring actor concurrency into context. After that, actor concurrency was analyzed in detail and also was compared to the aforementioned other concurrency models. As a next step, a prototype was implemented. The prototype allows developers to define synchronisation protocols on actors with the help of a domain-specific language and then subsequently verifies the message orders between actors based on these protocols through AspectJ aspects. As the first implementation of the verification algorithm, which is based on logging the actor's interaction history, proved to be not ideal, another improved one was implemented, which creates automatons to store and verify the interaction history. To test the prototype under real-world conditions, an implementation of the chameneos concurrency game and a token ring application were implmenented as example applications. Finally, important properties of the prototype, such as the worst-case runtime of the implemented verification algorithms were analyzed and benchmarks were performed to examine the prototype's runtime.

1.2 Thesis Structure

The rest of this thesis is structured as follows:

Chapter Concurrency first gives a motivation for why concurrency is becoming more and more important with the increasing advent of multi-core processors. Then the term concurrency is defined and challenges associated with concurrency as well as mechanisms to deal with them are discussed. This chapter also introduces some well-known theoretical models for describing concurrency, in order to bring the actor concurrency model into context.

Chapter Actor Systems describes the *actor concurrency model* and its approach at concurrency in detail. Benefits and limitations of the model are discussed and a comparison of actor concurrency to the theoretical concurrency models introduced in the previous chapter is performed.

Chapter Prototype introduces the proposed prototype and is therefore the core of this thesis. The prototype's basic concepts and its architecture in overview and in detail are described, as well as performed improvements and optimizations. Benefits and limitations are discussed and the prototype is compared to similar approaches that can be found in literature. The last part of the chapter describes the development process and the prototype's development history in iterations.

Chapter Evaluation first gives the worst-case runtime of the implemented verification algorithms. Next, the example applications implemented are introduced in detail. After that, the benchmarks used to measure the prototype's performance are discussed. The chapter closes with proposed topics for future research.

CHAPTER 2

Concurrency

This chapter describes fundamental concurrency basics and helps bringing *actor concurrency* and the prototype described in this thesis into context to other existing approaches to concurrency. The requirement for concurrency is motivated and a basic description of concurrency is given, an overview over challenges and basic mechanisms for dealing with them is provided and selected theoretical concurrency models are introduced.

When describing concurrency mechanisms and theories, concurrently running portions of a program are often called "‘parts of a program"’ in this chapter. This term was chosen, because there are various terms for concurrently running program parts (thread, actor, process, ...) and which one of them is used in literature often depends on the described mechanism or theory. Because we want to approach the topic concurrency in a consistent fashion, a neutral term was chosen.

2.1 Motivation

So what is concurrency and why do we need it? In order to increase the execution speed of a program we can basically do two things: We can either speed up each instruction or run several instructions in parallel. While for sequential execution there is one possible order in which to execute a program’s instructions, concurrent execution leaves some details of the instruction order unspecified [7], allowing instructions to be executed interleaved or in parallel. In uniprocessor systems, interleaving can be used to run programs seemingly in parallel. Interleaving also allows processing to be continued when one program is waiting for the results of operations currently performed, for example disk IO. In systems with multiple processing units concurrent program parts can run at the same time, thus being executed really in parallel.

One of the reasons why concurrency has received so much attention in the last years is that the end of performance increase for instructions in conventional processing units seems to be in sight [54]. While processor performance for decades was improved by increasing the speed of instructions and transistor density, nowadays the trend points toward improving processor performance by introduction of processors with multiple cores and multiprocessor systems [39].

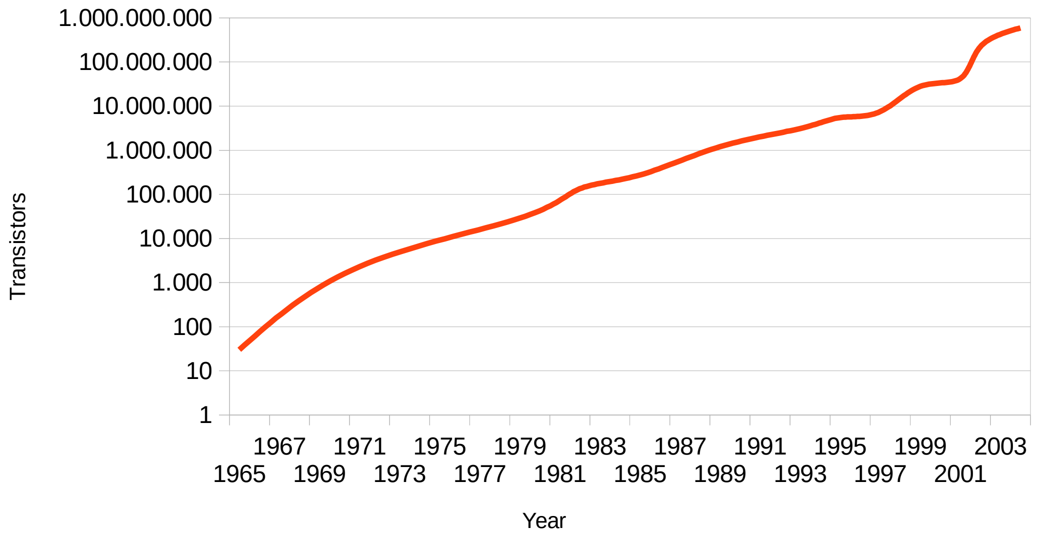

Figure 2.1: Transistor density according to *Moore's Law* (density in logarithmic scale, Source: [43])

2.1.1 Instruction Speedup

Speeding up instruction processing is one way of increasing the overall speed of a program. Instruction speed has increased since the very beginning of integrated circuit development. *Moore's Law* states that the density of transistors on processing units doubles every two years, increasing available computing power [51]. Being more an observation than a real law, *Gordon E. Moore* reviewed the progression of the semiconductor industry in 1965, coming to the conclusion that the number of transistors in chips grows exponentially [51]. Since then, more than 50 years long, transistor density more or less has continued to grow as predicted in Moore's initial observation.

It remains to be seen if and how long the exponential growth of processing power can be upheld. However, because of physical factors, it is unlikely that exponential growth of computing power for conventional processors will continue much longer. Because *Moore's Law* states that the density of transistors grows, transistors shrink in size. Since transistor size begins to become small enough and transistor density high enough for quantum effects to occur [49], the question remains how long until very fundamental physical limits are reached; educated guesses predict one or two more decades of exponential growth or less [49] [51]. Thermal management problems, leakage current and thermal noise might put a hold on exponential growth of processing power even earlier [49].

2.1.1.1 Physical Limits

In the following section the most important physical limits which could put a hold on growth of processing power are presented in order of their fundamentality. While the *speed of light* and *Heisenberg uncertainty* are fundamental physical limits to any classical computing, *heat dissipation*, *leakage current* and *thermal noise* are limits specific to semiconductor technology.

Speed of Light The most serious physical limitation is for sure the finite *speed of light*. During the duration of a 1.8 GHz clock period, light can travel approximately an 8cm round trip, for a 5GHz clock period it even drops to about 3 centimeters [54]. Unfortunately, electrons in silicon move from 3 to 30 times slower than light does in vacuum [54] which effectively limits processing speeds for conventional semiconductor devices considerably.

Heisenberg Uncertainty Another severe limiting factor is the *Heisenberg uncertainty principle*. It states a fundamental limit on the precision of measuring an electron's position and spin [49]. An electron's characteristic dimension for *Heisenberg uncertainty* is the *Compton wavelength* [49] [56] that describes the fundamental limit of measuring an electron's position and spin based on *quantum mechanics* and *special relativity* [49].

Heat Dissipation An issue that may put an end to increase of instruction speedup much earlier than the aforementioned very fundamental physical limitations is *heat dissipation*. When current flows through a processing unit, it radiates heat [28]. This is the reason why CPUs and GPUs have to be cooled with heat sinks passively or with air or water actively. If the number of transistors in chips continues to grow at the current rate, cooling down processors might soon reach practical limits [28].

Leakage Current *Leakage current* is spontaneous movement of electrons between emitter and collector of a transistor [28]. Such *leakage current* does nothing than to consume power. With smaller transistor sizes more *leakage current* occurs because electrons can more easily pass small distances, resulting in higher energy consumption and more heat dissipation without providing any benefit [28].

Thermal Noise *Thermal noise* is the result of *heat dissipation* of the elements of an integrated circuit [28]. The *thermal noise* level is proportional to the resistance and bandwidth of a transistor [28]. *Thermal noise* can influence tiny integrated circuits so considerably, that they stop working stable and predictably, rendering them completely useless for computing.

2.1.1.2 New Approaches

The aforementioned issues have led to active research on very promising technologies that could provide much more efficient computing and may replace current integrated circuit technology in the future. Two especially interesting technologies are *optical computing* and *quantum computers*.

Optical Computing Because electrons move only 3 to 30 percent of the vacuum *speed of light* in conventional semiconductor materials [54], there is ongoing effort to use optical materials inside processing units as in such materials 60 percent of the vacuum *speed of light* could be reached [54]. *Optical computing* uses tiny optical fibers as connections within and between processing units. Such a *transphasor* could improve processing speed over conventional *transistors* considerably. While in optical materials *heat dissipation* is considerably less of a problem, the inefficiency of the conversion between electricity and light results in power-consumption and heat dissipation problems [54] hindering practical applicability of the technology. Also *optical computing* is still bound by the *speed of light* and *heisenberg uncertainty* as fundamental limits for the increase of instruction speed.

Quantum Computing *Richard Feynman* observed in the 1980's that quantum mechanical effects are hard to simulate on conventional computer systems which led to the theory that quantum effects could be utilized for more effective computing [64]. However it proved to be difficult to build computer systems using quantum effects and to develop algorithms for them. In 1994, *Peter Shor* described what is now known as the first quantum algorithm. *Shor's algorithm* can factor integers in polynomial time [64].

In quantum computing systems computational space increases exponentially with the quantum system's size [64]. Because this observation enables exponential parallelism, *quantum computing* does not suffer from the physical limits of conventional computer systems. A *quantum bit* in *quantum computing* is the equivalent to a *bit* in classical computing. *Quantum bits* are in a superposition of the states 0 and 1 and a register of *quantum bits* is in superposition of all 2^n possible values, which supports computation with all 2^n possible values at once.

One of the big issues holding back *quantum computing* to become effectively usable is that the access to results of a quantum computation is restricted as accessing a result requires measurement which disturbs the quantum state the system is in [64]. Moreover measured results are probabilistic, so they have to be interpreted to be of any use. In the past few years progress was made and non-traditional programming techniques were developed to exploit the parallelism provided by *quantum computing*. One such technique is to manipulate the state of a quantum program so that a property common to all output values can be read, another important one is based on the transformation of a program's quantum state to increase the likelihood of reading the wanted output [64].

2.1.2 Parallelization

The other way of speeding up program execution is to run instructions in parallel. This requires the program to be written in a way such that its instructions can be run concurrently and that the underlying system provides multiple processing units where instructions can be executed in parallel. As we have already seen, speeding up program instructions becomes harder and harder, so executing program instructions in parallel will become increasingly important if we want to improve performance of programs further. As everything does, parallelization has its challenges and limitations. Writing concurrent programs that retain the wanted semantics can be tricky as we will see in more detail later in this chapter. Running as much of a program in parallel is also important as the overall performance gain obtained depends on the degree of possible parallelization as described by *Amadahl's law*.

Concurrency Being able to execute a program in parallel requires it to be specified in a way that at least some of its instructions can be executed concurrently. Introduction of concurrency into complex programs requires considerable effort by the developer, so that the semantic of the program does not get changed by accident. The rest of this chapter describes what concurrency and parallelity are (2.2), goes into detail regarding challenges (2.3) and common solution approaches (2.4) as well as describing theoretical models of concurrency (2.7).

2.2 Definition

This section introduces informal definitions of concurrency as used throughout this thesis. In order to define what concurrency means, we first have to define what a program is and what executing the program means.

2.2.1 Program

A *program* consists of a *list of atomic instructions*, a *current state* and a set of *initial states*. The program's instructions together and in total order describe the semantics of the program. A program state is defined as the values of all memory addresses associated with the program. The program starts in one of the states from the initial state set. The program then is executed by feeding its instructions into one or multiple processing units, where the atomic instructions are executed with the given parameters and change the program's current state.

Sequential Execution When executed sequentially, one instruction of a program gets executed at a time on one processing unit. *Sequential execution* specifies the order in which to execute instructions as a *total order* [7]. So for a program running sequentially, there is exactly one order in which the program's instructions will be executed.

Concurrent Execution In contrast to sequential execution, concurrent execution leaves some details of the instruction order unspecified, so that instructions can be executed in multiple sequential orders or truly simultaneous. Only if parts of a concurrent program are being executed at the same time, we say that the program is executed in parallel. Concurrent execution does not necessarily imply that instructions get executed in parallel at the same time. Concurrent programs can be executed

- preemtively and time-shared on a single processor
- on multiple cores at the same processor
- on physically separated processors
- on physically separated machines

2.2.2 Concurrency Definitions

The two most common definitions of concurrency are *interleaved concurrency* and *true concurrency* [14] [63] [20]. Concurrency is said to be *interleaved*, if only one atomic instruction can take place at each computation step. Concurrency is said to be *true*, if multiple atomic instructions can take place at each computation step. For both definitions, it has to be ensured, that the program's semantic meaning does not change.

So interleaved concurrency basically reduces concurrency to nondeterminism by modeling parallel actions as choice between their possible sequentializations [14] [20]. True concurrency is more powerful than interleaved concurrency, because with true concurrency, behavior can be expressed, that cannot be expressed with interleaved concurrency [63]. Therefore true concurrency cannot be reduced to interleaved concurrency. Because interleaved concurrency is however easier to handle in proofs, some theoretical models of concurrency use it over true concurrency [67].

The following example demonstrates the difference between interleaved and true concurrency: Take the computing actions $i = j + 1;$ and $j = i + 1;$. For simplicity we assume here, that these computing actions are executed atomically (this might not always be the case, but for example on very long instruction word architectures [29] it is). With interleaved concurrency these computing actions take place one at a time, so either

$$\begin{aligned} i &= j + 1; \\ j &= i + 1; \end{aligned} \tag{2.1}$$

or

$$\begin{aligned} j &= i + 1; \\ i &= j + 1; \end{aligned} \tag{2.2}$$

gets executed. With true concurrency these computing actions (may) get executed at the same time:

$$i = j + 1;\ j = i + 1; \tag{2.3}$$

Let us assume as an example that $i = 2;\ j = 5;$ before the aforementioned statements get executed. With interleaved concurrency there are two possible outcomes: For the instruction order 2.1 the result is $i == 6;\ j == 7;$, with the instruction order 2.2 the result $i == 4;\ j == 3;$. With true concurrency however the outcome $i == 6;\ j == 3$ is also possible.

2.2.3 Performance Gain

When a program's atomic instructions are executed in parallel, the execution is sped up. Usually there however exist parts of a program, which have to be executed sequentially, because atomic instructions depend on each other. Given the amount of instructions that have to be run sequentially, *Amadahl's law* determines the overall increase of speed when executing a program in parallel. *Gustavson's law* argues against *Amadahl's law* with the argument that the law uses wrong assumptions.

2.2.3.1 Amadahl's Law

Amadahl's law [41] [11] describes the overall performance gain achievable by executing parts of a program in parallel. Amadahl's Law states, that if we speed up part of a program by a certain factor, the overall speed up is

$$Speedup_{real}(f, S) = \frac{1}{(1 - f) + \frac{f}{S}} \tag{2.4}$$

with *f* being the fraction of the program sped up and *S* being the amount of speedup [11] [41].

Implications Amadahl's Law has important implications [41]. The overall speedup of a program running on multiple processing units in parallel is limited by the fraction of the program that can be executed only sequentially. This makes clear how important it is to make as much of a program concurrently executable if we want to improve processing speed through parallelization. Another implication of the law is that when the fraction *f* sped up is small, even big optimizations of that fraction will have little effect on the overall program's performance. Moreover, the fractions we ignore also limit performance, as the speedup is bound by

$$\frac{1}{1 - f} \tag{2.5}$$

when the amount of speedup *S* approaches infinity [41].

Gustafson's Law *John L. Gustafson* argued against Amadahl's Law [37] [41]. His arguments were that Amadahl's Law assumes the parallelizable fraction of a program to be a fixed value and that the law does not consider the machine the program runs on. Gustafson stated in what is now known as *Gustafson's Law* [37] [41] that machines with greater parallel computation power allow instructions to operate on larger data sets in the same amount of time than machines with smaller parallel computation power, thus increasing performance gain through parallelization.

2.3 Challenges

In the following, challenges associated with concurrency are described. Finding solutions for these challenges leads the way as to why and how the actor concurrency model and the thesis prototype came into being and where they fit in. On the lowest abstraction level, *finding the semantically valid partial orders* for a program's instructions is the ultimate challenge regarding concurrency. Seen on a higher abstraction level, issues regarding concurrency can be described more specifically: A *race condition* leads to unpredictable results when multiple concurrent program parts work in an uncoordinated way. *Deadlock*, *livelock* and *starvation* hinder concurrent program parts to progress.

2.3.1 Finding Valid Partial Orders

The existence of multiple possible instruction orders for a concurrent program introduces issues about correctness which are not present when executing the same program in a predefined order.

The overall goal when writing concurrent programs is to find and separate wanted, correct orders from unwanted, incorrect ones. This means finding the ones that can be run concurrently, but do not change the program's meaning, which can be done on different abstraction levels. As we will see, there exists a variety of different mechanisms for introducing concurrency while asserting program correctness (see sections 2.4 and 2.7).

A program's instructions have relationships called *program dependencies* defined between them which are relevant for their semantic meaning. Program dependencies imply that the affected instructions have to be executed in a certain order to behave as expected [19]. Program dependencies come in two forms: *control dependencies* and *data dependencies*.

Control Dependencies *Control dependencies* [15] are caused by control statements. An instruction x_2 that depends on another instruction x_1 through a control dependency has to be executed after x_1 because the result of x_1 determines if x_2 will be executed. An example for this form of order restriction is a simple if statement: The condition evaluation has to be evaluated before the then statement because the condition determines whether the then statement gets executed at all.

Data Dependencies *Data dependencies* [15] [19] are caused by instructions that refer to data of previous instructions. If an instruction x_1 writes into a variable and x_2 reads from the very same variable, the statement x_2 is data dependent on statement x_1 and has to be executed before it.

Program Dependencies versus Concurrency Making a program as concurrent as possible and satisfying its program dependencies are somewhat antagonistic to each other. On the one hand we want instructions to be able to run in as many orderings as possible, but on the other hand, control- and data dependencies limit the amount of allowed instruction orderings. In conclusion, what we want is to *support many orderings semantically equal to the original one* and to *prevent all orderings changing the program's meaning*.

2.3.2 Race Conditions

A race condition occurs when two concurrent parts of a program share some resource and inconsistently read and write on it dependent on the timing and in the order their instructions get executed [59]. Such a shared resource could for example be a variable which both concurrent program parts hold and can write and read on. As consequence of a race condition, the same program can behave differently and non-deterministically for different executions with the same input.

As an example of a race condition, suppose two concurrent parts of a program t_1 and t_2. t_1 and t_2 both read data from the same memory address. Then t_1 and t_2 both perform some operations and write to that memory address. Depending on which of the two program parts writes as last one, the memory address contains a different value. This can lead to unwanted behavior as the result highly depends on the timing of the instructions executed. So the time the processing of each instruction requires and the order the instructions are executed in determines the value of the memory address.

2.3.3 Deadlock and Livelock

Deadlock and *livelock* can occur if two or more concurrent parts of a program share some resources and the program's parts can request exclusive access to a memory location [58] [54].

Suppose again two concurrent parts of a program t_1 and t_2. t_1 wants to write memory addresses a_1 and a_2 and therefore requests exclusive access to both. t_2 also wants to write to a_1 and a_2 and so also requests exclusive access to them. Unfortunately the instructions get interleaved so that t_1 gains exclusive access to a_1 and t_2 gains exclusive access to a_2. Now both t_1 and t_2 wait for the other's resource to continue processing, but neither can acquire it, effectively waiting for each other to continue.

A *livelock* is similar to a deadlock. In a livelock, however, the involved concurrent program parts continually change their states and in contrast to a deadlock do seem to progress. However, they only change their states, again waiting for all required shared resources to be exclusively available to them without ever getting them.

2.3.4 Starvation

Starvation can occur when multiple parts of a program try to gain access to the same resource, but one or some parts never get it [58].

Suppose three concurrent program parts t_1, t_2 and t_3. All three require the shared memory address a_1 exclusively and periodically. t_1, t_2 and t_3 all request exclusive access to a_1. t_1 receives exclusive access to the resource, performs operations on it and returns the exclusive access. Suppose, t_3 receives exclusive access as the next one. In the meantime, t_1 again requests exclusive access to the resource. Now the operating system decides to grant exclusive access again to t_1. While t_1 is using a_1, t_3 again requests exclusive access. After t_1 gives back exclusive access to a_1, the operating system again grants t_3 exclusive access. This goes on and on. t_2 never gets exclusive access to a_1 and therefore never can continue in its instruction flow.

2.4 Solution Approaches

There are various approaches for mitigating and solving the aforementioned issues associated with concurrency. In some way or another, all these mechanisms provide means for *synchronisation*. We first informally define the term as used in this thesis and then go into more detail.

2.4.1 Synchronisation

Synchronisation is the coordination of actions and events in a program divided into multiple, concurrently executed parts. Synchronisation has the goal to constrain the execution to wanted instruction orders and avoid undesired ones that would change the program's semantics and would lead to incorrect or undefined behavior [59]. Synchronisation can be *blocking* or *non-blocking* and consists of two separate aspects: *Instruction ordering* and *mutual exclusion*. Mechanisms for synchronisation can also be categorized depending on their abstraction level into *hardware synchronisation mechanisms* and *synchronisation primitives*.

Blocking and Non-Blocking Synchronisation Synchronisation can be performed in *blocking* or *non-blocking* fashion [54]. *Blocking synchronisation* lets concurrent program parts hold when reaching a certain point in execution, waiting until some condition is met or some resource is available. This can unfortunately lead to deadlock or livelock. *Non-blocking synchronisation* in contrast permit execution without blocking, so concurrent program parts run wait-free and neither deadlock nor livelock issues can arise [54]. In theory, any parallel algorithm can be expressed in non-blocking form [54] and depending on how strong the guarantees provided are [54], an algorithm can be *wait-free* where every program part makes progress in finite time, *starvation-free* so program parts make progress in finite time in absence of failures or *deadlock-free* where at least one program part progresses in finite time in absence of failures.

Instruction Ordering Regardless of the concurrency abstraction used, we always want to coordinate program parts so that their instructions execute in a semantically correct order. *Instruction ordering* is the task of forcing one of possibly multiple existing semantically valid instructions orders. Ideally this means to [59]

- preclude instruction orders that change the program's semantic and would lead to incorrect or undesired behavior

- on the other hand supports as many orders as possible that keep the same semantic meaning as the originally given sequential order

The first directive assures correctness of the program, while the second one permits a change of the instruction order for optimizing performance and, even more important, to run instructions in parallel.

Mutual Exclusion Concurrency concepts that supports shared resources between concurrent program parts also require a mechanism for providing *mutual exclusion* [34] [58]. A shared resource is a resource to which multiple program parts have simultaneous access to. The existence of shared resources requires ways to coordinate access to them, otherwise their data may become corrupted [34]. Concurrently running program parts want to work with shared resources without any external interference, executing multiple instructions in direct succession. This is called a *critical section* [58] and mechanisms that provide *mutual exclusion* make sure only one program part can be in a critical section for a specific resource at a time [58].

While in a *critical section*, the program part has exclusive access to the shared resource and can use it in an atomic and consistent way. After leaving the *critical section*, other program parts may again enter a *critical section* on their own. Failing to provide *mutual exclusion* can leave shared resources in an inconsistent state. Unfortunately mutual exclusion itself can become a source for concurrency issues as it may lead to *deadlock*, *livelock* or *starvation*.

2.4.2 Hardware Synchronisation Mechanisms

Usually hardware provides built-in synchronisation mechanisms. While it is possible to implement low level synchronisation mechanisms in software, providing them in hardware is usually much more efficient. High level *synchronisation primitives* build upon these low level *hardware synchronisation mechanisms*, providing more comfort and safety. Common hardware synchronisation mechanisms include:

- Disabling interrupts
- Atomic operations
- Memory barriers

Disabling Interrupts For single core processing units, the simplest way of providing hardware synchronisation is to *disable interrupts* [58]. Disabling interrupts prevents context switches of any kind and until interrupts are turned on again, the executed sequence of instructions runs atomically. Disabling interrupts unfortunately has various limitations. First, one cannot differentiate between processes that can influence each other and processes that are irrelevant to each other, which can lower efficiency considerably, but more importantly disabling interrupts does not work for providing synchronisation in multicore or multiprocessor environments at all.

Atomic Operations An *atomic operation* is a special machine instruction that performs multiple actions atomically without interruption [58] [54]. Since concurrency emanates at the instruction level, such an instruction is atomic and other parts of a program cannot interfere with its substeps. Advantages are that the concept is simple and applicable for multicore or multiprocessor environments. Disadvantages are that busy waiting and deadlock might occur with some instructions and atomic operations can only be used with single data elements and not with multiple ones. Which instructions are available depends on the specific chipset. Common ones provided include: [54] [58] [44]:

- Test-and-set
- Compare-and-swap
- Load-link and conditional-store

Test-and-set atomically loads and writes a variable. Only if the given variable is 0 the running concurrent program part is allowed to pass and the variable gets set to 1, other concurrent program parts have to wait until the variable is 0 again. *Compare-and-swap* first compares the content of a variable with a value given as parameter, changing the variable's content to another given value depending on the outcome. *Load-link* and *Conditional-store* are used together. *Load link* loads a variable from memory. The variable's content can be changed arbitrarily. For writing it back, the *conditional-store* operation is used. It guarantees that storing only succeeds if the variable has not been written by any code other than the one corresponding to the previous load-link operation.

Memory Barriers Modern processors can perform a variety of performance optimizations on instruction sequences. These optimizations include [54] changing order of instructions, deferring or combining them, providing branch prediction and various types of caching and have the goal to improve instruction throughput as well as to minimize the time spent waiting for memory access. When the program is executed sequentially, these optimizations leave the program's semantic meaning program intact, however when the program runs concurrently, they can cause inconsistencies and may lead to unexpected or unpredictable behavior [54].

Memory barriers [54] are special instructions guaranteeing that other instructions will not be moved over them when optimizations are performed, imposing a partial order over instructions on both sides of the barrier [54]. The memory barrier functions as a wall and the processor has to guarantee that it does not reorder or optimize instructions when this results in moving instructions over the barrier. Memory barriers exist in different varieties and the most appropriate one for a specific situation can be chosen to allow the processor to perform as many optimizations as possible. Four basic types of memory barriers can be found [54]:

- Write memory barriers
- Read memory barriers
- Data dependency barriers
- General memory barriers

Write memory barriers and *read memory barriers* give guarantees about memory write and memory read instructions respectively. *Data dependency barriers* give guarantees about interdependent read instructions only and *general memory barriers* give guarantees about both read and write instructions.

2.4.3 Synchronisation Primitives

Synchronisation primitives provide synchronisation and mutual exclusion on a higher abstraction level. Synchronisation primitives are provided by the operating system, the programming language, or associated libraries and use the hardware synchronisation mechanisms mentioned earlier for providing functionality. Synchronisation primitives are more robust and stride to be easier and more straightforward to use than their hardware counterparts. Common synchronisation primitives are:

- Locking
- Semaphores
- Mutices (mutex)
- Monitors
- Condition variables

Locking *Locking* means to wait until a certain event happens before execution can proceed. Locking can be implemented either *blocking* or *spinning* [70] [12]. A *blocking lock* tries to acquire the lock and when failing suspends the current process adding it to a queue of waiting processes, waking it up if the lock is released. A *spin lock* continually checks if the lock is available. The best choice between blocking and spinning depends on the ratio between expected spin time and time required for a context switch [70] as both operations add overhead. Most operating systems use a hybrid locking mechanism [59] polling first like a *spin lock* and after a timeout suspending the currently running program part like a *blocking lock*.

Semaphore The concept of a *semaphore* was first proposed by *Edgar W. Dijkstra* [25] [34]. A semaphore supports process synchronisation without busy waiting. It has a value, a queue of blocked processes and two operations *P()* and *V()*. *P()* decreases the value, *V()* increases it. A process that calls *P()* can proceed only if the semaphore's value is *>0* and gets blocked and added to the process queue otherwise. If the semaphore's value is *<0*, the next call of *V()* increases the value by *1*, which allows the next blocked process listed in the semaphore's queue to continue. In contrast to a lock, a semaphore can be increased by a different concurrent program part than the one that performed the last decrease operation. Semaphore implementations can be *binary semaphores* with a boolean value or *counting semaphores* with an integer value.

Mutices (Mutex) A *mutex* is an instruction that locks a shared resource in a concurrent program part. The shared resurce then can be used by a single program part uninfluenced. After that, the concurrent program part releases the mutex, signalling that it has finished working with the shared resource. In contrast to a semaphore, a mutex has an owner and only the concurrent program part which owns the mutex can release it. So while semaphores provide a solution to the instruction-ordering part of the synchronisation problem and are used for communication, mutices provide a solution to the mutual exclusion problem and should be used when a shared resource is to be protected for a critical section [16].

Monitors A *monitor* is a synchronisation mechanism found in object-oriented languages. The *monitor* concept was introduced by Hansen [38]. *Monitors* [34] are objects that give safe access to their methods and variables by more than one thread, using low-level synchronisation mechanisms. A *monitor* supports entry methods that guarantee *mutual exclusion* when executed [34].

Condition Variables *Monitors* also provide *condition variables* [34]. A *condition variable* is a construct that allows one to signal a specific condition to a process. It provides two operations *wait()* and *notify()* [34]. When a process calls *wait()* on the *condition variable*, the process gets blocked and written into a queue. Whenever *notify()* gets called, the next process in the queue is allowed to continue. *Hoare monitors* [34] [48] hand the control immediately to the first process in the queue, while *Mesa monitors* [48] let the thread that called *notify()* continue until it gets suspended.

2.5 Exchange of Information

In order to write useful concurrent programs, the concurrent parts of a program have to exchange information between each other. There are two ways how this can be accomplished:

- By *shared memory*
- By *message passing*

Depending on the underlying concurrency mechanism, programs either use one or the other, or a hybrid combination of both. Multicore processors and multiprocessor machines usually provide either *shared memory* or *message passing* in hardware [59] and both can be emulated in software if the hardware does not support it directly [59]. The two paradigms differ in how synchronisation is employed. While for shared memory systems synchronisation has to provided explicitly most of the time, message passing systems usually incorporate synchronisation in the message passing mechanism [59]. In any case, the mechanisms previously described can be used for accomplishing synchronisation.

2.5.1 Shared Memory

With *shared memory*, data communication is performed implicitly when data shared between concurrent program parts is accessed. Developers must assert that processes using the same memory do coordinate. Issues to consider include *race conditions*, *deadlock*, *livelock* and *starvation*. In contrast to *message passing*, *shared memory* models hide the need for explicit data communication. In order to work correctly, *shared memory* models require consistency on the operations performed by concurrent program parts.

Different consistency models have been described. Stronger consistency models are more restrictive, but also give us more guarantees over the instruction order while weaker ones are much less restrictive, but provide weaker guarantees [34], giving us more possible instruction orderings for performing optimizations and improving execution speed.

In a *sequential consistency* model [59] [34], memory accesses seem to appear in a total order. A read instruction to a memory address therefore always returns the last written value [59]. *Linearizability* describes an even stronger consistency model [34]. *Linearizability* looks for an external observer as if each operation on shared resources performed by the parts of a concurrent program take effect immediately at some point between beginning and end of its execution. *Causal consistency* is a common weaker consistency model [34]. With *causal consistency* concurrent program parts do not have to agree on the same ordering for write operations. However, each program part sees read operations that can affect it correctly. More common relaxations include allowing arbitrary reordering of reads and writes to different memory locations or allowing writes from multiple processes to become visible in inconsistent orders for different observing processes, as long as they are consistent for every single process [2].

2.5.2 Message Passing

In a *message passing* model, concurrent processes communicate by explicitly sending messages to each other [59]. This is a two-sided process: The sending process describes the data which should be sent and the receiving process describes how to receive the data and what to do with them. Both together have to describe at least what data are sent, by whom were they sent and to whom.

In the purest form, every part of the program has its own memory, not sharing memory resources with other parts. Note, that messages sent between concurrent program parts are not shared resources as long as these messages do not contain addresses of resource of the sending program part which the receiving program part could access [46]. The *actor concurrency* model leverages this form of communicating information in that it defines actors as completely independent, not sharing any memory resources.

Message passing can happen either *synchronously* or *asynchronously*. *Blocking*, or *synchronous message passing*, also called *request-reply messaging* means that the sender blocks and waits until the receiver either sends a receipt that it received the message or an answer [58]. Conceptually, most programmers are used to *synchronous message passing* as function calls in most programming languages work that way. *Non-blocking*, or *asynchronous message passing* means that the sender sends a message to the receiver and immediately continues without waiting [58]. *Asynchronous message passing* has the advantage that sender and receiver are decoupled from each other and that sender and receiver do not even have to be active at the same time. When required, synchronisation can be provided for *asynchronous message passing* through a *synchronizer*, which is an algorithm simulating synchronicity on asynchronous architectures [13].

2.6 Concurrency and Object-Orientation

2.6.1 Requirements

While many object-oriented languages incorporate mechanisms for dealing with concurrency, the availability of both object-orientation and support for concurrency as separate concepts is not enough to efficiently develop concurrent, object-oriented applications [53], and so paradigms or programming languages providing both concurrency and object orientation have additional requirements [53].

Active and Passive Objects Developers should have the possibility to define *active* and *passive objects*. *Active objects* are running concurrently while *passive objects* run sequentially and developers should be able to select from both depending on whether the current part of the system requires concurrency or not. The system then can provide optimizations for *passive objects* and remove code and checks required for concurrency so reducing overhead and greatly increasing performance.

Dynamic Allocation and Scheduling Objects should be *dynamically allocated* to processing units in order to leverage the underlying platforms' parallel processing capabilities as good as possible. *Load balancing* mechanisms have to be implemented as well. Moreover, a *scheduling* mechanism usable by objects should be available. Objects can use this *scheduling* mechanism for deferring the processing of messages based on their internal state and the content of the message.

Location Transparency Object instances should provide *location transparency*. Each object has an unique address and objects communicate with each other by calling upon that address. If an object gets relocated in memory or is allocated to another processing unit, it ideally should still be reachable through the same address as before, which provides flexibility in relocating object instances without worrying about updating all references to them.

Temporal Consistency *Temporal consistency* must be adhered for events and requires a synchronisation mechanism. The implementation of such a mechanism must not require the availability of a *global clock* as hardware and memory configuration should be abstracted away. Moreover, the synchronisation mechanism should support inheritance. Ideally, the synchronisation constraints should be separated from the application logic to help developers to concentrate on the currently important aspect when developing applications.

Maintaining Object-Orientation Developers used to object-oriented programming don't want to violate their principles when concurrency is required. Object-orientation helps structuring problems and reduces the complexity of tough problems by dividing them into more manageable sub-problems. Best-practices for object-orientation are well-known and a considerable amount of research has been provided to make object-orientation efficient. A concurrent object-oriented system must provide inheritance, but also must maintain encapsulation. Systems that require synchronisation constraints to be considered separately and do not allow us to inherit or require changing them when creating subclasses, do not satisfy this.

2.7 Theoretical Models

Actor concurrency is a theoretical mathematical model for describing concurrent computation. This section introduces actor concurrency and describes theoretical models competing with it. Actor concurrency itself is described in much more detail in Chapter 3, where it also gets compared to the concurrency approaches introduced here. There exists a wide variety of models formally describing concurrency and they differ in their goals as well as in their approaches. This section describes only important ones that can be compared to actor concurrency and is not intended as a complete enumeration.

The following theoretical models of concurrency will be introduced in this section:

- Actor concurrency
- Communicating sequential processes
- Calculus of communicating processes
- Pi calculus
- Petri nets

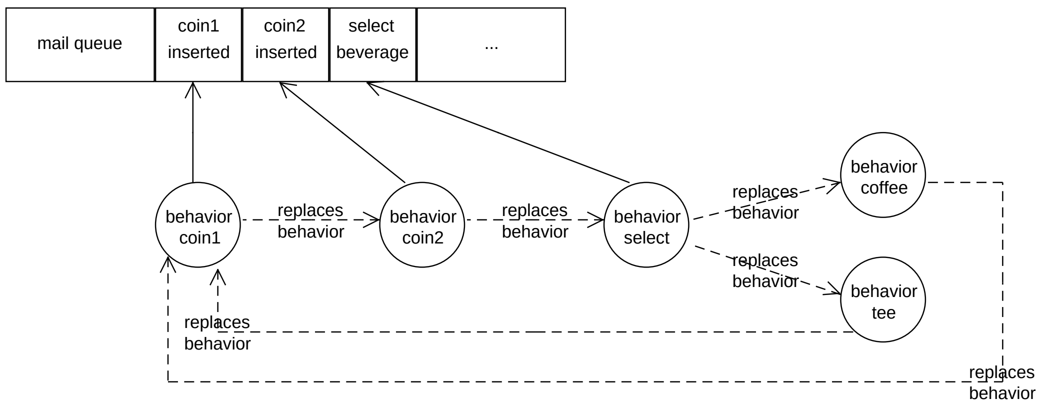

Figure 2.2: Example of an actor system composed of one actor describing a simple vending machine, changing its behavior (Example source: [24], notation source: [5])

Because communicating sequential processes, calculus of communicating processes and the pi calculus are all process calculi and share common concepts, these commonalities get discussed in a separate section before introducing each specific process calculus.

2.7.1 Actor Concurrency

Actors are self-contained entities, not unlike objects in the object-oriented paradigm [46]. Actors hold no shared information whatsoever and exchange information only through messages. Every actor has its own message box for buffering received messages [40]. An actor can send messages, replace its behavior and create new actors. An actor can only communicate with another actor if it knows the other actor's address. Because of the fact that actors do not hold shared information, actors are completely independent of each other and can be executed concurrently; different actors as well as multiple instances of the same actor can run concurrently. For determining the behavior of the whole system each actor can be viewed independently which simplifies analysis considerably. Figure 2.2 shows an exemplary actor system. For a much more detailed view on *actor concurrency* see Chapter 3.

2.7.2 Process Calculus: Common Concepts

A *process calculus* provides a formal mechanism for modelling various aspects of reactive systems [1], as for example concurrency. A process calculus can be used to compose a system out of processes and in the following allows one to reason about the system's properties with the help of mathematical techniques. A fundamental characteristic of the process calculus formalism is that processes communicate exclusively via *communication channels*. This property contrasts to actor concurrency and petri nets, where the communication medium is not explicitly represented [3] in the formalism.

Process Algebra *Robin Milner* first noticed that sequential and concurrent processes can be expressed in an algebraic structure and complex processes can constructively be built out of simpler ones [1]. Later, the term *process algebra* was introduced by *Jan Bergstra* and *Jan Willem Klop* [30]. Today, the notion of a process algebra is used in general for algebraic approaches of studying and formally describing systems composed out of processes. Existing process algebras differ in their terminology, purpose, approach and power of expression [27]. The algebraic approach helps to formally verify properties of concurrent systems as well as to avoid unwanted properties. Process algebra also can be used as the theoretical foundation for implementation of concurrent systems [42].

Atomic Actions In order to describe a system composed of processes, one starts by selecting a set of *atomic actions*. An action is an indivisible behavior which can be executed atomically and on its own [30]. These actions can be choosen freely as no further assumptions are made about an action and nothing except its identity is known about it [27].

Process Terms A *process term* consists of atomic actions linked by process operators. The simplest possible process term is a single atomic action. Process terms can be composed by combining existing process terms through process operators to express complex behavior by iterative composition.

Process Operators *Process operators* allow to iteratively combine simple atomic actions into more complex process terms. Process operators manipulate process terms in a well-defined way and obey algebraic laws, which makes stepwise formal reasoning possible [27]. Every process algebra provides a set of such well-defined process operators. Typical operators include [30]:

- Building operators for building finite processes
- Recursion operators for expressing possibly infinite behavior
- Communication operators for modeling communication
- Special constants, for example null, terminate, or deadlock
- Silent steps for abstracting away internal computations
- Concurrency operators for expressing concurrency

For expressing concurrent behavior with process algebras which we are especially interested in, the last kind of operator is important. Concurrent process algebras provide one or multiple concurrency operators which can be used to divide the system into multiple parts that may be executed concurrently [30].

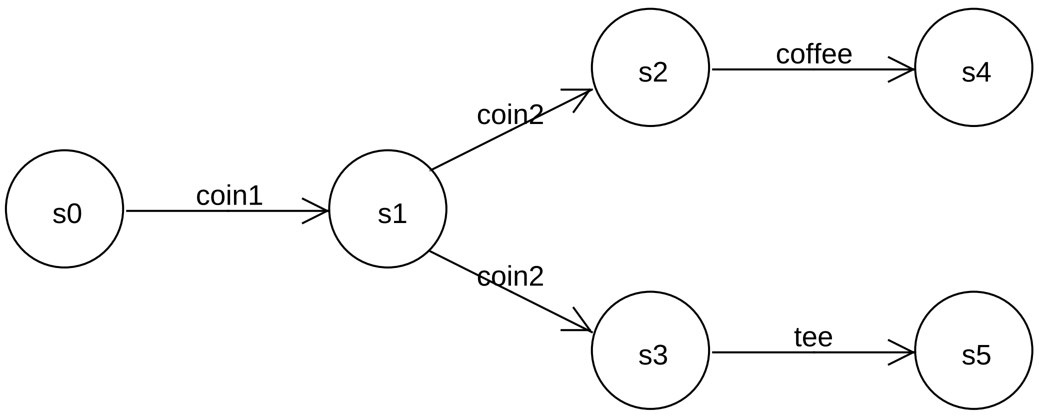

Figure 2.3: Example of a labelled transition system describing a simple vending machine (Example source: [24], notation source: [24])

Labelled Transition System As a starting point for describing systems with a process calculus often a *labeled transition system* (LTS) is used. Formally, a labeled transition system is a triple [1]

$$(Proc, Act, \xrightarrow{a} | a \subset Act) \tag{2.6}$$

where [24]

- *Proc* is a set of states
- *Act* is a set of transition labels or actions
- $a \subset Act$ with $\xrightarrow{a} \subseteq Proc \times Proc$ is a transition relation

Figure 2.3 depicts an exemplary labelled transition system describing a simple vending machine. Its states describe the conditions a process can be in [30]. Transitions between states allow a process to change from one state into the other. The annotated label at a transition is the action or event that leads to the change of state. The possible transitions between states are determined by the transition relation. A labelled transition system can be represented as a graph [30]. In such a graph, the system's states are represented by nodes, the possible transitions between states are the edges between nodes and the possible actions or events on those edges are represented by labels on those edges. One special state is called the initial state. The system starts its operation in this state.

Transformation to Process Terms In order to formally express a process graph in the form of a labelled transition system, it has to be transformed into process algebra terms in a well-defined way. To link parts of a labelled transition system to process algebra terms in a well-defined way, formal *transition rules* are used. Transition rules are inductive proof rules providing process algebra terms that equal parts of a labelled transition systems [30]. Complex labelled transition systems can so be inductively transformed into process algebra terms. Extensions to transformation rules have to be conservative, which means, that whenever a new operator is introduced, it has to be formally proved that transition rules added for the new operator do not change the behavior of already existing operators [30].

Behavioral Equivalence It is very useful to have methods for finding out if two processes have equivalent behavior. *Operational semantics* is one of the most successful ways of examining such equivalences in process algebra [24]. The approach of operational semantics is to describe processes with labelled transition systems and to examine these graphs for possible equivalence [24]. The literature describes different useful equivalences. Two processes are *traces equivalent* [24] if and only if they can perform exactly the same sequences of actions. Two processes are *bisimulation equivalent* [24] if they can simulate each other step by step, the basic idea being, that two states are considered equivalent if we are able to reach equivalent states from them by performing the same sequences of actions. *Testing equivalence* [24] was proposed as an alternative to bisimulation equivalence. Two processes are testing equivalent if they both behave the same seen from the outside.

2.7.3 Communicating Sequential Processes

Tony Hoare introduced and described *communicating sequential processes (CSP)* in the late 70's [42] [30]. Originally, communicating sequential processes were introduced as an imperative parallel programming language [17]. In the paradigm, imperative processes are executed concurrently and these processes communicate with each other by synchronized input and output. Communicating sequential processes takes a black box approach [17] and so a process can be completely described by the communication with its environment.

Processes In communicating sequential processes, a process [42] is defined as the overall behavior pattern of an object. A process has an alphabet describing the events the process can handle. Such an event is atomic and can be arbitrarily chosen by the modeler; events should be granular enough to model the system's behavior, but not too fine-grained for the sake of abstraction and simplification. A process can only engage in events which are contained in its alphabet. There are two special processes defined called $SKIP$ and $STOP$ [42]. The $SKIP$ process represents successful termination, while the $STOP$ process represents a deadlock situation.

Channels and Communications Processes communicate with each other by *communications* over *communication channels*. A communication is an event

$$c.v \tag{2.7}$$

where c is the name of the channel on which the communication is taking place and v is the communicated message's content, or value [42]. A channel can only be used for communication between two specific processes and only unidirectional. This means that a single channel can be used for input or for output, but not for both at once [42], and for modeling bidirectional communication two channels are required.

Operators The following basic operators have been introduced for communicating sequential processes [42] [27]:

- Prefixing: $a \rightarrow P$
- General Choice: $P \mid Q$
- Non-deterministic Or: $P \sqcap Q$
- Parallel: $P \mid\mid Q$
- Interleaving: $P \mid\mid\mid Q$

Prefixing If P is a process and a is an event in its alphabet, then

$$a \rightarrow P \tag{2.8}$$

describes a process in which a is offered until it is accepted and that afterwards behaves like P. This is called *prefixing* [42]. When used with recursion, the prefixing operator can create processes that communicate forever. For example

$$CLOCK = tick \rightarrow CLOCK \tag{2.9}$$

describes a clock that never stops ticking.

Choice Operators In contrast to for example calculus of communicating processes, there are two operators available for expressing choice in communicating sequential processes [42].

The first one is *general choice* [42]. With the general choice operator deterministic choice can be modeled. If a and b are distinct events and P and Q are processes then

$$a \rightarrow P \mid b \rightarrow Q \tag{2.10}$$

describes a process that participates in either event a or event b. When participating in event a, the subsequent behavior of the process is the one described by P, when participating in event b its behavior is described by Q.

The second choice operator is *non-deterministic or* [42]. If P and Q are processes, then

$$P \sqcap Q \tag{2.11}$$

describes a process that either behaves like P or like Q. The selection between P and Q is made arbitrarily and cannot be influenced by the external environment, which makes it impossible to determine in advance which choice will be made. The non-deterministic or-operator also does not provide any guarantee of fairness [27]; so for example P could always be selected in favor of Q.

Concurrency Operators In communicating sequential processes there are two operators for describing concurrency [42]. The first operator is the *parallel operator* [42]. If P and Q are processes and their alphabets share some events, then

$$P \mid\mid Q \tag{2.12}$$

describes a process that behaves like a system composed of P and Q interacting in *lock-step synchronisation*. For events that are in $P's$ and $Q's$ alphabets, both P and Q have to process the event in order for the overall composed process to continue.

The second concurrency operator is the *interleaving operator* [42]. If P and Q are processes, then

$$P \mid\mid\mid Q \tag{2.13}$$

describes a process where P and Q run in parallel without any interaction or synchronisation between them. Each event is processed exactly by one of the two processes. If P or respectively Q cannot process an event, the other one processes it on occurrence; if there are events that could have been processed by either P or Q, the processor is non-deterministically chosen.

2.7.4 Calculus of Communicating Processes

Calculus of communicating processes (CCS) was introduced by Robin Milner in the late 70's [30]. As in other process calculi, processes are the basic building blocks for modeling system behavior in calculus of communicating processes [1].

Processes and Actions The most basic process in calculus of communicating processes is the *0* process [1]. It does nothing at all and is used as starting point for constructing other processes. The *actions* a process can perform are written as *labels*. For example a label eat denotes the action of eating. Labels are atomic and can arbitrarily be chosen by the modeler to fit the system's capabilities. Processes can be given names for easier modeling. For example

$$Philosopher \overset{def}{=} eat \tag{2.14}$$

depicts a philosopher that eats. Naming also allows us to model recursive behavior.

In calculus of communicating processes, processes are described through *process interfaces* [1]. A process interface is a collection of channels or *communication ports* which a process can use to interact with other processes. A communication port has a name for identification and addressing and it is either used for input or for output [1].

Operators The following operators are defined for calculus of communicating processes [1]:

- Action Prefixing: $a.P$
- Choice: $P1 \ + \ P2$
- Parallel Composition: $P1 \mid P2$
- Restriction: $P1 \ \ L1$
- Relabeling: $P1[label1/label2]$

Action Prefixing With *action prefixing* [1] a process can be prefixed with a label which can be used to iteratively construct complex processes. The operator also can be used with the *0* process, acting as inductive start for process construction. If P is a process and a is a label, the expression

$$a.P \tag{2.15}$$

prefixes the process P with the label a. When the action the label represents occurs, the process is said to be stricken. When a process becomes the *0* process after being struck, it is said to be a match. For example

$$a.0 \tag{2.16}$$

is a process that executes action a and then dies.

Choice The *choice operator* [1] is used for describing processes that may follow different behavior depending on the system's state. For example if $P1$ and $P2$ are processes, then

$$P1 \; + \; P2 \tag{2.17}$$

creates a process which can either execute the behavior of $P1$ or $P2$. Once however an action from either $P1$ or $P2$ was performed, it will preempt the further execution of the process that was not chosen. While communicating sequential processes models deterministic and non-deterministic choice with separate operators, the choice operator in calculus of communicating processes can express both deterministic and non-deterministic behavior [27].

Parallel Composition The *parallel composition operator* [1] is used to describe processes running concurrently and possibly interacting with each other. For example

$$P1 \mid P2 \tag{2.18}$$

describes a process in which the sub-processes $P1$ and $P2$ can proceed independently of each other. $P1$ and $P2$ may choose to communicate with each other through their communication ports if they wish to at each time.

Restriction Operator The *restriction operator* [1] hides a communication port of a process from the outside, restricting the ports use to a well-defined set of processes. For example if $P1$ is a process and $L1$ is a communication port, then

$$P1 \; L1 \tag{2.19}$$

means that only process $P1$ can access communication port $L1$.

Relabelling Operator The *relabelling operator* [1] replaces one label with another one. It can be used to provide abstractions and allows one to define a process with basic general behavior patterns, whose generic actions can then be exchanged to specific ones later. For example if $P1$ is a process and $label1$ and $label2$ are labels, then

$$P1[label1/label2] \tag{2.20}$$

means that where $label2$ is present, it will be replaced recursively with $label1$.

2.7.5 Pi Calculus

The *pi calculus* is a relatively young offspring of the process calculus family. It was developed by *Robin Milner*, *Joachim Parrow* and *David Walker* in 1992 [55]. One of the main motivations for developing pi calculus was to support easier modeling of dynamically changing systems. The pi calculus is especially interesting here as it was influenced by how actor concurrency models dynamic systems and tries to combine the benefits of actor concurrency with the ones of process calculus.

Agents and Names Processes are called *agents* in pi Calculus [60]. The *empty agent*, written as *0* does not perform any actions and acts as the iterative starting block for constructing more complex agents. A set of arbitrarily chosen *names* functions as all of *communication ports*, *variables* and *data values*. As usual in a process calculus, communication between agents is performed over communication channels. A speciality of the pi Calculus however is that an agent can dynamically change with whom it communicates during its lifetime.

Identifiers The pi calculus provides *identifiers* [60] for agents. These identifiers are defined as a set of functions with fixed non-negative arities. Each identifier must have a definition, which can be seen as a process declaration. An identifier

$$A(y_1, \ldots, y_n) \tag{2.21}$$

where n is the arity of the identifier A and a definition

$$A(x_1, \ldots, x_n) \overset{def}{=} P \tag{2.22}$$

together behave as an agent P with y_i in the identifier replaced by x_i for each i and can be thought of as an invocation of the identifier with the actual parameters $y_1 \ldots y_n$.

Links The pi calculus also provides *links* [60]. A link is a reference to a communication channel and can be shared between agents over other communication channels. An agent holding a link can use it for communication over the communication channel referenced by the link as it would do with any other communication channel it already has access to. The link mechanism makes modeling agents with dynamically changing interactions possible [60]. Almost all of the increased complexity of pi calculus in comparison to other process models originates from links and the fact that they can be communicated between agents.

Operators Most of the operators in pi calculus have the same semantic meaning as in other process calculi. The basic variant of pi calculus has the following operators [60]:

- Input-, output- and silent prefix: $a(x).P$ and $\bar{a}x.P$ and $\tau.P$
- Sum and parallel composition: $P \;+\; Q$ and $P \mid Q$
- Restriction: $(\mathbf{v}x)P$

- Match and mismatch: $if\ x = y\ then\ P$ and $if\ x \neq y\ then\ P$

A relabelling operator does not exist in pi calculus, as the primary use of relabelling is the definition of new agents from existing ones, and agents in pi calculus are defined through identifiers and the parameters given to them instead.

Prefixing Prefixing comes in three variations in pi calculus. *Input prefix* and *output prefix* are for sending and receiving names, *silent prefix* evolves an agent's behavior.

The *input prefix* operator allows an agent to receive and store input over a communication channel. For example

$$a(x).P \tag{2.23}$$

means that a name is received as a value along the input port named a and will be stored in the placeholder x. After the input value has been received, the agent continues with the behavior described by P, but with the value received replacing the content of x.

The *output prefix* operator allows an agent to send away data over an output port. For example

$$\bar{a}x.P \tag{2.24}$$

means that value x is sent along the port named a. After the send operation, the agent continues with the behavior described by P.

Silent prefix can be used to construct an agent that evolves and changes its behavior without any interaction with the environment. For example

$$\tau.P \tag{2.25}$$

means that the agent P changes its behavior.

Sum and Parallel Composition *Sum* and *parallel composition* have the same meaning as the corresponding operators in other process calculi.

With the *sum operator*

$$P\ +\ Q \tag{2.26}$$

an agent is constructed which either behaves like P or Q.

Parallel composition

$$P \mid Q \tag{2.27}$$

describes an agent behaving as P and Q executed in parallel. P and Q can choose to communicate with each other if required.

Restriction The *restriction operator* hides names from the outside world. For example,

$$(\nu x)P \tag{2.28}$$

hides name x, which allows agent P to access it exclusively. The operator works mostly equivalent to the ones described for other process calculi. Remember however, that in pi calculus communication ports can be transmitted between agents. So while the communication port x mentioned in the example can so far only be used by P, P can choose to transmit it to other agents which then subsequently also can make use of it.

Match and Mismatch The *match operator* and the *mismatch operator* are used to compare names.

The *match operator* checks names for equality. For example

$$if\ x = y\ then\ P \tag{2.29}$$

states that if names x and y are equal, the agent will behave like P and otherwise will do nothing.

The *mismatch operator* checks for inequality. For example

$$if\ x\ \neq\ y\ then\ P \tag{2.30}$$

states that if names x and y are not equal, the agent will behave like P and otherwise will do nothing.

At first glance it seems to be restrictive to only provide match and mismatch as test operations, but actually those are the only meaningful ones that can be performed on names in the pi calculus, as transmitted names do not have any internal structure and can be arbitrarily chosen.

2.7.6 Petri Nets

Petri nets have been introduced by *Dr. Carl Adam Petri* in 1962 [68]. Petri nets are a flexible mathematical formalism and can be represented graphically for easier analysis. They can be used to model a varied assortment of aspects of static and dynamic systems and many applications for petri nets exist in computer sciences [68]. With petri nets, all important aspects of concurrency such as synchronisation and mutual exclusion can be modeled. While classical petri nets have no mechanism for representing priorities, the theory can be expanded to model priority aspects as well.

Petri Net Graphs A petri net is a bipartite directed graph with four kinds of objects in it [68]:

- Places
- Transitions
- Directed arcs
- Tokens

Places and *transitions* are connected through *directed arcs* labeled with a weight. Every place can hold null or a positive number of *tokens*. A directed arc always connects a place and a transition; it never can connect two places or two transitions directly without a place or respectively a transition in between. A transition that is not connected to an input place is called a *source transition*. It is always and unconditionally enabled for firing. A transition that is not connected to an output place is called a *sink transition*. When firing, a source produces new tokens without consuming any and a sink consumes tokens without producing any. A place and a transition are called a *self-loop* if they are connected in a loop through directed arcs. A petri net is called *pure* if there are no self-loops in it.

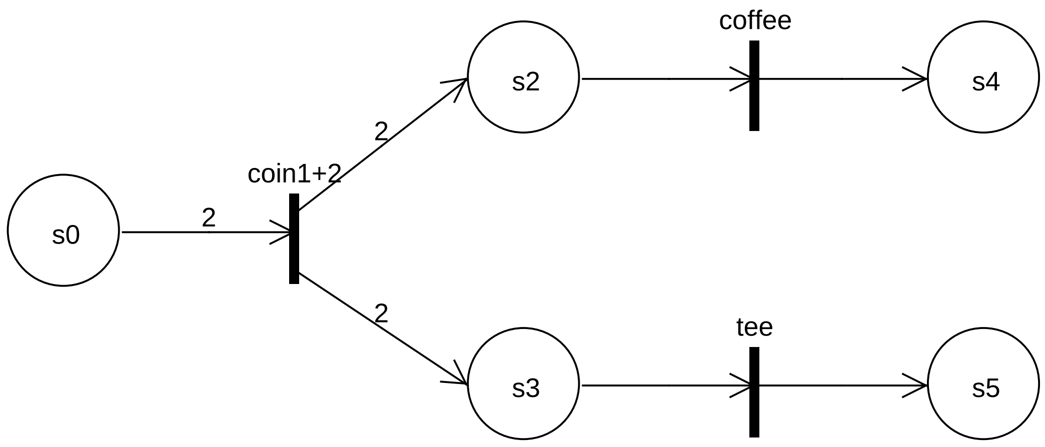

Figure 2.4: Graphical representation of a petri net describing a simple vending machine (Example source: [24], notation source: [59])

Graphical Representation For graphical representation petri nets can be drawn as a diagram. Figure 2.4 shows the graphical representation of an exemplary petri net. The usual convention is, that circles represent places, boxes or black bars represent transitions and edges with a weight represent directed arcs connecting them. Arcs can only be drawn between a circle and a box, respectively between a place and a transition, and vice versa.

Formal Definition A petri net can be represented formally as a 5-tuple [68]:

$$N = (P, T, I, O, M0) \tag{2.31}$$

where P and T are finite sets of places and transitions defined as

$$P = \{p1, \ldots, pm\} \tag{2.32}$$

and

$$T = \{t1, \ldots, tn\} \tag{2.33}$$

with the additional conditions that $P \cup T \neq \emptyset$ and $P \cap T = \emptyset$. I and O are an input and an output function, defined as

$$I : P \times T \to N \tag{2.34}$$

and

$$O : T \times P \to N \tag{2.35}$$

where the input function defines arcs from places to transitions and the output function defines arcs from transitions to places, with N being a set of non-negative integer values. Finally, an initial marking

$$M0 : P \times N \tag{2.36}$$

of tokens is required, assigning tokens to places.

Execution of Petri Nets A petri net is executed through changing the location and number of tokens associated to its places [68]. Firing transitions defines how that happens, and two rules determine the flow of tokens through the net.

The *enabling rule* models which transitions can fire. A transition is enabled if each of its input places contains at least the number of tokens equal to the weight of the directed arc. Formally, this is defined as

$$M(p) \geq I(t, p) \tag{2.37}$$

The *firing rule* models the flow of tokens from input place(s) through directed arc(s) to output place(s). Only enabled transitions can fire. Firing removes the number of tokens written as weight at the directed arc from place to transition from the input place and deposits the number of tokens written as the weight at the directed arc from transition to place at the output place. Formally, this is defined as

$$M'(p) = M(p) - I(t, p) + O(t, p) \tag{2.38}$$

Modeling Dynamic and Non-Dynamic Behavior For modeling non-dynamic behavior, the triple of places, transitions and directed arcs is sufficient and can be used to represent different non-dynamic aspects of the modeled system. For modeling dynamic behavior tokens are used as they can represent a system's state and state change over time. Every place can hold null or a positive number of tokens and presence or absence of tokens on a place indicate a condition previously associated with it to hold or not to hold at the current time. The specific meaning of a token is not predefined and it depends on the dynamic system and the properties the modeler wants to represent. For example, tokens can reflect events or the execution of operations. An assignment of tokens to the places of a petri net is called a *marking*. When executing a petri net, the tokens get moved from place to place representing the modeled system's dynamic behavior.

Modeling Examples With petri nets, various aspects of dynamic systems can be modeled. For this thesis, the following aspects are the most interesting ones [68]:

- Sequential execution
- Concurrency
- Synchronisation
- Mutual exclusiveness
- Priorities

For modeling *sequential execution*, transitions and places can be connected in series (Figure 2.5 a). For the second transition to be able to fire, the first one already must have fired. With this approach, precedence constraints and causal relationships can be modeled.

For modeling *concurrent execution* a forking transition is required (Figure 2.5 b). When firing, the forking transition plants tokens in multiple output places. From the forking on the execution happens concurrently.

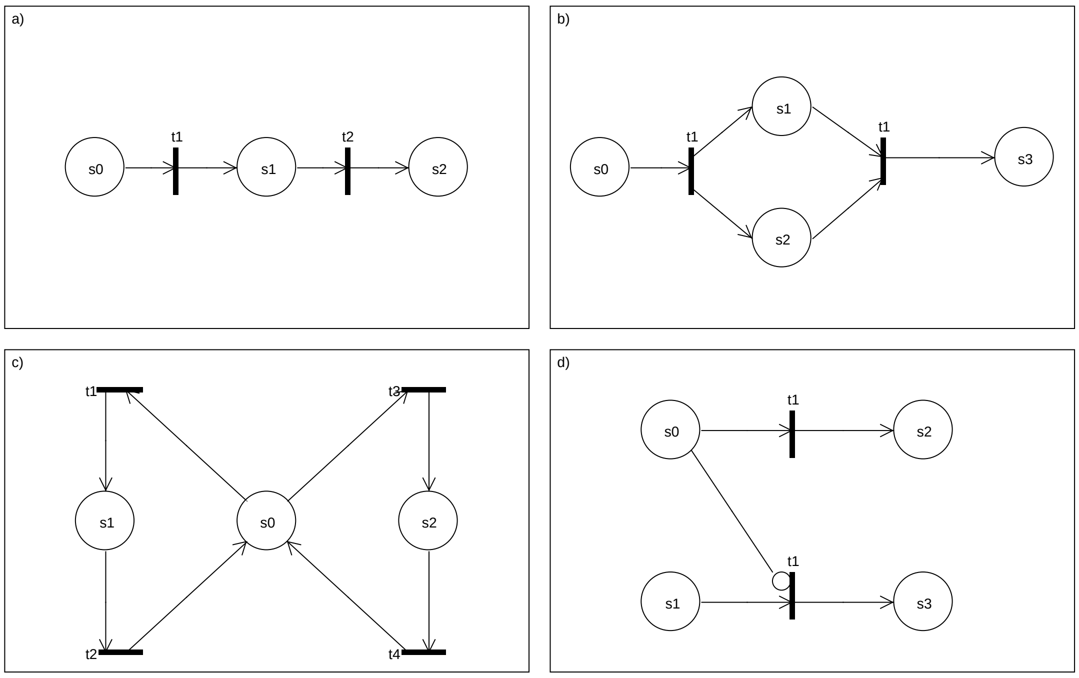

Figure 2.5: Modeling aspects of dynamic systems with petri net graphs. a) Sequential Execution, b) Concurrency and Synchronisation, c) Mutual Exclusion and d) Priorities (Source: [68])

The *synchronisation* process after concurrent execution can be represented by two or more places connected with a joining transition (Figure 2.5 b). The joining transition can only fire if it gets tokens from all connected input places.

Mutual exclusiveness can be modeled through places and transitions connected in a pattern that does not allow transitions between the two paths (Figure 2.5 c).

Modeling *priority constraints* requires introduction of *inhibitor arcs* (Figure 2.5 d), as standard petri nets have no mechanism for modeling such constraints. An inhibitor arc connects a place to a transition. The presence of an inhibitor arc changes the transition's firing conditions and the transition is enabled only if all input places have the necessary amount of tokens and no tokens are available at places connected through inhibitors arcs.

CHAPTER 3

Actor Systems

Actor systems are a powerful formalism built on the *actor concurrency model*, allowing us to describe and implement concurrent systems. Actor systems consist of self-contained, autonomous entities called actors [46]. The concurrency model was introduced by *Carl Hewitt* in the late 70ies [40] and is heavily influenced by advances of modern physics, especially *special relativity*. Actor concurrency provides *true concurrency* by nature and integrates concurrency very well with *object-oriented programming*. Actors provide abstraction from low-level concurrency issues, allow dynamically changing topologies, and make it easy to reason about each actor independently.

3.1 Overview

Actors are self-contained, autonomous entities [40] [46] similar to objects in the object-oriented paradigm. Figure 3.1 shows a graphical representation of an actor, its components and capabilities. Actor instances run completely independent of each other and so different actors can run concurrently as well as one actor can have multiple instances running concurrently [40]. As actors do not share any state among them, mechanisms for dealing with mutual exclusion are not required in the paradigm.

Exchanging Information Actors exclusively exchange information by sending and receiving asynchronous messages. Every actor has an uniquely identifiable *mailbox*, or *address* [40], which decouples sending and receiving of a message from each other. At its creation, an actor has an initial set of addresses of other actors it knows of. Through its lifetime, an actor can gain knowledge of further addresses by either creating actors or receiving messages containing actor addresses.

Operations An actor can perform the following four operations [40]:

1. Create: Create a new actor

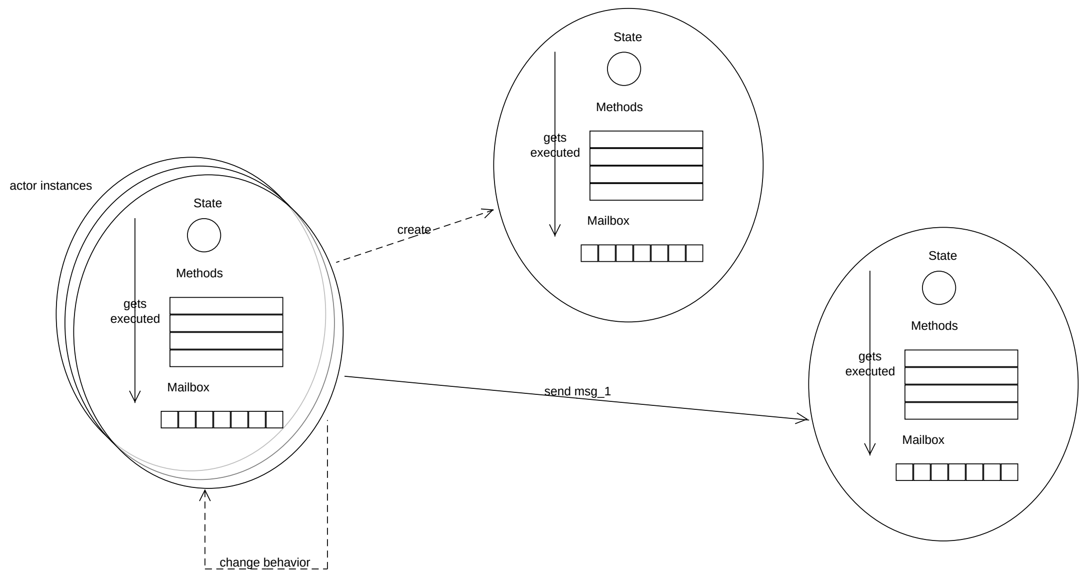

Figure 3.1: Graphical representation of actors. Actors can create new actors, send and receive messages and replace their own behavior (Source: [46])

2. Send message: Send a message to another actor
3. Receive message: Receive a message
4. Become: Replace the own behavior

Creating New Actors An actor can create a new actor. After creation, the creator immediately continues processing and the created actor concurrently starts waiting for messages. When an actor creates another actor, the creator gains knowledge of the new actor's address and can send messages to it.

Sending Messages Actors communicate exclusively over messages. As communication in the actor paradigm is asynchronous, the sender sends the message away and immediately continues processing. If an actor wants to send another actor a message, it requires the other actor's mailbox address, often simply called the actor's address. There are three ways an actor can gain knowledge of another actor's address: The address is contained in the actor's initial set of known actors, the actor is the other actor's creator and therefore knows its address, or the actor has received the address in a message.

Receiving Messages Every actor has its own mailbox and a received message gets stored in that mailbox until the actor chooses to process it. The mailbox mechanism decouples sender and receiver from each other. It allows the sender to immediately proceed after a send operation and the receiver to continue its current processing when receiving a message, processing it later when it has time to do so in a coordinated manner. Actors can run completely independent of each other, because receiving messages is the only way an actor can be influenced from the outside.

Changing Behavior Finally, an actor can decide to change its behavior, allowing actors to dynamically evolve over time depending on their message history. For example the actor can decide that its behavior is not appropriate any more after having received a certain message and can replace its current behavior with a different one. Every message the actor receives from now on gets processed with the actor's new behavior.

3.2 Event Orders and Laws

When Hewitt introduced actor concurrency, he noticed different event orders influencing the behavior of an actor system [40] and that the analysis of the order of events stands at the very foundation of the actor concurrency model. The following subchapter illustrates the topic in as much detail as necessary for this thesis.

Events An *event* is a discrete step in the history of computations in an actor system. Every event e consists of the receipt of a message by a recipient and both recipient and message are actors themselves in the theory (however, some implementations of the actor model do not require messages to be actors).

Event Order Relations There are three important order relations for events [40]:

- Activation Order: Causality between events
- Arrival Order: Order of message arrival at actors
- General Precedes Relation: Combination of activation and arrival order

3.2.1 Activation Order

The *activation order* [40] is derived from how events cause or activate one another. Intuitively the activation order can be seen as relation describing causality between two events. It is written as

$$E_0 \;{+}{+}{>}\; E_1 \tag{3.1}$$

which means that E_0 precedes E_1.

Example Figure 3.2 shows an example. If actor X receives a message M_0 as event E_0 and as a result of this sends a message M_1 which actor Y receives as event E_1 then the event E_1 is said to be activated by E_0 and E_0 is called the activator of $E1$. In other words, event E_0 caused event E_1. Every event can have one activator at most. The activation order is a partial order, because one event might activate several other ones.

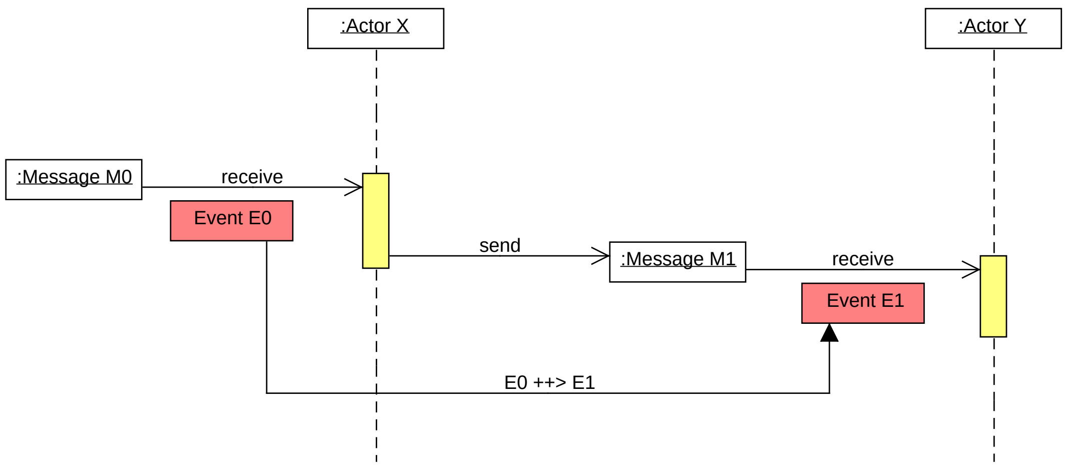

Figure 3.2: Exemplary activation order of two events $E0$ and $E1$. Actor X receives message $M0$ in event $E0$ and as a result sends message $M1$ which actor Y receives in event $E1$. Event $E1$ is said to be activated by event $E0$.

Laws The following laws were defined by Hewitt for activation order:

- Discreteness
- Events have at most one predecessor
- Events are successor to a finite set of events

First of all, activation order must be discrete. What this formally means is, that if E_1 ++> E_2 then the set

$$\{E \mid E_1 \text{ ++> } e \text{ ++> } E_2\} \tag{3.2}$$

of events between E_1 and E_2 has to be finite. Discreteness makes sure that the immediate successor and the immediate predecessor of an event are well-defined and that computing models which assume infinitely fast machines are eliminated from actor concurrency.

An event can be caused by only one event at most, which is called the *immediate predecessor* of the event; an event that has no predecessor is called an *initial event*. This means that two distinct events cannot be both the immediate cause of another event. Formally defined, for all events E the set of immediate predecessors

$$\{E \mid P1 \text{ ++> } E \land \neg\exists P2 \ : P1 \text{ ++> } P2 \text{ ++> } E\} \tag{3.3}$$

can have at most one element.

Finally, an event can only be successor to a finite set of other events. Formally this means, that for all events E the immediate successor set

$$\{E \mid E \text{ ++> } S\} \tag{3.4}$$

is finite.

3.2.2 Arrival Order

Activation order alone is not enough to describe actors with side effects. If the state of an actor changes when it receives a message, the order in which messages arrive is important. *Arrival order* [40] specifies the order of message arrival for an actor. It is important to note, that while the activation order is unambiguously and globally the same for the whole system, the arrival order is specific to each actor and is heavily affected by influences from the environment such as scheduling, message transmission and other factors. The arrival order is written as

$$E_1 =>_Y E_2 \tag{3.5}$$

meaning that event E_1 has arrived before event E_2 at actor Y.

Example Figures 3.3 and 3.4 show two possible arrival orders which could occur for an exemplary actor system. In both cases actor X receives a message M_0 in event E_0 and as resulting action sends two messages M_1 and M_2 in the order $M1$ M_2 to actor Y. Actor Y receives these messages in the events E_1 and E_2. In one instance (Figure 3.3) actor Y receives the events in the order E_1 E_2. In the other one (Figure 3.4) actor Y receives the events in the order E_2 E_1.

Consequences The consequences of this observation are immense. For actor Y message M_1 arrives before message M_2 in the first case, while message M_2 arrives before message M_1 in the second case. This means, that while actor X always sends the messages in the same order, actor Y may receive it in a different one and the actors cannot agree on the order in which the messages M_1 and M_2 happened.

Laws Hewitt specified the following laws for the arrival order:

- An event has finitely many predecessors
- The arrival order is a total order

Only a finite number of events precede an event. So for all actors X and all events E, the set

$$\{E' \mid E' =>_X E\} \tag{3.6}$$

has to be a finite set. As a corollary to this law, the arrival order is discrete.

The arrival order for each actor also has to be a total order. If $E1$ and $E2$ are distinct events and both events are received by actor X, then either

$$E_1 =>_X E_2 \tag{3.7}$$

or

$$E_2 =>_X E_1 \tag{3.8}$$

This means that the order of events has to be well-defined and one of the events has to precede the other one, so either E_1 arrives before E_2 or E_2 arrives before E_1 at actor X. The process of enforcing the order in which an actor receives events is called *arbitration*.

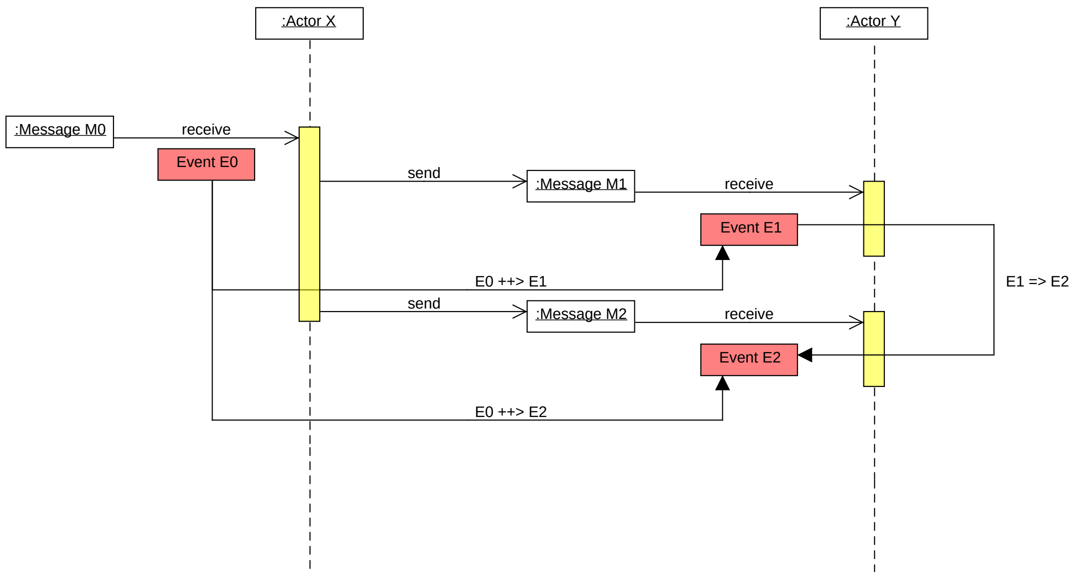

Figure 3.3: Example of a possible activation order. In this case, actor Y receives the messages M_1 and M_2 in the order M_1 M_2.

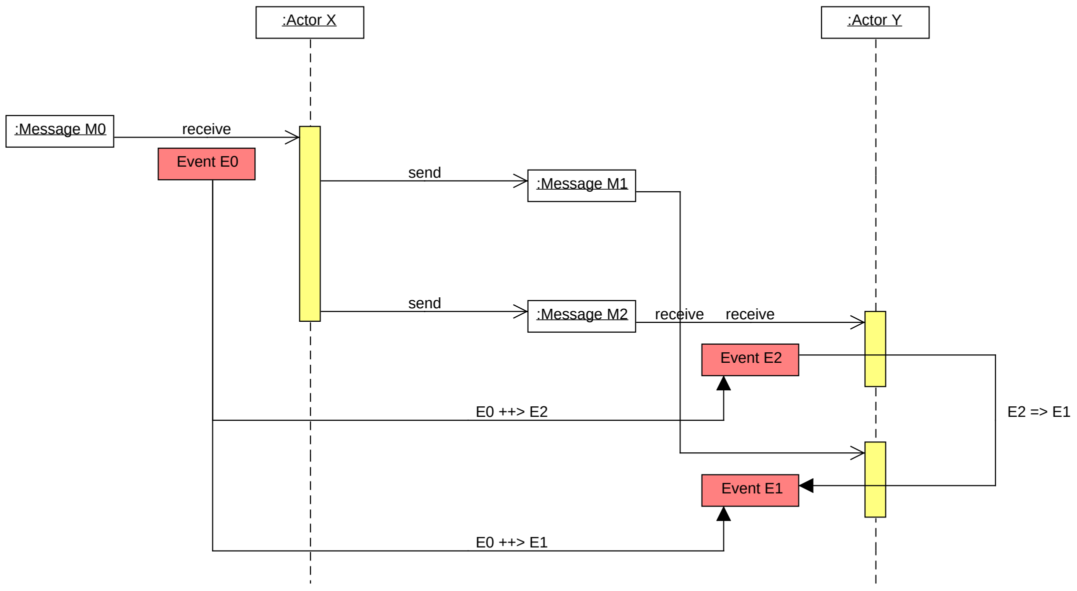

Figure 3.4: Example of another possible activation order. In this case, actor Y receives the messages M_1 and M_2 in the order M_2 M_1.

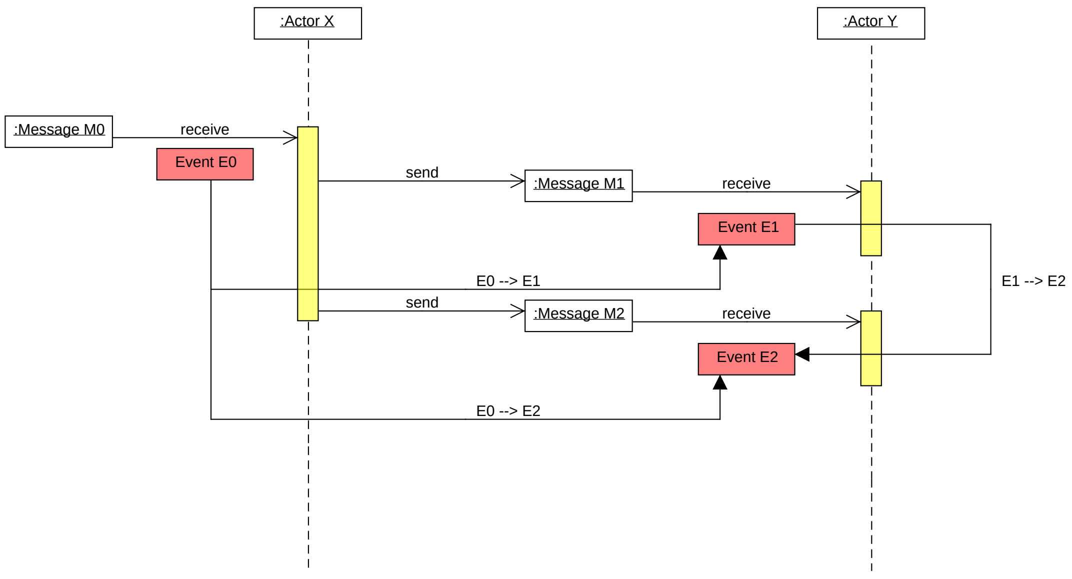

Figure 3.5: Example of the general precedes relation. In this specific example event E_0 generally precedes events E_1 and E_2 and event E_1 generally precedes event E_2.

3.2.3 General Precedes Relation

In order to relate activation order and arrival order to a notion of time, Hewitt defined another event order relation called the *general precedes* relation [40]. It is defined as the transitive closure of the union of activation order and arrival order for every actor. It is written as

$$E_1 \; --> E_2 \tag{3.9}$$

which means that event E_1 generally precedes event E_2. In the general precedes relation an event E can have at most two immediate predecessors: One that is the event's activator in the activation order relation and another one that is the event's immediate predecessor in the arrival order relation.

Example Figure 3.5 shows an example for how the general precedes relation works. Here actor X receives message M_0 in event E_0 and as a result sends two messages M_1 and M_2 to actor Y, which actor Y receives in the events E_1 and E_2. In the given example event E_0 generally precedes the events E_1 and E_2 because E_0 is the activator for E_1 and E_2 in the activation order, and event E_1 generally precedes event E_2 because E_1 is the direct predecessor of E_2 in the arrival order.

Laws For the general precedes relation Hewitt specified the following laws:

- Discreteness

- Law of causality

The general precedes relation is a discrete relation. While activation order relation and arrival order relation are discrete, this unfortunately does not imply that the general precedes relation is discrete as well. This is the reason that discreteness has to be formally stated as

$$--> \text{ is a discrete relation} \tag{3.10}$$

for the general precedes relation.

In order for the general precedes relation to correctly express event precedence, the activation and arrival of events have to be consistent. This is guaranteed by the law of causality which states that for no event E the statement

$$E --> E \tag{3.11}$$

holds, meaning that no cycles are allowed in the causal chain of an event, because an event cannot cause itself directly or indirectly.

3.2.4 More Actor Laws

Hewitt also specified two more important laws on actors that are not related to the aforementioned relations: The *law of creation* and the *law of locality*.

3.2.4.1 Law of Creation

The *law of creation* [40] specifies that an actor cannot be used before it has been created. While this seems trivial, it is nonetheless an important statement. In order to be able to formally express this statement, a unique special event $birth(X)$ is introduced for each actor X and for which

$$birth(X) --> E \tag{3.12}$$

holds for every event E in which actor X participates in. This formally states the notion that an actor X can only be used after its creation.

3.2.4.2 Law of Locality

The *law of locality* [40] states, that an actor can acquire an other actor's address only by either directly meeting the actor or by indirectly meeting an acquaintance of that actor which knows the actor's address. The law for general actor systems is quite subtle and here gets only presented for actor systems that do not change over time and with actors that have a fixed set of extended acquaintances. In order to formulate the law correctly, a few definitions are required.

Conception Event First, the birth of an actor during a computation has to be formalized as a *conception* event. If an actor X is conceived as result of a computation, then the conception can be formulated as

$$conception(X) = activator(birth(X)) \tag{3.13}$$

stating that the conception causes the birth event of actor X.

Acquaintances Every actor X has a set of immediate acquaintances defined as

$$acquaintances(X) \tag{3.14}$$

The acquaintances of actor X represent the set of actors it knows and can send messages to. An actor's set of acquaintances can change over time when the actor receives messages containing actor addresses which subsequently get included in the set. The extended acquaintances of actor X are defined as

$$acquaintances^{*}(X) = (X) \cup acquaintances(X) \cup acquaintances^{2}(X)) \cup \ldots \text{ (ad infinitu)} \tag{3.15}$$

So the set of extended acquaintances for an actor is the actor itself, its acquaintances, the acquaintances of its acquaintances and so on.

Event Participants If $target(E)$ is the actor targeted by an event E, $messenger(E)$ is the associated message and $conceived(E)$ is the possibly empty set of actors conceived during the event, then the extended participants $participant^{*}(E)$ of event E are defined as

$$participants^{*}(E) = acquaintances^{*}(target(E)) \cup acquaintances^{*}(messenger(E)) \tag{3.16}$$

So the extended participants of event E are the extended acquaintances of the actor that receives the event combined with the extended acquaintances of the message belonging to the event.

Law of Locality The law of locality itself is formulated in two parts. First, for all actors X, the statement

$$aquaintances(X) \subseteq participants^{*}(conception(X) \cup conceived(X)) \tag{3.17}$$

is required to hold. The equation states that an actor can only know about actors which were known when it was conceived (the actors are extended participants of its conception event).

Secondly, for all events E, the statement

$$participants(E) \subseteq participants^{*}(activator(E) \cup conceived(activator(E)) \tag{3.18}$$

is required to hold. This means, that the receiving actor and the message belonging to event E must have been known to the participants of its cause (the activator event).

3.3 Concurrency Model

Actor concurrency has wide-reaching consequences which hold for concurrent and distributed systems in general. The model shows, that it is impossible to define a global event order on which all components of a concurrent system can agree upon [40]. It also makes clear, that the arrival order of events in a realistic distributed concurrent system is undefined [4].

3.3.1 Impossibility of a Global Event Order

One of the central statements of the actor concurrency model is, that a single unique global clock cannot be defined for concurrent systems [4] [40] [21]. It is impossible to establish a single global order of events on which all components of the system can agree upon, because each component has a local understanding of the order in which events arrive. In the actor concurrency model this is expressed by the fact that the arrival order relation is local for each actor [40]. While all actors can agree on the causal relationships between events defined by their activation order, they cannot agree on the order in which events occur and events occurring at different components are inherently unordered unless they are connected by causality. Formally this means that the global ordering of events is a partial order [4].

The idea behind this statement is analogous to *special relativity* by Albert Einstein [4]. Special relativity states, that there can not exist a global clock every viewer can agree upon, because there is a fundamental upper limit on how quick information can travel from one viewer to the other (light speed), which in consequence dictates that different viewers may notice the same event at different times.

For components of a distributed concurrent system the same is true: Each component holds information in the form of its current state. If one assumes, that there is a limit to how fast information can be transmitted between components, the state as it is seen by the component itself can be different from the state seen by components watching it [4].

3.3.2 Arrival order indeterminacy

Another important statement of actor concurrency is, that because of the inevitable nondeterminism in any concurrent, distributed environment, the order in which events are regarded by components of such a system is not strictly defined [4]. In any such system it is therefore not possible to precisely predict when a sent message will arrive at one of its components. The order in which a component regards arriving messages or events is determined by their local arrival order, and a realistic model of concurrent systems must assume that the arrival order of messages or events is arbitrary and unknown [4]. Constructing synchronously functioning systems requires to define protocols for arbiting the arrival order [4] and the prototype described in this thesis is one example of how to do just that.

To make the issue clearer, here is an example (Figures 3.6 and 3.7). Suppose we have three actors A, B and C. Actor A sends a message to actor B and another one to actor C requesting a computation from both actors. Actors B and C receive the messages and do the processing. Actor B finishes before actor C and returns the result to actor A. Later actor C finishes as well and returns the result to actor A. Now from actor A's perspective two things can happen: Either it receives the result from actor B before the one of actor C (Figure 3.6) or it receives the result from actor C before that of actor B (Figure 3.7). Unfortunately in a distributed concurrent system, there is no guarantee on the order of events so both outcomes of the scenario are equally possible.

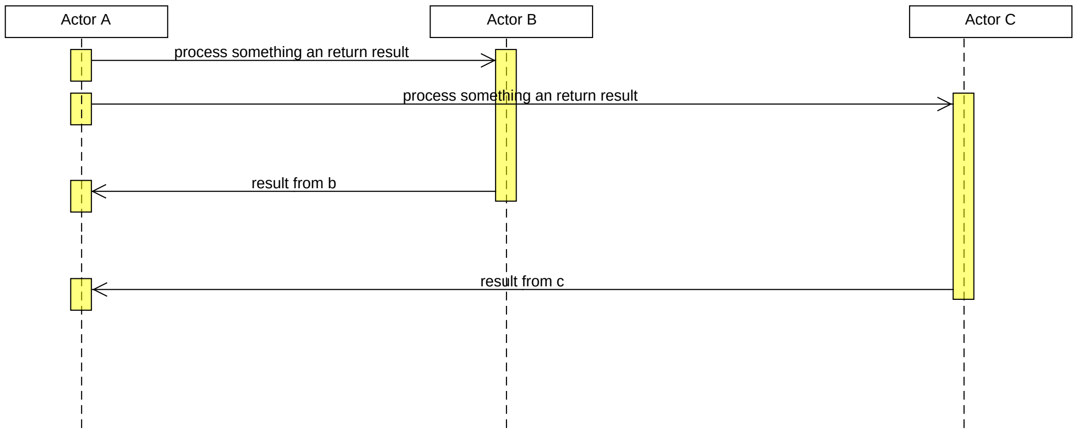
Actor A
Actor B
Actor C
process something an return result
process something an return result
result from b
result from c

Figure 3.6: Local event order for actor A with the result message from actor B arriving before the one from actor C

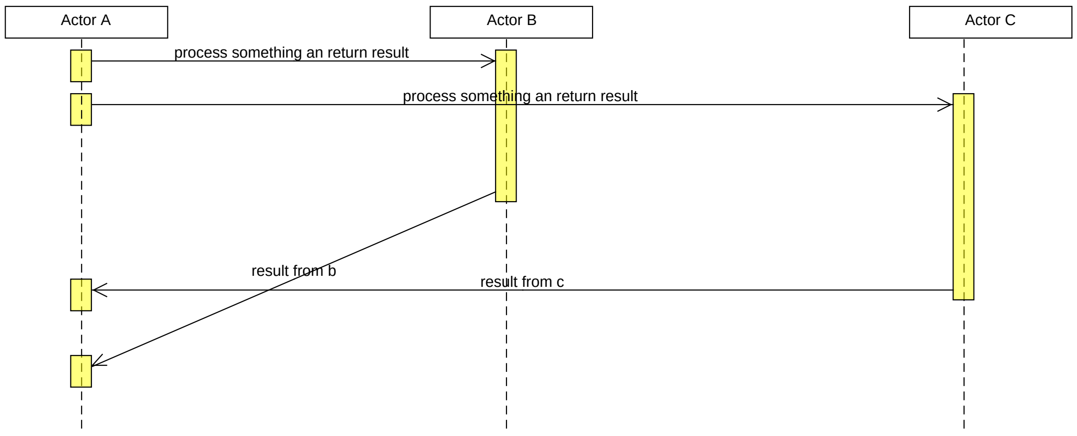
Actor A
Actor B
Actor C
process something an return result
process something an return result
result from b
result from c

Figure 3.7: Local event order for actor A with the result message from actor C arriving before the one from actor B

3.4 Benefits and Limitations

3.4.1 Benefits

It is no wonder that actor concurrency is gaining interest recently, as it provides a very elegant and hassle-free view of concurrent programming on a high abstraction level. The actor concurrency model has the following benefits [46] [61] [4] [8] [66] [52] [53] [6]:

- No shared state
- Asynchronous communication and lack of interruption
- Inherent concurrency and actor distribution
- Abstraction from low-level concurrency issues
- Support for object-orientation

No Shared State The actor concurrency model requires actors to avoid sharing of internal state [46]. Actors that adhere to that principle run completely separated and are influenced only by messages sent to them [8]. Actors can be examined separately, which makes analyzing such systems much easier. However sending messages containing internal state, for example references or pointers to an actor's variables, obliterates the actor concurrency model's benefits regarding the independence of actors. Messages sent between actors should therefore always be immutable or have at least call-by value semantics which requires making a copy of the message's content [46].

Asynchronous Communication and Lack of Interruption Communication between actors is asynchronous [46]. Each actor's message box buffers incoming messages and so decouples the send operation from the receive operation. Arriving messages get buffered in the actor's mailbox and the actor itself freely chooses when to process a message. This makes actors interruption-free [61]. Consequentially this means, that the computation time is a deterministic function of the incoming message [61], and the time the computation of a processed message requires depends only on the behavior of the invoked actor.

Inherent Concurrency and Actor Distribution Actors inherently support concurrency [4]. The only constraints on concurrent execution and execution speed are the logical dependencies between computations and the hardware's limits [8]. Different actors can be executed in parallel as well as an individual actor can have multiple instances of itself running in parallel if the system's current work load and resources permit it. Actors can take advantage of multiple threads and, even more important, of multiple cores and processors. In many actor concurrency implementations threads get pooled and actor instances are executed on these pooled threads [66] [46]. Some implementations also utilize light-weight threads building on operating system or host programming language threads, but requiring less overhead.

Because the actor concurrency operates with asynchronous communication [4], actors also very naturally support distribution. Actors can easily be distributed on different cores, processors or machines, which also makes them interesting for systems distributed over network. Some modern actor concurrency implementations, for example *AKKA* [66], provide support for distributing actors over common network protocols out-of-the box.

Abstraction from Low-Level Concurrency Issues Developers should not constantly have to think about where and how to synchronize shared data, how to design their application to prevent deadlocks and how to avoid inconsistencies and other problems occurring in concurrent systems. Actor concurrency reduces the details to consider as it raises the abstraction level programmers can use for writing concurrent programs and also allows them to separate scheduling aspects from logical aspects [61]. Actor concurrency only requires specification of the logical order of events, detailed scheduling aspects can be specified separately. Mutual exclusion issues do not arise in actor concurrency, because actors do not share any state and can be influenced only by sending them messages which get buffered if an actor is not able to process the message immediately [4].

Another important issue in concurrent systems is deadlock prevention [4]. In paradigms with synchronous communication, a deadlock is defined as a condition where no process can communicate with another. From a syntactical point of view, an actor system cannot deadlock, because communication between actors is always asynchronous, messages get buffered and actors never wait for a reply. Semantically, however, an actor system can be seen as deadlocked if its actors keep trying to send the same message over and over again without progressing, a condition similar to a livelock.

An actor has only two side effects [52]. An actor sends messages to other actors and it can change its internal behavior. As actors also are encapsulated and do not share state [46] it is easy to reason about the behavior of actor systems. To examine the behavior of a single actor, we only have to know which messages it can receive and how and when it changes its behavior. From there on we only have to take the behavior of the actor itself into consideration.

Support for Object-Orientation As previously discussed (see Section 2.6.1), any programming paradigms supporting concurrency and object-orientation in an integrated and consistent fashion has specific requirements [53]. Actor concurrency supports most of these requirements quite well. As actors are implemented as objects, provide encapsulation by design and most implementations provide inheritance, actors support an object-oriented development approach [6]. Actor concurrency naturally provides active objects in the form of actors running inherently concurrent and most implementations allow for the use of the host language's objects as passive objects avoiding most of the overhead for concurrency. Because actors run separately from each other and do not share internal state, scheduling and dynamic allocation to processing units can be implemented easily. Location transparency is also provided, as actors have an unique address which does not change on relocation.

In actor concurrency, temporal consistency has to be achieved through division into actors and through their behavior. In the purest form, the model does not provide a separate mechanism for specifying synchronisation constraints. Some implementations, as for example *Actor-*

Foundry, provide such a mechanism in the form of *local synchronisation constraints* [46] which allows us to defer processing of messages dependent on the message's content and the receiver's internal state. The prototype described in this thesis provides similar capabilities, but differs in some important aspects. In contrast to local synchronisation constraints, the prototype provides a domain specific language, and supports an application to discard messages as early as at the send operation and discards messages instead of disabling it. All these aspects are discussed in more detail in Section 4.5.2.

3.4.2 Limitations

In practice, actor concurrency has some limitations. The following issues are of the most importance [52] [46]:

- Ensuring valid message orders
- Performance
- Integration into existing languages

Ensuring Valid Message Orders Because actors run concurrently and communicate asynchronously with each other, there is no guarantee for the order in which messages arrive at an actor, and multiple actors can have a different perception on the order of messages not related by causality [46]. This becomes a problem when an actor requires synchronisation and wants to support or forbid specific message orders. An actor may for example want to support messages that lead to a change in the actor's internal state only in a well-defined way, because it would otherwise function incorrectly.

What we want is a mechanism that allows for some message orders and forbid other ones. The prototype proposed in this thesis provides a framework for defining and enforcing allowed message orders for actors depending on an actor's history of received messages. As there are various other approaches for synchronisation in actor concurrency, some important approaches are described and compared to the prototype in Section 4.5.

Performance Benchmarks performed on modern actor concurrency implementations show that actors are ready for use in real-world applications [46]. Nonetheless, performance was a problematic aspect of actor concurrency implementations for a long time, and it certainly will be an important future topic for research.

There are many factors influencing the performance of an actor system. If the actor model is used consequently, there may be many actors [52], which can affect the system's overall performance through the overhead required for creation and management of actors. In order to improve performance, many actor concurrency implementations relax one or multiple semantic properties of the actor concurrency model [46]. For example, messages may internally be passed by reference instead of copying them, thus sharing state between actors, or they may be scheduled through an unfair scheduling algorithm.

Integration into Existing Languages Many actor concurrency implementations expand on an already existing programming language, and to provide a consistent and well integrated implementation for actor concurrency can be tricky. When programming with actors in an object-oriented language, we want to be able to use inheritance for factoring out common base functionality and extending on it. Actors in their purest form do not have a direct notion of inheritance or hierarchy [52]. Also, behavior replacement for actors is hard to accomplish in statically typed languages, as they are by nature not well suited for behavior replacement, static analysis is not good at supporting it [52] and behavior replacement also makes optimisations more difficult. In dynamically typed languages, behavior replacement is of course easy to implement.

3.5 Comparison

Actor concurrency, process calculi and petri nets all provide different theoretical approaches for modeling concurrency, differing greatly in their properties. In the following we will compare actor concurrency with the other mentioned theoretical models by examining different properties:

- Concurrency model
- Communication model
- Channels
- Dynamic topologies

Concurrency Model Concurrency models can be distinguished by whether they represent concurrency as *true concurrency* or as *interleaved concurrency* [20]. Actor concurrency and petri nets model concurrency as true concurrency, treating concurrency as a primitive. With true concurrency, a system's behavior is determined by the causal relations between events happening at different places in the system. Process calculus variants model concurrency as interleaved concurrency. Interleaved concurrency reduces concurrency to nondeterminism by modeling parallel actions as choice between their possible sequentializations.

Communication Model Another distinguishing aspect is whether *synchronous communication* or *asynchronous communication* is considered to be the fundamental communication model [3]. While both can be modeled with each other, the one that is considered to be the fundamental one is a distinguishing factor characterizing a concurrency paradigm. Actor concurrency sees asynchronous communication as the fundamental communication primitive. The model requires communication between actors to guarantee eventual delivery of messages [3] [4], but does not make any other assumptions about the order of messages arriving at an actor. Petri nets also consider asynchronous communication as the fundamental communication model. In contrast, process calculus based models regard synchronous communication as the fundamental communication primitive, although there are variants of the pi calculus that provide a direct notion of asynchronous communication as well.

Channels *Channels* explicitly capture and represent the underlying communication medium required for exchanging information between distributed components. Neither actor concurrency nor petri nets have a channel concept. Actors have unique addresses over which they communicate directly with each other and addresses can be communicated between actors, but the communication medium over which this happens is not explicitly represented in the formalism [3]. Process calculus explicitly models the channels over which information can be exchanged and communication is performed exclusively over channels in process calculus [3]. Multiple processes can share and use the same communication channel which can lead to interference in unwanted or unanticipated ways.

Dynamic Topologies In actor concurrency, actors can be dynamically created and an actor system's topology can change dynamically through the exchange of actor addresses between actors [9]. While classical petri nets do not allow dynamically changing topologies, *Reconfigurable petri nets* allow us to model such systems [50]. While in older process calculus variants the topology is static and processes can not acquire knowledge of communication channels over their lifetime, in newer variants, as for example in pi calculus processes can exchange communication channels by sending them as values over other already known communication channels [60] [3].

3.6 AKKA

3.6.1 Capabilities

AKKA [66] is an actor concurrency implementation written in Scala. It provides lightweight actors, location transparency and fault tolerance, and has APIs for Scala and for Java. An actor system in AKKA can consist of millions of actors running concurrently without any problem. AKKA was chosen as basis for the prototype because it can comfortably be used with Java, is open source and has good documentation.

AKKA runs actors on its own lightweight threads for improving performance, under the hood lightweight threads get mapped onto a configurable pool of java threads. Interactions between actors use pure message passing for communication and actors are location transparent and can be distributed over networks. Actors can model behavior changes either through their state variables or by swapping their receive function with become and unbecome operations. Fault tolerance and self-healing are provided in the form of supervisor hierarchies. AKKA also provides different mailbox implementations which can be freely chosen from. The default one is FIFO, where the order of messages processed by an actor is the same as the order in which the messages have arrived. Another commonly used mailbox implementation queues messages by priority.

3.6.2 Java API

In the following, a very short overview of AKKA's Java API, the most important classes in it and their functionality is given. Table 3.1 shows the most important operations for these classes. *ActorSystem* represents a system composed of actors. This class allows us to create actors and also provides a method to shut down the actor system. Every actor in Java AKKA has to extend the class *UntypedActor* and has to provide an implementation of its *onReceive* method which is called whenever the actor receives a message. *ActorPath* represents an endpoint at which one or several instances of an actor listen for incoming messages. As usual for actor concurrency, an actor endpoint can have multiple instances of the same actor running concurrently. *ActorRef* represents the address of an actor and can be used to communicate with it.

Class	Operation	Description
ActorSystem	void actorOf(Props props)	Creates a new actor. At least the actor's class is required in the properties.
ActorSystem	void shutdown()	Shuts down the actor system.
UntypedActor	ActorRef getSelf()	Returns the ActorRef identifying the current actor itself.
UntypedActor	void onReceive(Object msg)	Called whenever the actor receives a message.
ActorRef	void tell(Object msg, ActorRef s)	Sends a message to the actor identified by this ActorRef.

Table 3.1: Important Operations in Java AKKA

CHAPTER 4

Prototype

The prototype developed in the course of this thesis enables developers to specify synchronization protocols and then makes sure actors comply with them. This chapter first gives an overview over the prototype and then describes the its architecture. After that the prototype is analyzed for benefits and limitations and gets compared to similar approaches found in literature. The chapter closes with a description of the development process.

4.1 Concept and Overview

Concept Actor concurrency is very powerful when it comes to divide concurrent applications into manageable pieces, and reasoning about an actor system can be done on a per-actor basis even in the presence of concurrency, so lowering the overall complexity of the resoning process. Because actors can influence each other exclusively through sending messages, the only way for unwanted behavior to be introduced is through the receipt of unanticipated messages or messages in an unexpected order. The prototype described in this thesis allows one to define rules for the message order on a per-actor basis and subsequently makes sure the actor system complies with them. The prototype asserts that an actor only processes specified, wanted messages in a wanted order and that unspecified, unwanted messages and message orders will be detected and that such messages will be filtered out.

Overview The thesis prototype uses *AKKA* [66] as underlying actor implementation and provides a layer for verification of the order of sent and received messages. Whenever an actor sends or receives a message, an *AspectJ* [26] aspect intercepts the call and checks the message against a protocol established by a set of configurable rules. The interception runs in the context of the sender or the receiver, thus making it a local operation of the actor. A verification context object gets built which contains all information necessary for the verification and subsequently the verification is performed. The outcome of the verification decides whether the message is allowed to be sent or received, or gets discarded.

It is important to note, that every actor has its own rule set and verification data and actors extending the class *PrototypeActor* store them locally. As the verification can be performed in the execution context of the actor and all data required for verification is locally available, the verification can be seen as a part of the actor's send or receive operation. In accordance with one of the central statements of actor concurrency, the prototype therefore does not assume a global clock between actors to be available.

4.1.1 Rules

A synchronisation protocol for an actor is described by a number of rules. Every rule has the following form:

$$condition \rightarrow action \tag{4.1}$$

When *condition* is met, the rule fires and *action* has to be considered in the validation process. Conditions are built from *atomic conditions*, complex conditions can be created by connecting atomic conditions with *composite conditions*. Rules always have to be interpreted in the context of the current actor and the performed operation. A rule can be used to enable or forbid both send or receive operations of actors. Table 4.1 gives an overview over currently available conditions and actions and their meaning. To make the syntax and semantics clear, here are three exemplary rules. The rule

when always() **then** allow(a, b)

means, that actor a always is permitted to send a message to actor b. The rule

when messageExists(a, b) **then** forbid(a, c)

states, that actor a is permitted to send a message to actor c if it previously has received a message from actor b. The rule

when messageExists(a, b) **and** messageExists(a, c) **then** allow(a, d)

determines, that actor a is permitted to send actor d a message if it previously has received messages from actors b and c.

Grammar Figure 4.1 shows the grammar of rules in EBNF. A rule, also called a *constraint* can have one or several atomic conditions and exactly one action. If more than one atomic condition is given, the conditions have to be connected by the composite conditions *and* or *or*. A composite condition *not* can only be placed directly before an atomic condition.

4.1.1.1 Atomic Conditions

Atomic conditions can be seen as the atomic pieces of information about the actor's former local message history that can be specified in rules. An atomic condition states that the actor has sent or received a certain message in the past. At the moment, there are two important ones implemented: *messageExists* and *always*

Name	Parameters	Description
	Atomic Conditions	
MessageExists	sender, receiver	a message between sender and receiver exists in the current actor's message history
Always	-	always holds
	Composite Conditions	
Not	condition	negates the atomic condition given as parameter
And	condition1, condition2	holds, if condition1 and condition2 hold
Or	condition1, condition2	holds, if either condition1 or condition2 holds
	Actions	
AllowAction	sender, receiver	rule allows an actor to perform the current action
ForbidAction	sender, receiver	rule forbids an actor to perform the current action

Table 4.1: Overview over Conditions and Actions for Rules

$$\begin{array}{rcl}
\langle\text{CONSTRAINT}\rangle & \models & when\ \langle\text{CONDITIONS}\rangle\ then\ \langle\text{ACTION}\rangle \\
\langle\text{CONDITIONS}\rangle & \models & \langle\text{CONDITION}\rangle\langle\text{COMPOSITE}\rangle \\
\langle\text{COMPOSITE}\rangle & \models & \epsilon\ \mid\ \mathit{and}\ \langle\text{CONDITION}\rangle\langle\text{COMPOSITE}\rangle\ \mid\ \mathit{or}\ \langle\text{CONDITION}\rangle\langle\text{COMPOSITE}\rangle \\
\langle\text{CONDITION}\rangle & \models & always()\ \mid\ messageExists(actor\_id, actor\_id)\ \mid\ \mathit{not}\ \langle\text{CONDITION}\rangle \\
\langle\text{ACTION}\rangle & \models & allow(actor\_id, actor\_id)\ \mid\ forbid(actor\_id, actor\_id)
\end{array}$$

Figure 4.1: EBNF grammar of rules

Message Exists *MessageExists* holds, if and only if the actor has sent or received a message as described by the condition. Note that one of the two parameters of this conditon has to be the current actor, because every actor has its own local understanding of time and message order, and also because of the law of locality [40]. As a consequence, the prototype can only validate communication directly performed with an actor itself. Constraints like "Actor A can send actor B a message only if actor C has sent another message to actor B before" cannot be expressed locally at actor A, but only if one assumes a clock synchronisation between these actors; as an example for such an approach, see *RTsynchronizer* in Chapter 4.5.

Always The *always* condition simply always holds. It was necessary to introduce such a condition because of the whitelist-approach the verification algorithm takes. Always tells the verification system that the rule always fires and the rule's action always has to be considered in the verification.

Not The condition *Not* simply negates its subcondition. It therefore holds, whenever the subcondition does not hold and vice versa.

4.1.1.2 Composite Conditions

Composite conditions help building more complex conditions constructively through combining atomic conditions with logical expressions. The following composite conditions exist: *and* and *or*. They have the usual meaning. *And* holds if both its sub-conditions hold. *Or* holds if one of its sub-conditions holds.

4.1.1.3 Actions

When the conditions of a rule holds, the rule fires and the specified action gets considered as outcome in the validation. There are two possible actions: *allow* and *forbid*. The semantic meaning of *allow* and *forbid* depends on the operation in which the current verification is taking place.

Allow *Allow* states, that the rule allows the actor to perform the operation currently under verification. If the current operation is a send operation, the rule allows the actor to send the message; if the current operation is a receive operation, the rule allows the actor to receive the message. Note that one rule that enables an operation does not mean the operation can be performed, as another rule might also fire and forbid the operation, thus overruling the *allow* action.

Forbid *Forbid* states, that the rule forbids the operation currently under verification to be performed. If the current operation is a send operation, the rule forbids to send the message; if the current operation is a receive operation, the rule forbids to receive the message. In contrast to *allow*, a rule that forbids an action means that the current operation will not be performed, as *forbid* has precedence over *allow*.

4.1.2 Verification Algorithm

The verification algorithm works by running over all rules in the current actor's rule set, checking for each one if the rule's conditions hold and whether the rule enables an actor to perform the current operation. Two verification algorithms were implemented for the prototype. Both use the same basic algorithm for the verification itself (see below), but they differ heavily in how they log interactions between actors and how they check if a rule fires. The *history logging verification implementation* logs interactions for each actor and uses this history to check if a rule fires. Because this proved to be problematic, a second algorithm was implemented, which circumvents the issues of this verification algorithm implementation. The *automaton verification implementation* creates an automaton for each rule and logs interactions with the actor by changing the automaton's active state, which is used to check wether a rule fires.

4.1.2.1 Basic Verification Algorithm

Listing 4.1 shows the verification algorithm. It enables or forbids an actor to perform an operation in the following way:

1. If no rule exists whose conditions hold and which enables or forbids the operation: Forbid the operation
2. If a rule exists, its conditions hold and it forbids the operation: Forbid the operation
3. If a rule exists, its conditions hold, it enables the operation and no other rule forbids it: Enable the operation

Listing 4.1: Verification Algorithm

```
allowed = false;
for(constraint: constraints){
   if(constraint forbids interaction)
      forbid action;
   if(constraint allows interaction)
      allowed = true;
}
if(!allowed)
   forbid action;
```

So for an operation to be enabled, no rule that fires forbids it and at least one rule that fires enables it. If the operation was forbidden by at least one rule, the operation is considered forbidden. If a rule specifically enables an actor to perform an operation, this gets noted, but the operation is considered enabled only if no other rule later on forbids it. Therefore all rules containing at least one condition or action involving the current operation's sender-receiver combination have to be checked before making a call about enabling or forbidding an operation. If the operation is enabled, the verification lets the actor send or receive the message in question, if not, an *InteractionNotAllowedException* is thrown.

When the algorithm has decided on an overall action, it will be performed for the currently verified operation. It has to be noted, that the outcome of the verification does not state anything about the action taken for an equivalent future send or receive operation. So for example a rule can fire and forbid an actor to send a message at first. But later the rule does not fire because its conditions are not met any more or it has been removed from the actor's rule set, so the actor is enabled to send such a message.

Send versus Receive The verification always uses the rule set and message history of the actor that performs the currently examined operation. When verifying a send operation, rule set and message history of the sender are used. When verifying a receive operation, rule set and message history of the receiver are used. Except that, the algorithm works the same for send and receive operations.

Whitelist-Approach The algorithm is written with a whitelist-approach in mind: Operations that are not explicitly enabled are treated as forbidden. This is generally considered the safer approach [65]. However this means, that an actor with no rules can neither send nor receive any messages because the verification algorithm forbids all operations, thus the need of the always condition.

4.1.2.2 History Logging Verification Implementation

The history logging verification algorithm keeps a log of interactions already performed for each actor. When checking rules to determine whether the actor may perform the current operation, this implementation of the verification algorithm does it by checking the rule's conditions against the stored history of interactions; if the message history contains interactions as described by the conditions, the rule fires and the action of firing rules is considered in the overall outcome of the verification, rules that do not fire have no influence.

Memory Overhead Problem This algorithm requires memory to store an actor's message history. The size of an actor's message history grows linearly with the amount of messages sent and received, which can become a serious problem for long-running applications because memory is of course limited, and as memory consumption goes up, the verification's performance degrades considerably. Because of this issues a second algorithm was implemented which works by creating automatons for rules and does not suffer from memory and performance degradation issues.

4.1.2.3 Automaton Verification Implementation

This implementation of the verification algorithm creates an automaton for each rule and logs interactions with the actor by changing the automaton's active state, which is used to check whether a rule fires.

Verification Algorithm The automaton verification algorithm creates an autmaton for each rule that gets added. The autmaton represents the rule and consists of multiple states. Whenever a message is about to be sent or received, the prototype relays it to each automaton for the current actor. The automaton decides based on its current state if an actor may process a specific message or not. If all automatons allow the message to be processed, the actor can process the message.

Automaton Creation Creation of an automaton as representation for a rule is done recursively by composition of state patterns which are available for each *atomic condition* and through a composition pattern wich is used for the *composite conditions*. Creation of automatons is explained in detail in Section 4.2.3.

4.2 Architecture

In this section, the prototype's architecture is described. First a short overview is given and then each component gets examined in detail, including excerpts from the prototype's source code. Used software patterns are described as well.

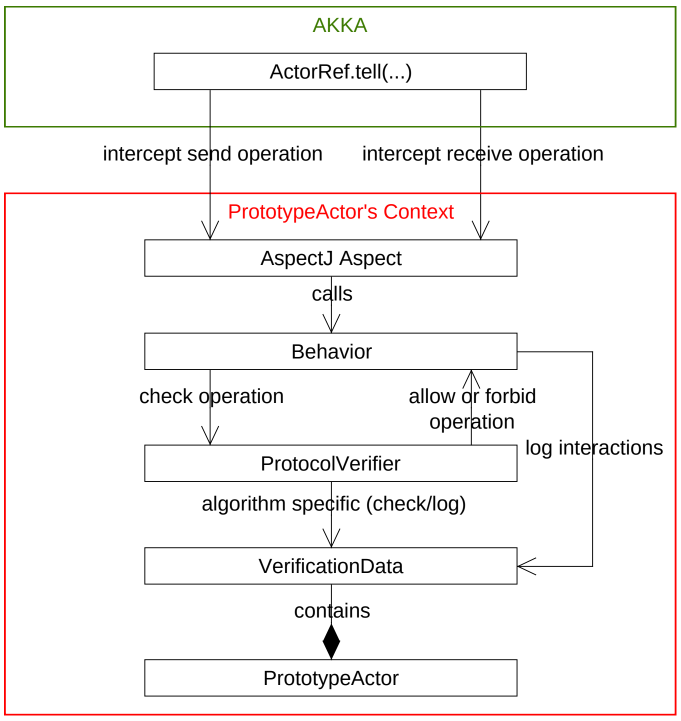

Figure 4.2: Architecture of the prototype.

4.2.1 Overview

Figure 4.2 gives an overview over the prototype's architecture. The prototype provides a framework for specifying a synchronisation protocol to decide if a send or receive operation should take place. AspectJ aspects hook into actors, intercepting send and receive operations. The prototype provides a class *PrototypeActor*, which is a subclass of AKKA's *UntypedActor* and locally holds *VerificationData* required by the specific verification algorithm for the actor. Whenever an actor performs a send or receive operation, the aspect calls a *Behavior* object which subsequently calls a *ProtocolVerifier* object. The *ProtocolVerifier* uses the actor's *VerificationData* object to decide if the current operation can be enabled. If it is, the operation is performed, if not the prototype dismisses the operation and it simply does not happen. As already mentioned, the whole verification takes place in the context of the actor and no external communication is required, so seen from the outside the prototype's verification looks like the actor's local behavior.

4.2.2 Components

The simplified class diagram in Figure 4.3 shows the prototype's components and the relations between them. Table 4.2 gives an overview about the components, including a short description. For better overview and readability, the components are classified in three categories in the table: *Base components*, *automaton verification algorithm* and *history verification algorithm*.

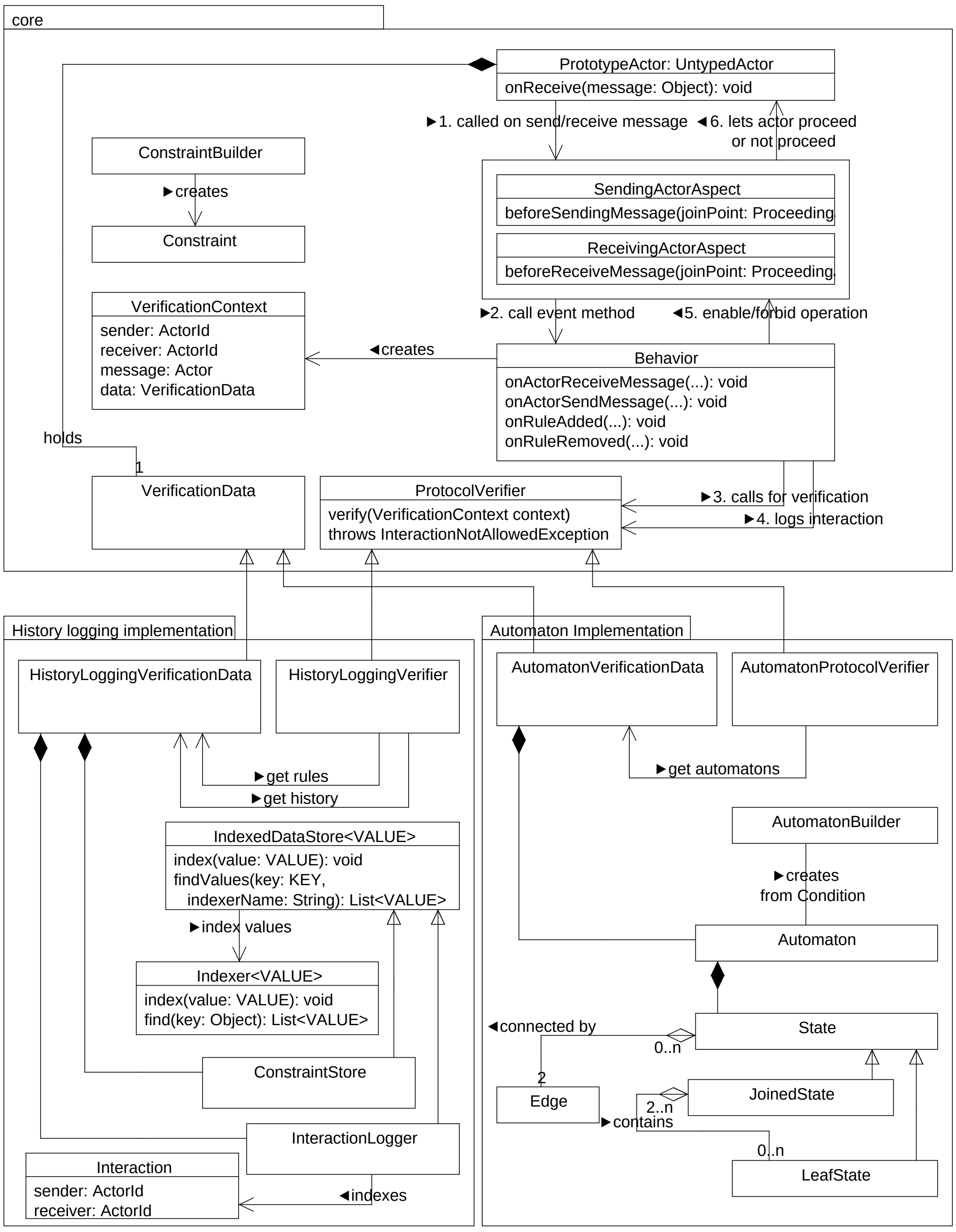
core
PrototypeActor: UntypedActor
onReceive(message: Object): void
►1. called on send/receive message
◄6. lets actor proceed or not proceed
ConstraintBuilder
►creates
Constraint
SendingActorAspect
beforeSendingMessage(joinPoint: Proceeding
ReceivingActorAspect
beforeReceiveMessage(joinPoint: Proceeding
VerificationContext
sender: ActorId
receiver: ActorId
message: Actor
data: VerificationData
►2. call event method
◄5. enable/forbid operation
◄creates
Behavior
onActorReceiveMessage(...): void
onActorSendMessage(...): void
onRuleAdded(...): void
onRuleRemoved(...): void
holds
1
VerificationData
ProtocolVerifier
verify(VerificationContext context) throws InteractionNotAllowedException
►3. calls for verification
►4. logs interaction
History logging implementation
HistoryLoggingVerificationData
HistoryLoggingVerifier
►get rules
►get history
IndexedDataStore<VALUE>
index(value: VALUE): void
findValues(key: KEY, indexerName: String): List<VALUE>
►index values
Indexer<VALUE>
index(value: VALUE): void
find(key: Object): List<VALUE>
ConstraintStore
InteractionLogger
Interaction
sender: ActorId
receiver: ActorId
◄indexes
Automaton Implementation
AutomatonVerificationData
AutomatonProtocolVerifier
►get automatons
AutomatonBuilder
►creates from Condition
Automaton
State
◄connected by
0..n
2
Edge
JoinedState
2..n
►contains
0..n
LeafState

Figure 4.3: Simplified class diagram showing the prototype's components and relations between them.

Component	Short Description
Base Components	
Prototype	Facade as API for easy usage of the prototype
PrototypeActor	Base class for actors using the prototype
SendingActorAspect	Aspect triggered by send operations
ReceivingActorAspect	Aspect triggered by receive operations
Behavior	Prototype's behavior encapsulated
ProtocolVerifier	Verification algorithm
VerificationContext	Holder for passing around verification information
Constraint	A rule with conditions and an action
ConstraintBuilder	Provides a fluent API for rule creation
NoActor and AnyActor	Special Actor Identifiers
Automaton Verification Algorithm	
AutomatonProtocolVerifier	Implementation of the verification algorithm
Automaton	Automaton describing the possible history for a rule
AutomatonBuilder	Creates an automaton of a given rule
ConditionBuilderContext	Holder to pass around information when building an automaton
History Verification Algorithm	
HistoryProtocolVerifier	Implementation of the verification algorithm
IndexedDataStore	Stores and indexes objects
InteractionLogger	Logs an actor's message history
ConstraintStore	Holds an actor's rule set

Table 4.2: Overview over the prototype's components with a short description

Prototype For better usability, a *Prototype* class was introduced as facade, which provides a simple interface for using all of the prototype's capabilities that are divided over various sub-components. The *Prototype* class provides developers with easy access to rule creation, addition and removal of rules to an actor and to an actor's rules set and message history.

PrototypeActor The class *PrototypeActor* (Listing 4.2) extends AKKA's *UntypedActor* class. In order to allow implementations of the verification algorithm to store custom data at an actor, every PrototypeActor can hold a *VerificationData* object. Implementations of the verification algorithm can extend this class and fill it with data during verification and logging to perform the verification.

The *History Verification Algorithm* stores the actor's rule set in the form of a *ConstraintStore* instance and the actor's message history in the form of an *InteractionLogger* instance into the *VerificationData* of an actor. The *Automaton Verification Algorithm* stores a map of automatons keyed by *Constraint* it belongs to into the *VerificationData*.

Rules should only be added and removed locally by the actor if possible, but for convenience reasons the *PrototypeActor* class also provides methods for configuring and reconfiguring the actor with the help of *ConfigMessage* messages from the outside if required.

Listing 4.2: Class PrototypeActor

```
public abstract class PrototypeActor extends UntypedActor {
   public <T extends VerificationData> T getData();
   public <T extends VerificationData> void setData(T data);
   public ActorRef getActorId();
   public boolean isFirstConfigure();
   public void configuredOnce();
   public void configure(ConfigMessage message)
         throws ConfigurationException{};
   public void reconfigure(ConfigMessage message)
          throws ConfigurationException{};
}
```

SendingActorAspect and ReceivingActorAspect The classes *SendingActorAspect* and *ReceivingActorAspect* (Listing 4.3) are AspectJ aspects which are configured to intercept the send or respectively the receive operations of all actors. The aspects interject the actor's operation at the moment it gets started and then call upon the prototype's behavior as defined in the currently used implementation of the *Behavior* class. *Behavior* is then responsible to verify the operation and log it in the actor's local message history. The outcome of the verification specifies if the actor will proceed with the operation or not. Interception of receive operations is necessary for verifying the synchronisation protocol defined by an actor's rule set. While the interception of send operations is technically not required for the verification, it is still convenient to provide it as this allows developers to allow or forbid send operations as well.

SendingActorAspect defines a method with a pointcut intercepting any calls to the method

```
ActorRef.tell(Object message, ActorRef sender)
```

of *ActorRef* objects as well as calls to the same method of objects sub-classing *ActorRef*. So the aspect gets called whenever an *ActorRef* object is used to initiate a send operation. Messages of type *ConfigMessage* are excluded by the pointcut definition, because they will always be accepted for delivery. Note, that the pointcut definition as seen in Listing 4.3 uses the *call* directive which means that the pointcut intercepts the method invocation when the call to *ActorRef.tell(...)* is made and the *this* reference given to the pointcut points to the method's caller. *SendingActorAspect* then finds out if the call was made from inside an actor. If it was, the actor instance is located through the *ProceedingJoinPoint* instance given as join point parameter. Then *SendingActorAspect* calls the method

```
void onActorSendMessage(
         UntypedActor contextActor,
         ActorRef sender,
         ActorRef receiver)
throws InteractionNotAllowedException;
```

on the prototype's current *Behavior* with the actor instance, the sender's ActorRef and the receiver's ActorRef as parameters.

ReceivingActorAspect defines two methods with pointcuts. The first one intercepts all config messages arriving at an actor and then calls the method

```
boolean processConfigurationMessage(
          UntypedActor actor,
          ConfigMessage message)
throws ConfigurationException;
```

on the prototype's current *Behavior* which is responsible to correctly inform actors of configuration messages. The second method intercepts all other messages arriving at an actor and then calls the method

```
void onActorReceiveMessage(
          UntypedActor contextActor,
          ActorRef sender,
          ActorRef receiver)
throws InteractionNotAllowedException;
```

on the prototype's current *Behavior* with the actor instance receiving the message, the sender's *ActorRef* and the receiver's *ActorRef* as parameters. Note, that the pointcut definitions in *ReceivingActorAspect* use the *execution* directive, which means that the method invocation gets intercepted at the callee, which is the actor receiving the message.

ConfigMessage messages can be send to an actor allowing it to react to them and change its rule set, adding or removing rules. The first message of type *ConfigMessage* is directed to the *configure(ConfigMessage message)* method of a *PrototypeActor*, all following ones are directed to the *reconfigure(ConfigMessage message)* method. *ReceivingActorAspect* differentiates between instances of *ConfigMessage* and any other kinds of messages an actor receives because configuration messages are not subject to the verification as they should always reach the actor.

Listing 4.3: Classes SendingActorAspect and ReceivingActorAspect

```
@Aspect
public class SendingActorAspect {
   @Around(
      "call(void ActorRef+.tell(Object, ActorRef+))
       && !args(ConfigMessage, ActorRef+)"
   )
   public void beforeSendingMessage(
                  ProceedingJoinPoint joinPoint)
          throws Throwable { ... }
}

@Aspect
public class ReceivingActorAspect {
   @Around(
      "execution(void UntypedActor+.onReceive(Object))
       && args(configMessage)"
   )
   public void beforeReceiveConfigMessage(
                  ProceedingJoinPoint joinPoint,
                  ConfigMessage configMessage)
          throws ConfigurationException{ ... }

   @Around(
      "execution(void UntypedActor+.onReceive(Object))
      && !args(ConfigMessage)"
   )
   public void beforeReceiveMessage(
                  ProceedingJoinPoint joinPoint)
          throws Throwable { ... }
}
```

Behavior *Behavior* (Listing 4.4) is an interface describing all situations the prototype has to provide behavior for. So far this includes

- sending a message
- receiving a message
- processing *ConfigMessage* messages
- Adding and Removing rules

By encapsulating the prototype's behavior into an interface and implementing classes it is possible to easily exchange the prototype's current behavior if required later. The behavior currently used by the prototype is implemented in the class *DefaultBehavior*. In the following, the behavior as provided by *DefaultBehavior* is described, but any implementation of the *Behavior* interface should provide similar functionality.

In order to monitor sending and receiving messages, *Behavior* gets called by an aspect whenever it intercepts an actor operation. When processing send or receive operations, *DefaultBehavior* creates a *VerificationContext* holder object that contains all information required for further verification so the information can be passed around easily. Information included contains the kind of operation (send or receive), sender's and receiver's addresses and the verification data to use for the verification, which are taken locally from the actor if it is an instance of *PrototypeActor*; as a fallback there exists a global holder which is used for storing and verifying data for actors not extending the prototype's *PrototypeActor* class. Note however, that the use of this global holder requires conventional synchronisation and therefore its use is discouraged and instead all actors used with the prototype should extend the class *PrototypeActor*.

Next, *DefaultBehavior* triggers the validation process by calling *ProtocolVerifier*. *ProtocolVerifier* uses the information passed in the *VerificationContext* parameter to determine if the current operation is allowed or not. The verification algorithm already has been described in detail in Section 4.1.2. If the verification determines that the current operation is forbidden, an *InteractionNotAllowed* exception is thrown by *ProtocolVerifier* which is passed along to *DefaultBehavior* that returns the thread of execution back to the calling aspect which then cancels the actor's send or receive operation. If the current operation is enabled, *DefaultBehavior* calls the actor's *InteractionLogger* to log the operation as interaction between sender and receiver and then returns back to the aspect which at this point lets the actor proceed with the operation. As every actor has its local message history and the moment of sending a message does not provide any information if and when the message arrives at the receiver, sending and receiving of each message is logged separately at the sender and at the receiver.

For processing a *ConfigMessage* message, *DefaultBehavior* simply forwards the message to the *configure()* or the *reconfigure()* method of the actor receiving the message, depending on whether this is the first *ConfigMessage* the actor receives.

When a rule is added or removed the *Behavior* informs the actor's *VerificationData* object, so that it can perform operations if necessary. The *history logger verification* adds or removes the given rule to the rule set of the actor which is saved in the corresponding *VerificationData* object, the *automaton verification* creates an automaton whenever a rule gets added and adds it to the set of automatons wich is also saved in *VerificationData*.

Listing 4.4: Interface Behavior

```
public interface Behavior {
   void onActorReceiveMessage(
               UntypedActor contextActor,
               ActorRef sender,
               ActorRef receiver)
   throws InteractionNotAllowedException;

   void onActorSendMessage(
               UntypedActor contextActor,
               ActorRef sender,
               ActorRef receiver)
   throws InteractionNotAllowedException;

   boolean processConfigurationMessage(
               UntypedActor actor,
               ConfigMessage message)
   throws ConfigurationException;
}
```

ProtocolVerifier The *ProtocolVerifier* (Listing 4.5) interface has only one method

```
void verify(VerificationContext context)
throws InteractionNotAllowedException;
```

The method describes the operation of verifying an actor's send or receive operation to determine if it is enabled or forbidden. The *VerificationContext* instance passed as parameter holds all information required for the verification. By contract, the method is supposed to throw an *InteractionNotAllowed* exception if the verification algorithm detects the current operation to be forbidden to make sure the normal operation is interrupted. The prototype provides two implementations: *HistoryProtocolVerifier* and *AutomatonProtocolVerifier*; future versions of the prototype could provide more verification algorithms by implementing the *ProtocolVerifier* interface. The verification algorithm is described in section 4.1.2.

Listing 4.5: Interface ProtocolVerifier

```
    void verify(VerificationContext context)
    throws InteractionNotAllowedException;
}
```

VerificationContext *Behavior* constructs a *VerificationContext* object which holds all information about the actor operation currently analyzed. The approach of encapsulating this information into a holder object makes it easy to pass it around between the prototype's internal components. A *VerificationContext* object contains the following information:

- Type of verification (send or receive)
- Message sender
- Message receiver
- The local actor's rule set
- The local actor's message history

Constraint *Constraint* objects (Listing 4.6) represent the rules of an actor's synchronisation protocol as described before. A *Constraint* consists of conditions in the form of a *ConditionConstraint* object and an action in the form of an *ActionConstraint* object. *ConditionConstraint* can either be an atomic condition expressing a simple condition or a composite condition expressing a complex condition containing sub-conditions. Table 4.3 lists all *ConditionConstraint* and *ActionConstraint* subclasses implemented for the prototype so far.

Most of the methods provided in *Constraint* are simple getter and setter methods for accessing and changing the rule's content. The method

```
AllowInteraction allowsInteraction(VerificationContext
context)
```

however is different. When given a *VerificationContext* object as parameter, the method checks whether the rule holds for the operation described by the parameter. First the method tests if the *Constraint's* condition holds by using *ConditionConstraint's* method

```
boolean hold(InteractionLogger logger)
```

and subsequently it determines the *Constraint's* action and if the operation between sender and receiver may be performed with the help of *ActionConstraint's* method

```
AllowInteraction allow(ActorRef sender, ActorRef receiver)
```

The value returned describes the outcome and is either *AllowInteraction.ALLOW* or *AllowInteraction.FORBID*, stating whether this rule allows or forbids an actor to perform the current operation.

Constraint Class	Required Informations	Short Description
	ConditionConstraint	
AlwaysCondition	-	Always holds
MessageExistsCondition	sender, receiver	Holds if $\exists$ msg between sender, receiver
ConditionNegation	condition	Holds if negation of condition holds
ConditionAnd	condition1, condition2	Holds if condition1 and condition2 hold
ConditionOr	condition1, condition2	Holds if condition1 or condition2 holds
	ActionConstraint	
AllowAction	sender, receiver	Allows current action
ForbidAction	sender, receiver	Forbids current action

Table 4.3: ConditionConstraint and ActionConstraint subclasses implemented for the prototype

Listing 4.6: Class Constraint and interfaces ConditionConstraint and ActionConstraint

```
public class Constraint {
   private ConditionConstraint conditions;
   private ActionConstraint actions;

   public Set<ActorRef> getParticipatingSenders();
   public Set<ActorRef> getParticipatingReceivers();
   public ConditionConstraint getConditions();
   public void setConditions(
                    ConditionConstraint conditions);
   public ActionConstraint getActions();
   public void setActions(ActionConstraint actions);
   public AllowInteraction allowsInteraction(
                         VerificationContext context)
         throws InteractionNotAllowedException;

   @Override
   public String toString();
}

public interface ConditionConstraint extends ConstraintItem
{
   boolean hold(InteractionLogger logger);
}

public interface ActionConstraint extends ConstraintItem {
   AllowInteraction allow(
                    ActorRef sender,
                    ActorRef receiver);
}
```

ConstraintBuilder In order to make the creation of synchronisation rules in the form of *Constraint* objects more straightforward and comfortable, a *ConstraintBuilder* component was introduced. With the *ConstraintBuilder* only semantically correct rules can be created. This is accomplished by utilizing the *Builder* pattern [33] in conjunction with the *Fluent interface* pattern [31]. The *ConstraintBuilder* component consists of multiple interfaces and classes implementing one or multiple of these interfaces which together form a *Domain-Specific Language* (DSL) for creating *Constraint* objects.

Figure 4.4 shows the allowed workflows which lead to semantically correct rules. The interface *ConstraintBuilder* and its implementation *ConstraintBuilderImpl* are the entry point at which creation of a new *Constraint* begins. The *ConstraintBuilder* interface provides the method

```
When when() throws ConstraintBuilderException;
```

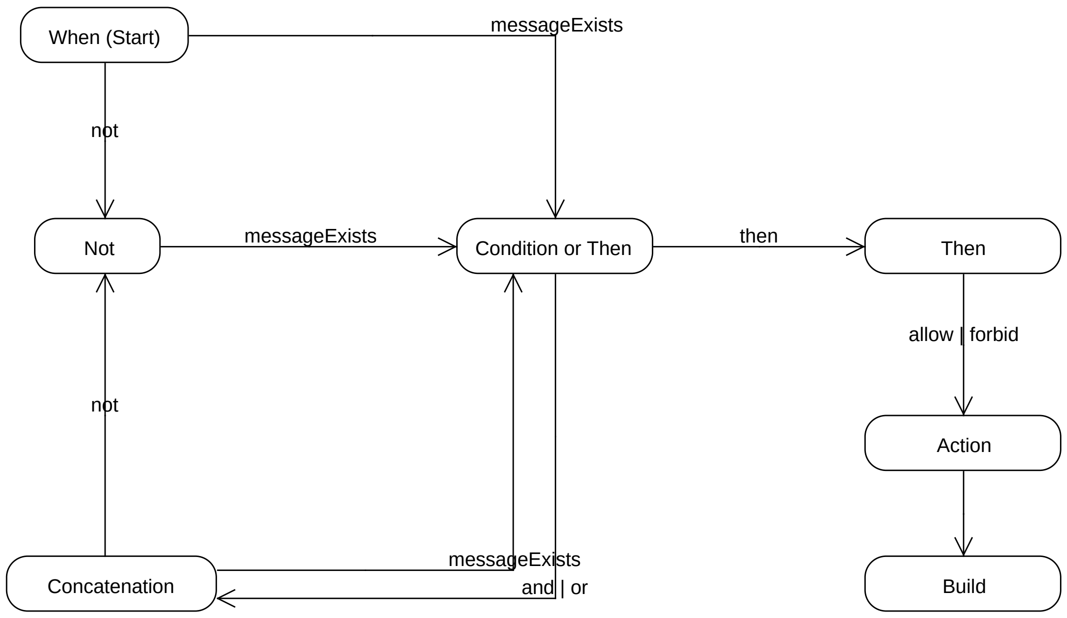

Figure 4.4: Workflow allowed by the *ConstraintBuilder* for creating semantically valid rules.

which starts the building process by returning an object implementing the builder interfaces that describe valid operations at this point, which are the creation of atomic conditions or the composite condition *not*. By calling one of the provided methods with the required parameters, the condition is added to the new rule and another builder object is returned. This new object again only allows one to choose from semantically valid options. This process continues until a state is reached in which the *build()* method can be called, which creates and returns the fully configured *Constraint* object. In the background, the classes forming *ConstraintBuilder* pass an instance of *BuilderContext* around, which holds the *Condition's* state so far. Listing 4.7 shows an example of how the *ConstraintBuilder* is used.

Listing 4.7: Example of how the ConstraintBuilder can be used to create Constraints

```
Constraint actorNoneToA
            = builder.when().always()
                    .then().allow(
                            Actors.noActor(),
                            actorA)
                    .build();
Constraint actorAtoB
            = builder.when().always()
                    .then().allow(
                            actorA,
                            actorB)
                    .build();
Constraint sendBtoC
            = builder.when().messageExists(
                            actorA,
                            actorB)
                    .then().allow(
                            actorB,
                            actorC)
                    .build();
```

NoActor and AnyActor An *ActorRef* can be used to uniquely identify an actor and interact with it. An *ActorRef* contains a path representing the actor's address. Unfortunately the use of *AKKA's* implementation of *ActorRef* alone proofed to be insufficient for the prototype's requirements, as it could not cope with two special situations. First, the *DeadLetterbox* actor is represented by an *ActorRef* with the path attribute set to null which is problematic when trying to compare the *DeadLetterbox* actor's *ActorRef* to other *ActorRef* instances, because *ActorRef's equals(...)* method tries to call the method *compareTo(...)* of the *ActorRef's* path attribute which fails with a *NullPointerException*. Second, in order to express rules that enable send or receive operations to or from any actor, an *ActorRef* instance that represents any possible actor is required.

For these use cases a *NoActor* class containing a path attribute of type *NoActorPath* and an *AnyActor* class containing a path attribute of type *AnyActorPath* were created for internal use in the prototype. *NoActor* represents the *DeadLetterbox* actor and *AnyActor* represents any actor possible. Instances of both classes can be compared to other *ActorRef* objects and also can be identified by the prototype through an instanceof check if required. Both *NoActor* and *AnyActor* are used in *Constraint* objects as sender or receiver in conditions and actions. *NoActor* objects are also used in *Interaction* objects when logging messages sent or received to or from the *DeadLetterbox* actor.

4.2.3 Automaton Verification Algorithm

AutomatonProtocolVerifier This implementation of the verification algorithm creates an automaton for each rule that is added to an actor and uses this automatons to verify interactions and to log sent and received messages. The automaton verification implementation of the algorithm circumvents the problems of the *History Logging Verification Algorithm* as each automaton requires finite space and performance does not degrade. Checking a rule works in O(1) which is also a considerable improvement. However, for each rule added to an actor, an automaton has to be created, which is a more expensive operation. The implementation uses a component *AutomatonBuilder* to create *Automaton* (Listing 4.8) objects which contain *States*.

Automaton and State An automaton represents a rule. It holds

- sender/receiver pair of the corresponding rules action
- the current state the automaton is in.

Each *State* contains an action (*ALLOW* or *FORBID*), and edges leading to other states. *LeafStates* express the occurrence or non-occurrence of a specific message. *JoinedStates* express a complex state in which some messages have occurred while others have not. Besides the aforementioned attributes all states have, *JoinedStates* contain *LeafStates* as substates. *Edges* represent a message from sender to receiver. They lead from one state to another and are annotated with a sender/receiver pair.

The automaton provides methods to check if a message is enabled in the current state and to log the occurrence of messages. The states of an automaton are connected by edges which represent possible state changes.

Listing 4.8: class Automaton

```
public class Automaton {
   public void logMessage(ActorRef sender, ActorRef receiver
);
   public AllowInteraction allows(ActorRef sender, ActorRef
receiver);
}
```

AutomatonBuilder *AutomatonBuilder* creates automatons from rules. As previously mentioned, the creation of an automaton is done recursively by composition. Each *condition* is represented by a distinct graph of states. State graphs for *atomic conditions* get connected by the state graphs for *composite conditions*. In the following, first the state graphs for atomic conditions and then the ones for compositions are described. For the latter the creation algorithm is also presented in detail. In the figures (4.5 and 4.6) + means the action is allowed and - means the action is forbidden.

Figure 4.5 depicts the state patterns used for the *atomic conditions*. For *Always* (1a.), the automaton simply consists of one state with the action of the rule. For *MessageExists* (1b.), two

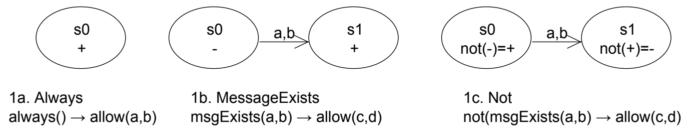

Figure 4.5: State patterns for *MessageExists*, *Always* and *Not*

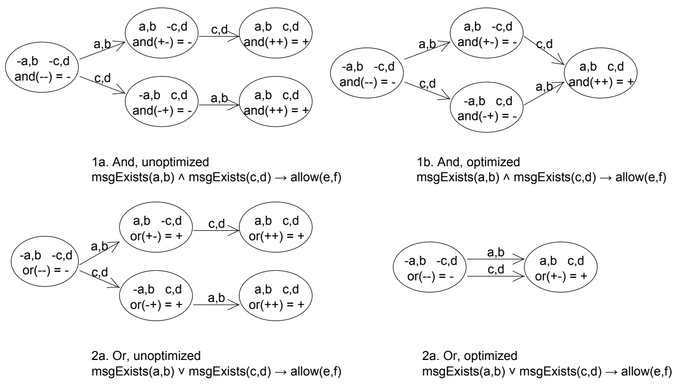
1a. And, unoptimized
msgExists(a,b) ∧ msgExists(c,d) → allow(e,f)
1b. And, optimized
msgExists(a,b) ∧ msgExists(c,d) → allow(e,f)
2a. Or, unoptimized
msgExists(a,b) ∨ msgExists(c,d) → allow(e,f)
2a. Or, optimized
msgExists(a,b) ∨ msgExists(c,d) → allow(e,f)

Figure 4.6: State patterns for *And* and *Or*

states are created. The first one contains the negated action of the rule. The second one contains the action of the rule. Both states are connected by an edge annotated with sender and receiver from the atomic condition. A *Not* (1c.) simply negates the actions of all given states.

Figure 4.6 depicts the state patterns for *And* (1a., 1b.) and *Or* (1a., 1b.) first in unoptimized and then in optimized form. Both, *And* and *Or* can be handled with the same algorithm. Listing 4.9 shows a simplified version (pseudo-code, for shortness and readability) of the algorithm.

Listing 4.9: Creation of state graph for *composite conditions*

```
public State build(condition, action) {
   if(isAtomicCondition(condition))
      return statesForConditionType(condition);

   leftState = build(condition.left(), action);
   rightState = build(condition.right(), action);
   startState = leftState.join(rightState);
   unprocessed_states = { startState };
   graph = {};
   //follow edges, create state graph
   while (unprocessed_states.hasMoreStates()) {
      state = unprocessed_states.next();
      for(edge in state) {
         if (edge is outgoing) {
                newState = state.followEdge(edge);
                graph.updateState(newState);
                unprocessed_states.add(newState);
            }
        }
        unprocessed_states.remove(state);
    }
    //simplify graph, join states where possible
    while(simplifieableStatesExist(graph.nodes)) {
        newState = join(simplifieableStates);
        graph.update(newState);
    }
}
```

As first step, the state graphs for the left and the right subconditions are recursively created. After that, a new start state for the *composite condition* is created by joining the start states of both subcondition graphs into a new state, which contains all the substates and edges of both states. The action of the new start state is determined by a boolean combination of the actions of the two joined substates. For an *And* they are joined with a boolean and, for an *Or* with a boolean or.

As the next step, as long as there exist states with unprocessed outgoing edges, a new state is created by following one of these edges, starting with the new start state. Following an edge means copying the current state and then changing the substate that represents the state change of following the edge. After this step, the state graph for the *composite condition* is complete.

In order to minimize the state graph, two optimizations are performed. The first one happens during state creation; as states with the same substates are equal, only one state gets created. The second time a state is referenced, the new edges are added to the already existing state. This is implemented by defining *equals()* and *hashCode()* accordingly for states and putting them into a map which gets checked whenever a state should be created.

The second optimization joins groups of states with the same action if they have no outgoing edges to states with a different action. To do this, the graph is taken and searched for such state groups. When one is found, the states get joined into one new state.

ConditionBuilderContext *AutomatonBuilder* constructs a *ConditionBuilderContext* object which holds information during automaton creation. Again, this is done to make passing around of information easy. The context holds the following information:

- Type of composite condition (*And* or *Or*)
- Graph
- Unprocessed States

4.2.4 History Verification Algorithm

HistoryProtocolVerifier This implementation of the verification algorithm verifies interactions by analyzing the previously stored message history of an actor, thus determining wether an interaction should be enabled or not. This implementation of the algorithm was the first one implemented. As described earlier, its performance degrades considerably with time and long-running applications can also run out of memory. The implementation uses an *InteractionLogger* component to record interactions with an actor and a *ConstraintStore* component to store rules at the actor, both of which extend a basic *IndexedDataStore* component.

IndexedDataStore Both *InteractionLogger* and *ConstraintStore* require functionality for indexing and efficiently querying objects by different attributes. The interface *IndexedDataStore* (Listing 4.10) and the implementation *IndexedDataStoreImpl* were designed as a common base providing these capabilities. *IndexedDataStore* uses generics to support type-safe usage. The component provides the capability to store objects of a certain type and to retrieve a subset of them by filtering out the ones that have certain attributes. Listings 4.10, 4.11 and 4.12 show the interfaces describing *IndexedDataStore* and the associated interfaces *Filter* and *Indexer*.

The component supports custom indexers and filters. An *Indexer* decides on which attribute to index objects passed to the *IndexedDataStore* and later can be used to query for objects containing a certain value of that attribute. A *Filter* decides which objects passed to an *IndexedDataStore* it will add to the collection of filtered objects based on each object's attributes. Filters were later removed because indexers proofed to be sufficient for implementation of the currently existing conditions and removal of them improved performance. *Indexer* and *Filter* instances get registered as listeners at an *IndexedDataStore* instance. Registered ones are notified whenever an object is put into the store and later can be used to query for objects.

Listing 4.10: Interface IndexedDataStore

```
public interface IndexedDataStore<VALUE> {
   void registerIndexer(Indexer<?, VALUE> indexer);
   void registerFilter(AbstractFilter storage);
   void index(VALUE value);
   <KEY> List<VALUE> findValues(
                        KEY key,
                        String indexerName);
   List<VALUE> findFilteredValues(String filterName);
   <KEY> List<VALUE> findValues(KEY key);
}
```

Listing 4.11: Interface Filter

```
public interface Filter<VALUE> {
   String name();
   void store(VALUE value);
   List<VALUE> getValues();
}
```

Listing 4.12: Interface Indexer

```
public interface Indexer<KEY, VALUE> {
   String name();
   Class<KEY> type();
   void index(VALUE value);
   boolean contains(Object key);
   List<VALUE> find(KEY key);
   List<VALUE> findByKey(Object key);
}
```

InteractionLogger The *InteractionLogger* interface (Listing 4.13) describes a component capable of recording sent and received messages for an actor. Interactions can then be looked up by sender, receiver or a combination of both. Information about an actor's interactions with other actors is required by the *HistoryProtocolVerifier* component when checking which rules hold for the actor's current operation. Each send or receive operation is represented with an *Interaction* containing

- a unique UUID (Universally Unique Identifier)
- the sender's address
- the receiver's address

The *InteractionLogger* (Listing 4.13) instance associated with an actor gets informed whenever the actor sends or receives a message and then is responsible to log the interaction. All *Interaction* objects stored in the *InteractionLogger* instance of an actor listed in the arrival order and together accurately depict the actor's message history.

The default implementation of *InteractionLogger* is *InteractionLoggerImpl*. It implements the *InteractionLogger* interface and extends *IndexedDataStoreImpl*, using its capabilities. Three indexers index *Interaction* objects for sender, receiver and the combination of both.

Listing 4.13: Interface InteractionLogger

```
   void logMessage(ActorRef sender, ActorRef receiver);
   List<Interaction> findInteractions(
                        ActorRef sender,
                        ActorRef receiver);
   List<Interaction> findAllInteractions(
                        ActorRef senderOrReceiver);
   List<Interaction> findInteractionsForSender(
                        ActorRef sender);
   List<Interaction> findInteractionsForReceiver(
                        ActorRef receiver);
}
```

ConstraintStore The *ConstraintStore* interface (Listing 4.14) describes a component for storing synchronisation rules for an actor. The component is used to manage an actor's synchronisation protocol. Synchronisation rules can be added, removed and queried. The actor itself can add and remove rules at will to the *ConstraintStore* instance it owns to change its synchronisation protocol. From the outside, *ConfigMessage* messages can be used to influence an actor's synchronisation protocol, triggering the actor's *configure* or *reconfigure* method where the actor can handle the message and may decide to adapt its synchronisation protocol. Each synchronisation rule is represented with a *Constraint* object, which contains

- the rule's conditions
- the rule's action

Together the rules stored in an actor's *ConstraintStore* instance build the synchronisation protocol the actor adheres to when performing send and receive operations.

The default implementation of *ConstraintStore* is provided in the class *ConstraintStoreImpl*. *ConstraintStoreImpl* implements the *ConstraintStore* interface and extends *IndexedDataStoreImpl*, using its capabilities. Three indexers index *Constraint* objects for sender, receiver and the combination of both.

Listing 4.14: Interface ConstraintStore

```
    void add(Constraint constraint);
    void remove(Constraint constraint);
    Collection<Constraint> getForSender(ActorRef sender);
    Collection<Constraint> getForReceiver(ActorRef receiver);
    Collection<Constraint> getForSenderReceiver(
                                ActorRef sender,
                                ActorRef receiver);
}
```

4.2.5 Software Design Patterns

The following software design patterns were used for design and implementation of the prototype. For each pattern a short description is given and its use in the prototype is explained.

Observer Pattern The *IndexedDataStore* component provides the capability to register *Indexer* and *Filter* objects which get notified whenever a new value gets added to the data store, as described by the *Observer Pattern* [33]. The pattern allows one to define a many-to-one dependency between objects, so that the many objects get notified whenever the one object changes its state. The notified objects, or listeners, can use the information about the object's new state to react to the state change.

Builder and Fluent Interface To make creation of rules safe and comfortable, the *Constraint-Builder* component uses the *Builder Pattern* [33] in combination with the *Fluent Interface Pattern* [31]. The *Builder Pattern* separates construction of a complex object from its representation, so that both can be adapted independently. This considerably eased adaptation of the building process during development and also supported changes of the internal representation of *Constraint* objects as necessary. The *Fluent Interface Pattern* introduced by *Martin Fowler* makes sure, that only semantically correct *Constraint* objects can be built with the *ConstraintBuilder*. *ConstraintBuilder* returns an object with methods that represent all semantically valid next operations for creating a *Constraint* object. When called, these methods return a new object with methods that represent the semantically valid operations after the last operation. This goes on, until a call to the method *build()* constructs the *Constraint* object.

Domain-Specific Language The *ConstraintBuilder* in combination with *Constraint-*, *ConditionConstraint-* and *ActionConstraint* objects form a *Domain-Specific Language* [32]. While maybe not a software pattern in the strictest form, the approach of defining a *Domain-Specific Language* can be seen as a method of designing systems and architecting software. The implemented *Domain-Specific Language* allows developers to express an actor's synchronisation protocol as rules in a natural and concise way.

Interpreter *ConditionConstraint* uses the *Interpreter Pattern* [33] to evaluate whether a constraint holds. The pattern is used to define a grammar for a given language and an interpreter that can interpret sentences in that language. The interpreter was implemented with *recursive descent parsing* [10], a top-down approach for syntax analysis, and consists of recursive procedures which are used to process the incoming input. The sequence of procedure calls defines a parse tree that evaluates whether the condition holds. Recursive descent parsing was chosen, because it is easy to implement and is efficient enough, as *ConditionConstraints* are usually not nested very deeply.

Facade In order to provide a simple, central API for accessing the prototype's capabilities, the *Facade pattern* [33] was used and the *Prototype* class was implemented. A *Facade* provides a consistent interface to functionality divided over a number of sub-components of an object-oriented system. The *Prototype* facade class simplifies the use of the prototype's functionality and provides central access to it.

Strategy In order to encapsulate the verification algorithm so it can be easily adapted or exchanged in the future, the *Strategy pattern* [33] was used. The interface *ProtocolVerifier* describes the verification operation and the class *HistoryProtocolVerifier* provides the default implementation. Whenever an operation has to be verified, the *ProtocolVerifier* instance registered in the system gets called, receives contextual information and verifies the operation.

4.3 Optimizations

This section summarizes optimizations performed in the course of development. Optimization targets were identified by visual code inspection and by profiling the application's code. As profiler *VisualVM* was used, which comes with the *Java SDK*.

4.3.1 General Prototype Optimizations

Replacing joinPoint.getArgs() with Pointcut *SendingActorAspect* as well as *ReceivingActorAspect* used *joinPoint.getArgs()* for finding out if a given message has the type *ConfigMessage*. Those messages are for configuring actors, have to pass verification in any case and are routed to the method *configure()* or *reconfigure()* of the *PrototypeActor* instance. The call to *joinPoint.getArgs()* proved to be very expensive, but fortunately it could be avoided by dividing the one pointcut defined for *SendingActorAspect* and respective *ReceivingActorAspect* into two more specific pointcuts. The first one only intercepts *ConfigMessages* and pipes them through to the *PrototypeActor's configure()* or *reconfigure()* method. The second pointcut intercepts any other messages, calling verifier and interaction logging.

Optimizing SendingActorAspect.findActor() The method *SendingActorAspect.findActor()* could also be improved. For once, reflection was used for finding the correct actor through *getClass().isAssignableFrom()* which could be avoided. Also finding the actor calling a send operation for actors defined as inner classes was removed. The costs for checking this are high and the restriction not to declare *PrototypeActors* as inner classes is a relatively moderate one. When no actor could be found around a send operation, the method now simply returns. Moreover, calls to *joinPoint.getThis()* were minimized to a minimum, as they are expensive.

Removing ActorId In the course of development, the wrapper *ActorId* introduced for *ActorRef* addresses could be removed from the prototype in favor of the classes *NoActor* and *NoActorPath*. The wrapper class was developed because *ActorRef* objects representing the *DeadLetterbox* actor contain a *Path* property defined as null which prohibits the use of the *equals()* method to compare *ActorRef* objects identifying the *DeadLetterbox* actor with other *ActorRef* objects. Now messages coming from the *DeadLetterBox* actor are recognized by the prototype and get stored with an instance of *NoActor* in the *InteractionLogger* which can be compared with *equals()*. The *ActorId* could be removed completely from the prototype which avoids the overhead object creation and indirect access.

4.3.2 Automaton Verification

AutomatonBuilder: Join Duplicate States Whenever following an *Edge* from one *State* to another during automaton creation, a new *State* object representing this new automaton state is created. Two automaton states which contain the same substates are equal to each other: in both states the same messages have occurred/not occurred. Such states can be joined to minimize the automaton's state count. To implement this optimization it is checked if an equal *State* object already exists in the automaton's state graph whenever a new *State* object should be created.

If this is the case, the *Edges* – which represent the change to the new state – are added to the already existing *State* object and *Edge.to* is updated on each *Edge* to point to the *State* object.

AutomatonBuilder: Join States with Same Action After the automaton has been created, the state graph of the autmaton can be optimized further without changing its semantics; Groups of *State* objects which only lead to the same action can be joined. Such *State* objects contain only edges that lead to *State* objects with the same *AllowInteraction*. As the outcome of all future checks clearly always has to be the same *AllowInteraction* even after following an *Edge*, the group of *State* objects can be joined into a single *State*. To find all such state groups, the graph is traversed and whenever a group is found which can be joined it immediately gets joined.

4.3.3 History Logging Verification

Optimizing IndexedDataStore's findValues() IndexedDataStore's *findValues(. . .)* method gets called very often. This method could be improved so that indexers do not have to be iterated if the *Indexer's* name is given and the key only gets looked up in this one. If no *Indexer* name is given, then we of course do have to iterate over all *Indexers*, aggregating the values found by each *Indexer* for the given key. Because most of the time we're only interested in values from one *Indexer*, this optimization will cut the costs of the operation most of the time it is called.

Removing Filters *Filters* for *IndexedDataStore* were removed. The feature was not used in production code besides for debugging output. Because registered *Filters* had to be called every time a value is to be indexed, this feature was deemed too expensive and was removed.

4.4 Benefits and Limitations

Most of the prototype's characteristics were already discussed in one or the other form while describing its design and implementation. This section summarizes these characteristics and describes them as benefits and limitations.

4.4.1 Benefits

Ensuring Message Order The order in which messages arrive at participants of an actor system is generally arbitrary and unknown [4] and has to be limited through synchronisation to prevent unwanted behavior. The prototype described in this thesis provides a rule-based approach to actor synchronisation. For each actor, developers can state rules which limit the arrival order of messages allowed in a certain situation. The prototype ensures, that the protocol built by these rules is maintained and that any other message order gets rejected, preventing the actor to enter an invalid or undefined state.

Local Verification Verification of the synchronisation protocol is performed locally at the actor. Rule set and message history are stored at the actor and the verification runs in the actor's context, as send and receive operations are intercepted with AspectJ. The actor's behavior effectively gets interrupted until the outcome of the verification is determined and the current operation is allowed to continue or gets canceled. From the outside, verification seems to be behavior of the actor itself and it is performed with the same message order and message history that the actor sees when it performs the verified operation.

No Assumption of Global Time The presented prototype does not assume the existence of a global order of messages for actors and so also does not assume the existence of a global time between them. This complies with the actor concurrency model's insight, that for a system with real concurrency neither a global message order nor a global time could be defined in a way that all actors could agree upon them [4] [40] [21]. The prototype's approach stays in contrast to for example the approach taken with RTSynchronizers [61], which assumes that at least for nearby actors times are sufficiently synchronized to support meaningful interpretation of timing constraints.

Support for Behavior Change An actor can replace its current behavior dynamically, changing how it reacts to arriving messages. The prototype's rule mechanism is flexible enough to support the synchronisation protocol to be adapted when the actor performs a behavior change operation. Rules can be added and removed at runtime to change the actor's synchronisation protocol in accordance with the new behavior.

4.4.2 Limitations

Only Conditions over Direct Interactions Rules for the synchronisation protocol of an actor can only be formulated over actors and messages directly affecting the actor in question. It is for example impossible to state the following rule for actor C:

$$when\ messageExists(A, B)\ then\ allow(B, C) \tag{4.2}$$

The condition part of this rule cannot be expressed, because the information required to verify the statement is not available at actor C and never can be, as the prototype does not assume existence of a unique global message order. To eliminate this limitation, (part of) the message history of actor A could be sent to actor C, however this operation would also be term to the arrival order and it would not be clear when the information arrives and is processed, which may lead to synchronisation issues.

Limited Expressiveness So far only basic conditions were defined and implemented for the prototype. In order to be able to use more details about events and properties of the message history of an actor to define the actor's synchronisation protocol, new conditions will have to be implemented. More advanced conditions could for example limit the firing of a rule depending on whether a message was sent to one actor of a group of actors, whether another rule has fired in the past, or whether a certain number of messages was sent or received to or from an actor.

Discarding of Messages The prototype discards messages violating an actor's synchronisation protocol. This behavior was chosen on purpose, because if an actor would process messages not allowed in the current state, it might enter an invalid or undefined state. Discarding of messages however can lead to situations where a message gets sent to an actor, but is discarded and thus is never processed, if the sender does not retransmit the message. The behavior of the prototype stands in contrast to the one of other approaches. ActorFoundry for example disables messages which an actor is not allowed to process in the current state, storing them in a queue with the possibility to be re-enabled later if the actor's state changes.

No Selective Verification It is currently not possible to selectively use the prototype for a subset of actors in an actor system, as the prototype injects itself into the system and gets called for every actor. Some applications might require the prototype's capabilities only for a small set of critical actors that require consistent state, while the overhead introduced by the prototype could be spared out for all other actors in the system. Selective verification could be implemented by providing an annotation that can be used to inform the prototype of actors requiring verification.

4.5 Comparison

This section describes approaches for actor synchronisation found while researching the topic. Each approach is first described and then compared to the prototype.

4.5.1 Set-Constraint-based Analysis

4.5.1.1 Description

Colaço et al propose in [23] and [22] a way to detect orphan messages in actor-based languages, which is compareable to the approach taken in this thesis, especially for the *automaton verification algorithm*; the paper however was unknown during design and implementation phases of the prototype and was found much later when writing this thesis.

In the paper, an orphan message is defined as a message which in an execution path cannot be handled by the receiving actor. There are safety orphans, which occur when no behavior of the receiver does know how to handle a message, and there are liveness orphans which occur if a behavior in the execution path knows how to handle a message but the receiver is deadlocked and never changes its behavior to the aforementioned one. The proposed type system associates each of an actor's behaviors with an interface; for a specific program execution the actor then can be described by a sequence of said interfaces. All possible executions of an actor can be described with a regular tree, where the nodes represent the behavior interfaces and the edges represent behavior changes.

The first approach taken by the authors was to calculate the intersection of all possible behavior interfaces [23]. As it turns out, this is too restrictive and forbids use of messages which could be enabled in some behaviors. The improved approach proposes to use multi-union of all behavior interfaces along a given tree branch and multi-intersection of these [22]. A message can be sent to an actor if in each future execution path from this moment on there exists a behavior which accepts the message, otherwise the message should not be allowed to be sent to the actor.

4.5.1.2 Comparison

Rule Creation While the basic approach is very similar to the one proposed for this thesis, nontheless there exist some differences. For the approach taken in [23] and [22] rules are inferred by a type inference system. In contrast to that, the prototype proposed in this thesis lets the programmer manually define and change the rules at runtime. Each rule is also treated as separate part of the verification protocol of the current actor's behavior, which makes it possible to dynamically add and remove rules at runtime in order to adapt the verification.

Static versus Runtime Checking The approach proposed in [22] tries to detect orphan messages statically if possible. Although another goal of the proposed inference system is, according to the authors, to produce information that helps to dynamically detect orphan messages as well, the main focus is on static checking. The prototype of this thesis on the other hand focuses on runtime verification, as it checks at runtime if an actor is enabled or forbidden to receive a message.

4.5.2 ActorFoundry: Local Synchronisation Constraints

4.5.2.1 Description

Messages between actors are sent and received asynchronously in *ActorFoundry* [46], an actor concurrency implementation. In order to deal with the nondeterminism of the order of messages arriving at actors, ActorFoundry provides the concept of *local synchronisation constraints* [46]. Local synchronisation constraints are special methods defined at actors. They allow the developer to restrict the order in which messages get processed by disabling them based on the actor's state and the messages previously received. A disabled message will not be processed, but instead gets stored in a queue. Local synchronisation constraints are reevaluated continuously and when the actor's state and message history change and allow to re-enable a message, the message gets removed from the queue and is delivered to the actor. This approach is similar to the one taken with the prototype proposed in this thesis, but differs in some areas, as explained below.

4.5.2.2 Comparison

Domain Specific Language To define a local synchronisation constraint in ActorFoundry, the programmer implements a method and manually has to specify the conditions which lead to disabling of messages. In contrast, our prototype provides a domain specific language that allows developers to construct rules which relate to the message history of an actor and against which arriving messages can automatically be validated. This approach provides developers with a structured approach for defining acceptance and denial criteria for messages in a well-defined way. While the implemented domain specific language lacks the possibility to include the actor's current state so far, future iterations of the prototype could extend it to provide this functionality.

Discard at Sending In contrast to local synchronisation constraints, our prototype also can handle rules for sending of messages. While this is not required for correctness, as actors only can be influenced by reception of messages, it allows the prototype to discard incorrect messages as early as possible.

Disabling versus Discarding With synchronisation constraints, disabled messages get buffered until the actor's state and message history allow it to process the message. The synchronisation constraint methods have to be run continuously in order to find out if a previously disabled message has to be reenabled because the actor's status has changed. The prototype described in this thesis takes a different approach here and discards messages which fail the validation. As explained before, this can lead to situations, where messages get discarded and never are retransmitted.

Dynamic Rule Reconfiguration The prototype provides the possibility to reconfigure and change the set of synchronisation rules for an actor at run-time. This capability makes it possible to dynamically add and remove rules as required when an actor should change its behavior or the state it is in.

4.5.3 RTSynchronizer and QoS Synchronizers

4.5.3.1 Description

RTSynchronizer and QoS Synchronizers extend the basic actor concurrency concept by supporting the definition of certain aspects of synchronization between a group of actors. Because both approaches have great similarities and QoS Synchronizers also rely on RTSynchronizers for time related constraints, the models are evaluated together.

RTsynchronizer *RTSynchronizer* [61] is an actor concurrency implementation, that allows developers to define timing constraints on groups of actors. Actors are provided with a local clock, and it is further assumed, that local clocks of nearby actors are sufficiently synchronized to support meaningful interpretation of timing constraints between them. In order to separate functional concerns from synchronisation, such timing constraints are defined in special actors called RTSynchronizers. Instead of sending and receiving messages, RTSynchronizers provide a specification of the temporal order of messages over a given group of (nearby) actors.

QoS Synchronizers *QoS Synchronizers* [62] are a proposal on how to specify quality of service aspects between actors for multimedia applications. A QoS Synchronizer is a special actor, that allows developers to define quality of service constraints between a group of actors. For time related quality of service constraints, QoS Synchronizers use RTSynchronizers.

4.5.3.2 Comparison

Definable Constraints As already explained, constraints in the presented prototype can only be formulated over actors that directly interact with each other. As RTSynchronizers assume equality of local clocks of nearby actors, they are more expressive and also support the definition of constraints between actors interacting indirectly with each other.

Locality of Constraints In the proposed prototype, constraints are stored at each actor and constraint verification is performed locally in the actor's execution context. For an external observer, the verification process seems to be behavior of the actor itself. In contrast, synchronisation logic in RTSynchronizers and QoS Synchronizers is defined externally in a special actor, which is responsible to perform coordination.

4.5.4 Actors with Temporal Constraints

4.5.4.1 Description

Actors with temporal constraints (ATC) [47], a model strongly influenced by temporal process algebra, allows developers to introduce active and passive temporal constraints on events and takes special emphasis on realtime systems. An active temporal constraint states, that certain actions must be executed in a specified time interval, because otherwise the system reaches a deadlock. A passive temporal constraint states, that if certain actions are not executed before a specified time limit, an exception will be thrown. Actors with temporal constraints uses the urgency-operator [57] to model active temporal constraints and the watchdog-operator [57] [69] to model passive temporal constraints.

4.5.4.2 Comparison

Expressible Constraints The described prototype provides a mechanism to introduce synchronization constraints on actors, whereas actors with temporal constraints provide a mechanism to introduce timing constraints on actors. While synchronization constraints support the expression of synchronisation protocols, timing constraints support the expression of timing protocols.

Real-time Applications The prototype of this thesis is built around AKKA, an actor concurrency implementation optimized for non-real-time applications. Actors with temporal constraints on the other hand was specifically proposed for use in real-time applications. While non-real-time applications are optimized for data throughput, real-time applications are optimized to fulfill their deadlines.

4.6 Development Process

4.6.1 Overview

For the development of the prototype, a software development process very similar to *SCRUM* [35] was used. Development was performed in iterations, with an iteration being two to three weeks long, depending on the chosen tasks and the author's time table. A short analysis of the current status was performed after each iteration, and the features to implement in the next iteration were chosen. The development was performed in a test-centric way: Unit tests, bug tests and integration tests were implemented and continuously executed over the span of the development process. Development and evaluation of the prototype were split in two separate phases: A *design and development phase* and an *evaluation phase*.

Design and Development Phase In the *design and development phase*, the prototype's architecture was planned and the prototype was realized. In Iteration 0, basic interception of messages between actors was implemented, in Iteration 1 the history protocol verification was implemented. In Iteration 2 and 3 the prototype was further improved, internal structures were optimized and the usability was revisited. In Iteration 4, a new implementation of the verification algorithm was implemented, as this proved to be necessary. For details on each iteration see Section 4.7.

Evaluation Phase In the *evaluation phase*, the prototype was tested and analyzed. In Iteration 5, two example applications using the prototype were implemented. The first one is a version of the Chameneos concurrency game [45], the second one a token ring application suitable for testing. For analysis, special focus was directed at measuring the prototype's performance. In Iteration 6, the prototype's performance was examined and evaluated. For details on each iteration see Section 4.7, for details on the implemented applications and the results of the performance evaluation see Chapter 5.

4.6.2 Technology

Programming Language The prototype was developed with *Java 7* [36] and *AspectJ 1.7.3* [26]. Besides Java, AspectJ is a key technology, as with it code can be executed in an actor's execution context whenever the actor sends or receives a message.

Underlying Actor System As the underlying actor concurrency implementation for the prototype *AKKA 2.10* [66] was chosen. Kilim, ActorFoundry and AKKA were considered. AKKA was selected because of various reasons: It runs on java, is open source and has detailed documentation. Because the prototype has to hook into the actor system's messaging algorithm, availability of the source code and good documentation helped considerably during development. AKKA is also actively maintained, has a solid developer base and is used for writing real world applications.

Used Libraries and Frameworks Logging was performed with *log4j 1.2.17*. For writing tests and for the actual testing *jUnit 4.11* was used. When mocking was required in unit tests or for the prototype in the early stages of development, *Mockito 1.9.5* was used.

4.6.3 Tool-set

Development Tools Development itself was performed with the help of a variety of tools. As integrated development environment *Eclipse Kepler* in the *Java EE* configuration was used. Additional Eclipse plugins installed and used were *AspectJ Development Tools 2.2.3*, *Scala IDE* *for Eclipse 3.0.3* and *m2e 1.4.1*. For automation of the build process and for dependency management *maven 3.0.5* was used. The profiler utilized for analysis and optimization in the later iterations of development was *VisualVM 1.7.0_45*.

Organizational Tools *Redmine 2.5.1* was employed not only as an issue tracker, but also as an overall organizational tool. *Git 1.7.9* was used for source control and an instance of *Gitlab 6.5.1* provided a remote repository for backup purposes.

Testing Toolset For writing tests and for testing *jUnit 4.11* and *Mockito 1.9.5* were used. Eclipse jUnit integration in combination with maven were used to continuously run tests during the development process.

4.6.4 Testing

4.6.4.1 General Approach

Testing was performed continually throughout the whole development process. Quality and correctness of the prototype were asserted by creating jUnit tests, performance was evaluated with a performance application. Tests were written before or immediately after production code and were continuously adapted to structural and behavioral changes. In order to write cleanly separated unit tests mocking was used. For reusing code required at multiple tests, jUnit *TestRules* were factored out.

4.6.4.2 Tests

Table 4.4 lists implemented tests. Important test cases are shortly described below. For details about the tested classes, see Section 4.2.

ConstraintTest This class implements tests which make sure that the *Constraint* class provides the expected grammar for constraints. Other tests verify that atomic conditions, negations and composite conditions calculate and return the correct senders and receivers. More tests assert that *Constraint* returns the correct outcome value as specified in the *Action*.

ConstraintBuilderTest and BuilderContextTest Test cases make sure that *ConstraintBuilder* supports all features of the *Domain Specific Language* used to describe creation of verification rules. Other test cases assert that the component creates constraints with the expected properties when used.

IndexedDataStoreTest and IndexerTests Test cases for *IndexedDataStore* verify, that the component uses registered indexers when it stores or queries values. Test cases for *Indexer* make sure that the generic methods provided for subclasses work as expected. Test cases for indexer implementations assert, that values are indexed and retrieved correctly.

Test Class	Classes Tested
ConstraintTest	Constraints Conditions Actions
ConstraintBuilderTest	ConstraintBuilder
BuilderContextTest	ConstraintBuilder
IndexedDataStoreTest	IndexedDataStoreImpl
IndexerTests	Indexed Data Store AbstractIndexer Indexer Implementations
InteractionLoggerImplTest	Interactions InteractionLoggerImpl
AutomatonBuilderTest	AutomatonBuilder
ConstraintStoreImplTest	ConstraintStore
BehaviorTest	Behavior
HistoryProtocolVerifierTest AutomatonProtocolVerifierTest	HistoryProtocolVerifier AutomatonProtocolVerifier
SendingActorAspectTest	SendingActorAspect
ReceivingActorAspectTest	ReceivingActorAspect

Table 4.4: Implemented Tests

InteractionLoggerImplTest and ConstraintStoreImplTest Test cases for *InteractionLogger* verify, that logging and querying for interactions works. Test cases for *ConstraintStore* check, if constraints can be added, retrieved and removed.

AutomatonBuilderTest Test cases to make sure that the creation of automatons works as expected. Tests assert that each *atomic condition* is implemented correctly and that the algorithm used for composition works as expected. Moreover there are tests for complexer combinations of the conditions and tests that assert the implemented optimization of automatons works as expected.

BehaviorTest Test cases for *Behavior* check, if the component triggers protocol verification when a message is sent or received and also if it triggers logging after an operation was allowed.

HistoryProtocolVerifierTest and AutomatonProtocolVerifierTest Test cases for the two implementations of the verification algorithm. They assert that the implementations implement the verification algorithm as specified. The implemented tests assert, that the algorithm implementations are correct and forbids an interaction if no rule enables or forbids it, forbids an interaction if a rule explicitly forbids it, and enables an interaction if a rule enables it and none forbids it.

SendingActorAspectTest and ReceivingActorAspectTest Test cases for the *AspectJ* aspects check, if the pointcut definitions are correct and if the aspects get called when a send or receive operation is performed by an actor. Other test cases make sure, that the prototype's behavior, verification and logging are called.

4.6.4.3 Performance Tests

For testing the prototype's performance, the *token ring* application was instrumented with *VisualVM* and executed with different parameter settings. In order to be able to execute performance tests easier, a launcher was developed which can be configured to execute the *token ring* application a number of times with specified actor and cycle count. For details about the performance tests and the prototype's performance, see Section 5.3.

4.7 Iterations

This section describes the evolution of the prototype during development. Each development iteration is shortly described and the performed changes are documented.

4.7.1 Iteration 0: Proof-of-Concept

Iteration 0 was all about a proof-of-concept. The first parts implemented were an *AspectJ* aspect for the interception of messages, classes to describe synchronization rules, a storage component for storing constraints and an interaction logger. Verification was not implemented before *Iteration 1*.

Intercepting Received Messages One of the first things implemented was an *AspectJ* aspect to test if an aspect would be able to hook into the message reception process of an actor and if it would run in the actor's context. The first test aspect only logged out an information message to assert that the pointcut intercepts the actor's receive operation correctly.

Synchronization Rule Structure The next steps were to create classes to describe synchronization protocol rules and interactions between actors, as well as components to store and retrieve them. *Constraint* describes a synchronization rule with conditions and an action. All interfaces and classes related to the description of synchronization protocol rules were created in this iteration, with exception of the *Always* condition. Manual creation of *Constraint* objects proved to be cumbersome – Listing 4.15 shows an example – this was however improved in the next iteration.

Listing 4.15: Creation of a Constraint without ConstraintBuilder

```
// When not received message and not sent message
//then do not AllowSendMessage and
//do not ForbidSendMessage
Constraint constraint2 =
      new Constraint();
```

```
ConditionNegation condition2_1_1 =
      new ConditionNegation();
condition2_1_1.setCondition(
      new MessageExistsCondition()
);
ConditionNegation condition2_1_2 =
      new ConditionNegation();
condition2_1_2.setCondition(
      new MessageExistsCondition()
);
ConditionConcatenation condition2_1 =
      new ConditionAnd();
condition2_1.setLeft(condition2_1_1);
condition2_1.setRight(condition2_1_2);
ActionNegation action2_1_1 =
      new ActionNegation();
action2_1_1.setAction(new AllowAction());
ActionNegation action2_1_2 =
      new ActionNegation();
action2_1_2.setAction(new ForbidAction());
ActionConcatenation action2_1 =
      new ActionAnd();
action2_1.setLeft(action2_1_1);
action2_1.setRight(action2_1_2);

constraint2.setConditions(condition2_1);
constraint2.setActions(action2_1);
```

Interaction Data Structure The data structure for storing interactions between actors was also created in this iteration. An *Interaction* object consists of a unique UUID and the addresses of the sender and the receiver. In the first draft, *Interaction* objects also contained the message itself, but this was later discarded as storing all messages is memory intensive and not required to analyze the currently available conditions.

Storing Synchronization Rules and Interactions The first implementation of *ConstraintStore* simply held a list with all synchronization rules for an actor in the form of *Constraint* objects and two maps containing lists with sender and receiver as map keys.

For logging of interactions between actors, the *InteractionLogger* component was developed. Early on it became clear, that indexing and filtering the message history efficiently would be important, so the indexer concept described before was introduced. The first implementation of *InteractionLogger* contained a list of the indexers, a map by name and another one by type. To log an interaction the *InteractionLogger* creates an *Interaction* object and runs it through registered indexers. Each *Indexer* indexes the *Interaction* on specific attributes and later can be used to find the *Interaction* by these attributes.

4.7.2 Iteration 1: Implementing HistoryLoggingVerification

This iteration's goal was to provide a first implementation of a synchronization verification algorithm. The basic verification algorithm as described before was designed and the *HistoryLoggingVerification* implementation was created, as well as a simple example application.

Verification *ProtocolVerifier* and the *HistoryProtocolVerifier* implementation as described earlier were developed in this iteration. The first version of the verification algorithm allowed messages from actors to be accepted if no rules for it were specified in an actor's rule set. This approach is not ideal and was later changed.

PrototypeActor Until now, *InteractionLogger* and *ConstraintStore* for each actor were stored and received centrally. This was a major problem, because that required explicit synchronisation and defies the fundamental idea of the prototype. In order to store rules and interactions locally at actors, a class *PrototypeActor* that extends *UntypedActor* was created. It can hold the actor's *InteractionLogger* and *ConstraintStore*. For *UntypedActor* actors, which still can be used with the prototype, *InteractionLogger* and *ConstraintStore* still have to be stored centrally though.

Simple Example Application To assess if the verification implemented works as expected, a simple example application was written. It consists of three actors sending each other messages in various combinations and with different rule sets.

4.7.3 Iteration 2: Improving Concept and Usability

Iteration 2 had the goal to improve the prototype's structure and usability.

Improve Protocol Verification While in *Iteration 1* interactions between actors were allowed by default, they became forbidden by default in *Iteration 2*. From a correctness and security point of view, this whitelist-approach is by far the better one [65]. Unfortunately, this however requires developers usually to define more rules to allow messages to be sent and received.

Creating an ActorId *AKKA* provides the possibility to send messages to an actor from *ActorRef.noActor()* [66] without specifying a sender, which is represented by an *ActorRef* with the path set to null. This dead letterbox actor can for example be used to tell an actor to start when the system is initialized and no other actor has yet been created. Unfortunately when comparing two *ActorRef* objects, *AKKA* tries to access their path variables without checking if they are set. As the prototype logs interactions by creating an *Interaction* object and stores it in a map with *ActorRef* as the key, this can result in a *NullPointerException*. In order to solve the issue, a wrapper class *ActorId* was created, which treats dead letterbox actor references correctly. Luckily it later could be removed in favor of a *NoActor* class that extends *ActorRef* and represents the dead letterbox actor.

Factor out IndexedDataStore *ConstraintStore* and *InteractionLogger* both need to efficiently store and retrieve objects of a certain type. This is a case, where factoring out a base component with common functionality obviously made sense. The component *IndexedDataStore* is the result of this consideration. *IndexedDataStore* provides functionality to efficiently store and retrieve objects and to add custom indexers as required.

Intercepting Send Operations While intercepting the receipt of messages is sufficient from a theoretical point of view to support synchronization validation, a developer might also want to provide rules to tell an actor not to send a message away as well. To support this, another *AspectJ* aspect was created, that hooks into the actor's process of sending messages. As a consequence, a rule now has to be added to the sender as well as to the receiver in order to allow a message to be transmitted. As another consequence, the method *ActorRef.tell(...)* should only be called from within an actor, as the prototype otherwise tries to retrieve *ConstraintStore* and *InteractionLogger* from the current *this* object, fails, and subsequently creates and stores new instances centrally.

Introduction of ConstraintBuilder Construction of synchronization rules by hand proved quickly to become cumbersome. To improve the prototype's usability, a *Builder* [33] using the *Fluent interface* pattern [31] was introduced. The builder provides a *domain specific language* for the construction of rules. With the aforementioned software patterns, the object creation process can be implemented in a context sensitive way, and only provides meaningful options at each step. Listing 4.16 shows creation of a rule with the new builder.

Listing 4.16: Creation of a Constraint with the ConstraintBuilder

```
final ConstraintBuilder builder =
        new ConstraintBuilderImpl();
final Constraint constraint =
        builder.when().messageExists(actors.a, actors.b)
                .then().allow(actors.b, actors.c)
                .build();
```

4.7.4 Iteration 3: Refactoring, Clean-Up and Improvements

Iteration 3 was mostly about refactoring and cleaning up existing code to improve object-oriented design, maintainability and clarity as well as improvements.

Creation of a Behavior Class The interjection of send and receive operations, the gathering of data and the verification were so far mangled together. To separate these concerns, an interface *Behavior* and a default implementation *DefaultBehavior* were introduced. The *AspectJ* aspect's only responsibility now is to interject the actor operation, then calling *Behavior*, which builds a *VerificationContext* object that contains all data required for verification and then triggers the verification.

Creation of ProtocolVerifier The synchronization protocol verification itself was factored out from the rest of the prototype's behavior by creating an interface *ProtocolVerifier*. Encapsulating the verification improves the code quality, but more importantly it also enables an easy exchange of the verification algorithm implementation. In a later iteration this process was brought to finish, so that as of now the verification implementation can be completely exchanged. There are currently two implementations of the *ProtocolVerifier* interface: *HistoryProtocolVerifier* and *AutomatonProtocolVerifier*.

Introduction of VerificationContext When separating message interception from behavior and verification, a holder class *VerificationContext* was introduced. With it contextual information about the currently intercepted operation can easily be passed around. It contains the type of the examined operation as well as the current actor's verification data.

Removal of Rules Because actors can dynamically change their behavior, and with it their interaction profile, a necessary improvement was to provide functionality to remove rules at runtime from an actor's synchronization rule set. As *ConstraintStoreImpl* extends and uses *IndexedDataStore*, removal of rules was provided by enhancing that component. A *remove(...)* method was added to the interfaces *IndexedDataStore*, *Indexer* and *Filter*, and then was implemented in *IndexedDataStoreImpl*, *AbstractIndexer* and *AbstractFilter*. *IndexedDataStoreImpl's remove(...)* method iterates over all registered *Filters* and *Indexers* and calls their *remove(...))* method, which removes the given object from the *Indexer* or the *Filter*.

Introduction of Configuration Messages Configuring actors from a central point at startup proved to become a common use case, and so a special type of message called *ConfigMessage* was introduced to easily allow it. A *ConfigMessage* contains a number of actor addresses, which the actor in question requires to construct its synchronization rules. When such a message gets sent to an actor, *ReceivingActorAspect* intercepts it as usual and *Behavior* processes it, but it gets treated specially and is forwarded to the actor's *configure(...)* or *reconfigure(...)* method instead of to the *onReceive(...)* method. The first *ConfigMessage* is forwarded to *configure(...)*, all subsequent ones are forwarded to *reconfigure(...)*, which allows developers to separate initial configuration from later reconfigurations.

Introduction of AnyActor While experimenting with the prototype, a new actor wildcard *AnyActor* for use in synchronization rules was created. Where it is used as a parameter, any actor is allowed to occur. *AnyActor* was implemented like *NoActor* by creating a class that extends *ActorRef*. The prototype specifically checks for instances of *NoActor* and *AnyActor* and behaves accordingly in the verification process.

Creation of a Prototype Facade as API To improve usability, a *Prototype* class was introduced. It was designed with the *Facade Pattern* [33] and provides a simple to use API for accessing all the prototype's capabilities. *Prototype* provides developers easy access to a *ConstraintBuilder* instance for creating new rules, methods to add and remove rules to an actor, and it allows developers to get an actor's *ConstraintStore* and *InteractionLogger*. Introduction of the *Prototype* class made it possible to remove significant amounts of boilerplate code in applications that use the prototype.

4.7.5 Iteration 4: Implementation of Automaton Verification

In *Iteration 4*, the *automaton verification* implementation of the verification algorithm was created. This proved to be necessary, because the *History Logging Verification* implementation has conceptual issues with memory and long-running applications. In order to support easy exchange of the verification algorithm implementation, the prototype was further refined.

Introduction of VerificationData In order to allow the different implementations of the verification algorithm to store custom data into actors, a *VerificationData* class was introduced. Each *PrototypeActor* now stores such an object, which can be used by the verification algorithm. An implementation of the verification algorithm has to provide a method which creates such a data object; this method is used by the prototype when it is required to create a *VerificationData* object for the currently used verification algorithm implementation.

Adapt Behavior and DefaultBehavior Because verification algorithms have to be informed about adding and removing rules to an actor, *Behavior* and *Verification* interface and their implementations were adapted to provide this kinds of events in the form of methods. This was the last step to make the implementation of the verification algorithm completely exchangeable. When a rule gets added, the *History Logging Verification* adds/removes the rules to/from its *ConstraintStore*, the *automaton verification* creates and stores a new *Automaton* when a rule gets added and removes it when the rule gets removed.

Implementation of AutomatonVerificationProtocol The *AutomatonVerificationProtocol* implementation of the verification algorithm implemented here creates an automaton for each rule that is added and logs interactions with the actor by changing the automaton's active state, which then is used to check the firing of rules.

4.7.6 Iteration 5: Creating Examples

In *Iteration 5*, example applications were created, which show and analyze the prototype's capabilities and performance.

Chameneos Application The *chameneos application* implements the *chameneos concurrency* game [45] with AKKA and the prototype. *Chameneos* was chosen as example application, because it provides a prototypical example of an application with concurrently running participants. Chameneos are creatures with red, blue or yellow skin. When two chameneos interact with each other, both change their colors into the third possible one. Chameneos find each other with the help of a meet-up service called the pall mall. The first implementation of the application created only 2 chameneos actors interacting with each other a single time for testing purposes. After asserting that the application behaved as expected, the chameneos count was increased to 4 and the chameneos were changed to interact endlessly. Later the chameneos count was increased to 100.

Token Ring Application The *token ring* application lets actors send a message in a circle from actor to actor. Both the number of actors and the number of circles is configurable. The *token ring* application was implemented in this iteration and was later used for performance tests and to find optimization targets.

4.7.7 Iteration 6: Performance Evaluation

The goal of *Iteration 6* was to examine the prototype's performance. In order to do that, the *token ring application* was used and a test-runner was implemented.

Test-runner To make performance testing easier, a test-runner was implemented, which can be configured by parameters for different test profiles. Supported parameters are the algorithm implementation used, the number of actors and the loop count. The test-runner uses the cartesian product on these parameters and lets the *token ring application* run in all these configurations. The test-runner also is able to (re-)compile the application with or without the prototype, dependent on the currently used profile.

Performance Comparison In order to measure the performance without prototype, with *HistoryProtocolVerifier* and with *AutomatonProtocolVerifier*, the test-runner with the *token ring* application was run in different parameter configurations. Each configuration was executed 100 times on an idle test system in order to gain a relevant result. For details about the results, see Section 5.3.

CHAPTER 5

Evaluation

This chapter describes how the developed prototype was evaluated. First, the worst-case runtimes of the implemented algorithms are examined. Then the sample applications which were created during development for the purpose of evaluating the prototype are described. After that, performed benchmarks and the results of them are discussed. Finally, some future topics of research are given which would improve the prototype further.

5.1 Algorithm Runtime

This section examines the worst-case runtimes for both implemented verification algorithms as well as the runtime of automaton creation for the *automaton verification algorithm*.

5.1.1 Runtime History Logging Verification

We split the algorithm into subtasks, look at each task's runtime independently and then determine the overall runtime by combining the subresults. The *history logging verification* first finds all applying rules and then checks for each found rule if it fires.

Find Applying Rules Finding the applying rules in the actor's local *ConstraintStore* can be performed in

$$O(stored\_rules) \tag{5.1}$$

where *stored_rules* is the number of rules stored in the *ConstraintStore*, as finding an indexed value in a Java *HashMap* has linear worst-case runtime (while the average-case runtime is constant if *equals* and *hashCode* are implemented correctly).

Checking Rules When checking the selected rules, the algorithm loops over them and for each one checks if it fires. While the algorithm does break the loop if it finds a rule holding and forbidding the current operation, in the worst-case the algorithm has to loop over all rules selected.

To check if a rule fires it has to be determined wether the rule's condition holds. Because conditions are represented as a tree structure with *composite conditions* as inner nodes and *atomic conditions* as leafes and because all subconditions have to be considered, the worst-case runtime depends on the number of subconditions and the biggest worst-case runtime to check any existing *atomic condition*.

Always and *Not* have runtime $O(1)$ because the first one always holds and immediately returns and the second one simply negates the outcome of its subcondition. For *MessageExists* the runtime is $O(interaction\_count)$ where *interaction_count* is the number of *Interactions* saved for that particular pair of sender and receiver, because the *Indexer* class also uses Java's *HashMap*. So regarding the runtime, the worst *atomic condition* to have is a *MessageExists*.

In conclusion, the worst-case runtime for checking one rule is

$$O(\sum_{selected\_rules} \sum_{conditions} interactions[condition])) \tag{5.2}$$

where

- *selected_rules* are the selected rules
- *conditions* are the conditions in the condition tree
- *interactions* are the interactions for the sender/receiver pair of a specific condition

Overall Worst-Case Runtime In conclusion, the overall worst-case runtime for the *history verification algorithm* is

$$O(|stored\_rules| + (\sum_{selected\_rules} \sum_{conditions} interactions[condition])) \tag{5.3}$$

where

- *stored_rules* are the rules stored for the current actor
- *selected_rules* are the rules filtered out for the current sender/receiver pair
- *conditions* are the conditions per rule
- *interactions* are the interactions for the sender/receiver pair of a rule

5.1.2 Automaton Verification Runtime

The automaton verification algorithm has to loop over all found rules for the current actor and then has to check if a rule fires, which can be performed in $O(1)$. The overall worst-case runtime of a verification with the *automaton verification algorithm* simply is

$$O(rules) \tag{5.4}$$

where *rules* is the number of rules for an actor. Again, the overall average runtime might be less, because the algorithm breaks the loop if it finds a rule that holds and forbids the current operation.

5.1.3 Automaton Creation Runtime

Whenever a rule gets added to an actor, a corresponding automaton gets created. In order to examine the overall runtime for automaton creation, we split the creation into two subtasks and examine them separately, combining the results. In the following section n always references the number of substates in the start state.

Creation In order to create all states of an automaton, the algorithm has to iteratively follow each existing edge beginning from the start state and create a new node whenever following an edge.

The runtime to create the corresponding states for an automaton that represents one of the *atomic conditions* is $O(1)$, because the pattern of states to create is fixed. For the *composite conditions And* and *Or* the worst-case runtime depends on the number of states that have to be created, which is equal to the number of overall edges to follow. This number can be determined easily, as the start state of a *composite condition* is a *JoinedState* which contains a number of *LeafStates* where each one represents the occurrence or non-occurrence of a message. As each *LeafState* has at most one outgoing edge to another *LeafState*, the state graph of an automaton with conditions basically becomes a decision tree where for each depth level the possible number of decisions gets reduced by one. The overall worst-case runtime for automation creation is therefore determined by the sum of nodes in each depth level which is

$$1 + (\sum_{m=1}^{n} \prod_{k=m}^{n} k) \tag{5.5}$$

Optimization After creation, the automaton gets optimized to reduce the number of states as much as possible. Discounting duplicate states, the number of remaining states in the automaton is

$$2^n \tag{5.6}$$

because this is the number of possible permutations of n binary decisions. The optimization loops over all states in the automaton and outgoing from each state tries to find a group of joinable states by following all edges (direct and indirect) of that state. For the worst-case runtime we have to assume, that no groups of states can be found that can be joined. So the operation's overall runtime is determined by the sum of direct and indirect outgoing edges between states for all states, which varies over each depth level and is in total

$$\sum_{m=1}^{n} \prod_{k=1}^{m} k \tag{5.7}$$

If a group of states has been found, it can be joined to one state.

Overall Worst-Case Runtime The overall worst-case runtime of automaton creation depends on the number of substates in the start state and is

$$O((1 + (\sum_{m=1}^{n} \prod_{k=m}^{n} k)) + (2^n * \sum_{m=1}^{n} \prod_{k=0}^{m} k)) \tag{5.8}$$

5.2 Sample Applications

For testing the prototype under real world conditions, two sample applications were implemented. The first one is a token-ring of actors, the second one is an implementation of the chameneos concurrency game.

5.2.1 Token-Ring

5.2.1.1 Overview

The token ring application lets a number of actors send messages in a circle from actor to actor. Both the number of actors and the number of loops are configurable by parameters. The token ring application was selected, because it can be quickly implemented and can easily be used for performance testing and debugging purposes. The token ring application consists of a *main class*, *token ring actors*, a special *statistic token ring actor* and *message objects* sent between the actors.

5.2.1.2 Main Class

The main class creates the actor system and the actors in the token ring. As parameters it expects two long values. The first one specifies the amount n of actors to create, the second one the number of loops l that should be performed. The main class creates and configures $n - 1$ *TokenRingActor* objects and 1 special *StatisticTokenRingActor* object. Every actor receives the last actor's address from which it will receive messages and the next actor's address which it will send messages to. Every actor locally creates and adds a rule which allows it to receive messages from its predecessor and to send messages to its successor. The *StatisticTokenRingActor* object also receives l as parameter. Then the main class sends a *StartMessage* to the *StatisticTokenRingActor* object. Listing 5.1 shows the creation and initial configuration of the *StatisticTokenRingActor* actor and the *TokenRingActor* actors.

Listing 5.1: Creation and Initial Configuration of Participating actors

```
public class TokenRingMainWith{
   ...
   private static void createTokenActors() throws
   ConstraintBuilderException {
      statisticActor =
         actorSystem.actorOf(
      Props.create(StatisticTokenRingActor.class),
      "StatisticTokenRingActor");
   for (int i = 0; i < actorCount; i++) {
      ActorRef current =
         actorSystem.actorOf(
            Props.create(TokenRingActor.class),
            "TokenActor" + (i));
      tokenActors.add(current);
   }

   tokenActors.get(0).tell(
      new ConfigMessage(
         new ConfigPair("lastActor", statisticActor),
         new ConfigPair("nextActor", tokenActors.get(1)))
,
         Prototype.noActor());
   ActorRef last = tokenActors.get(0);
   ActorRef current = tokenActors.get(1);
   ActorRef next = tokenActors.get(2);

   for (int i = 2; i < actorCount; i++) {
      if (i > 2) {
         last = current;
         current = next;
         next = tokenActors.get(i);
      }
      current.tell(
         new ConfigMessage(
            new ConfigPair("lastActor", last),
            new ConfigPair("nextActor", next)),
            new NoActor());
   }

   tokenActors.get(actorCount - 1).tell(
      new ConfigMessage(
         new ConfigPair(
            "lastActor",
            tokenActors.get(actorCount - 2)),
         new ConfigPair(
            "nextActor",
            statisticActor)),
         Prototype.noActor());
```

```
        last = tokenActors.get(actorCount - 1);
        next = tokenActors.get(0);
        statisticActor.tell(
           new ConfigMessage(
              new ConfigPair("lastActor", last),
              new ConfigPair("nextActor", next)),
              Prototype.noActor());

        Constraint noActorToStatisticActor =
           Prototype.builder()
              .when().always()
              .then().allow(Prototype.noActor(),
statisticActor)
           .build();
        Prototype.addRule(new NoActor(),
noActorToStatisticActor);
}
```

5.2.1.3 StatisticTokenRingActor

When receiving the *StartMessage*, the *StatisticTokenRingActor* object logs the time of beginning the execution and then sends a *TokenMessage* to its successor. The successing actor does the same with its successor and so on. When the *StatisticTokenRingActor* object receives the next *TokenMessage*, one loop has been performed. After the specified number of loops, it prints out the time required for the execution and shuts down the Akka *ActorSystem* instance. Listing 5.2 shows the *onReceive* method of the *StatisticTokenRingActor* class.

Listing 5.2: onReceive method of StatisticTokenRingActor

```
public class StatisticTokenRingActor extends PrototypeActor{
   ...
   @Override
   public void onReceive(Object msg) throws Exception {
      if (msg instanceof StartMessage) {
         this.actorSystem = ((StartMessage) msg).actorSystem;
         this.cycles = ((StartMessage) msg).cycles;
         this.startTime = System.nanoTime();
         nextActor.tell(new TokenMessage(), this.getSelf());
      }
      if (msg instanceof TokenMessage) {
         long counter = times.incrementAndGet();
         printOutStatistics(counter);
         nextActor.tell(msg, this.getSelf());
      }
   }
}
```

5.2.1.4 TokenRingActor

Every *TokenRingActor* configures itself to be enabled to receive messages from its predecessor and to send messages to its successor. Other than that, a *TokenRingActor* actor waits for incoming *TokenMessages*, sending them immediately along to its successor. Listing 5.3 shows the *onReceive* method of the *TokenRingActor* actor class.

Listing 5.3: onReceive method of TokenRingActor

```
public class TokenRingActor extends PrototypeActor{
   ...
   @Override
   public void onReceive(Object msg) throws Exception {
      if(msg instanceof TokenMessage){
         nextActor.tell(msg, this.getSelf());
      }
   }
}
```

5.2.1.5 Messages

The token ring application requires two messages to be exchanged besides the prototype's *ConfigMessage* messages. The first one, *StartMessage*, tells the *StatisticTokenRingActor* object when to start and send the first *TokenMessage* along. It contains the reference to the *ActorSystem* object which is required for shutting down the actor system when the application finishes. The *TokenMessage* fungates as the token passed along between the actors and does not contain further information inside.

5.2.2 Chameneos

5.2.2.1 Overview

As an example on how to use the prototype under real world conditions, the *Chameneos game* [45] was implemented with rules that only allow specified messages. The game was specifically designed as an example for examining the properties of concurrent languages and frameworks. It works as follows. *Chameneos* are little lizards, which may have red, blue or yellow skin. They like to live doing all kinds of things like eating, sleeping and they really like playing pall mall — an early precursor of croquet — with each other. Each *chameneos* lives it's live doing random stuff for a random period of time. At each moment it can decide that it now wants to play a game of pall mall for company. In order to find a partner to play with, the *chameneos* have a meeting place they go to, the *pall hall*. When a partner was found at the *pall hall*, the *chameneos* start playing with each other. Whenever two *chameneos* play pall mall together, they change their own and the other's color into the third possible one.

Chameneos Behavior The behavior of a *chameneos* is as follows. Before a playdate, a *chameneos* does asynchroneous actions for a random period of time. When a *chameneos* chooses it wants to play pall mall, it sends a message to the *pall hall*, afterwards waiting to be teamed up. When receiving a message from the *pall hall* specifying a play partner, the two *chameneos* directly interact with each other in order to play and change their colors.

Pall Hall Behavior The *pall hall* waits for playdate requests from *chameneos*. Whenever two requests were received, the *pall hall* teams the two *chameneos* up and sends notification messages to both of them. More than two *chameneos* might want to play at a given time so multiple rendez-vous requests can happen and all requests have to be registered for later processing. Notifications for playdates have to be sent as soon as possible to the participating *chameneos*.

Implementation The performed implementation has three important classes which hold most of the functionality. These are *ChameneosMain*, *PallHall* and *Chameneos*. Other classes involved are either message objects or they encapsulate services and utilities used by the aforementioned three classes. The first implementation of the *chameneos* example only used 2 *chameneos* actors for testing purposes, which also only tried to play pall mall one time. After the application behaved as expected, the number of *chameneos* actors was increased to 4 and the *chameneos* actors were changed to play pall mall endlessly. Then the *chameneos* count was increased again to 100 and the application was run and monitored 20 minutes without interruption to make sure it behaved as expected in the longer run.

5.2.2.2 ChameneosMain

ChameneosMain contains the main method. It creates the *PallHall* actor and a configurable number of *chameneos* actors. Then it configures the *chameneos actors* allowing them to receive messages from *NoActor*, sending the *chameneos ConfigureMessage* objects so they can locally configure their rule sets accordingly. Listing 5.4 shows the creation of the *chameneos* actors. After that, a *StartMessage* is sent to all *chameneos* actors, indicating they can start living, doing their asynchroneous tasks and request partners for playing pall mall.

Listing 5.4: Creating the Chameneos actors

```
public class ChameneosMain {
   ...
   private static void createChameneos()
   throws ConstraintBuilderException {
      final SecureRandom random = new SecureRandom();
      for (int i = 0; i < chameneosCount; i++) {
         final Color chameneosColor =
            Color.fromValue(random.nextInt(3));
         final String chameneosName =
            "Chameneos_" + Integer.toString(i);
         final ActorRef currentChameneos =
            actorSystem.actorOf(
                Props.create(
                    Chameneos.class,
                    chameneosName,
                    chameneosColor),
                chameneosName);

            currentChameneos.tell(
                new ConfigMessage(pallHall),
                Prototype.noActor());
            chameneos.add(currentChameneos);

            Constraint anonymeousToChameneos =
                Prototype.builder()
                    .when().always()
                    .then().allow(
                        Prototype.noActor(),
                        currentChameneos)
                .build();

            Prototype.addRule(
                new NoActor(),
                anonymeousToChameneos);
        }
    }
}
```

5.2.2.3 PallHall Actor

PallHall is the actor that realizes the component teaming up *chameneos* actors for playdates. It simply waits for *WantToPlay* messages from *chameneos*. Whenever two have arrived, the *PallHall* sends the both sender *chameneos* a *HereIsYourPartner* message, teaming them up. The *chameneos* actors then are responsible for playing and mutating their color on their own and the *PallHall* again waits for incoming *WantToPlay* messages. Listing 5.5 show the *onReceive()* method of the *PallHall* actor class.

Listing 5.5: onReceive method of PallHall

```
public class PallHall extends PrototypeActor{
    ...
    @Override
    public void onReceive(Object message) throws Exception {
        if (message instanceof WantToPlay) {
            WantToPlay playMessage =
                (WantToPlay) message;
            if (first == null) {
                first = playMessage.getChameneos();
            } else {
                second = playMessage.getChameneos();
                first.tell(
                    new HereIsYourPartner(second),
                    this.getSelf());
                second.tell(
                    new HereIsYourPartner(first),
                    this.getSelf());
                first = null;
                second = null;
            }
        }
    }
}
```

Rules The *PallHall* has a fixed set of rules that get added when the first and only *ConfigureMessage* is received. These rules are to

- Always enable receipt of messages from any actor
- Always enable sending of messages to any actor

Listing 5.6 depicts the code used for building and adding the rules in the *PallHall* actor class.

Listing 5.6: Code for Adding Initial Rules for the PallHall Actor

```
Constraint receiveFromAny =
    Prototype.builder()
        .when().always()
        .then().allow(new AnyActor(), this.getSelf())
    .build();
Constraint sendToAny =
    Prototype.builder()
        .when().always()
        .then().allow(this.getSelf(), new AnyActor())
    .build();

Prototype.addRule(this, receiveFromAny);
Prototype.addRule(this, sendToAny);
```

5.2.2.4 Chameneos Actor

Chameneos is the class implementing *Chameneos* actors. A *Chameneos* waits for *StartMessage* messages to arrive. When the message arrives the *Chameneos* waits for a random time. After that it wants to play with another *Chameneos* and contacts the *PallHall* actor with a *WantsToPlay* message. When receiving a *HereIsYourPartner* message in return, the *Chameneos* adds the required rules so it can send messages to and receive messages from its partner. Then it sends a *ColorQuestion* message to the partner *Chameneos*, requesting the partner's color. Immediately after the message has been sent, the rule allowing to send messages to the partner gets removed. When a *ColorAnswer* message from the partner arrives, the *Chameneos* mutates its color with the partner's color and removes the rule allowing the actor the receipt of messages from the partner. After that, the *Chameneos* again waits a random time, later again contacting the *PallHall* to play pall mall. Listing 5.7 shows the *onReceive()* method of the *Chameneos* actor class.

Listing 5.7: onReceive method of Chameneos

```
public class Chameneos extends PrototypeActor{
   ...
   @Override
   public void onReceive(Object message) throws Exception {
      if (message instanceof Start) {
         doOtherStuff();
         sendWantToPlay();
      } else if (message instanceof HereIsYourPartner)
         sendColorQuestion((HereIsYourPartner) message);
      else if (message instanceof ColorQuestion)
         sendColorAnswer((ColorQuestion) message);
      else if (message instanceof ColorAnswer) {
         processColorAnswer((ColorAnswer) message);
         doOtherStuff();
         sendWantToPlay();
      }
   }
}
```

Rules The following rules get added when the first *ConfigureMessage* arrives and in the rule set for the time the *Chameneos* exists.

- Enable receiving of a messages from *NoActor* so the Start message can be received
- Always enable sending messages to the *PallHall*
- Always enable receiving messages from the *PallHall*

Listing 5.8 shows the code for adding those rules to the *Chameneos actor*.

When the *Chameneos* gets a partner for playing, it adds two more rules to its rule set.

- Enable sending messages to the partner *Chameneos*
- Enable receiving messages from the partner *Chameneos*

Those rules get removed after one message has been sent and received respectively. Listing 5.9 shows the code for adding those rules to the *Chameneos actor*.

Listing 5.8: Code for Adding Initial Rules for the Chameneos Actor

```
Constraint receiveFromNoActor =
   Prototype.builder()
      .when().always()
      .then().allow(Prototype.noActor(), this.getSelf())
   .build();
Constraint receiveFromPallHall =
   Prototype.builder()
      .when().always()
      .then().allow(pallHall, this.getSelf())
   .build();
Constraint sendToPallHall =
   Prototype.builder()
      .when().always()
      .then().allow(this.getSelf(), pallHall)
   .build();

Prototype.addRule(this, receiveFromNoActor);
Prototype.addRule(this, receiveFromPallHall);
Prototype.addRule(this, sendToPallHall);
```

Listing 5.9: Rules Added to Chameneos actor when Partnered Up

```
Constraint sendToPartner =
   Prototype.builder()
      .when().always()
      .then().allow(myself, partner)
   .build();
Constraint receiveFromPartner =
   Prototype.builder()
      .when().always()
      .then().allow(partner, myself)
   .build();

Prototype.addRule(this, sendToPartner);
Prototype.addRule(this, receiveFromPartner);
```

5.2.2.5 Messages

The following messages were defined and are used in the *chameneos* application for communication.

StartMessage Message The *StartMessage message* is sent by *ChameneosMain* to every *chameneos* actor indicating that the configuration phase is finished and the *chameneos* should start its live by waiting for a random period of time and then requesting a playdate at the *PallHall* actor.

WantToPlay Message The *WantToPlay* message is sent to the *PallHall* by a *chameneos* when it wants a playdate and requires a partner. The *WantToPlay message* contains the address of the *chameneos* actor making the request.

HereIsYourPartner Message The *HereIsYourPartner message* is sent to a *chameneos* by the *PallHall* when partnered up two *chameneos*. The message contains the address of the *chameneos* that was chosen as a playdate partner.

ColorQuestion Message A *ColorQuestion* is sent to *chameneos* by their partner *chameneos* after they have been matched up to request each other's color. The message contains the partner *chameneos'* address to allow checking the consistency of the conversation.

ColorAnswer Message A *ColorAnswer message* is sent to a *chameneos* by its current partner *chameneos* after the partner has received a *ColorQuestion message*. The message only contains the color of the *chameneos*. When received by the *chameneos*, it mutates its color determining the new color by using its own color and the color provided in the partner's *ColorAnswer message*.

5.3 Benchmark

This section describes how performance of the prototype was measured and shows the benchmark results.

5.3.1 Setup

Test Application For measuring the prototype's performance, the token ring application was used. It was chosen, because this kind of application provides meaningful benchmark results. Moreover it also can be configured with the number of participating actors and the number of ring cycles, which makes testing the prototype in different configurations easy. The chameneos application was not used for benchmarks because the chameneos actors wait a random time in between interactions with each other, so benchmarks would not be comparable.

Test Machine Testing was performed on a machine with the following specifications:

- Processor: Core i7 Q820 1.73 GHZ, QuadCore with HyperThreading
- RAM: 8 GB RAM
- Disk: 120GB System-SSD
- OS: Windows 7 x64

Test Configurations The token ring application was run

- without the prototype
- with the *automaton verification algorithm*
- with the *history logging verification algorithm*

Each of these configurations was run with 10 as well as with 100 actors and for 1.000, 10.000 and 100.000 token ring cycles to analyze the prototype's performance under different conditions. All configurations were run 100 times on the idle test machine and the performance comparison was done against the mean value over the run times to compensate for possible scheduling or performance inconsistencies of the test machine.

5.3.2 Result and Interpretion

5.3.2.1 Result Data

Results Table 5.1 lists the runtime for different benchmark configurations in seconds. Figure 5.1 and Figure 5.2 graphically depict the results for 10 and 100 actors. The x-axis shows the amount of loops performed, the y-axis shows the runtime in seconds.

Actors	10			100		
Loops	1.000	10.000	100.000	1.000	10.000	100.000
Without Verification	3.230	6.949	26.393	5.984	24.741	197.961
Automaton Verification	5.726	11.701	44.341	10.066	42.539	342.927
History Logging Verification	6.812	13.273	51.190	11.400	47.345	562.112

Table 5.1: Performance measurements, averaged and in seconds

Interpretation When compared to the overall runtime of the token ring application without the prototype, the overall runtime for both verification algorithm implementations is similar for small numbers of actors and ring cycles. In such configurations message interception, creation of the verification context and calling the verification seem to contribute significantly to the overall runtime.

For higher numbers of actors and ring cycles however, the *automaton verification algorithm* obviously performs better than the *history logging algorithm*. This is to be expected, because of two reasons: First of all the check whether a message is enabled or forbidden is for the *automaton verification algorithm* – in contrast to the *history logging algorithm* – a constant operation. Secondly, and even more important, memory requirement of the former does not increase for longer interaction histories and therefore performance does not degrade over time.

Generally, the *automaton verification algorithm* provides significant improvements over the *history logging algorithm*. The *automaton verification algorithm* can verify and log interactions in $O(1)$, because the information for both operations is hashed efficiently in an automaton's current state. Only addition and removal of rules to actors have a significant runtime overhead

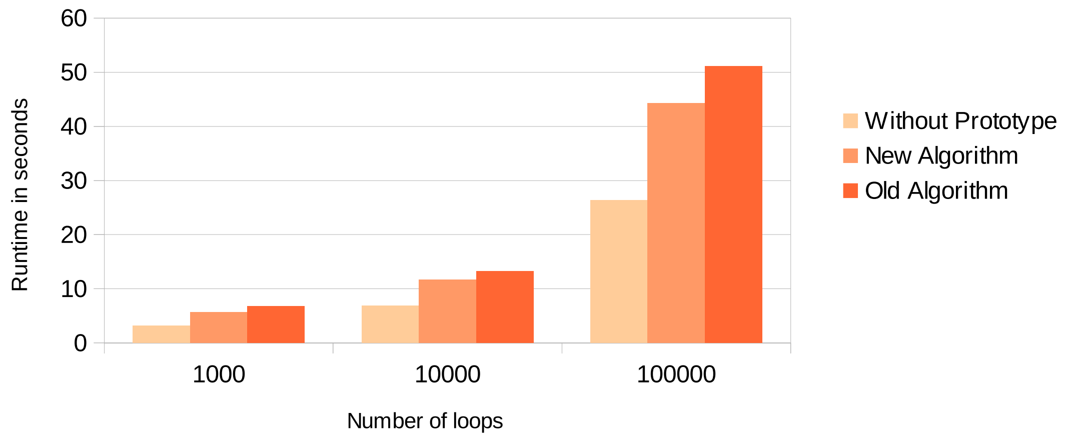

Figure 5.1: Performance with 10 Actors

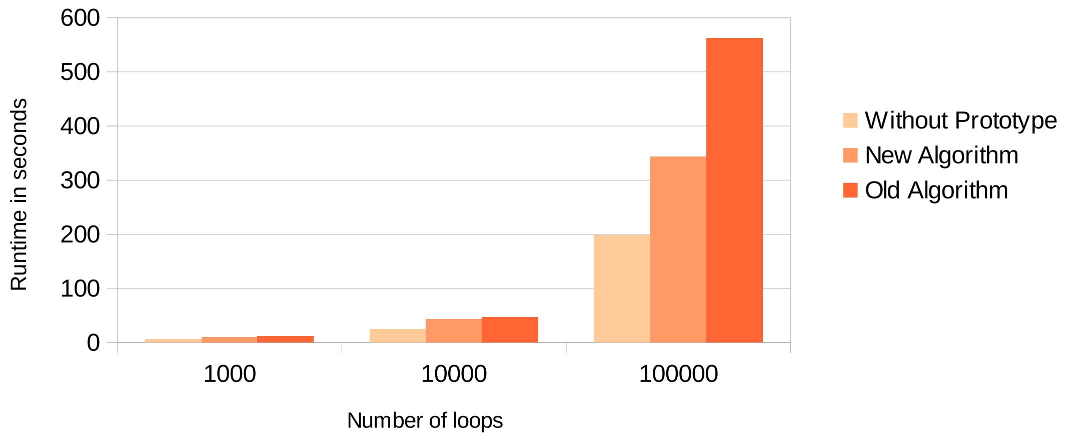

Figure 5.2: Performance with 100 Actors

in this verification implementation. The memory requirement only depends on the complexity of a rule, so performance doesn't degrade over time.

In contrast, the *history logging algorithm* has to check each condition from each rule against the previously stored message history of the current actor, which is a much more expensive operation. Moreover, an actor's message history grows linearly with each message sent and received, which can become a serious problem for long-running applications.

5.4 Future Research

This section describes topics for possible future areas of research which were identified but not tackled or implemented in the scope of this thesis.

5.4.1 New Rule Conditions

In order to improve expressibility of the prototype's rules, more rule conditions should be implemented. It might for example be useful to be able to define conditions over the order in which messages were sent or received, or to take a message's content into consideration. The prototype was designed to support easy extension. Implementation of a new condition only requires the extension of logging, if information not recorded so far is required, and the implementation of the condition itself.

5.4.2 Actor Wildcard Parameters

So far only addresses of specific actors or the ones of *AnyActor* and *NoActor* can be used as parameters in rule conditions. Parameter wildcards that match multiple actors would provide a comfortable way to define rules for a group of actors. For example, all actors that implement a certain interface, extend a specific class, or have certain other properties could be selected by such a wildcard.

5.4.3 Automatic Replacement of UntypedActor

For already existing AKKA applications, an automatic mechanism to exchange each occurrence of *UntypedActor* with *PrototypeActor* could be provided. Such a mechanism could be implemented either with a small pre-compiler application that changes the source-code, or with a library like *ASM* [18] that operates on the byte-code level. Which approach is the more appropriate one will require careful consideration.

5.4.4 Actors Distributed over Network

Real-World applications that use the prototype might want to leverage AKKA's ability to distribute actors over multiple devices. Networking aspects were not considered for the prototype in this thesis in order to lower the complexity. While the prototype conceptually should work the same when used with actors distributed over network, it was not tested in such environments.

Tests will have to be performed and based on the outcome, improvements and optimizations should be designed and implemented.

CHAPTER 6

Conclusion

This thesis describes design, implementation and evaluation of a prototype that allows a user of the prototype to define synchronisation protocols for the verification of message orders between actors. The prototype extends *AKKA* [66], an existing *actor concurrency* implementation, provides a verification layer on top of it and a domain-specific language to write synchronization rules. Rules can be defined on a per-actor basis and the prototype subsequently makes sure the actor system complies with them and detects unwanted message orders. The thesis consists of three parts: The first one examines concurrency and especially *actor concurrency*, the second one details the proposed prototype and the third one describes performed evaluations and future research.

Concurrency In order to bring the proposed prototype in context, the first part of this thesis examines concurrency mechanisms and theoretical models and relates them to *actor concurrency*. On the lowest abstraction level concurrency means finding semantically valid *partial orders* of instructions without changing a program's meaning. On a higher abstraction level issues can be described more specifically: *Race condition*, *deadlock*, *livelock* and *starvation* are specific situations that arise in concurrent programs. *Synchronisation* and data exchange between concurrent program parts are other aspects of concurrency. Concurrent object-oriented programming paradigms moreover have additional requirements [53].

Actor concurrency is one of various existing mathematical models to describe concurrency. Important other ones are *petri nets* and *process algebras*; the *pi calculus* is especially interesting here, as it was partly motivated by *actor concurrency*. An *actor system* consist of self-contained, autonomous entities called actors [46], which can send and receive messages, create new actors and replace their current behavior [40]. They provide inherent true concurrency, are independent of each other, support dynamic topology changes [40] and have no shared state [8] [6] [46]. Communication is performed asynchronously and interruption-free [4] [46] [61]. One of the central statements of *actor concurrency* is, that no single, unique global clock can be defined for a concurrent system [4] [21] [40]; Actors can agree on the causal relationship between messages, but they can not agree on the total order in which messages occur, because it is impossible to predict when a sent message will arrive [4].

Prototype The second part of this thesis describes and evaluates the proposed prototype. The prototype provides a layer for verification on top of *AKKA* [66], an implementation of *actor concurrency*. It intercepts message commmunication between actors with *AspectJ* aspects [26] and verifies them against rules which the application developer can define. Rules have the form $condition \rightarrow action$. When *condition* is met, the rule fires and *action* influences the overall validation outcome. The verification runs in the actor's own execution context with local verification data, so no external communication is required and the whole verification is a local operation. The actor's behavior gets interrupted until the outcome of the verification is determined and so from the outside verification looks like the actor's own behavior. An actor's synchronization protocol can be adapted at runtime by adding or removing rules to adapt to an actor's behavior changes. Because actors can only have a local understanding of time and because of the *law of locality* [40] only communication performed directly by an actor can be validated, so in all conditions and actions either sender or the receiver has to be the current actor.

Two verification algorithms were implemented for the prototype. Both use the same basic verification logic, but differ in how they log interactions between actors and how they check firing of rules. *History logging verification* keeps a log of interactions for each actor and uses it to check if a rule fires. Because this approach proved to be memory intensive and not performant enough, a second algorithm was implemented which circumvents these issues. *Automaton verification* creates an automaton for each rule. Interactions with the actor are logged by changing the automaton's active state and the active state of an automaton is used to determine if a rule fires.

In order to bring the prototype into context with existing work, other approaches for actor synchronisation in literature were examined. Colaço et al propose a way to detect orphan messages in actor-based languages, which is compareable to the approach taken in this thesis [23] [22]. *Local synchronisation constraints* [46], *RTSynchronizers* [61] *QoS constraints* [62] and *actors with temporal constraints* [47] are other proposed approaches for actor synchronisation.

Evaluation and Future Research Evaluation of the prototype consisted of worst-case runtime determination, tests with real world applications and benchmarking. For both implemented verifiction algorithms the worst-case runtime of the verification itself was determined. For the *automaton verification* also the worst-case runtime of automaton creation and -optimization was determined. To test the prototype under real world conditions, two sample applications were written. The *chameneos application* implements the *chameneos concurrency game* [45], a prototypical example for test-driving concurrent systems. The *token ring application* sends messages around in a circle, actor and loop counts are configurable. The latter application also was used to perform benchmark tests with the prototype. For this, different configurations of the application were run without the prototype, with the *history logging verification* and with the *automaton verification* and the resulting overall runtimes were measured and graphically evaluated.

Proposed future research includes the addition of new, more powerful conditions for rules and wildcard parameters to broaden the expressability of the rules, automatic conversion of plain AKKA programs to the prototype and the evaluation of the prototype in environments distributed over networks.

Bibliography

[1] L. Aceto, K. G. Larsen, and A. Ingolfsdottir, “An Introduction to Milner ’ s CCS,” 2005.

[2] S. V. Adve and M. D. Hill, “Weak Ordering - A New Definition,” *SIGARCH Computer Architecture News*, vol. 18, no. June 1990, pp. 2–14, 1990.

[3] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, “A Foundation for Actor Computation,” *Journal of Functional Programming*, vol. 7, no. January 1997, pp. 1–72, 1997.

[4] G. Agha, *A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1986.

[5] ——, “Concurrent Object-Oriented Programming,” *Communications of the ACM*, vol. 33, no. 9, pp. 125–141, 1990.

[6] G. Agha and C. J. Callsen, “ActorSpace : An Open Distributed Paradigm,” in *PPoPP ’93: Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 1993, pp. 23–32.

[7] G. Agha, S. Frolund, W. Y. Kim, R. Panwar, A. Patterson, and D. Sturman, “Abstraction and Modularity Mechanisms for Concurrent Computing,” in *Research Directions in Concurrent Object-oriented Programming*. Cambridge, MA, USA: MIT Press, 1993, no. May, pp. 3–21.

[8] G. Agha and C. Hewitt, *Concurrent Programming Using Actors: Exploiting Large-Scale Parallelism*. Cambridge, MA, USA: Massachusetts Institute of Technology, 1985.

[9] G. Agha, S. Smith, I. A. Mason, and C. Talcott, “Towards a Theory of Actor Computation,” in *CONCUR ’92: Proceedings of the Third International Conference on Concurrency Theory*. London, UK, UK: Springer-Verlag, 1992, pp. 565–579.

[10] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. [Online]. Available: http://www.amazon.com/Compilers-Principles-Techniques-Tools-2nd/dp/0321486811

[11] G. M. Amdahl, “Validity of the Single-Processor Approach to Achieving Large-Scale Computing Capabilities,” in *AFIPS ’67 (Spring): Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. New York, NY, USA: ACM, 1967, pp. 483—-485.

[12] T. E. Anderson, E. D. Lazowska, and H. M. Levy, “The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors,” *IEEE Transactions on Computers*, vol. 38, no. 12, pp. 1631–1644, 1989. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=40843

[13] B. Awerbuch, “Complexity of Network Synchronization,” *Journal of the ACM*, vol. 32, no. 4, pp. 804–823, 1985.

[14] J. C. M. Baeten and T. Basten, “Partial-Order Process Algebra (and its Relation to Petri Nets),” pp. 1–79, 2001.

[15] A. K. Bansal, *Introduction to Programming Languages*, 1st ed. Chapman & Hall/CRC, 2013. [Online]. Available: http://books.google.at/books?id=531cAgAAQBAJ&pg=PA282

[16] M. Barr, “Mutexes and Semaphores Demystified,” pp. 1–6, 2008. [Online]. Available: http://www.barrgroup.com/Embedded-Systems/How-To/RTOS-Mutex-Semaphore

[17] S. Brookes, “Retracing the Semantics of CSP,” in *CSP’04: Proceedings of the 2004 International Conference on Communicating Sequential Processes: The First 25 Years*. London, UK: Springer-Verlag, Berlin, Heidelberg, 2005, pp. 1–14.

[18] E. Bruneton, “ASM 4.0 - A Java bytecode engineering library.” [Online]. Available: http://download.forge.objectweb.org/asm/asm4-guide.pdf

[19] J. Cheng, “Dependence Analysis of Parallel and Distributed Programs and Its Applications,” in *APDC ’97: Proceedings of the 1997 Advances in Parallel and Distributed Computing Conference*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 370–377. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=574057

[20] R. Cleaveland and S. A. Smolka, “Strategic Directions in Concurrency Research,” *ACM Computing Surveys*, vol. 28, no. 4, pp. 607–625, 1996.

[21] W. D. Clinger, *Foundations of Actor Semantics*. Cambridge, MA, USA: Massachusetts Institute of Technology, 1981.

[22] J.-L. Colaco, M. Pantel, F. Dagnat, and P. Sallé, “Static safety analysis for non-uniform service availability in Actors,” 1998.

[23] J.-L. Colaco, M. Pantel, and P. Sallé, “A Set-Constraint-based analysis of Actors,” no. Proceedings FMOODS ’97, 1997.

[24] R. De Nicola, *A Gentle Introduction to Process Algebras*. Lucca, IT: IMT - Institute for Advanced Studies Lucca, no. ii.

[25] E. W. Dijkstra, “Co-Operating Sequential Processes,” in *Programming Languages: NATO Advanced Study Institute*, F. Genuys, Ed. Academic Press, 1968, pp. 43–112.

[26] Eclipse Foundation, “AspectJ,” 2014. [Online]. Available: http://eclipse.org/aspectj

[27] C. Fidge, *A Comparative Introduction to CSP, CCS and LOTOS*. Queensland, Australia: Software Verification Research Centre, Department of Computer Science The University of Queensland, 1994.

[28] A. Filatova, “Moore’s Law,” p. 3, 2003. [Online]. Available: http://www.xbitlabs.com/articles/cpu/display/moore_2.html

[29] J. Fisher, “Retrospective: very long instruction word archtectures and the ELI-512,” *Solid-State Circuits Magazine, IEEE*, vol. 1, no. 2, pp. 34–36, 2009.

[30] W. Fokkink, “Process Algebra: An Algebraic Theory of Concurrency,” in *CAI ’09: Proceedings of the 3rd International Conference on Algebraic Informatics*. Thessaloniki, Greece: Springer-Verlag Berlin, Heidelberg, 2009, pp. 47–77.

[31] M. Fowler, “FluentInterface,” p. 1, 2005. [Online]. Available: http://martinfowler.com/bliki/FluentInterface.html

[32] ——, *Domain Specific Languages*, 1st ed. Addison-Wesley Professional, 2010, vol. 5658. [Online]. Available: http://www.amazon.com/Domain-Specific-Languages-Addison-Wesley-Signature-Fowler/dp/0321712943

[33] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Boston, MA, USA: Addison-Wesley Longman Publishing, 1995, vol. 47. [Online]. Available: http://www.amazon.co.uk/exec/obidos/ASIN/0201633612/citeulike-21

[34] V. K. Garg, *Concurrent and Distributed Computing in Java*, 1st ed. Hoboken, NJ, USA: John Wiley & Sons, Jan. 2004, vol. 2, no. February 1984.

[35] B. Gloger, *SCRUM: Produkte zuverlässig und schnell entwickeln*, 4th ed. Carl Hanser Verlag GmbH & Co. KG, 2013.

[36] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java Language Specification Java SE 7 Edition*. Redwood City, California: Oracle America, Inc, 2011.

[37] J. L. Gustafson, “Reevaluating Amdahl’s Law,” *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[38] B. P. Hansen, “Structured Multiprogramming,” *Communications of the ACM*, vol. 15, no. 7, pp. 574–578, 1972.

[39] M. Herlihy, “Taking Concurrency Seriously: the Multicore Challenge,” in *IISWC ’07: Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization*. Washington, DC, USA: IEEE Computer Society, Sep. 2007, pp. 2–. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4362175

[40] C. Hewitt and H. G. Baker, “Laws for Communicating Parallel Processes,” in *IFIP-77: International Federation for Information Processing 1977*, no. November, Toronto, Canada, 1977, pp. 987–992.

[41] M. D. Hill and M. R. Marty, “Amdahl’s Law in the Multicore Era,” *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[42] C. A. R. Hoare, “Communicating Sequential Processes,” *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 2004.

[43] Intel, “Moore’s Law Timeline,” p. 1, 2005. [Online]. Available: http://download.intel.com/pressroom/kits/events/moores_law_40th/MLTimeline.pdf

[44] E. H. Jensen, G. W. Hagensen, and J. M. Broughton, “A New Approach to Exclusive Data Access in Shared Memory Multiprocessors,” in *Prepared for Submittal to the 15th Annual International Symposium on Computer Architecture*. Honolulu, Hawaii: UCRL-Preprint, 1987, p. 11.

[45] C. Kaiser and J.-F. Pradat-Peyre, “Chameneos , a Concurrency Game for Java , Ada and Others,” in *AICCSA’03: International Conference on Computer Systems and Applications, 2003. Book of Abstracts.* Tunis, Tunisia: IEEE, 2003, p. 62ff.

[46] R. K. Karmani, A. Shali, and G. Agha, “Actor Frameworks for the JVM Platform : A Comparative Analysis,” in *PPPJ ’09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*. Calgary, Alberta, Canada: ACM New York, NY, USA, 2009, pp. 11–20.

[47] B. Laichi, “ATC: actors with temporal constraints,” in *ISORC ’01: Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*. Washington, DC, USA: IEEE Comput. Soc, 2001, pp. 306–313. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=922854

[48] B. W. Lampson and D. D. Redell, “Experience with Processes and Monitors in Mesa,” *Communications of the ACM*, vol. 23, no. 2, pp. 105–117, 1980.

[49] M. Law, B. Y. J. R. Powell, and M. S. L. Aw, “The Quantum Limits to Moore’s Law,” *Proceedings of the IEEE*, vol. 96, no. 8, pp. 1247–1248, 2008.

[50] M. Llorens and J. Oliver, “Structural and Dynamic Changes in Concurrent Systems: Reconfigurable Petri Nets,” *IEEE Transactions on Computers*, vol. 53, no. 9, pp. 1147–1158, Sep. 2004. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1315608

[51] C. A. Mack, “Fifty Years of Moore’s Law,” *IEEE Transactions on Semiconductor Manufacturing*, vol. 24, no. 2, pp. 202–207, May 2011. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5696765

[52] P. Mackay, "Why has the actor model not succeeded?" in *SURPRISE 97: Surveys and Presentations in Information Systems Engineering*, vol. 2. London, UK: Department of Computing, Imperial College of Science Technology and Medicine, 1997, pp. 2–4.

[53] D. M. Marsh and L. M. Ott, "Distributed Processing: Requirements for an Object-Oriented Approach," *Proceedings of the Thirtieth Hawaii International Conference on System Sciences 1997*, vol. 1, no. January, pp. 73–80, 1997. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=667198

[54] P. E. Mckenney, *Is Parallel Programming Hard, And, If So, What Can You Do About It?* Corvallis, OR, USA: Linux Technology Center BM Beaverton, 2014.

[55] R. Milner, *The Polyadic π-Calculus: A Tutorial.* Edinburgh, UK: Laboratory for Foundations of Computer Science, University of Edinburgh, 1991.

[56] P. J. Mohr, B. N. Taylor, and D. B. Newell, "CODATA recommended values of the fundamental physical constants: 2006," *Rev. Mod. Phys.*, vol. 80, no. 2, pp. 633–730, 2008. [Online]. Available: http://physics.nist.gov/cuu/Constants/archive2006.html

[57] X. Nicollin, "ATP : Une algebre pour la specification et l'analyse des systemes temps reel," Ph.D. dissertation, Institut National Polytechnique de Grenoble, 1992.

[58] J. Niu, *CSc33200-0460 Course: Operating Systems.* Jinzhong Niu, 2003.

[59] D. A. Padua, *Encyclopedia of Parallel Computing*, 1st ed. Springer, 2011. [Online]. Available: http://link.springer.com/book/10.1007/978-0-387-09766-4

[60] J. Parrow, "An Introduction to the Pi Calculus," in *Handbook of Process Algebra*, J. A. Bergstra, A. Ponse, and S. A. Smolka, Eds. New York, NY, USA: Elsevier Science Inc, 2001.

[61] S. Ren and G. A. Agha, "RTsynchronizer : Language Support for Real-Time Specifications in Distributed Systems," in *LCT-RTS 1995: Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, & Tools for Real-Time Systems*, R. Gerber and T. J. Marlowe, Eds., no. June, 1995, pp. 50–59.

[62] S. Ren, N. Venkatasubramanian, and G. Agha, "Formalizing Multimedia QoS Constraints Using Actors," in *IFIP 1997: Proceedings of the Second IFIP International Conference on Formal Methods for Open, Object-Based Distributed Systems*, no. Mm. Canterbury, UK: Chapman & Hall, 1997, pp. 139–153.

[63] L. Ribeiro Korff and M. Korff, "True Concurrency = Interleaving Concurrency + Weak Conflict," *Electronic Notes in Theoretical Computer Science*, vol. 14, pp. 204–213, 1998. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S1571066105802373

[64] E. Rieffel and W. Polak, "An Introduction to Quantum Computing for Non-Physicists," *ACM Computing Surveys*, vol. 32, no. 3, pp. 300–335, 2000.

[65] J. H. Saltzer and M. D. Schroeder, “The Protection of Information in Computer Systems,” *Proceedings of the IEEE*, vol. 63, pp. 1278–1308, 1975.

[66] Typesafe Inc., “Akka 2.2.3. Java Documentation,” p. 1, 2013. [Online]. Available: http://doc.akka.io/docs/akka/2.2.3/java.html

[67] R. van Glabbeek and F. Vaandrager, “Petri Net Models for Algebraic Theories of Concurrency,” in *PARLE Parallel Architectures and Languages Europe*. Amsterdam: Springer Berlin Heidelberg, 1987, pp. 224–242.

[68] J. Wang, “Petri Nets for Dynamic Event-Driven System Modeling,” in *Handbook of Dynamic System Modeling*, P. A. Fishwick, Ed. Boka Raton, FL: Chapman & Hall, 2007, no. 4, pp. 24–1–24–16.

[69] S. Yovine, *Méthodes et outils pour la vérification symbolique de systèmes temporisés*. Grenoble, FR: Institut National Polytechnique de Grenoble, 1993.

[70] J. Zahorjan, E. D. Lazowska, and D. L. Eager, *Spinning Versus Blocking in Parallel Systems with Uncertainty*. University of Washington and University of Sasatchewan, 1988.