ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/266461870

# Security Analysis in Cloud Computing using Rewriting Logic

**Article**

CITATIONS
2

READS
27

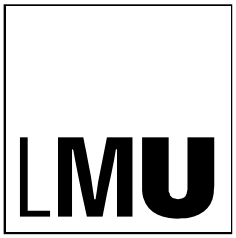
**1 author:**

Jonas Eckhardt
Technische Universität München
**22** PUBLICATIONS **94** CITATIONS

SEE PROFILE

All content following this page was uploaded by Jonas Eckhardt on 12 May 2016.

The user has requested enhancement of the downloaded file.





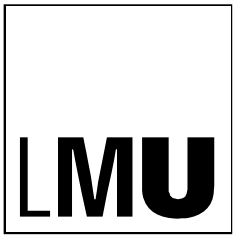
**Institut für Informatik**
Lehr- und Forschungseinheit für
Programmierung und Softwaretechnik
Oettingenstraße 67 D-80538 München

# Security Analysis in Cloud Computing using Rewriting Logic

Jonas Eckhardt

**Masterarbeit im Elitestudiengang Software Engineering**

Software Engineering
Elite Graduate Program






**Institut für Informatik**
Lehr- und Forschungseinheit für
Programmierung und Softwaretechnik
Oettingenstraße 67 D-80538 München

# Security Analysis in Cloud Computing using Rewriting Logic

| | |
|---|---|
| Matrikelnummer: | 983270 |
| Beginn der Arbeit: | 4. August 2011 |
| Abgabe der Arbeit: | 24. Januar 2012 |
| Erstgutachter: | Prof. Dr. Martin Wirsing |
| Zweitgutachter: | Prof. Dr. Alexander Knapp |
| Betreuer: | Prof. José Meseguer |

Software Engineering
Elite Graduate Program

## ERKLÄRUNG

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, den 23. Januar 2012 Jonas Eckhardt

*Für Petra und Martin*

# Acknowledgements

I would like to thank many people for their support, friendship, and encouragement indispensable for the successful completion of my thesis. Though words cannot express my gratitude, it is a pleasure for me to thank those who made this thesis possible.

I owe my deepest gratitude to Tobias Mühlbauer. Tobias, thank you for being a good friend in the last two years and I honestly hope that there are many further joint years to come. Thank you for the inspiring scientific as well as political discussions. Thank you for being patient with me.

My sincere thanks go to Martin Wirsing who initiated the thesis project and gave me the unique opportunity to participate in cutting-edge research. I also would like to say thanks to Alexander Knapp who always found the time to answer my questions and who gave advice during my entire master's studies. I would like to show my gratitude to José Meseguer who offered me a warm welcome in his group, integrated me in his research, and taught me the true meaning of being a scientist; it was an honor to work with you.

My sincere thanks go to many friends and colleagues for scientific discussions, advices, and continuous support. Among them, particularly Musab AlTurki, who introduced me to the secrets of statistical model-checking with PVeSta, and Michael Katelman, who taught me the relationship between Functional Verification and Puppies. Special thanks go to Rakesh Bobba and Jashua Gupta: thank you for our inspiring discussions about the group-key management approach using ZooKeeper. I would also like to thank Nana Arizumi, Alexandre Duchâteau, Santiago Escobar, Raúl Gutiérrez, Fredrik Kjølstad, and Ralf Sasse for their endorsement and friendship during my stay in Urbana-Champaign.

Finally, and the most personal for me, I would like to thank my parents and family. The following words are dedicated to you in German:

> *Petra und Martin, Danke für eure immerwährende Unterstützung, Danke für eure Liebe, Danke dafür, dass ihr mir ein Halt auf meinem Weg seid. Besonderer Dank gilt außerdem meinem „großen“ Bruder Lukas und meiner „kleinen“ Cousine Anna, die beide immer wieder einen sechsten Sinn erwiesen haben und mich mit ihren Ratschlägen stets wieder auf den rechten Pfad geführt haben — nicht unbedingt wissenschaftlich, aber im Alltag unabdingbar.*

This research is partially supported by NSF Grant CCF 09-05584, AFOSR Grant FA8750-11-2-0084, the ASCENS project (FP7-257414), the NESSoS project (FP7-256980), the PROSA$^{\text{LMU}}$ scholarship for research stays abroad, the Deutschland Stipendium scholarship, and the Software Engineering Elite Graduate Program.

# Abstract

Cloud Computing is an emerging topic with high potentials in the IT-industry. Services that are offered in the Cloud are often safety- and security-critical services which need to dynamically adapt to changes in in highly unpredictable and potentially hostile environments, such as the Internet. These aspects increase the complexity of designing secure and safe services for the Cloud and this raises the question how to analyze the desired security and safety of those services. This thesis tries to tackle these two issues by proposing two new approaches for modelling and analyzing secure and safe Cloud-based services based on the rigorous formal semantics of Rewriting Logic and (statistical) model-checking. Furthermore, we demonstrate the usefulness of the proposed approaches in multiple case-studies.

# Contents

**1. Security Analysis in Cloud Computing using Rewriting Logic** **1**
1.1. Objectives of the Thesis . . . 2
1.2. Outline of the Thesis . . . 3

**2. Prerequisites on Cloud Computing and Rewriting Logic** **5**
2.1. Cloud Computing — an Introduction . . . 5
2.2. Security Challenges in Cloud Computing . . . 8
2.3. Rewriting Logic and Maude . . . 8

**3. A Formal Language for the Design and Analysis of Cloud-based Architectures** **11**
3.1. Introduction to Coordination Languages . . . 12
3.1.1. Linda . . . 12
3.1.2. Advantages of Coordination Languages . . . 14
3.2. KLAIM . . . 14
3.2.1. Overview of KLAIM . . . 14
3.3. M-KLAIM — a Maude-based specification of KLAIM . . . 20
3.3.1. Overview . . . 20
3.3.2. Description of modules . . . 21
3.4. Application of the CINNI calculus . . . 36
3.4.1. Implementation of $\text{CINNI}_{\text{KLAIM}}$ . . . 38
3.5. OO-KLAIM — an extension of M-KLAIM for object-oriented specifications 43
3.5.1. Object-based programming in Maude . . . 44
3.5.2. OO-KLAIM syntax . . . 44
3.5.3. OO-KLAIM semantics . . . 45
3.6. D-KLAIM — an extension of OO-KLAIM for distributed specifications . . . 47
3.6.1. Rewriting with external objects in Maude . . . 48
3.6.2. D-KLAIM specification overview . . . 48
3.6.3. D-KLAIM modules . . . 48
3.6.4. The socket interface . . . 54
3.6.5. Example of a Cloud-based architecture specification based on D-KLAIM 60
3.7. Maude-based formal analysis of *-KLAIM . . . 62
3.7.1. Maude LTL model checking . . . 62
3.7.2. A *-KLAIM-based token-based mutual exclusion algorithm . . . 62

3.7.3. Model checking using the Maude search command . . . . . 66
3.7.4. A D-KLAIM-based load balancer . . . . . 67

**4. A Modularized Actor Model for Statistical Model Checking 71**
4.1. Introduction to the *Actor Model of Computation* . . . . . 72
4.1.1. A Maude-based Specification of the *Actor Model* . . . . . 74
4.2. Introduction to Statistical Model Checking . . . . . 76
4.2.1. Probabilistic Rewrite Theories . . . . . 76
4.2.2. Maude specification of Actor PMaude . . . . . 78
4.2.3. Statistical Analysis using the PVeStA model checker . . . . . 82
4.3. Introduction to the *Reflective Russian Dolls* Model . . . . . 84
4.4. The Modularized Actor Model . . . . . 86
4.4.1. The Hierarchical Addressing Scheme . . . . . 86
4.4.2. The Actor Model and the Name Generator . . . . . 87
4.5. Multi-level scheduling for the Modularized Actor Model . . . . . 90
4.5.1. The Absence of unquantified non-determinism . . . . . 95
4.6. Using PVeStA to Statistically Analyze Specifications based on the Modularized Actor Model . . . . . 96
4.6.1. The module *APMAUDE* . . . . . 96
4.6.2. Running PVeStA . . . . . 96

**5. Guaranteeing High Availability under Distributed Denial of Service Attacks 99**
5.1. Introduction to Denial of Service Attacks . . . . . 100
5.2. The ASV Protocol . . . . . 101
5.3. Maude-based Analysis of the ASV Protocol . . . . . 102
5.3.1. Description of the ASV specification in Maude . . . . . 103
5.3.2. Statistical Model Checking Results . . . . . 114
5.4. ASV$^{++}$ — a 2-Dimensional Protection Mechanism against DDoS Attacks . 115
5.4.1. The Server Replicator meta-object and the ASV$^{++}$ protocol . . . . . 116
5.4.2. Description of the ASV$^{++}$ specification in Maude . . . . . 118
5.4.3. Statistical Model Checking Results . . . . . 123
5.5. Conclusion . . . . . 126

**6. Specification and Analysis of a Group-Key Management System using the ZooKeeper Service 127**
6.1. The ZooKeeper Service . . . . . 127
6.2. Group-Key Management using ZooKeeper . . . . . 129
6.2.1. Abstraction of a Group-Key Management System . . . . . 129
6.2.2. Using ZooKeeper's Notification Mechanism for Key Distribution . . 131
6.3. Maude specification of the Group-Key Management on top of ZooKeeper . . 133
6.3.1. Maude specification of the ZooKeeper Service . . . . . 133
6.3.2. Maude specification of the Group-Key Management Service . . . . . 157
6.4. Statistical Analysis of the Group-Key Management Service . . . . . 164
6.4.1. QuaTEx Formulas . . . . . 165
6.4.2. Discussion and Analysis of the Results . . . . . 166

6.5. Related Work . . . . . 168
6.6. Conclusion . . . . . 168

**7. Conclusion & Future Work** **169**
7.1. Conclusion . . . . . 169
7.2. Future Work . . . . . 170

**A. KLAIM Appendix** **173**

**B. Additional Maude Specifications** **177**
B.1. The *SAMPLER* module . . . . . 177
B.2. Maude Specification of a Generic Actor Generator . . . . . 180

**Bibliography** **183**

Chapter 1

# Security Analysis in Cloud Computing using Rewriting Logic

Cloud Computing is an emerging topic with high potentials in the IT-industry. Resources are provided as a service over a large network, such as the Internet, to customers on a use-by-need and pay-per-use basis. Since computing services are hosted in huge data centers and accessible almost everywhere, the Cloud becomes a single point of access for the customer's computation and data storage needs. In 2010, BBC published a news report stating that the number of Google Mail users was at that time around 170 Million users [26]. This is not the exception. Cloud computing offers innovative software developers and companies the opportunity to deploy new services without having to invest heavily in hardware and in maintenance personnel in advance. Additionally, the application developer does not need to bother about over- or under-provisioning since the Cloud automatically performs the scaling and the developer only pay for what is actually needed.

This new paradigm not only prompts questions about traditional security properties like the security and privacy of information stored in the Cloud, but also raises questions about performance-based **Q**ality **o**f **S**ervice (QoS) properties like the availability of a service in the Cloud.

Cloud providers like Google, Amazon, and Microsoft guarantee high percentage of availability in their **S**ervice **L**evel **A**greements (SLAs). For instance, Amazon guarantees 99.95% availability of the service within a region over a trailing 365 day period for EC2 [21], Microsoft provides a 99.9% uptime Service Level Agreement for Exchange Online, SharePoint Online, Office Live Meeting, and Office Communications Online [63], and Google guarantees the Google Apps Covered Services web interface to be operational and available to customer at least 99.9% of the time in any calendar month [40].

Cloud-based services are running in potentially hostile and unpredictable environments. For example, the Cloud-based file storage service Dropbox reported that "Yesterday we made

a code update at 1:54pm Pacific time that introduced a bug affecting our authentication mechanism. We discovered this at 5:41pm and a fix was live at 5:46pm." [22] on June 20, 2011. During these nearly four hours, the broken authentication mechanism granted access to possibly private data stored on some accounts using any chosen password. Issues like this are not the exception as one can see in the the Gartner report "Assessing the Security Risks of Cloud Computing" [46] where several security risks have been identified that are important for Cloud Computing: (i) data segregation, (ii) data privacy, (iii) privileged user access, (iv) service provider viability, (v) availability and recovery.

Zhang et al. make the following observation about Cloud Computing in [84]:

> *However, despite the fact that Cloud computing offers huge opportunities to the IT industry, the development of Cloud computing technology is currently at its infancy, with many issues still to be addressed.*
>
> — Cloud Computing: state-of-the-art and research challenges

The above quote points out two important observations on Cloud Computing. On the one hand, Cloud Computing is an very profitable topic to the IT industry, and on the other hand, Cloud Computing is still new and not much is known about it. Thus, there are many issues that still need to be addressed.

## 1.1. Objectives of the Thesis

The main goal of this thesis is to tackle the complexity that emerges for the design and analysis of secure and safe Cloud-based services. We aim to find an general approach that bases on a rigorous formal basis in which (i) models of Cloud-based services can be specified, (ii) security and safety properties of such services can be expressed, and (iii) quantitative as well as qualitative analyses of such properties can be performed. In addition to that, we want to conduct case studies on Cloud-based systems with different sizes, e.g., different number of participants and interactions, to evaluate the usefulness and expressiveness of the approach.

Therefore, we present two new approaches: First, we present an approach that is based on the coordination language KLAIM with which one can specify a model of, and model-check specific temporal logic formulas on Cloud-based services. On a high level, we see *Cloud Computing* as the distribution of tasks and data across multiple computing sites, which are often referred to as nodes. A related issue is the communication and coordination between participating nodes to achieve quality of service properties for the software and services running in the network. Using this formal language, we model and analyse various simple Cloud Computing systems. We show that this approach provides a flexible way of specifying Cloud-based systems. The use of model-checking algorithms provides the exact verification of the satisfaction of temporal logic properties. However, due to the state-explosion problem of model checking, the size of the specified systems are limited. This limitation is indefensible for Cloud-based systems.

The second approach uses the standard *actor model of computation* as foundation for the specification and statistical model-checking of quantitative properties in Cloud-based systems. In contrast to exact model-checking algorithms, we use statistical model-checking, which allows for the analysis of quantitative properties on large system. As the absence of

un-quantified non-determinism is required to perform statistical model checking, we provide the description of a multi-level scheduling approach for the modularized actor model which fulfills this requirement. In contrast to the first approach, in which only small systems could be analyzed, by using this approach, large systems with a multitude of participants and interactions can be analyzed. We show the usefulness of this approach by means of two case studies.

## 1.2. Outline of the Thesis

We give an short introduction to Cloud Computing, Rewriting Logic, and the Maude system in Chapter 2.3. In Chapter 3, we develop a formal language in which *Cloud Computing* architectures can be specified and analyzed. Additionally, various Cloud-based systems are specified and analyzed, but due to the state-explosion problem of model checking, the size of the specified systems are limited. In order to specify modular and adaptive systems in an unified way, we developed in Chapter 4 a modularization of the actor model that can be used with statistical model checking. In Chapter 5, we present solutions to how Cloud-based services can be made resilient to **D**istributed **D**enial **o**f **S**ervice (DDoS) attacks with minimum performance degradation. We specify and statistically analyse thereby a modular and adaptive defense mechanism against DDoS attacks. Chapter 6 shows an additional application of the modularized actor model an a group-key management system. Finally, the thesis is concluded and further work is presented in Chapter 7.

# Chapter 2

# Prerequisites on Cloud Computing and Rewriting Logic

## 2.1. Cloud Computing — an Introduction

The idea of Cloud Computing — computing as a utility — is not new and there exist a multitude of definitions for the term *Cloud Computing*. The **N**ational **I**nstitute of **S**tandards and **T**echnology (NIST), the measurement standards laboratory of the United States of America[1], define the term in [50] as follows:

> *"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."*
>
> — NIST definition of Cloud Computing

Furthermore, they provide five essential characteristics for their model of Cloud Computing, that are described in the following.

**On demand self-service.** Consumers can provision computing resources as needed without any human interaction with the service's provider.

**Broad network access.** Applications and resources are available over the network and accessible through standard mechanisms.

[1] The NIST is is a non-regulatory agency of the United States Department of Commerce. See: `www.nist.gov`.

**Resource pooling.** Provider pool their computing resources to serve multiple consumers using dynamic and virtual resource assignment. Generally, the consumer has no control over the exact location of the provided resources.

**Rapid elasticity.** Resources can be rapidly and elastically provisioned to provide quick scalability (scale out as well as scale in). This results in the illusion of unlimited scalability of the Cloud.

**Measured Service.** The Cloud automatically controls and optimizes resources usage.

In order to achieve a complete definition of the term Cloud Computing, Vaquero et al. discuss the concept in [79] and result in the following definition:

> *Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs.*
>
> — Proposed Definition (Vaquero et al.)

Furthermore, Vaquero et al. propose a minimum set of features that is common to most Cloud definitions:

**Scalability:** Scalability allows a service to serve an increasing amount of users without noticeable performance loss. Most Cloud systems provide this feature by dynamically allocating virtualized resources adapting to the service's workload.

**Pay-per-use utility model:** A pay-per-use utility model provides the ability to pay for the use of resources on a short-term basis as they are used. Instead of providing a huge data center, the provider can use the Cloud and pay only for the resources that are actually needed to provide the service. Thus, service provides no longer need to invest heavily and encounter difficulties in building and maintaining huge data centers.

**Virtualization:** In order to abstract from specific hardware and to provide an uniform interface for services, Clouds offer virtualized resources, e.g., configurable **V**irtual **M**achines (VMs), that can be used to run the service on.

On an architectural point of view, Cloud Computing can be divided into three layers: The application layer, the platform layer, and the infrastructure and hardware layer. Figure 2.1 illustrates these three layers. In the following, we discuss the architectural layers in more detail.

**Infrastructure & hardware layer.** The infrastructure and hardware layer is responsible for managing the physical resources and providing storage and computing resources by partitioning the physical resources using virtualization technologies such as Xen [14], KVM [7] and VMWare [12]. The infrastructure and hardware layer is hosted in big data centers.

**Platform layer.** The platform layer consists of operating systems and application frameworks that promise to minimize the work to deploy applications. For example, the Google App Engine [5] and the Windows Azure Platform [13] operate at the platform level and provide API support for databases and business logic.

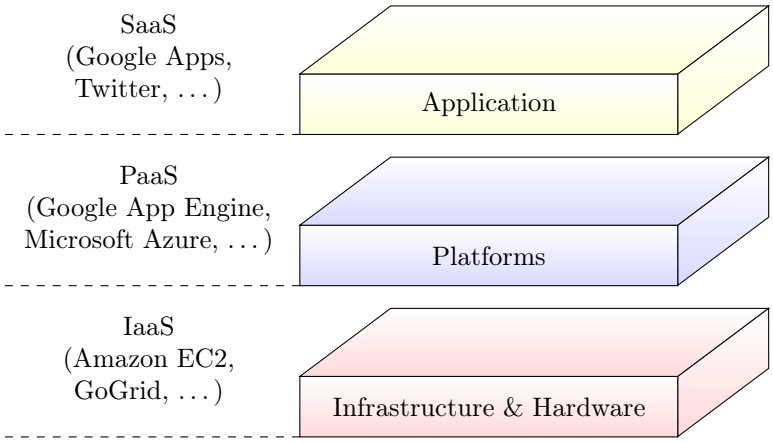



**Figure 2.1.:** Architectural layers of a Cloud Computing environment.

**Application layer.** The application layer consists of the actual applications hosted in the Cloud. These applications leverage the scalability of the Cloud to provide better *Service Level Agreements (SLAs)* such as, for example, performance or availability. The services provides by Google Apps for business [6] and Twitter [11] are well known examples of applications hosted in the Cloud.

Each of the architectural layers is loosely coupled with the layers above and below, which provides more modularization in comparison with traditional service hosting environments.

On a business point of view, Cloud Computing employs a service-driven business model, i.e., resources are provided as services on a pay-per-use basis. These services can be grouped into three categories: **S**oftware **a**s **a** **S**ervice (SaaS), **P**latform **a**s **a** **S**ervice (PaaS), **I**nfrastructure **a**s **a** **S**ervice (IaaS).

**Infrastructure as a Service.** IaaS provides infrastructural resources on demand, usually VMs. Examples of IaaS providers are Amazon EC2 [1], GoGrid [4] and Flexiscale [2].

**Platform as a Service.** PaaS provides platform layer resources such as operating system support and software development frameworks. PaaS providers are for example Google App Engine [5], Windows Azure Platform [13] and Force.com [3].

**Software as a Service.** SaaS provides on-demand applications, hosted in the Cloud. Examples of SaaS providers include Salesforce.com [9], Rackspace [8], and SAP Business By Design [10].

Besides the architectural and business point of view on Cloud Computing, the visibility of data plays an important role in Cloud Computing. Concerning the visibility of data, Cloud Computing can be characterized in Public Clouds, Private Clouds, Hybrid Clouds, and Virtual Private Clouds.

**Public Clouds.** Public Clouds offer their services to the general public. Since Public Clouds are open for the general public, they lack fine-grained control over the comprised data and network and security settings.

**Private Clouds.** Private Clouds (aka. Internal Clouds) are designed to be used exclusively by a single organization and provide a high level of control over performance, reliability and security. Private Clouds can be managed by an external provider or the organization itself.

**Hybrid Clouds.** Hybrid Clouds combine Public and Private Cloud models, trying to address the limitations of each approach. The service infrastructure is divided in parts that run in a Private Cloud and parts that run in a Public Cloud.

**Virtual Private Clouds.** Similar to Hybrid Clouds, **V**irtual **P**rivate **C**louds (VPCs) combine Public and Private Cloud models. In contrast to Hybrid Clouds, the whole service infrastructure is hosted in a Public Cloud, but VPCs leverage **V**irtual **P**rivate **N**etwork (VPN) technology to design a customized topology and security settings.

## 2.2. Security Challenges in Cloud Computing

The new paradigm of Cloud Computing prompts questions about traditional security properties like the security and privacy of information stored in the Cloud. Not only this, application that are hosted in the Cloud are accessible from almost everywhere and are running in potentially hostile and unpredictable environments. As for example, in December 2010, the Mastercard website was attack by an **D**enial **o**f **S**ervice (DoS) attack and as a result unavailable for most customers [56]. This raises questions about performance-based **Q**ality **o**f **S**ervice (QoS) properties like the availability of a service in the Cloud.

Cloud providers like Google, Amazon, and Microsoft guarantee high percentage of availability in their **S**ervice **L**evel **A**greements (SLAs). For instance, Amazon guarantees 99.95% availability of the service within a region over a trailing 365 day period for EC2 [21], Microsoft provides a 99.9% uptime Service Level Agreement for Exchange Online, SharePoint Online, Office Live Meeting, and Office Communications Online [63], and Google guarantees the Google Apps Covered Services web interface to be operational and available to Customer at least 99.9% of the time in any calendar month [40].

The Gartner report "Assessing the Security Risks of Cloud Computing" [46] identifies several security risks that are important for Cloud Computing: (i) data segregation, (ii) data privacy, (iii) privileged user access, (iv) service provider viability, (v) availability and recovery. In this work, we try to help tackling these problems by using an approach that generates safe- and secure-by-construction Cloud-based systems.

## 2.3. Rewriting Logic and Maude

Rewriting logic [59] is a very simple computational logic to specify concurrent systems as *rewrite theories*, that is, as triples $(\Sigma, E, R)$, where $(\Sigma, E)$ is an *equational theory* with syntax and type structure specified by the signature $\Sigma$, and with (possibly conditional) $\Sigma$-equations $E$; and where $R$ is a set of (possibly conditional) *rewrite rules* of the form

$$t \rightarrow t' \; \textit{if} \; \textit{cond},$$

with $t, t'$ $\Sigma$-terms, and *cond* the rule's condition. The way $(\Sigma, E, R)$ models a concurrent system is as follows:

1. the *states* of the system are modeled as elements of the initial algebra (algebraic data type) $T_{\Sigma/E}$ associated to the equational theory $(\Sigma, E)$, and

2. the *local atomic transitions* of the concurrent system are parametrically modeled by the rewrite rules in $R$; that is, a rewrite rule

$$t \to t' \; \textit{if} \; \textit{cond}$$

specifies that if a state fragment is a substitution instance of the pattern $t$ and satisfies condition *cond*, then that system fragment can perform a local transition to a new state which is the corresponding substitution instance of the pattern $t'$.

Many such transitions can happen *concurrently* in the system; rewriting logic models *all* the concurrent transitions possible in the system [59].

The Maude system [34] executes rewrite theories under these assumptions, with a self-explanatory typewriter syntax almost isomorphic to the mathematical syntax. The key concept in Maude is that of a module. A module can be a *functional module*, an equational theory, and a *system module*, a rewrite theory. Both functional and system modules $M[X :: P]$ can be *parameterized* by a parameter theory $P$, and can be instantiated by theory interpretations $V : P \longrightarrow Q$ to instances $M[V]$ called *views*, with the usual pushout semantics (see [34]).

Rewriting logic can naturally model concurrent systems, which can be both *real-time* and *probabilistic*. We crucially exploit these two features in this paper. Real-Time systems are supported by rewrite theories $(\Sigma, E \cup A, R)$ whose underlying equational theory $(\Sigma, E \cup A)$ includes among its types an algebraic data type *Time* representing time instants (which may be either discrete or continuous), and whose global states are pairs of the form $(t, r)$, with $t$ a term representing a "discrete" state, and $r$ a time value of sort *Time* representing the global clock. The rewrite rules in $R$ can then be either *instantaneous* rules, which do not change the global clock, or *tick* rules, which advance the global time (see [69]). Probabilistic concurrent systems, which may also be real-time systems, are modeled by *probabilistic rewrite rules* of the form

$$l : t(\vec{x}) \to t'(\vec{x}, \vec{y}) \;\; \textit{if} \;\; \textit{cond}(\vec{x}) \;\; \textit{with probability} \;\; \vec{y} := \pi_l(\vec{x})$$

where the righthand side term $t'$ has new variables $\vec{y}$ disjoint from the variables $\vec{x}$ appearing in $t$ which make the application of the rule non-deterministic. The probabilistic nature of the rule is expressed by the probability distribution $\pi_l(\vec{x})$ with which values for the extra variables $\vec{y}$ are chosen; where $\pi_l(\vec{x})$ is in general not fixed, but parametric on the righthand side variables $\vec{x}$ (see [34]).

The probabilistic real-time distributed systems we consider are *object-based* systems, made up of objects that communicate with each other by asynchronous message passing. We briefly explain how such systems are formally specified in rewriting logic. Each object belongs to a certain class $K$ and has a unique name, say $o$, identifying it. Furthermore, its state is a record structure of the form $a_1 : v_1, \ldots, a_n : v_n$, with $a_1 \ldots, a_n$ the object's *attributes* (state variables), and $v_1 \ldots, v_n$ the corresponding values currently stored in those attributes. Therefore, an object in a given state can be represented as a term of the form

$$\langle o : K \mid a_1 : v_1, \ldots, a_n : v_n \rangle$$

All objects in a system are then terms of sort *Object*. A *message* addressed to object $o$ with contents $d$ can be represented as a term $(o \leftarrow d)$; and all messages in a system are then terms of sort *Message*. The *distributed state* of such an object-based system is a *multiset* or "soup" of objects and messages, called a *configuration*. Mathematically, this is specified by declaring a sort *Configuration* with subsort inclusions $Object, Message < Configuration$, and an associative and commutative multiset union operator with empty syntax: $\_\ \_ : Configuration\ Configuration \longrightarrow Configuration$ and with identity element *null*. The dynamic behavior of a distributed object-based system can then be specified by rewrite rules that describe how an object behaves upon receiving a certain type of message. In their simplest, Actor-like form, they are rules of the form

$$(o \leftarrow d)\ \langle o : K \mid a_1 : v_1, \ldots, a_n : v_n \rangle \ \rightarrow \\ \langle o : K \mid a_1 : v'_1, \ldots, a_n : v'_n \rangle\ (o_1 \leftarrow d_1) \ldots (o_n \leftarrow d_n)$$

That is, upon receiving message $(o \leftarrow d)$ object $o$ can change its state, and can send several messages to other objects. Although not indicated in the rule above, the righthand side may also include new objects, so that dynamic object creation is also supported.

Chapter 3

# A Formal Language for the Design and Analysis of Cloud-based Architectures

In this chapter, our goal is to develop a formal language in which *Cloud Computing* architectures can be specified and analyzed. At a high level, *Cloud Computing* is the distribution of tasks and data across multiple computing sites, which are often referred to as nodes. A related issue is the communication and coordination between participating nodes to achieve various service properties for the software and services running in the network. However, questions regarding the architectural design and the satisfaction of service properties (e.g. security and liveness properties) of such systems arise. In the following, we

1. give an introduction to coordination languages (Section 3.1),
2. introduce the KLAIM language specification (Section 3.2),
3. develop M-KLAIM, a Maude-based formal executable specification of the KLAIM coordination language (Section 3.3),
4. extend M-KLAIM to OO-KLAIM for object-oriented specifications (Section 3.5),
5. extend OO-KLAIM to D-KLAIM for distributed object-oriented specifications (Section 3.6),
6. and lastly show how specifications based on the aforementioned languages can be formally analyzed (Section 3.7).

We will show that the definition of the formal languages based on KLAIM provide a way to specify *Cloud Computing* architectures, and that these specifications are not only executable, but also analyzable using model checking.

## 3.1. Introduction to Coordination Languages

According to the article "Coordination Languages and their Significance" [39], where the term *coordination language* was first mentioned, the term was created "to designate the linguistic embodiment of a coordination model" and to identify such languages as members of a class of complete coordination languages "in their own rights, not mere extensions to some host language". Part of the proposal is to separate the concerns of *computation* and *communication* (also coordination) into two different models. The computation model is used to express computational activities, whereas the coordination model provides operations to create computational activities and to support communication and synchronization among them. Both models can be integrated into a single language, or they can be separated into two distinct languages. Gelenter et al., the originators of the Linda coordination language, prefer the second alternative, where one chooses specialized languages for the different concerns of computation and coordination.

### 3.1.1. Linda

Linda is a coordination language developed by Gelernter et al. at Yale University [39, 38] and is an independent coordination model that can be added to any base language with no change to the base language semantics. Communication in Linda relies on an asynchronous and associative mechanism which is based on a global environment that can be compared to a logically shared object memory. This environment is called a *tuple space*, which itself represents a bag (multiset) of tuples, i.e., a bag of ordered sequences of typed data items. All Linda processes have access to the tuple space and are able to generate tuple-structured data objects and read tuples from the tuple space. Tuples are selected by processes using associative pattern-matching, where two tuples match if they have the same number of fields and the corresponding fields match. Fields are either values or variables, where two values match if they are equal and variables match any value of the same type. At a high level, tuple selection can be compared to a query on a relational database. Linda offers four primitives for the interaction of processes with the tuple space:

| | |
|---|---|
| $\mathbf{out}(t)$ | writes a tuple into the tuple space |
| $\mathbf{eval}(t)$ | dynamically creates a process that evaluates $t$ and writes the result into the tuple space |
| $\mathbf{in}(t)$ | evaluates $t$ and, if existent, consumes, i.e., reads and removes, a matching tuple $t'$ from the tuple space |
| $\mathbf{rd}(t)$ | evaluates $t$ and, if existent, reads a matching tuple $t'$ while keeping $t'$ in the tuple space |

$\mathbf{out}(t)$ and $\mathbf{eval}(t)$ are non-blocking operations, whereas $\mathbf{in}(t)$ and $\mathbf{rd}(t)$ block until a matching tuple $t'$ for the evaluated tuple $t$ is available in the tuple space. When several tuples match the evaluated tuple, one of the matching tuples is selected non-deterministically. Furthermore, if several $\mathbf{rd}(t)$ or $\mathbf{in}(t)$ operations are waiting for the same evaluated tuple, the operation that is selected to proceed is chosen non-deterministically. The original Linda language proposal further introduces two non-blocking predicative operations, namely $\mathbf{rdp}(t)$

and **inp**($t$), which evaluate to false if no tuple $t'$ that matches the evaluation of $t$ is present in the tuple space. If a matching tuple $t'$ is found, the tuple is read (**rdp**($t$)) or consumed (**inp**($t$)). The Linda model is a simple, yet powerful abstraction of communication and allows processes to use the primitives for data manipulation and synchronization alike. For example, one-to-one, broadcast, many-to-one, and other communication patterns can be created by a combination of the Linda primitives. Implementations of the Linda model exist for various programming languages including Java [83] and C++ [37].

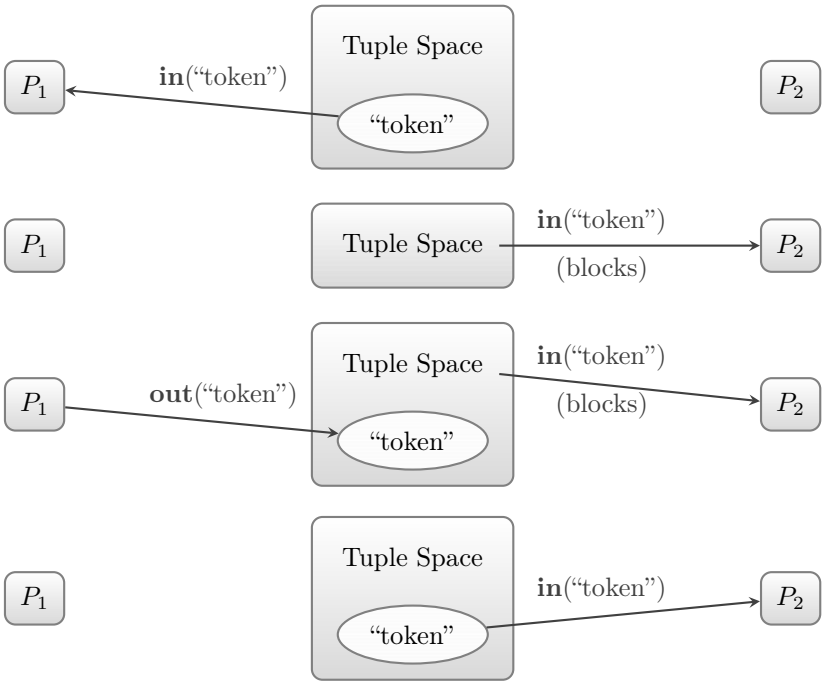



**Figure 3.1.:** Example of process synchronization using the Linda model

**Example 3.1: Synchronization using the Linda model**
Figure 3.1 gives an example of how two processes $P_1$ and $P_2$ can be synchronized using the Linda coordination model. A tuple that consists only of the String field "token" is already stored in the tuple space. It is removed by $P_1$ using the **in**("token") operation. While $P_1$ holds the "token"-tuple, process $P_2$ tries to acquire the token by performing the **in**("token") operation. For our example, we assume that only one such "token"-tuple exists. As such, the operation called by $P_2$ on the tuple space blocks. Only after $P_1$ writes the "token"-tuple back into the tuple space using the **out**("token") operation, the blocked **in**("token") operation of $P_2$ unblocks and is able to acquire the tuple from the tuple space. In summary, the "token"-tuple in our example acts as a mutex (binary semaphore) [36], where **in**("token") corresponds to the P-operation and **out**("token") corresponds to the V-operation on general semaphores.

### 3.1.2. Advantages of Coordination Languages

According to [39], the strengths of Linda and coordination languages in general arise from two generic principles – *separation* and *generality*:

**Separation:** Separation of computation and coordination favors *portability* and support for *heterogeneity*. Portability means that acquired knowledge, tools, and implementations that deal with coordination can be reused when moving from one platform, language, or parallelism model to another. Having two separate languages for the concerns of computation and coordination allows the programmer to switch the base language without giving up the coordination model. The support for heterogeneity is a generalization of portability, because a single coordination model may be used to combine systems built on different base languages.

**Generality:** The advantages of having a general-purpose coordination language are *economy*, *flexibility*, and *intellectual focus*. Generality expresses a language's ability to be used for all kinds of concurrent applications "from multi-threaded applications executing on a single processor, through tightly-coupled, fine-grained parallel processing applications, to loosely-coupled, coarse-grained distributed applications" [82]. It favors simplicity (conceptual economy) and enables the possibility of focusing on a general model for several related problems. In terms of flexibility, a coordination language such as Linda is able to logically express different forms of communication, including message passing, shared memory, or RPC.

The advantages of coordination languages address many properties of cloud computing architectures. Cloud computing often operates upon a highly heterogeneous system were different computing sites collaborate to achieve common goals. Applications in such an environment may be built upon different base languages and run on different platforms. Using a coordination language as a foundation for our models of cloud computing architectures, we are able to express various problems in a unified model.

## 3.2. KLAIM

In [68], De Nicola et al. present KLAIM, a coordination language for mobile computing which supports the specification of processes that can be moved between computing environments. We develop a Maude-based formal executable specification of KLAIM in rewriting logic, named M-KLAIM, that is used as a foundation for the specification of further coordination languages and *Cloud Computing*. In the following, we give an overview of KLAIM and discuss the general design of our executable specification. Furthermore, we present CINNI [75], a calculus of explicit substitutions, and apply it to the M-KLAIM specification.

### 3.2.1. Overview of KLAIM

KLAIM (**K**ernel **L**anguage for **A**gents **I**nteraction and **M**obility) is a kernel programming language for mobile computing. The language's basic operators were influenced by process algebras like CSP [49], CSS [65] and the $\pi$-calculus [66]. Additionally, Linda's primitives are used and enriched with explicit localities. These localities allow distinguishing between

multiple computing sites and the distribution of the tuple space across such sites. A locality can be either a *physical* or a *logical* locality. This separation allows for a program to be written independently from the network's physical setup. Again, the network structure, the mapping between logical and physical localities, and the distribution of processes, can be rearranged without any changes to the program. The specification of KLAIM also includes a type system that statically checks security properties, i.e., whether the intended operations of a process comply with its access rights. For reasons of simplicity, we have omitted the type system in our Maude-based specification of KLAIM.

### Net syntax

At the highest level of abstraction, the KLAIM model specifies a soup of nodes, called a *net*. A *node* is a triple $(s, P, \rho)$ where $s$ is a *site*, $P$ is a *process*, and $\rho$ defines an *allocation environment*. A *site* can be thought of as a globally valid identifier for a node. Logical localities, i.e., symbolic names for a site, allow programs to reference nodes while ignoring the precise allocation between these names and actual sites. The distinguished logical locality *self* refers to the current execution site. These localities are considered to be first-order data which can be created dynamically and shared using the tuple space. Each node has a specific *allocation environment*, which is a (partial) function from logical localities to sites. $[s/l]$ denotes the environment that maps the logical locality $l$ to the site $s$. $\rho_1 \bullet \rho_2$ denotes the allocation environment that combines the environments $\rho_1$ and $\rho_2$ and is defined by:

$$\rho_1 \bullet \rho_2(l) = \begin{cases} \rho_1(l) & \text{if } \rho_1(l) \text{ is defined} \\ \rho_2(l) & \text{otherwise} \end{cases}$$

In KLAIM, sites are also considered to be logical localities for which the allocation environment acts as the identity function. Additionally, it is assumed that for an allocation environment $\rho_s$ at site $s$ the equation $\rho_s(self) = s$ holds. Nodes that fulfil this property are said to be well-formed. Finally, a net is composed of a set of Nodes:

$$\begin{aligned} N ::= {} & s ::_\rho P && \text{(A single node } (s, P, \rho)\text{ )} \\ & \mid N_1 \parallel N_2 && \text{(Net composition)} \end{aligned}$$

A KLAIM net is said to be *legal* if each node is well-formed and is assigned a distinct site in the net.

**Example 3.2: A legal KLAIM net**
In the following example let $s_1, s_2$ be sites, $l_1, l_2$ be logical localities, $\rho_1, \rho_2$ be allocation environments, and $P, Q$ be processes:

$$s_1 ::_{\rho_1 := [s_1/self] \bullet [s_1/l_1] \bullet [s_2/l_2] \bullet [s_2/l_1]} P \parallel s_2 ::_{\rho_2 := [s_2/self]} Q$$

This net is a legal net as each node is well-formed ($\rho_1(self) = s_1$, $\rho_2(self) = s_2$) and is being assigned a distinct site in the net. Evaluating $\rho_1(l_1)$ yields $s_1$ according to the definition of composed allocation environments. The mapping $[s_2/l_1]$ in $\rho_1$ is never considered for the evaluation on $l_1$ as $[s_1/self] \bullet [s_1/l_1](l_1)$ is already defined.

**Process syntax**

KLAIM processes are built using operators borrowed from Milner's CCS [65]. The *nil* process term represents the process that cannot perform any action. Given processes $P_1$ and $P_2$, $P_1 \mid P_2$ (respectively $P_1 + P_2$) stands for the parallel composition of (respectively the non-deterministic choice between) the two processes $P_1$ and $P_2$. A process can be a process variable or a process invocation $A\langle\tilde{P}, \tilde{l}, \tilde{e}\rangle$, where $\tilde{P}$ is a sequence of processes, $\tilde{l}$ a sequence of localities, and $\tilde{e}$ a sequence of expressions. KLAIM assumes that a process identifier $A$ has a unique defining equation $A(\tilde{X}, \tilde{u}, \tilde{x}) =_{\text{def}} P$, with $\tilde{X}$ a sequence of process variables, $\tilde{u}$ a sequence of locality variables, $\tilde{x}$ a sequence of expression variables, and P being a process. It is further assumed that all free variables in $P$ are contained in the set of variable sequences $\{\tilde{X}, \tilde{u}, \tilde{x}\}$ and that each process identifier is guarded within the scope of a blocking **in** or **read** action prefix to prevent the immediate re-execution of a process invocation, which would result in an infinite loop. **read** and **in** are two of four actions that can prefix a process. The KLAIM actions $out(t)@l$, $eval(P)@l$, $read(t)@l$, and $in(t)@l$ correspond to the Linda operations to generate tuples (**out**), spawn processes (**eval**), read tuples (**rd**), and consume tuples (**in**). In KLAIM, the operations have logical localities as a postfix, which denote the sites the actions address. $t$ stands for a tuple which is a list of expressions, processes, localities (including locality variables) and formal fields. Formal fields are of the form $!v$, where $v$ is either an expression variable, a process variable, or a locality variable. In addition to the operations borrowed from Linda, the $newloc(u)$ action is used to create fresh sites. The locality variable $u$ refers to that fresh site in the prefixed process. Figure 3.2 gives an overview of the process syntax.

$$
\begin{array}{rl}
P ::= & nil \qquad \text{(null process)} \\
\mid & a.P \qquad \text{(action prefixing)} \\
\mid & P_1 \mid P_2 \qquad \text{(parallel composition)} \\
\mid & P_1 + P_2 \qquad \text{(nondeterministic choice)} \\
\mid & X \qquad \text{(process variable)} \\
\mid & A\langle\tilde{P}, \tilde{l}, \tilde{e}\rangle \qquad \text{(process invocation)}
\end{array}
$$

$$
\begin{array}{l}
a ::= out(t)@l \mid in(t)@l \mid read(t)@l \mid eval(P)@l \mid newloc(u) \\
t ::= e \mid P \mid l \mid !x \mid !X \mid !u \mid t_1, t_2
\end{array}
$$

**Figure 3.2.:** KLAIM process syntax

In KLAIM, the $newloc(u).P$, $read(t)@l.P$ and $in(t)@l.P$ actions act as binders for variables that are used in the process $P$. For example, in $newloc(u).P \mid in(!x, !X).Q$, the locality variable $u$ is bound in process $P$, and the variables $x$ and $X$ are bound in process $Q$. KLAIM specifies a process to be a term without free variables. Therefore, each variable must be bound by a binder.

**Example 3.3: A legal KLAIM process term**

In the following example, let $l_1, l_2$ be logical localities, $X$ a process variable, $x$ an expression variable, $u$ a locality variable, and 7 a value expression:

$$P := in(!X)@self.A\langle X, l_1, 7\rangle + out@l_2(7).nil \quad A(X, u, x) =_{]mathrmdef} out(x)@u.X$$

The process identifier $A$ has the unique definition $out(x)@u.X$. The free variables $x, u, X$ are contained in $\{X, u, x\}$ and the process identifier occurs within the scope of a blocking in prefix which also binds the free process variable $X$. As no other free variables occur in the term, $P$ is a legal KLAIM process term.

**Operational Semantics**

KLAIM's operational semantics is given in the structural operational semantics (SOS) style and differentiates between two semantics: the *symbolic semantics* and the *reduction relation*. The semantics proceeds in two steps. First, the *symbolic semantics* specifies the effects of actions on the tuple space which, in KLAIM, is reflected at the process level and defines the process commitments related to localities and the allocation environment. In a second step, the *reduction relation* fully describes the process behavior in a net.

The structural rules of the the *symbolic semantics* specify the possible transitions of KLAIM processes. The resulting labeled transition system does not take the physical location of processes and the tuple space into account. In the transition system, the labeled transition

$$P \xrightarrow[\rho]{\mu} P'$$

describes how process $P$ evolves to process $P'$. The label $\mu$ gives an abstract description of what activity is performed and the label $\rho$ stands for the allocation environment that records the local bindings that must be taken into account to evaluate $\mu$. For example, the rules to send and consume a tuple

$$out(t)@l.P \xrightarrow[\phi]{s(t)@l} P$$

$$in(t)@l.P \xrightarrow[\phi]{i(t)@l} P$$

specify that a process $P$ with the prefix $out(t)@l$ or $in(t)@l$ is able to evaluate to process $P$ with the side effect of sending the tuple $t$ to $l$ ($\mu = s(t)@l$) or consuming the tuple $t$ from $l$ ($\mu = i(t)@l$). Both rules also state that the allocation environment does not have to be taken into account to evaluate the activities, i.e., the empty allocation environment has to be taken into account ($\rho = \phi$). The whole set of structural rules is depicted in Figure 3.3.

KLAIM reflects the tuple space at the process level, where tuples are modeled as processes. The auxiliary process $out(et)$, whose symbolic semantics is given by the structural rule

$$out(et) \xrightarrow[\phi]{o(et)@self} nil$$

denotes the presence of the evaluated tuple $et$ in the tuple space. Tuples are evaluated using the tuple evaluation function $\mathcal{T}[\![.]\!]\rho$, which exploits the allocation environment to resolve locality names. The rules for $\mathcal{T}[\![.]\!]\rho$ are depicted in Figure 3.4, where $\mathcal{E}[\![.]\!]$ evaluates closed

$$out(t)@l.P \xrightarrow[\phi]{s(t)@l} P \qquad eval(t)@l.P \xrightarrow[\phi]{e(t)@l} P$$

$$in(t)@l.P \xrightarrow[\phi]{i(t)@l} P \qquad read(t)@l.P \xrightarrow[\phi]{r(t)@l} P$$

$$newloc(u).P \xrightarrow[\phi]{n(u)@self} P$$

$$\frac{P \xrightarrow[\rho]{\mu} P'}{P + Q \xrightarrow[\rho]{\mu} P'} \qquad \frac{Q \xrightarrow[\rho]{\mu} Q'}{P + Q \xrightarrow[\rho]{\mu} Q'}$$

$$\frac{P \xrightarrow[\rho]{\mu} P'}{P \mid Q \xrightarrow[\rho]{\mu} P' \mid Q} \qquad \frac{Q \xrightarrow[\rho]{\mu} Q'}{P \mid Q \xrightarrow[\rho]{\mu} P \mid Q'}$$

$$\frac{P \xrightarrow[\rho']{\mu} P'}{P\{\rho\} \xrightarrow[\rho' \bullet \rho]{\mu} P'\{\rho\}} \qquad \frac{P[\tilde{P}/\tilde{X}, \tilde{l}/\tilde{u}, \tilde{e}/\tilde{x}] \xrightarrow[\rho]{\mu} P'}{A\langle \tilde{P}, \tilde{l}, \tilde{e} \rangle \xrightarrow[\rho \bullet \rho]{\mu} P'}$$

**Figure 3.3.:** Structural rules of KLAIM's symbolic semantics

expressions to values. The evaluation of a process $P$, $\mathcal{T}[\![P]\!]\rho$ introduces the concept of the process closure $P\{\rho\}$, which combines the process $P$ with the allocation environment $\rho$.

Nets are identified up to the smallest congruence such that the net composition $\|$ is associative and commutative. The *reduction relation* describes the process behavior in a net and provides rules for actions that affect the local node and rules for actions that affect a remote node. Syntactically, the reduction transition

$$N \rightarrowtail N'$$

describes the evolution of net $N$ to $N'$. The local and remote rules for the *out* operation

$$(1)\ \frac{P \xrightarrow[\rho']{s(t)@l} P' \qquad s = \rho' \bullet \rho(l) \qquad et = \mathcal{T}[\![t]\!]_{\rho' \bullet \rho}}{s ::_\rho P \rightarrowtail s ::_\rho P' \mid out(et)}$$

$$(2)\ \frac{P_1 \xrightarrow[\rho]{s(t)@l} P_1' \qquad s_2 = \rho \bullet \rho_1(l) \qquad et = \mathcal{T}[\![t]\!]_{\rho \bullet \rho_1}}{s_1 ::_{\rho_1} P_1 \parallel s_2 ::_{\rho_2} P_2 \rightarrowtail s_1 ::_{\rho_1} P_1' \parallel s_2 ::_{\rho_2} P_2 \mid out(et)}$$

add a new auxiliary process to the local (rule (1)) or to a remote (rule (2)) process and thereby put a new tuple into the tuple space. In rule (2), the tuple $t$ is evaluated using the allocation environment $\rho \bullet \rho_1$, which means that if the process has a closure $P\{\rho\}$, its closure is used in conjunction with the local allocation environment $\rho_1$ to evaluate the tuple. If the

$$
\begin{aligned}
\mathcal{T}[\![e]\!]\rho &= \mathcal{E}[\![e]\!] \\
\mathcal{T}[\![P]\!]\rho &= P\{\rho\} \\
\mathcal{T}[\![l]\!]\rho &= \rho(l) \\
\mathcal{T}[\![!x]\!]\rho &= !x \\
\mathcal{T}[\![!X]\!]\rho &= !X \\
\mathcal{T}[\![!u]\!]\rho &= !u \\
\mathcal{T}[\![t_1, t_2]\!]\rho &= \mathcal{T}[\![t_1]\!]\rho, \mathcal{T}[\![t_2]\!]\rho
\end{aligned}
$$

**Figure 3.4.:** Inductive definition of KLAIM's tuple evaluation function

process has no closure, the equation $\rho = \phi$ holds, and the tuple is evaluated using only the local allocation environment $\rho_1 = \phi \bullet \rho_1$. Finally, if a tuple is sent to a remote node, the sending process' closure and the sending node's allocation environment are used to evaluate the tuple. The other rules of KLAIM's reduction relation (Figure A.1) evaluate tuples in a similar way.

Pattern matching is used to identify appropriate tuples for an *in* or *read* operation. For example, the rule to consume a tuple from a remote node

$$
(6)\ \frac{P_1 \xrightarrow[\rho]{i(t)@l} P_1' \qquad s_2 = \rho \bullet \rho_1(l) \qquad P_2 \xrightarrow[\phi]{o(et)@self} P_2' \qquad match(\mathcal{T}[\![t]\!]_{\rho \bullet \rho_1}, et)}{s_1 ::_{\rho_1} P_1 \parallel s_2 ::_{\rho_2} P_2 \rightarrowtail s_1 ::_{\rho_1} P_1'[et/\mathcal{T}[\![t]\!]_{\rho \bullet \rho_1}] \parallel s_2 ::_{\rho_2} P_2'}
$$

uses pattern matching to match the remotely available evaluated tuple with the evaluation of the tuple that is the argument of the *in* operation. KLAIM's matching rules are shown in Figure 3.5, where $v$ denotes a value, $P$ a process, $s$ a site, $!x$ an expression variable, $!X$ a process variable, $!u$ a locality variable, and $et_i$ with $i \in \{1, 2, 3, 4\}$ denote evaluated tuples.

$$
match(v, v) \qquad match(P, P) \qquad match(s, s)
$$

$$
match(!x, v) \qquad match(!X, P) \qquad match(!u, s)
$$

$$
\frac{match(et_2, et_1)}{match(et_1, et_2)} \qquad \frac{match(et_1, et_3) \qquad match(et_2, et_4)}{match((et_1, et_2), (et_3, et_4))}
$$

**Figure 3.5.:** KLAIM's matching rules

To reflect the fact that a site is present only once in a net, the rule

$$
(11)\ \frac{N_1 \rightarrowtail N_1' \qquad st(N_1') \cap st(N_2) = \emptyset}{N_1 \parallel N_2 \rightarrowtail N_1' \parallel N_2}
$$

states that a net within a composed net may only make a step if no site is used twice. The

helper function $st(N)$ thereby simply returns the set of sites in $N$. All rules of the reduction relation are listed in Figure A.1 in the Appendix.

**Example 3.4: Communication between nodes in KLAIM**
In the following example, we consider a net consisting of three nodes that are located at the sites $s_1, s_2$ and $s_3$. The nodes are all well-formed, since their allocation environments $\rho_i,\ i \in \{1,2,3\}$ are well-defined ($rho_i(self) = s_i$). Furthermore, the logical locality $l_2$ is mapped to $s_2$ in $\rho_1$ and $\rho_3$. An *out* operation at site $s_1$ first triggers a transition as described in the *symbolic semantics*:

$$s_1 ::_{\rho_1} out(7)@l_2.nil \quad \| \quad s_2 ::_{\rho_2} nil \quad \| \quad s_3 ::_{\rho_3} in(!x)@l_2.P$$
$$\downarrow s(7)@l_2$$
$$nil$$

Now the corresponding rule of the reduction relation adds the evaluated tuple 7 to the process at $s_2$, which is the site that the logical locality $l_2$ maps to in the allocation environment at site $s_1$. Simultaneously, the *symbolic semantics* allows for transitions to be made by the action at site $s_3$ and the auxiliary process at site $s_2$:

$$s_1 ::_{\rho_1} nil \quad \| \quad s_2 ::_{\rho_2} out(7) \quad \| \quad s_3 ::_{\rho_3} in(!x)@l_2.P$$
$$\downarrow o(7)@self \qquad \downarrow i(!x)@l_2$$
$$nil \qquad P$$

In a last step, the rule for a remote consumption of a tuple allows the tuple 7 of site $s_2$ to be consumed by the *in* operation at site $s_3$ since the expression variable $!x$ matches with any value:

$$s_1 ::_{\rho_1} nil \quad \| \quad s_2 ::_{\rho_2} nil \quad \| \quad s_3 ::_{\rho_3} P[7/!x]$$

## 3.3. M-KLAIM — a Maude-based specification of KLAIM

Rewriting logic [60] supports the executable specification of KLAIM's syntax and structural operational semantics. This can be done in several definitional styles [74], which can exactly mirror any desired SOS style. In this work, we aim at an efficient executable specification of KLAIM and therefore do *not* follow the SOS style of Section 3.2 *au pied de la lettre*. That is, the inference rules of the structural operational semantics have not been specified as given, but have been transformed to rewrite rules that allow for better executability. In terms of syntax, the Maude-based implementation was designed to be as close as possible to KLAIM's notation. In the following, we give an overview of the modules in the Maude-based executable specification and describe each module in more detail.

### 3.3.1. Overview

Figure 3.6 gives a basic overview of the Maude modules and their submodule dependencies. The module *LOCALITY* contains the main sorts and operations for logical and physical localities. KLAIM's allocation environment, i.e., the mapping between logical and

physical localities, is described in the module *ALLOCATION-ENVIRONMENT*. The syntax and semantics of expressions are given in the modules *EXPRESSION-SYNTAX* and *EXPRESSION-SEMANTICS*. The syntactic elements of processes and tuples are described in the module *PROCESS-SYNTAX*. The module *PROCESS-CONTEXT* defines the sorts, operations and conditional requirements for process definitions which are used for process invocations. The module *TUPLE-EVALUATION* includes the description of the tuple evaluation and matching mechanisms. Substitutions are handled by the CINNI calculus, whose implementation is specified in the module *CINNI*. Section 3.4 gives an introduction to CINNI and further describes the application of the calculus to KLAIM and its implementation. The top level sorts and operators of KLAIM are described in the module *KLAIM-SYNTAX*. Finally, the module *KLAIM-SEMANTICS* defines the rewriting semantics of nets. For the evaluation of process invocations, the process context is used. To allow for greater flexibility, the context is statically defined and is passed to *KLAIM-SEMANTICS* as a parameter.

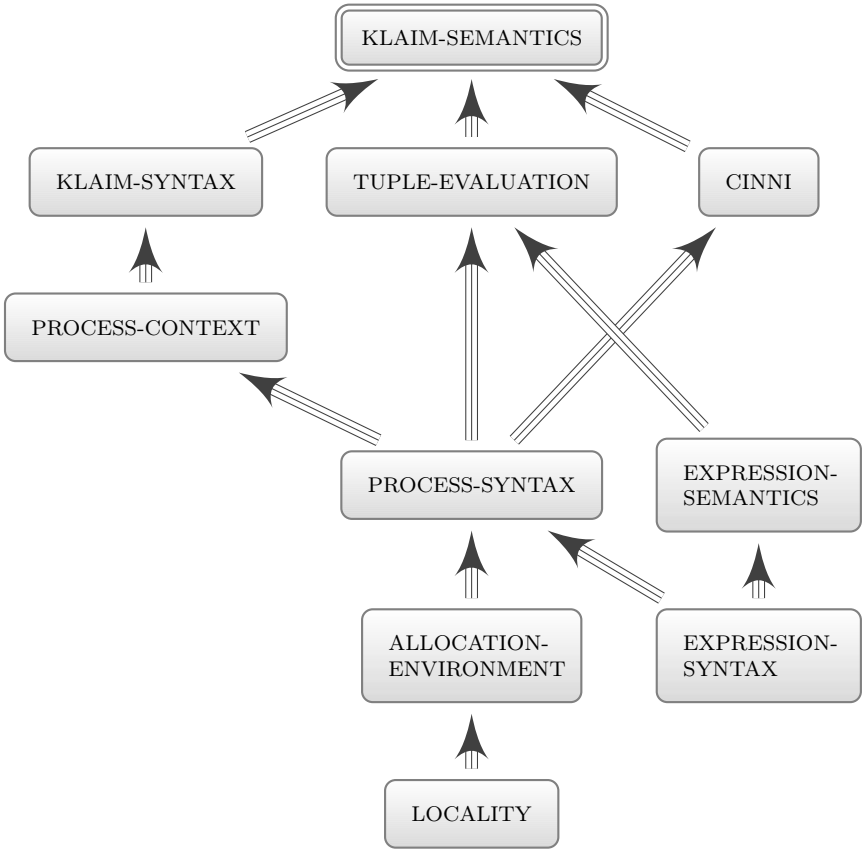



**Figure 3.6.:** Overview of Maude modules of the M-KLAIM specification

### 3.3.2. Description of modules

The following descriptions of modules show how KLAIM's SOS specification style of Section 3.2 is translated to Maude.

### *LOCALITY*

The module *LOCALITY* includes the sort and constructor operator declarations for KLAIM localities. The sort `Locality` denotes logical localities. The sort for quoted identifiers, `Qid`, is defined to be a subsort of `Locality` and, as such, provides a simple way to construct logical localities. Physical localities are represented by the sort `Site`, which is defined as a subsort of the sort `Locality`, as KLAIM states that physical localities can be used as logical localities. Sites are constructed using the operator `site`, with a single argument of sort `Qid`.

```
sorts Locality Site LocalityVar LocalityVarName .
subsort Qid Site LocalityVar < Locality .

op self : -> Locality [ctor] .
op site_ : Qid -> Site [ctor prec 15] .
```

### *ALLOCATION-ENVIRONMENT*

The allocation environment describes the mapping between logical and physical localities. Terms of sort `AllocationEnvironment` are constructed by the empty allocation environment `{}`, the mapping `[S/L]` assigning to a logical locality `L` a site `S`, and the concatenation $\rho_1$ `*` $\rho_2$ of two allocation environments $\rho_1$ and $\rho_2$.

```
sort AllocationEnvironment .

op {} : -> AllocationEnvironment [ctor] .
op [_/_] : Site Locality -> AllocationEnvironment [ctor] .
op _*_ : AllocationEnvironment AllocationEnvironment
  -> AllocationEnvironment [ctor assoc id: {}] .
```

The evaluation operator

```
op _(_) : AllocationEnvironment Locality -> Locality [memo] .
```

takes an allocation environment and a locality as arguments and, if a mapping for the locality is present in the allocation environment, returns the site the locality is mapped to. In the following description of the behavior of the evaluation operator, the variables

```
var RHO : AllocationEnvironment .
vars L L1 L2 : Locality .
var S : Site .
```

are used. The equation

```
eq [evaluation-base] : {}(L) = L .
```

deals with the base case of the evaluation, where the evaluation operator is applied to an empty allocation environment. In this case, the locality is itself is returned. The two equations

```
eq [evaluation-rec1] : [S / L] * RHO(L) = S .
ceq [evaluation-rec2] : [S / L1] * RHO(L2) = RHO(L2) if L1 =/= L2 .
```

recursively decompose the allocation environment. Finally, the Equation

```
eq [evaluation-site] : RHO(S) = S .
```

specifies that the evaluation function acts as an identity on sites.

### *EXPRESSION-SYNTAX*

KLAIM does not explicitly specify a syntax and semantics for expressions. The only requirement it makes for expressions is that (fully evaluated) ground terms of expressions should be either values or variables. In M-KLAIM, we therefore developed a simple expression model based on natural numbers.

Expressions are of sort `Expression`. The sort is a supersort of the sorts `Val` for values and `Var` for variables, i.e. values and variables are basic expressions.

```
sorts Expression Val Var .
subsorts Var Val < Expression .
```

Values are constructed by the operator

```
op [_] : Nat -> Val [ctor] .
```

which takes a natural number as an argument. Variables are named representatives of expressions. The Maude-based KLAIM specification used the CINNI calculus to handle variables and its substitutions. The CINNI calculus and its application to M-KLAIM are described in Section 3.4. In addition to values and variable, expressions are constructed by the operator

```
op _+e_ : Expression Expression -> Expression
  [assoc comm ctor prec 16] .
```

which takes two expressions as arguments.

### *EXPRESSION-SEMANTICS*

In KLAIM, expressions are tuples which can be evaluated using a tuple evaluation function. When a tuple evaluation function is applied to an expression, the tuple evaluation function calls an expression evaluation function. The expression evaluation operator

```
op E[|_|] : Expression -> Nat .
```

takes an expression as an argument and, in our expression model, returns a natural number. For the description of the semantics of the evaluation operator, the variables

```
vars E1 E2 : Expression .
var N : Nat .
```

are used.

The expression evaluation (partial) function is only defined for closed expressions, i.e., expressions that do not contain expression variables. If the expression evaluation operator is applied to a single value, it returns that value.

```
eq [expression-evaluation-base] : E[|[N]|] = N .
```

If the operator is applied on a sum of two expressions, it is recursively applied to the sum's arguments. The result is the sum of the evaluation of the arguments. The sum operator for natural numbers is predefined in Maude.

```
eq [expression-evaluation-rec] : E[| E1 +e E2 |]
  = E[| E1 |] + E[| E2 |] .
```

This generic specification of a expression model based on natural numbers can easily be replaced by a more expressive specification of arbitrary expressions on other data types of values, provided the operators used are fully defined (sufficiently complete) on such data types.

### *PROCESS-SYNTAX*

KLAIM processes serve two purposes. First, they capture the behavior of a node in the net, i.e., the actions it performs, and, second, a node's tuple space is syntactically represented in the process. We distinguish between the sorts `SyntacticProcess` and `AuxiliaryProcess`. The supersort `Process` encapsulates these two different types of processes.

```
sorts Action SyntacticProcess AuxiliaryProcess Process .
subsort SyntacticProcess AuxiliaryProcess < Process .
```

Terms of sort `SyntacticProcess` specify processes that are constructed using the KLAIM syntax for processes and process closures. The sort `AuxiliaryProcess` defines the auxiliary process terms constructed by the operator

```
op out(_) : EvaluatedTuple -> AuxiliaryProcess [ctor] .
```

which can only interact with corresponding `in` or `read` actions of syntactic processes. Auxiliary processes represent the tuple space in KLAIM: Evaluated tuples are stored inside auxiliary processes, which in turn are part of a process.

In order to distinguish between a tuple and a tuple on which the evaluation function has been applied, we introduce the two sorts `Tuple` and `EvaluatedTuple`. The sort `Tuple` represents KLAIM tuples and `EvaluatedTuple` the results of the tuple evaluation function. However, every evaluated tuple is itself again a tuple. Therefore, the sort `EvaluatedTuple` is a subsort of the sort `Tuple`. Tuples or evaluated tuples that cannot be decomposed any further are denoted by the two sorts `TupleElt` and `EvaluatedTupleElt`. Figure 3.7 gives an overview of the subsort relation between the sorts in the module *PROCESS-SYNTAX*. An arrow from one sort to another represents the subsort inclusion relationship, e.g., `Locality` → `TupleElt` states that the sort `Locality` is a subsort of `TupleElt`.

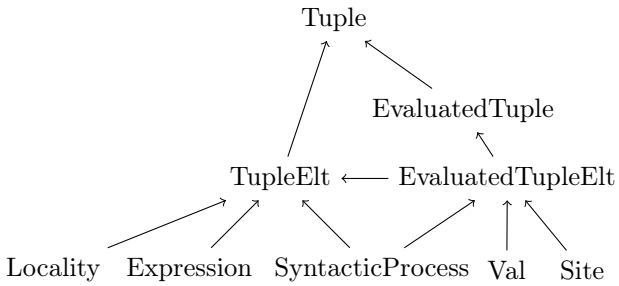



**Figure 3.7.:** Subsort hierarchy in the *PROCESS-SYNTAX* module

The sort `SyntacticProcess` and its constructors capture KLAIM's process constructs:

- The constant

  ```
  op nil : -> SyntacticProcess [ctor] .
  ```

constructs the null process.

- The operator

```
op _._ : Action SyntacticProcess -> SyntacticProcess
  [frozen ctor prec 25] .
```

is used to create a process with an action prefix.

- The operator

```
op _|_ : Process Process -> Process
  [frozen assoc comm ctor id: nil prec 30] .
```

creates the parallel composition of two processes. The constant term `nil` is the identity for this operator, because a `nil` process can never perform an action. M-KLAIM specifies a second composition operator

```
op _|_ : SyntacticProcess SyntacticProcess -> SyntacticProcess
  [frozen assoc comm ctor id: nil prec 30] .
```

specifically for syntactic processes. A process therefore is only of sort `SyntacticProcess` if it contains no auxiliary processes.

- The operator

```
op _+_ : SyntacticProcess SyntacticProcess -> SyntacticProcess
  [frozen assoc comm ctor prec 35] .
```

creates the nondeterministric choice of two syntactic processes.

- The operator

```
op _<_,_,_> : Qid ProcessSeq LocalitySeq ExpressionSeq
  -> SyntacticProcess [ctor prec 25] .
```

constructs a process invocation. The behavior of process invocations is defined in the process context in more detail (Section 3.3.2).

- Proccess variables construct a syntactic process. Consequently, the sort `ProcessVar` is defined to be a subsort of `SyntacticProcess`.

```
sorts ProcessVar ProcessVarName .
subsort ProcessVar < SyntacticProcess .
```

- Additionally, we add the constructor

```
op _{_} : SyntacticProcess AllocationEnvironment
  -> SyntacticProcess [prec 28] .
```

to create a *process closure*, i.e., a process that is closed under an allocation environment. Such an operator is not defined in the original KLAIM specification. It is used by the M-KLAIM specification to syntactically denote closed processes.

As done by Verdejo et al. in [80], we add the `frozen` attribute to and declare a precedence value for the process constructors. Declaring a given operator as frozen forbids rewriting with rules in all proper subterms of a term having such an operator as its top operator. Hence, we prevent rewriting rules to be executed on subterms and only allow rewriting rules for the top-level operator to proceed, which reflects KLAIM's process semantics. In order to prevent ambiguity, each operator is assigned a precedence value that defines how, i.e., in what precedence order, a term is parsed in the presence of several operators.

The sorts `Tuple` and `EvaluatedTuple` respectively represent KLAIM tuples and the results of the tuple evaluation function.

```
sorts Tuple EvaluatedTuple .
```

The associative concatenation operators

```
op _,_ : Tuple Tuple -> Tuple [ctor assoc prec 40] .
op _,_ : EvaluatedTuple EvaluatedTuple -> EvaluatedTuple
  [ctor assoc prec 40] .
```

are defined for the sorts `Tuple` and `EvaluatedTuple`.

As shown in Figure 3.7, the sort `TupleElt` is a subsort of the sort `Tuple`, and the sorts `EvaluatedTupleElt`, `Locality`, and `Expression` are subsorts of the sort `TupleElt`. Additionally, the sort `EvaluatedTuple` is a subsort of `Tuple` and the sort `EvaluatedTupleElt` is a subsort of `EvaluatedTuple`. Finally, the sorts `Val`, `SyntacticProcess`, and `Site` are subsorts of the sort `EvaluatedTupleElt`.

```
sort TupleElt EvaluatedTupleElt .
subsort EvaluatedTupleElt Locality Expression < TupleElt < Tuple .
subsort Val SyntacticProcess Site
  < EvaluatedTupleElt < EvaluatedTuple < Tuple .
```

KLAIM's formal fields are constructed by the operators

```
op !_ : VarName -> EvaluatedTupleElt [ctor prec 15] .
op !_ : LocalityVarName -> EvaluatedTupleElt [ctor prec 15] .
op !_ : ProcessVarName -> EvaluatedTupleElt [ctor prec 15] .
```

which take an expression variable name, a locality variable name, or a process variable name as an argument. In M-KLAIM, a formal field is an evaluated tuple element.

Finally, the operator definitions for KLAIM's process prefixes (*out*, *in*, *read*, *eval*, and *newloc*)

```
op out(_)@_ : Tuple Locality -> Action [ctor prec 20] .
op in(_)@_ : Tuple Locality -> Action [ctor prec 20] .
op read(_)@_ : Tuple Locality -> Action [ctor prec 20] .
op eval(_)@_ : SyntacticProcess Locality -> Action [ctor prec 20] .
op newloc(_) : LocalityVarName -> Action [ctor prec 20] .
```

construct the terms of sort `Action`.

### *TUPLE-EVALUATION*

The module *TUPLE-EVALUATION* provides fuctions for the evaluation of tuples and the matching of evaluated tuples. The tuple evaluation operator

```
op T[|_|]_ : Tuple AllocationEnvironment -> EvaluatedTuple [memo] .
```

takes a tuple and an allocation environment as arguments and returns the evaluated tuple. The commutative matching operator

```
op match : EvaluatedTuple EvaluatedTuple -> Bool [comm memo] .
```

takes two evaluated tuples as arguments and returns a boolean value of Maude's predefined sort `Bool`.

For the specification of the tuple evaluation semantics, the variables

```
var ET : EvaluatedTuple .
var E : Expression .
var SP : SyntacticProcess .
var L : Locality .
vars T1 T2 : Tuple .
```

are used. The equation

```
eq [tuple-evaluation-id] : T[| ET |]RHO = ET .
```

defines that the tuple evaluation function acts as the identity function on evaluated tuples. As described by the equation

```
eq [tuple-evaluation-base1] : T[| E |]RHO = [E[| E |]] .
```

expressions are evaluated using the expression evaluation operator introduced in the module *EXPRESSION-SEMANTICS*. A syntactic process $P$ is evaluated to the process closed under the evaluation function's allocation environment $\rho$.

```
eq [tuple-evaluation-base2] : T[| SP |]RHO = SP{RHO} .
```

Localities are evaluated using the evaluation function of the allocation environment (Section 3.3.2).

```
eq [tuple-evaluation-base2] : T[| SP |]RHO = SP{RHO} .
```

The evaluation of a tuple sequence evaluates each tuple element and yields a sequence of evaluated tuples. The equation

```
eq [tuple-evaluation-rec] : T[| T1,T2 |]RHO
  = (T[| T1 |]RHO), (T[| T2 |]RHO) .
```

defines the recursive application of the evaluation function on all tuple elements in a tuple sequence.

The matching operation for evaluated tuples is used by KLAIM's semantics in order to identify appropriate tuples for *in* and *read* operations. For the definition of the semantics of the matching function, the variables

```
var V : Val .
var SP : SyntacticProcess .
var S : Site .
var VN : VarName .
var PVN : ProcessVarName .
var LVN : LocalityVarName .
vars ET ET1 ET2 : EvaluatedTuple .
vars ETE1 ETE2 : EvaluatedTupleElt .
```

are used.

Equal terms of values, syntactic processes, or sites do match.

```
eq [tuple-matching-base1] : match(V, V) = true .
eq [tuple-matching-base2] : match(SP, SP) = true .
eq [tuple-matching-base3] : match(S, S) = true .
```

Formal fields, i.e., placeholders for any term of a certain sort, match with any term of that specific sort. The equations

```
eq [tuple-matching-var1] : match(!(VN), V) = true .
eq [tuple-matching-var2] : match(!(PVN), SP) = true .
eq [tuple-matching-var3] : match(!(LVN), S) = true .
```

specify the matching of formal fields for expression variable names, process variable names, and locality variable names with any expression variable, syntactic process, and site term.

Tuple sequences match if each evaluated tuple element in the first sequence matches the associated evaluated tuple element in the second sequence.

```
eq [tuple-matching-rec] : match((ETE1, ET1), (ETE2, ET2))
  = (match(ETE1, ETE2) and match(ET1, ET2)) .
```

KLAIM does not specify the behavior of the tuple matching operator when it is applied to sequences of different length. Our design decision for M-KLAIM is to return `false` if the matching operator is applied to sequences of different length. Finally, the equation

```
eq [tuple-matching-otherwise] : match(ET1, ET2) = false [owise] .
```

states that a pair of evaluated tuples does not match if it is not covered by one of the aforementioned base cases.

### *PROCESS-CONTEXT*

KLAIM does not specifically define a process context but uses process identifiers and defining equations for process invocations. For M-KLAIM, we decided to specify an explicit construct, namely a process context of sort `Context`, to map between process identifiers (of sort `ProcessId`) and defining syntactic processes.

```
sort Context .
sort ProcessId .
```

Process identifiers are named by quoted identifiers and are used by process invocations to reference a specific mapping to a defining syntactic process. They are constructed by the operator

```
op _(_,_,_) : Qid ProcessVarNameSeq LocalityVarNameSeq VarNameSeq
  -> ProcessId [ctor] .
```

which takes the name, a sequence of process variable names, a sequence of locality variable names, and a sequence of expression variable names as arguments. The variable names declare the variable names that are used in the defining syntactic process.

The constructors for defining equations

```
op _=def_ : ProcessId SyntacticProcess -> [Context] [ctor prec 45] .
```

and the context composition

```
op _&_ : Context Context -> [Context]
  [ctor assoc comm id: nilContext prec 50] .
```

are partial functions, because not every context that can be constructed is regarded a valid context by the KLAIM specification. Thus, the result of the constructor operators is in general of kind `[Context]` (see [33]), and is only of sort `Context` for valid contexts. Additionally, contexts are constructed by the constant

```
op nilContext : -> Context [ctor] .
```

which defines the empty process context.

A process context is valid if the context provides no more than one defining equation for a process identifier and each defining equation is closed. In the following, we describe how the semantics of M-KLAIM handles the validity of process contexts. For the description of the semantics, the variables

```
var PID : ProcessId .
var SP : SyntacticProcess .
var CK : [Context] .
```

are used.

We define the `isValidContext` predicate and provide a conditional membership to define the legal members of the sort `Context`.

```
op isValidContext : [Context] -> Bool .
cmb CK : Context if isValidContext(CK) .
```

As a first base case, the equation

```
eq [isValidContext-base1] : isValidContext(nilContext) = true .
```

defines the empty context to be a valid context. Second, the equation

```
eq [isValidContext-base2] : isValidContext(PID =def SP)
  = isClosed(PID, SP) .
```

defines a single process definition to be valid if it is closed under the process identifier. The specification of the `isClosed` predicate, which takes a process identifier and a syntactic process as arguments, is not shown here. It evaluates to `true` if all unbound process, locality, and expression variable names of the syntactic process are contained in the respective sequences for the process, locality, and expression variable names of the process identifier. Lastly, the equation

```
op _definedIn_ : ProcessId Context -> Bool [memo] .
eq [definedIn-base] : PID definedIn nilContext = false .
eq [definedIn-rec] : PID1 definedIn (PID2 =def SP & C) = (PID1 == PID2)
  or (PID1 definedIn C) .

eq [isValidContext-rec] : isValidContext((PID =def SP) & CK)
  = not(PID definedIn CK) and isClosed(PID, SP) and isValidContext(CK) .
```

recursively evaluates a process context composed of multiple process identifiers. For each process identifier, it checks if it is the only definition for that identifier using the `definedIn` operator and if the syntactic process is closed under the process identifier.

A process context is queried by the predicate `definedIn` and the operations `processId` and `def`. The predicate

```
op _definedIn_ : Qid Context -> Bool [memo] .
```

takes a quoted identifier as an argument and returns whether a context contains a process identifier whose name is equal to the specified quoted identifier. The operator

```
op processId : Qid Context -> ProcessId [memo] .
```

is used to retrieve the process identifier for a specific quoted identifier from the context. Finally, the operator

```
op def : ProcessId Context -> SyntacticProcess [memo] .
```

retrieves the defining syntactic process for a specific process identifier from a context.

For the definition of the semantics of the context querying operators, the variables

```
vars PID1 PID2 : ProcessId .
vars A1 A2 : Qid .
var SP : SyntacticProcess .
var C : Context .
var PVNS : ProcessVarNameSeq .
var LVNS : LocalityVarNameSeq .
var EVNS : VarNameSeq .
```

are used.

The `definedIn` predicate returns `false` if the context that is provided as an argument is the empty context.

```
eq [definedIn-base] : A1 definedIn nilContext = false .
```

Otherwise, it recursively traverses the process context and returns `true` if a defining equation for the process identifier is found.

```
eq [definedIn-rec] : A1 definedIn (A2(PVNS, LVNS, EVNS) =def SP & C)
  = (A1 == A2) or (A1 definedIn C) .
```

Finally, the `def` operator recursively traverses a process context to find the defining syntactic process for a process identifier.

```
eq [def] : def(PID1, ((PID2 =def SP) & C)) =
  if PID1 == PID2 then SP else def(PID1, C) fi .
```

The specification of M-KLAIM's semantics uses the constant

```
op context : -> context [ctor] .
```

as a reference to the process context. By default, the process context is defined to be an empty context.

```
eq [default-Context] : context = nilContext [owise] .
```

Specifications based on M-KLAIM can specify their own process context by adding an equation that defines the individual process context. Examples for the definition of model-specific process contexts are shown in Sections 3.6.5 and 3.7.

### *KLAIM-SYNTAX*

The *KLAIM-SYNTAX* module defines the sort

```
sort Net .
```

and constructors for KLAIM nets.

The operator to construct a node

```
op (_{_}::{_}_) : Site Nat AllocationEnvironment Process
  -> [Net] [ctor] .
```

extends the KLAIM notation for nodes with a natural number enclosed in curly braces after the site. This number is not mentioned in the KLAIM specification. However, our specification uses it as a counter of a node's children. This number provides a way to guarantee the creation of a fresh site when a *newloc* action is executed by one of the nodes.

Additionally, the associative and commutative net constructor operator

```
op _||_ : Net Net -> [Net] [config assoc comm ctor] .
```

combines two nets.

The constructors for nodes and nets are partial and thus have kind `[Net]`. This is due to the fact that not all nodes and nets that can be constructed are well-formed and legal. For the description of the semantics, the predefined Maude parametrized data structure `SET` is imported in protecting mode, with appropriate renamings of sorts and operators.

```
protecting SET{Site} * (sort Set{Site} to Sites,
  op _,_ : Set{Site} Set{Site} -> Set{Site} to _;;_) .
```

and the variables

```
var KN : [Net] .
var S : Site .
var SI : Sites .
var M : Nat .
var RHO : AllocationEnvironment .
var P : Process .
var N : Net .
```

are used.

The conditional membership

```
op isValidNet : [Net] Sites -> Bool [memo] .
cmb KN : Net if isValidNet(KN, empty) .
```

checks for the validity of nodes and nets. For a single node and a set of other sites in the net, the predicate `isValidNet` returns `true` if and only if the evaluation of self on the node's allocation environment yields the site of the node, the site is not in the set of other sites in the net, the node's process is closed, and the node's process does not loop indefinitely. The equations

```
eq isValidNet((S { M }::{ RHO } P) || N, SI)
  = isValidNet((S { M }::{ RHO } P), SI)
    and isValidNet(N, S ;; SI) .
```

and

```
eq isValidNet((S { M }::{ RHO } P), SI)
  = noInfiniteLoops(P)
  and isClosed(P)
  and RHO(self) == S
  and not(S in SI) .
```

define the predicate's semantics.

The predicate `isClosed`, which determines if a process is closed, is not shown here. For the definition of the semantics of the operator

```
op noInfiniteLoops : Process -> Bool [memo] .
```

which determines if a process loops indefinitely, the variables

```
var AP : AuxiliaryProcess .
var T : Tuple .
vars SP SQ : SyntacticProcess .
var L : Locality .
var LV : LocalityVarName .
vars P Q : Process .
var RHO : ALLOCATIONENVIRONMENT .
var PS : ProcessSeq .
var LS : LocalitySeq .
var ES : ExpressionSeq .
```

are used.

To guarantee that a process does not loop indefinitely, the KLAIM specification requires that each process invocation in a process is guarded by a blocking action. The base cases are described by the equations

```
eq noInfiniteLoops(nil) = true .
eq noInfiniteLoops(AP) = true .
eq noInfiniteLoops(in(T) @ L . SP) = true .
eq noInfiniteLoops(read(T) @ L . SP) = true .
```

which specify that the `nil`-process, an auxiliary process, and processes that are guarded by the blocking *in* or *read* action do not loop indefinitely.

Otherwise, the equations

```
eq noInfiniteLoops(newloc(LV) . SP) = noInfiniteLoops(SP) .
eq noInfiniteLoops(out(T) @ L . SP) = noInfiniteLoops(SP) .
eq noInfiniteLoops(eval(SQ) @ L . SP) = noInfiniteLoops(SP) .
ceq noInfiniteLoops(P | Q) =
  noInfiniteLoops(P) and noInfiniteLoops(Q)
if P =/= nil /\ Q =/= nil .
eq noInfiniteLoops(SP + SQ) =
  noInfiniteLoops(SP) and noInfiniteLoops(SQ) .
eq noInfiniteLoops(SP { RHO }) = noInfiniteLoops(SP) .
```

specify the recursive evaluation of the operator for the process constructors.

Finally, the equation

```
eq noInfiniteLoops(A < PS, LS, ES >) = false .
```

specifies that a process invocation by itself is not guaranteed to be non-looping.

The *KLAIM-SYNTAX* module further specifies the high-level substitution operators

```
op _[_/_] : SyntacticProcess EvaluatedTuple EvaluatedTuple
  -> SyntacticProcess [prec 20] .
op _[_/_] : SyntacticProcess ProcessSeq ProcessVarNameSeq
  -> SyntacticProcess [prec 20] .
op _[_/_] : SyntacticProcess LocalitySeq LocalityVarNameSeq
  -> SyntacticProcess [prec 20] .
op _[_/_] : SyntacticProcess ExpressionSeq VarNameSeq
  -> SyntacticProcess [prec 20] .
```

for each type of substitution that occurs in the semantics of KLAIM. Substitutions in KLAIM occur in two different places. First, evaluated tuples are substituted for evaluated tuples when an `in` action consumes or a `read` action reads a tuple. Second, when a process invocation is processed, sequences of process, locality, and expression variable names are substituted with sequences of processes, localities, and expressions in the defining process.

### *KLAIM-SEMANTICS*

The *KLAIM-SEMANTICS* module specifies KLAIM's SOS style rules using rewriting logic.

**Specification of $\mathcal{R}_{\mathbf{KLAIM}}$.** Works by Braga and Meseguer [31], Verdejo et al. [80], and Serbanuta et al. [74] have shown that SOS rules can naturally be mapped to rewrite rules with different styles. Basically, inference rules of the form

$$\frac{P_1 \to Q_1 \ldots P_n \to Q_n}{P_0 \to Q_0}$$

become conditional rewrite rules of the form

$$P_0 \to Q_0 \ \textit{if} \ P_1 \to Q_1 \wedge \ldots \wedge P_n \to Q_n$$

where the condition may include rewrite conditions. Some technical details may be added to capture the one-step semantics of some SOS rules. In our approach defining the rewrite semantics of KLAIM, $\mathcal{R}_{\text{KLAIM}}$, we combine the rules of the symbolic semantics and the reduction relation. Transitions in $\mathcal{R}_{\text{KLAIM}}$ only happen at the net level. The structural rules for action prefixes are not reflected by the rewriting semantics. Inference rules of the reduction relation with premises that require a process to perform a transition according to the symbolic semantics are mapped to inference rules with no such condition. The symbolic semantics transition for the action prefix is built-in into the rewrite rule. Using this approach, the conditional rewrite rules obtained by the transformation of the reduction relation's rules include no rewrite conditions as the remaining premises of these inference rules are expressed by equational and matching conditions. One advantage of this approach is that the specification can be executed with higher performance, because equational and matching conditions are evaluated faster than rewrite conditions and therefore avoid expensive non-deterministic rewrite searches in conditions.

The variables

```
vars COUNT COUNT1 COUNT2 : Nat .
vars RHO RHO1 RHO2 : AllocationEnvironment .
vars P PP Q : Process .
vars SP SQ : SyntacticProcess .
var L : Locality .
vars S S1 S2 NEWSITE : Site .
vars ET1 ET2 : EvaluatedTuple .
var PS : ProcessSeq .
var LS : LocalitySeq .
var ES : ExpressionSeq .
var PVNS : ProcessVarNameSeq .
var LVNS : LocalityVarNameSeq .
var VNS : VarNameSeq .
var E : Expression .
var T : Tuple .
var A ID : Qid .
var C : Context .
var PID : ProcessId .
```

are used by the specification of the semantic rewrite rules in Maude.

**Rules for the *out* action.** The rules

```
crl [out-self] :
  (S {COUNT} :: {RHO} (out(T) @ L) . SP | PP )
=>
  (S {COUNT} :: {RHO} SP | PP | out(T[| T |]RHO) )
if S := RHO(L) .
```

and

```
crl [out-remote] :
  (S1 {COUNT1} :: {RHO1} (out(T) @ L) . SP | PP)
  || (S2 {COUNT2} :: {RHO2} P)
=> (S1 {COUNT1} :: {RHO1} SP | PP)
  || (S2 {COUNT2} :: {RHO2} P | out(T[| T |]RHO1) )
if S2 := RHO1(L) .
```

correspond to rule (1) and rule (2) of KLAIM's reduction relation (Figure A.1). `out-self` states that if the process of the node at site `S` contains a process which can perform an *out* action and the action's locality evaluates to `S`, a new auxiliary process is added to the process at that node. Otherwise, if the action's locality evaluates to a different site than the node's site and the remote node at that site exists, the rule `out-remote` adds the auxiliary process to the process at that node. In both rules, the process `PP` reflects the possible existence of a parallel process to the process that is prefixed with the *out* action.

**Rules for the *eval* action.** The rules

```
crl [eval-self] :
  (S {COUNT} :: {RHO} (eval(SQ) @ L) . SP | PP)
=>
  (S {COUNT} :: {RHO} SP | PP | SQ)
if S := RHO(L) .
```

and

```
crl [eval-remote] :
  (S1 {COUNT1} :: {RHO1} (eval(SQ) @ L) . SP | PP)
  || (S2 {COUNT2} :: {RHO2} P)
=>
  (S1 {COUNT1} :: {RHO1} SP | PP)
  || (S2 {COUNT2} :: {RHO2} P | SQ )
if S2 := RHO1(L) .
```

correspond to rules (3) and (4) of KLAIM's reduction relation (Figure A.1).

**Rules for the *in* action.** The rules

```
crl [in-self] :
  (S {COUNT} :: {RHO} (in(T) @ L) . SP | out(ET1) | PP )
=>
  (S {COUNT} :: {RHO} (SP [ ET1 / ET2 ]) | PP)
if ET2 := T[| T |]RHO /\ S := RHO(L) /\ match(ET1, ET2) .
```

and

```
crl [in-remote] :
  (S1 {COUNT1} :: {RHO1} (in(T) @ L) . SP | PP)
  || (S2 {COUNT2} :: {RHO2} P | out(ET1) )
```

```
=>
  (S1 {COUNT1} :: {RHO1} (SP [ ET1 / ET2 ]) | PP)
  || (S2 {COUNT2} :: {RHO2} P )
if ET2 := T[| T |]RHO1 /\ S2 := RHO1(L) /\ match(ET1, ET2) .
```

correspond to rules (5) and (6) of KLAIM's reduction relation (Figure A.1).

**Rules for the *read* action.** The rules

```
crl [read-self] :
(S {COUNT} :: {RHO} (read(T) @ L) . SP | out(ET1) | PP)
=> (S {COUNT} :: {RHO} (SP [ ET1 / ET2 ]) | out(ET1) | PP)
if ET2 := T[| T |]RHO /\ S := RHO(L) /\ match(ET1, ET2) .
```

and

```
crl [read-remote] :
  (S1 {COUNT1} :: {RHO1} (read(T) @ L) . SP | PP)
  || (S2 {COUNT2} :: {RHO2} P | out(ET1) )
=>
  (S1 {COUNT1} :: {RHO1} (SP[ ET1 / ET2 ]) | PP )
  || (S2 {COUNT2} :: {RHO2} P | out(ET1))
if ET2 := T[| T |]RHO1 /\ S2 := RHO1(L) /\ match(ET1, ET2) .
```

correspond to rules (7) and (8) of KLAIM's reduction relation (Figure A.1).

**Rules for the *newloc* action.** The rule

```
crl [newloc] :
  (site ID {COUNT} :: {RHO} (newloc(LVN)) . SP | PP)
=>
  (site ID {s(COUNT)} :: {RHO} (SP [NEWSITE / LVN]) | PP)
  || (NEWSITE {0} :: {[NEWSITE / self] * RHO} nil)
if NEWSITE := site (qid(string(ID) + "." + string(COUNT, 10))) .
```

corresponds to rule (10) of KLAIM's reduction relation (Figure A.1). It creates a new node with at a fresh site. The fresh site consists of the identifier, i.e., the site of the node that evaluates the *newloc* action, followed by a dot and the current count of children. The rule furhter increases the count of children by one.

**Other rules.** Rules (9), (11), and (12) of KLAIM's reduction relation (Figure A.1) have no corresponding rules in $\mathcal{R}_{\text{KLAIM}}$. Rule (9), which specifies that any subprocess of a node's process can perform a step, is captured by the parallel process `PP` in the rewrite rules. Rule (11), which only allows legal nets to perform a transition, is captured by the conditional membership for legal nets. Finally, rule (12), which specifies how the reduction behaves with respect to structural congruence, is captured by the signature of nets.

In addition to the mapping of the rules of the reduction relation, $\mathcal{R}_{\text{KLAIM}}$ defines the rule

```
crl [process-invocation] :
  (S {COUNT} :: {RHO} (A < PS, LS, ES >) | PP)
=>
  (S {COUNT} :: {RHO}
    def(PID, context)[PS / PVNS][LS / LVNS][ES / VNS] | PP)
if (A definedIn context)
  /\ A( PVNS, LVNS, VNS) := processId(A, context)
  /\ PID := A( PVNS, LVNS, VNS) .
```

to describe the evaluation of process invocations and the rule

```
rl [process-choice] : SP + SQ => SP .
```

to describe the semantics of process choices. Only one rule is needed to define the semantics of the process choice operator because of the operator's associativity and commutativity.

## 3.4. Application of the CINNI calculus

Many formal languages, including KLAIM, use the concept of variables to range over essential entities of the language. More specifically, KLAIM uses variables to range over processes, localities, or expressions. Variables can appear in tuples, sequences, or processes. The semantic rules of KLAIM use substitutions of such variables for the evaluation of *in* and *read* operations, for the instantiation of process identifiers by a process invocation, and to bind a fresh site to the locality variable of a *newloc* operation.

CINNI is a generic calculus of explicit substitutions that contributes a first-order representation of terms which takes variable bindings into account and captures free substitutions [75]. For a given language $L$ and its defining syntax, the instantiation of CINNI for $L$ is denoted by $\text{CINNI}_L$. Stehr tries to stay as close as possible to the standard name notation while at the same time including the canonical representation of the de Bruijn notation [35] as the special case, in which a single name is used. CINNI uses the Berklin notation [27, 28] that unifies indexed and named notations. In the Berklin notation, each variable name $X$ is annotated with an index $i \in \mathbb{N}$ which represents the position of the binder in the term that binds $X_i$. The index $i$ of $X_i$ thereby indicates that the binder that binds the variable $X$ is the $i$th binder to the left of the variable in the term.

**Example 3.5: Berklin notation**
The following example illustrates the Berklin notation. Variable $X_0$ is bound by the second binder, while variable $X_1$ is bound by the first binder in the term.

$$\forall X.\ \forall X.\quad f(X_0)\ \wedge\ f(X_1)$$

CINNI extends a given language with explicit substitutions as shown in equations 3.1, 3.2 and 3.3. The simple substitution $[X := M]$ replaces variable $X_0$ with value $M$ and reduces the index of any other equally named variable $X_{n+1}$ to $X_n$. The shift substitution for variable $X$, $\uparrow_X$, increases the index of variables with the same name $X$. The lifted substitution $\Uparrow_X (S)$ is defined in the equations 3.4, 3.5, and 3.6. It decreases the index of variables with the name $X$, performs the substitution $S$, and finally lifts the variable.

$$[X := M] \qquad \text{(simple substitution)} \tag{3.1}$$

$$\uparrow_X \qquad \text{(shift substitution)} \tag{3.2}$$

$$\Uparrow_X (S) \qquad \text{(lifted subsitution)} \tag{3.3}$$

$$\Uparrow_X (S)X_0 = X_0 \tag{3.4}$$

$$\Uparrow_X (S)X_{n+1} = \uparrow_X (SX_n) \tag{3.5}$$

$$\Uparrow_X (S)Y_n = \uparrow_X (SY_n) \text{ if } X \neq Y \tag{3.6}$$

For each syntactical constructor $f$ of the language $L$, CINNI adds a *syntax-specific equation* which automatically shifts the bound variables in each argument of the constructor. Let $j_{i,1}, \ldots j_{i,m_i}$ be the arguments that are bound by $f$ in argument $i$, then the *syntax specific equation* is defined by:

$$S\, f(P_1, \ldots, P_n) = f(\Uparrow_{P_{j_{1,1}}} (\ldots \Uparrow_{P_{j_{1,m_1}}} (S))P_1, \ldots, \Uparrow_{P_{j_{n,1}}} (\ldots \Uparrow_{P_{j_{n,m_n}}} (S))P_n)$$

**Example 3.6: Application of CINNI on** *SimpleKLAIM*
As an example of how the CINNI calculus can be applied to a formal language, we introduce a subset of the KLAIM coordination language, *SimpleKLAIM*. Let $P$ be a process, $E$ an expression and $X$ a name for an expression variable. Expressions are natural numbers, expression variables, or the sum of two expression using the + operator.

$$\begin{aligned} E ::= {} & n \in \mathbb{N} \\ & \mid X_{i \in \mathbb{N}} \\ & \mid E_1 + E_2 \end{aligned}$$

Processes are either the null process *nil* or the parallel composition of two processes. Additionally, a process can be prefixed by the *out* or *in* actions. The process $in(X).P'$ binds the variable name $X$ in $P'$.

$$\begin{aligned} P ::= {} & nil \\ & \mid P_1 \mid P_2 \\ & \mid out(E).P' \\ & \mid in(X).P' \end{aligned}$$

Processes may communicate according to the following rule:

$$out(V).Q \mid in(X).P \quad \rightarrow \quad Q \mid [X := V]P$$

$\text{CINNI}_{SimpleKLAIM}$ adds the operations and equations for explicit substitutions to *SimpleKLAIM*. Additionally, a *syntax specific equation* is added for each constructor.

$$S(n) = n,\ n \in \mathbb{N} \tag{3.7}$$
$$S(nil) = nil \tag{3.8}$$
$$S(E_1 + E_2) = S(E_1) + S(E_2) \tag{3.9}$$
$$S(P_1 \mid P_2) = S(P_1) \mid S(P_2) \tag{3.10}$$
$$S(out(E).P') = out(SE).(SP) \tag{3.11}$$
$$S(in(X).P') = in(X)(\Uparrow_X (S)P') \tag{3.12}$$

Rules 3.7 and 3.8 eliminate the substitution if it is applied to a natural number or the *nil*-process. The rules 3.9, 3.10, and 3.11 pass substitutions down to subterms. If a substitution is applied to a process $P'$ with the *in* action prefix $in(X).P'$, rule 3.12 enforces that the substitution is lifted to ensure that the substitution is applied to the correct variables.

To show how a reduction in $\text{CINNI}_{SimpleKLAIM}$ is processed, let us consider the following example:

$$in(X).in(X).out(X_0 + X_1).nil \mid out(3).out(4).nil$$

$$\rightarrow [X := 3]in(X).out(X_0 + X_1).nil \mid out(4).nil \tag{3.13}$$
$$\rightarrow in(X).(\Uparrow_X ([X := 3])out(X_0 + X_1).nil) \mid out(4).nil \tag{3.14}$$
$$\rightarrow in(X).(out(\Uparrow_X ([X := 3])(X_0 + X_1)).(\Uparrow_X ([X := 3])nil)) \mid out(4).nil$$
$$\rightarrow in(X).(out(\Uparrow_X ([X := 3])(X_0) + \Uparrow_X ([X := 3])(X_1)).nil) \mid out(4).nil$$
$$\rightarrow in(X).(out(X_0 + \uparrow_X ([X := 3]X_0)).nil) \mid out(4).nil \tag{3.15}$$
$$\rightarrow in(X).(out(X_0 + 3).nil) \mid out(4).nil$$
$$\rightarrow [X := 4](out(X_0 + 3).nil) \mid nil$$
$$\rightarrow (out([X := 4](X_0 + 3)).([X := 4](nil))) \mid nil$$
$$\rightarrow (out(([X := 4]X_0) + ([X := 4]3)).nil) \mid nil$$
$$\rightarrow out(4 + 3).nil \mid nil$$

Step 3.13 of the reduction applies the substitution $[X := 3]$ to the process term $in(X).out(X_0 + X_1).nil$, where the variable $X_0$ is bound to the binder $in(X)$. The substitution does not change bound variables and is only applied to the variable $X_1$. In order to apply the substitution correctly, is is lifted as shown in reduction step 3.14. Finally, step 3.15 applies the lifted substitution which replaces the correct variable, i.e., $X_1$, with the value 3.

### 3.4.1. Implementation of CINNI$_{\text{KLAIM}}$

KLAIM uses variables to range over processes, localities and expressions. In order to be consistent in the notation and to differentiate between the three different types of variables, we introduce the three constructors

```
op x_ : Qid -> VarName [ctor prec 15] .
op u_ : Qid -> LocalityVarName [ctor prec 15] .
op X_ : Qid -> ProcessVarName [ctor prec 15] .
```

for the variable names.

As required by CINNI, we extend names for variables with an index $n \in \mathbb{N}$. The operators

```
op _{_} : ProcessVarName Nat -> ProcessVar [ctor prec 15] .
op _{_} : VarName Nat -> Var [ctor prec 15] .
op _{_} : LocalityVarName Nat -> LocalityVar [ctor prec 15] .
```

define the constructors for variables that range over processes (`ProcessVar`), variables that range over expressions (`Var`), and variables that range over localities (`LocalityVar`).

Additionally, we add a sort for each type of substitution:

```
sorts ProcessSubst ExpressionSubst LocalitySubst .
```

CINNI requires the basic substitution operators to be defined for each of the three substitution types. The operators

```
op [_:=_] : ProcessVarName SyntacticProcess -> ProcessSubst .
op [shift_] : ProcessVarName -> ProcessSubst .
op [lift__] : ProcessVarName ProcessSubst -> ProcessSubst .
op __ : ProcessSubst SyntacticProcess -> SyntacticProcess .
```

```
op [_:=_] : VarName Expression -> ExpressionSubst .
op [shift_] : VarName -> ExpressionSubst .
op [lift__] : VarName ExpressionSubst -> ExpressionSubst .
op __ : ExpressionSubst Expression -> Expression .

op [_:=_] : LocalityVarName Locality -> LocalitySubst .
op [shift_] : LocalityVarName -> LocalitySubst .
op [lift__] : LocalityVarName LocalitySubst -> LocalitySubst .
op __ : LocalitySubst Locality -> Locality .
```

define the constructors for the three types of substitution and an operation that allows the substitution to be concatenated with the corresponding type.

The syntax for localities and expressions does not provide a binding construct for variables. However they may be bound at the processes level, e.g., by a $newloc(u)$ action. Consequently, we have to handle substitutions of locality and expression variables as well as substitutions of process variables at the process level. Keeping this restriction in mind, the *syntax-specific equations* for expression and locality variables are defined in a straightforward manner.

In the following description of substitution operators and *syntax-specific equations*, the variables

```
vars Y Z : Qid .
var EST : ExpressionSubst .
var PST : ProcessSubst .
var SP : SyntacticProcess .
vars E E1 E2 : Expression .
var V : Val .
var N : Nat .
```

are used.

The equations

```
eq [expr-subst0] : ([x Y := E]) x Y{0} = E .
eq [expr-subst1] : ([x Y := E]) x Y{s(N)} = x Y{N} .
ceq [expr-subst2] : ([x Y := E] x Z{N}) = x Z{N} if Z =/= Y .
eq  [expr-subst3] : EST V = V .

eq [expr-subst-shift0] : ([shift x Y] x Y{N}) = x Y{s(N)} .
ceq [expr-subst-shift1] : ([shift x Y] x Z{N}) = x Z{N} if Z =/= Y .

eq [expr-subst-lift-base] : ([lift (x Y) EST] x Y{0}) = x Y{0} .
eq [expr-subst-lift-rec0] : ([lift (x Y) EST] x Y{s(N)})
  = [shift x Y] (EST x Y{N}) .
ceq [expr-subst-lift-rec1] : ([lift (x Y) EST] x Z{N})
  = [shift x Y] (EST x Z{N}) if Z =/= Y .

eq [expr-subst-addition] : EST (E1 +e E2) = (EST E1) +e (EST E2) .
```

define the behavior of the CINNI substitution operator and the *syntax-specific equations* for expressions. The corresponding equations for locality variables are defined similarly and are not shown here. Equation `expr-subst3` handles substitutions that are applied to values. Equation `expr-subst-addition` passes the substitution down to the subterms. The other equations depict the implementation of the CINNI behavior for substitutions.

In order to handle substitutions at the process level, we introduce the sort `Substitution` as a supersort of the sorts for the three types of substitution.

```
sort Substitution .
subsort ProcessSubst ExpressionSubst LocalitySubst < Substitution .
```

In KLAIM, substitutions occur at three different occasions. First, an evaluated tuple in a process may be replaced by another evaluated tuple. This happens when a process $in(T)@L.P$ or $read(T)@L.P$ synchronizes with a corresponding auxiliary process $out(ET)$ at the node located at locality $L$. Then, the evaluation of the tuple $T$, $\mathcal{T}[\![T]\!]$, in $P$ is replaced by the evaluated tuple $ET$. Syntactically, the substitution is denoted by process, expression, and locality variables in the defining process, which are substituted for the processes, expressions, and localities in the corresponding sequences of the invocation. Last, the evaluation of a process $P$ that is prefixed with a $newloc(u)$ action substitutes the locality variable $u$ in $P$ with the fresh site generated by the action prefix.

In summary, at the highest level, substitutions in KLAIM are applied to processes and to sequences of processes, localities, or expressions. Additionally, tuples that may occur in *read*, *in*, *out* or *eval* actions have to be taken into account for the substitution. To correctly apply a substitution to a term, it has to be passed down to subterms that are likely to be affected by the substitution. The operators

```
op __ : Tuple Substitution -> Tuple .
op __ : SyntacticProcess Substitution -> SyntacticProcess .
op _[_] : ProcessSeq Substitution -> ProcessSeq .
op _[_] : LocalitySeq Substitution -> LocalitySeq .
op _[_] : ExpressionSeq Substitution -> ExpressionSeq .
```

describe the application of substitutions to the different concepts in KLAIM.

For the substitution for process variables we need to specify the following *syntax-specific equations*:

```
eq [process-subst0] : [X Y := SP] X Y{ 0 } = SP .
eq [process-subst1] : [X Y := SP] X Y{s(N)} = X Y{N} .
ceq [process-subst2] : [X Y := SP] X Z{N} = X Z{N} if Z =/= Y .

eq [process-subst-shift0] : ([shift X Y] X Y{N}) = X Y{s(N)} .
ceq [process-subst-shift1] : ([shift X Y] X Z{N}) = X Z{N} if Z =/= Y .

eq [process-subst-lift-base] : ([lift (X Y) PST] X Y{0}) = X Y{0} .
eq [process-subst-lift-rec0] : ([lift (X Y) PST] X Y{s(N)})
  = [shift X Y] (PST X Y{N}) .
ceq [process-subst-lift-rec1] : ([lift (X Y) PST] X Z{N})
  = [shift X Y] (PST X Z{N}) if Z =/= Y .

eq (PST SP) = SP [owise] .
```

These are all such equations because process substitutions are only applied to process terms that cannot be decomposed any further. The operations for the sort `Substitution` take care of of all types of substitutions in a unified way.

**Example 3.7: Substitution for an expression**
Let us consider the the following process:

$$(in(!x)@self.in(!x)@self.out(x_0 + x_1)@self.nil) \mid out(7)$$

The $in(!x)$ actions bind the variables $x$ in the subsequent processes. If a first synchronization rule is applied, the process evolves to

$$in(!x)@self.out(x_0 + x_1)@self.nil$$

and the substitution $[7/x]$ is applied to the resulting process.

In terms of our Maude specification, the substitution `[x 'x := [7]]` that is of the sort `ExpressionSubst < Substitution` is appended to the process and yields the following term:

```
(in(!x 'x)@self.out(x 'x {0} +e x 'x {1})@self.nil)) [x 'x := [7]]
```

The CINNI rules lift the substitution because of the *in* action:

```
(in(!x 'x)@self.out((x 'x {0} +e x 'x {1}) (lift x' x ([x 'x := [7]])))@self.nil)
```

Finally, the substitution is passed down to the expression inside the *out* action and the substitution for terms of the sort `ExpressionSubst` is applied to the term:

```
(in(!x 'x)@self.out( x 'x {0} +e x [7])@self.nil)
```

### Top level substitutions

In the following description of the behavior of top level substitutions, the variables

```
var A Y : Qid .
var PS : ProcessSeq .
var LS : LocalitySeq .
var ES : ExpressionSeq .
var LST : LocalitySubst .
var EST : ExpressionSubst .
var PST : ProcessSubst .
var S : Substitution .
vars SP SQ : SyntacticProcess .
var T : Tuple .
var TE : TupleElt .
var L : Locality .
var RHO : AllocationEnvironment .
var PV : ProcessVar .
var LV : LocalityVar .
```

are used.

The equations

```
eq [klaim-subst-processinvocation-locality] :
  (A < PS, LS, ES >) LST = A < (PS [LST]), (LS [LST]), ES > .
eq [klaim-subst-processinvocation-expression] :
  (A < PS, LS, ES >) EST = A < (PS [EST]), LS, (ES [EST]) > .
eq [klaim-subst-processinvocation-process] :
  (A < PS, LS, ES >) PST = A < (PS [PST]), LS, ES > .

eq [klaim-subst-out-locality] :
  ((out(T) @ L). SP) LST = (out(T LST) @ (L LST)) . (SP LST) .
eq [klaim-subst-out-owise] :
  ((out(T) @ L). SP) S = (out(T S) @ L) . (SP S) [owise] .

eq [klaim-subst-eval-locality] :
  ((eval(SQ) @ L). SP) LST = (eval(SQ LST) @ (L LST)) . (SP LST) .
```

```
eq [klaim-subst-eval-owise] :
  ((eval(SQ) @ L). SP) S = (eval(SQ S) @ L) . (SP S) [owise] .

eq [choice-composition-substitution] : (SP + SQ) S = (SP S) + (SQ S) .
ceq [parallel-composition-substitution] : (SP | SQ) S = (SP S) | (SQ S)
  if SP =/= nil /\ SQ =/= nil .

eq [closure-substitution-processvar] : (PV {RHO}) S = (PV S) {RHO} .
```

show the handling of substitutions for the process invocation, for the `out` and `eval` action prefixes, for the choice and the parallel composition of two processes and for the process closure. The equations for the process invocation apply the substitution to the affected sequences. The equations for the `out` and `eval` actions differentiate between locality substitutions and other types of substitutions because substitutions of locality variables need also be applied to the location postfix of an action. The equations for the choice and parallel composition pass the substitution down to the subprocesses. Finally, if a substitution is applied to a closed process variable, the substitution is directly applied to the variable. This is an exception to the normal handling of closures.

The equations

```
eq [klaim-subst-in-locality] :
  ((in(T) @ L). SP) LST
    = (in(T LST) @ (L LST)) . (SP liftedSubstitution(T, LST)) .
eq [klaim-subst-in-owise] :
  ((in(T) @ L). SP) S = (in(T S) @ L)
  . (SP liftedSubstitution(T, S)) [owise] .
eq [klaim-subst-read-locality] :
  ((read(T) @ L). SP) LST
    = (read(T LST) @ (L LST)) . (SP liftedSubstitution(T, LST)) .
eq [klaim-subst-read-owise] :
  ((read(T) @ L). SP) S
    = (read(T S) @ L) . (SP liftedSubstitution(T, S)) [owise] .
```

for the *in*, *read* and *newloc* actions differ substantially from the equations for the other actions as they can bind variables in the subsequent process.

The equation

```
eq [klaim-subst-newloc-locality] :
  (newloc(u Y). SP) LST = newloc(u Y) . (SP [lift (u Y) LST]) .
eq [klaim-subst-newloc-owise] :
  (newloc(u Y) . SP) S = newloc(u Y) . (SP S) [owise] .
```

for the *newloc* action lifts the substitution if it is of sort `LocalitySubst`.

The operator

```
op liftedSubstitution : Tuple Substitution -> Substitution .
```

which takes a tuple and a substitution as arguments, computes the correct substitution by lifting the substitution if necessary. The operation recursively decomposes tuples, taking tuple elements, i.e., tuples that cannot be decomposed any further, from the front of the tuple sequence and computes the resulting substitution. The name of the variable is lifted if the substitution is applied to a tuple that binds the corresponding type of variable.

```
eq [liftedSubstitution-base0] : liftedSubstitution(!(X Y), PST)
  = [lift (X Y) PST] .
```

```
eq [liftedSubstitution-base1] : liftedSubstitution(!(u Y), LST)
  = [lift (u Y) LST] .
eq [liftedSubstitution-base2] : liftedSubstitution(!(x Y), EST)
  = [lift (x Y) EST] .
eq [liftedSubstitution-base-owise] : liftedSubstitution(TE, S)
  = S [owise] .
eq [liftedSubstitution-other-tuple-rec] :
  liftedSubstitution((TE, T), S)
  = liftedSubstitution(TE, liftedSubstitution(T, S)) .
```

**Example 3.8: The liftedSubstitution operation**
Let $P$ be process $in(!X, !x, !u, nil, [7]).Q$, and assume that the substitution $[X := nil]$ is applied to $P$. As the *in* action already binds $X_0$ in $Q$, the substitution has to be lifted. The reduction

```
liftedSubstitution(!X 'X, !x 'X, !u 'U, nil, [7], [X 'X:= nil])
liftedSubstitution(!X 'X,
  liftedSubstitution(!x 'X, !u 'U, nil, [7], [X 'X := nil]))
...
liftedSubstitution(!X 'X, [X 'X := nil])
  [lift (X 'X) [X 'X := nil]] .
```

shows the application of the `liftedSubstitution` operation to the example tuple. Besides the first tuple element `!X 'X`, all other tuple elements do not alter the substitution.

Finally, the equations

```
eq [klaim-subst-base-nil] : nil S = nil .
eq [klaim-subst-base-processvar] : PV PST = PST PV .
eq [klaim-subst-base-processvar-owise] : PV S = PV [owise] .
eq [klaim-subst-base-localityvar] : LV LST = LST LV .
eq [klaim-subst-base-localityvar-owise] : L S = L [owise] .
eq [klaim-subst-base-experssionvar] : E EST = EST E .
eq [klaim-subst-base-expressionvar-owise] : E S = E [owise] .
ceq [klaim-subst-tuple-owise] : TE S = TE
  if not(TE :: SyntacticProcess) .
```

specify the base cases of substitutions for tuples. As shown in equation `[klaim-subst-base-nil ]`, any substitution applied to the null process yields the null process. The equations in lines 2 – 7 prepend the substitution to the term if it is of the right type and otherwise discard it. Finally, equation `[klaim-subst-tuple-owise]` says that the substitution is discarded, if it is not applied to a `SyntacticProcess`.

## 3.5. OO-KLAIM — an extension of M-KLAIM for object-oriented specifications

Maude supports modeling of distributed object-based systems in which objects communicate via message passing. In the following, we extend M-KLAIM by adding the object-oriented paradigm to the specification. This extensions allows for a more natural specification of cloud-based architectures.

### 3.5.1. Object-based programming in Maude

The predefined module *CONFIGURATION* supports modelling of object-based systems in Core Maude. Terms of the sort `Configuration` consist of objects and messages and can be thought of as a soup. More specifically, a configuration is a multiset of objects (defined by the sort `Object`) and messages (defined by the sort `Msg`) that describe a possible system state. To address objects, an object's first argument is usually an object identifier that is unique in the system. Object identifiers are terms of the sort `Oid`. Messages usually contain such an identifier to address specific objects. An object's state is described by terms of the sort `Attribute`. These attributes are usually, as a set of attributes (defined by the sort `AttributeSet`), part of an object.

In OO-KLAIM, we decided to introduce a slightly modified syntax for objects and configurations. The syntax of OO-KLAIM is described in the following Section and resembles the original syntax of KLAIM. Nonetheless, the definition is based on the module *CONFIGURATION*. Except for the modules *KLAIM-SYNTAX* and *KLAIM-SEMANTICS*, the specification of OO-KLAIM shares all modules with the M-KLAIM specifications defined in Section 3.3.

### 3.5.2. OO-KLAIM syntax

In OO-KLAIM, KLAIM nodes are objects which are identified by their site. To better demonstrate that these objects describe the entities of a distributed system, a new constructor for more specific sites of sort `IPSite` is introduced. An address which represents the physical host of a KLAIM node is specified by a term of sort `Address`. The address itself is made up of an IP address (a term of sort `String`) and a port (a term of sort `Nat`).

```
sort Address .
op _:_ : String Nat -> Address [ctor prec 5] .
```

Finally, a term of sort `IPSite` is constructed by an address and an identifying term of sort `Qid`. This additional term allows for the specification of systems where more than one KLAIM node is addressed by the same IP address and port (e.g. this way a multi-threaded application can be modeled).

```
sort IPSite .
subsorts IPSite < Oid Site .
op _#_ : Address Qid -> IPSite [ctor prec 10] .
```

The constructor for OO-KLAIM nodes slightly differs from the constructor for nodes in M-KLAIM. As aforementioned, a more specific site which includes an IP address and a port is used as a node's site. Additionally, an OO-KLAIM node is of sort `Object`.

```
op (_{_}::{_}_) : IPSite Nat AllocationEnvironment Process
  -> Object [ctor object] .
```

In OO-KLAIM, a KLAIM net corresponds to a configuration. To better reflect the syntax of the original KLAIM specification, the concatenation operator for configurations `__` is renamed to `_||_`. It is of note that the conditional membership which determines if a KLAIM net is a valid net is omitted in the object-oriented specification. Instead, the module *INITIALCHECK*, which is not shown here, provides an operator `legalNet` which determines if an object-configuration forms a valid KLAIM net.

The OO-KLAIM specification uses messages for the communication between nodes. These messages are syntactically reflected by terms of the sort `Msg`, which are made up of an object identifier which represents the addressed object and message contents of sort `MsgContents`. In OO-KLAIM, the configuration that forms a KLAIM net does not only contain KLAIM nodes but also the messages that nodes use to communicate with each other. This configuration can be thought of as a soup of objects and messages.

```
sort MsgContents .
op msg(_,_) : Oid MsgContents -> Msg [ctor message] .
```

Message contents exist for the *out*, *eval*, *read*, and *in* actions respectively. The message contents to request a tuple using a *read* or *in* action include the evaluated tuple to match with and an object identifier to send to response to. The response contains a matched tuple in addition to the object identifier the response is from.

```
op remote-out(_) : EvaluatedTuple -> MsgContents [ctor] .
op remote-eval(_) : SyntacticProcess -> MsgContents [ctor] .
op readRequest(_,_) : Oid EvaluatedTuple -> MsgContents [ctor] .
op readResponse(_,_) : Oid EvaluatedTuple -> MsgContents [ctor] .
op inRequest (_,_) : Oid EvaluatedTuple -> MsgContents [ctor] .
op inResponse (_,_) : Oid EvaluatedTuple -> MsgContents [ctor] .
```

At the process level, the OO-KLAIM specification adds two auxiliary actions, `blockRead` and `blockIn`. These actions are placeholders to block the continuation of a process that is waiting for a response from *read* or *in* actions that address remote KLAIM nodes. Both auxiliary actions carry the tuple the process is waiting for, and an object identifier, which represents the object the response is expected to arrive from.

```
op blockRead : Tuple Oid -> Action [ctor] .
op blockIn : Tuple Oid -> Action [ctor] .
```

### 3.5.3. OO-KLAIM semantics

OO-KLAIM's semantics differs from M-KLAIM's semantics in the rules where two nodes are involved and in the rule to create a new node with a fresh site. In the following description of the rules of the OO-KLAIM semantics, the variables

```
var COUNT PORT : Nat .
var RHO : AllocationEnvironment .
var PP : Process .
vars SP SQ : SyntacticProcess .
var L : Locality .
vars S S1 S2 NS : IPSite .
vars ET ET1 ET2 : EvaluatedTuple .
var T : Tuple .
var ID : Qid .
var IP : String .
```

are used.

The semantic rule for an *out* action which addresses a remote node consumes the *out* action and puts a new message into the configuration. The message contains the receiving node's site and the tuple of the *out* action.

```
crl [out-remote-produce] :
```

```
  (S1 {COUNT}::{RHO} (out(T) @ L) . SP | PP)
=>
  (S1 {COUNT}::{RHO} SP | PP) || msg(S2, remote-out(T[| T |]RHO))
if S2 := RHO(L) /\ S2 =/= S1 .
```

A node consumes a message that is addressed to it and has the `remote-out` message contents in it by putting the evaluated tuple that comes with the message in its tuple space.

```
rl [out-remote-consume] :
  (S {COUNT}::{RHO} PP) || msg(S, remote-out(ET))
=>
  (S {COUNT}::{RHO} PP | out(ET)) .
```

Similarly to the rule for a remote *out* action, the semantic rule for an *eval* action consumes the action and puts a new message with the syntactic process that should be evaluated and the remote node's site into the configuration.

```
crl [eval-remote-produce] :
  (S1 {COUNT}::{RHO} (eval(SQ) @ L) . SP | PP)
=>
  (S1 {COUNT}::{RHO} SP | PP)
    || msg(S2, remote-eval(SQ))
if S2 := RHO(L) /\ S2 =/= S1 .
```

A node consumes a message that is addressed to it and has `remote-eval` message contents in it by appending the syntactic process that comes with the message to its process.

```
rl [eval-remote-consume] :
  (S {COUNT}::{RHO} PP)
    || msg(S, remote-eval(SP))
=>
  (S {COUNT}::{RHO} PP | SP) .
```

It is of note that the process behavior of M-KLAIM and OO-KLAIM processes is slightly different. In M-KLAIM, if a remote *out* or *eval* action address a node which is not in the KLAIM net, the process that is prefixed by this action cannot proceed. In OO-KLAIM, however, the definition of the rules for the remote *out* and *eval* actions allow for a process to proceed even in the case when the action prefixing the process addresses a node that does not exist in the net. This semantic variation is not due to technical limitations, since an object-oriented specification that exactly captures the original semantics could be given. However, we decided to introduce this semantic variation from the original specification to achieve greater flexibility in regard to the design of distributed systems.

KLAIM's *in* and *read* actions are blocking actions, i.e., a process which is prefixed by such an action can only proceed if a tuple that matches the tuple of the *in* or *read* action is found. In the following we discuss OO-KLAIM's semantics rules for a remote *in* action. The rules for the remote *read* action are similar to the rules for the remote *in* action and are omitted here. A remote *in* action is consumed by a KLAIM node by putting a request message which addresses the remote node into the configuration. The message contains the tuple of the *in* action and the site of the node that processed the *in* action. To simulate the blocking behavior, the node's process is prefixed by a blockIn action that contains the node's site the message is sent to and the tuple of the in action.

```
crl [in-remote-request] :
  (S1 {COUNT}::{RHO} (in(T) @ L) . SP | PP)
```

```
=>
  (S1 {COUNT}::{RHO} blockIn(ET, S2) . SP | PP)
    || msg(S2, inRequest(S1, ET))
if ET := T[| T |]RHO /\ S2 := RHO(L) /\ S2 =/= S1 .
```

A node consumes a message that is addressed to it and contains an `inRequest` if a tuple that matches the tuple of the message is present in its tuple space by putting a response message that contains the found tuple into the configuration. The message also contains the node's site that processed the message and is addressed to the node where the request came from.

```
crl [in-remote-response] :
  (S2 {COUNT}::{RHO} SP | PP | out(ET1))
    || msg(S2, inRequest(S1, ET2))
=>
  (S2 {COUNT}::{RHO} SP | PP)
    || msg(S1, inResponse(S2, ET1))
if match(ET1, ET2) .
```

A node consumes a message that is addressed to it and contains an `inResponse` by matching the information of the sender and the tuple that come with the response with the information of a `blockIn` action in its process. If the tuples match, the received tuple is substituted for the tuple of the `blockIn` action in the process that the `blockIn` action prefixes. It is necessary for the receiving node to match the tuples, because a node can send two requests with non-matching tuples to the same remote node.

```
crl [in-remote-consume] :
  (S1 {COUNT}::{RHO} blockIn(ET1, S2) . SP | PP)
    || msg(S1, inResponse(S2, ET2))
=>
  (S1 {COUNT}::{RHO} SP [ ET2 / ET1 ] | PP)
if match(ET1, ET2) .
```

The rule to process a *newloc* action in OO-KLAIM constructs a fresh site using the parent's IP address, port, and the identifier attached with the parent's counter of children. A new node at the constructed fresh site is then added to the configuration. The node encodes the information about its parent in its site. Other than that, the implicitly introduced hierarchy plays no role in the definition of the semantics of OO-KLAIM.

```
crl [newloc] :
  (IP : PORT # ID {COUNT}::{RHO} newloc(LVN) . SP | PP)
=>
  (IP : PORT # ID {s(COUNT)}::{RHO} (SP [ NS / LVN ]) | PP)
    || (NS {0}::{[NS / self] * RHO} nil)
if NS := IP : PORT # (qid(string(ID) + "." + string(COUNT, 10))) .
```

## 3.6. D-KLAIM — an extension of OO-KLAIM for distributed specifications

D-KLAIM is an extension of OO-KLAIM that allows for specifications to be executed in a distributed environment. In essence, the D-KLAIM extension allows for multiple instances of Maude to execute specifications based on OO-KLAIM. The OO-KLAIM specification

instances communicate with each other through sockets which are handled by objects introduced in D-KLAIM. We use Maude's support for rewriting with external objects and the predefined implementation of sockets. In the following, we give an introduction to rewriting with external objects in Maude and show in more detail how D-KLAIM extends the OO-KLAIM specification. To simulate the behavior of a distributed system that communicates using sockets in a single Maude instance, we developed a socket abstraction for D-KLAIM that allows for such systems to be specified and executed. The socket abstraction further allows the model checking of such systems. Finally, we give an example of a Cloud-based architecture based on D-KLAIM.

### 3.6.1. Rewriting with external objects in Maude

For a configuration to communicate with external objects in Maude, the configuration must contain a so-called portal configuration. The default portal is part of the predefined module *CONFIGURATION*.

```
sort Portal .
subsort Portal < Configuration .
op <> : -> Portal [ctor] .
```

An example for external objects are sockets. Currently, Maude supports IPv4 TCP sockets. The predefined module *SOCKET* of the Maude distribution includes the definition of messages to create, close, and interact with sockets. The messages are consumed by the portal configuration which internally then handles the socket communication. Additionally, an object identifier for the external object

```
op socketManager : -> Oid [special (...)]
```

is defined. The `socketManager` object is a factory for socket objects. The operator

```
op socket : Nat -> Oid [ctor] .
```

provides object identifiers for the sockets.

### 3.6.2. D-KLAIM specification overview

Figure 3.8 gives an overview of the D-KLAIM specification. The modules *D-KLAIM-META-TOOLS* and *D-KLAIM-COMMUNICATION* define the core syntax and semantics of D-KLAIM. The theory *SOCKET-INTERFACE* is an abstraction of the socket behavior and is a parameter of the communication module *D-KLAIM-COMMUNICATION*. The module *D-KLAIM* instantiates the communication module with the view *Socket* and can be used as a foundation for specifications that should be executed in a distributed environment. The module *D-KLAIM-ABSTRACTION* instantiates the communication module with the view *Socket-Abstraction* and can be used as a foundation for specifications where model checking should be applicable.

### 3.6.3. D-KLAIM modules

D-KLAIM extends the OO-KLAIM specification with two main modules, *D-KLAIM-META-TOOLS* and *D-KLAIM-COMMUNICATION*. These two modules specify the behavior of socket-based communication. In the following we describe syntax and semantics provided by the modules.

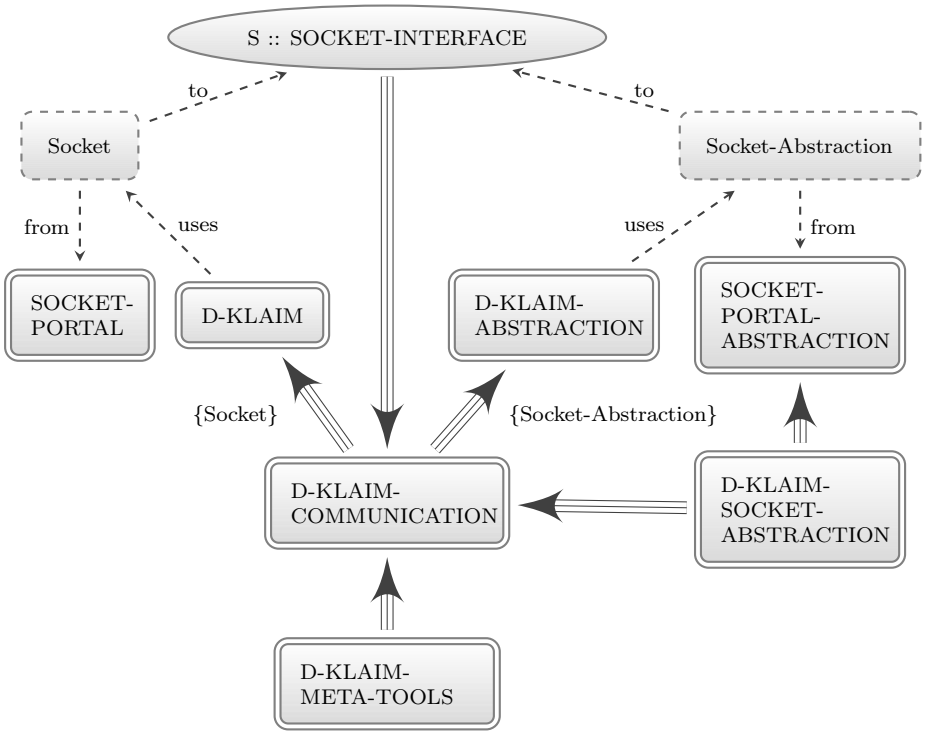



**Figure 3.8.:** D-KLAIM specification overview

### *D-KLAIM-META-TOOLS*

In Maude, terms which are sent through a socket have to be converted to terms of sort `String`. In D-KLAIM, we want to send terms of the sort `Msg`. The module *D-KLAIM-META-TOOLS* specifies the operators `msg2String` and `string2Msg` which convert messages to string representations and vice versa. The variables

```
var N : Nat .
var MSG : Msg .
var Q : Qid .
var QL : QidList .
vars DATA S S' S'' : String .
```

are used for the specification of these operators.

The operator `msg2String` takes a message as argument and calls the meta-level operator `metaPrettyPrint` which, in this case, takes the meta-representation of the module that defines the semantics of OO-KLAIM and the meta-representation of the message term as arguments. The meta-representation of the message term is obtained by calling the `upTerm` operator. `metaPrettyPrint` returns a list of quoted identifiers that meta-represent the string of tokens produced by pretty-printing the message term in the signature of the OO-KLAIM specification. Finally, the operator `qidList2String` converts the list of quoted identifiers to a string representation by concatenating the string representations of each list item separated by a space.

```
op msg2string : Msg -> String [memo] .
eq msg2string(MSG) =
```

```
    qidList2String(metaPrettyPrint(upModule('OO-KLAIM-SEMANTICS, false),
      upTerm(MSG))) .

  op qidList2String : QidList -> String .
  op qidList2StringRec : QidList String -> String .
  eq qidList2String(QL) = qidList2StringRec(QL, "") .
  eq qidList2StringRec(Q, S) = S + string(Q) .
  eq qidList2StringRec(Q QL, S) =
    qidList2StringRec(QL, S + string(Q) + " ") .
```

The operator `string2Message` takes the string representation of a message as an argument and first converts the string to a list of quoted identifiers by calling the operator `string2QidList`. `string2QidList` searches the string for spaces from left to right and extracts the substrings from the string. Each individual string is then turned into a quoted identifier and appended to the resulting list of quoted identifiers. Next, the `metaParse` operator takes the meta-representation of the module that defines the semantics of OO-KLAIM and the list of quoted identifiers as arguments and tries to parse the given list of quoted identifiers as a term of an arbitrary type that is defined in the OO-KLAIM specification. It returns, if successful, a tuple that consists of the meta-representation of the parsed term and its corresponding sort or kind. The `getTerm` operator extracts the meta-representation of the parsed term from the tuple. Finally, the `downTerm` operator takes the meta-representation of the parsed term and the message `error` as arguments. It returns the meta-representation of the canonical form of the term if it is a term in the kind `[Msg]`. Otherwise, it returns the error message `error`.

```
  op error : String -> Msg [ctor] .

  op string2msg : String -> Msg [memo] .
  eq string2msg(S) =
    downTerm(getTerm(metaParse(upModule('OO-KLAIM-SEMANTICS,false),
      string2QidList(S), anyType)), error(S)) .

  op string2QidList : String -> QidList .
  op string2QidListRec : String QidList -> QidList .
  eq string2QidList(S) = string2QidListRec(S, METAnil) .
  ceq string2QidListRec(S, QL) = string2QidListRec(S'', QL qid(S'))
  if N := find(S, " ", 0)
    /\ S' := substr(S, 0, N)
    /\ S'' := substr(S, N + 1, length(S)) .
  eq string2QidListRec(S, QL) = QL qid(S) [owise] .
```

### *D-KLAIM-COMMUNICATION*

The module *D-KLAIM-COMMUNICATION* specifies how socket communication is handled in D-KLAIM. We first define the syntactical elements of the extension. The `communicator` object keeps track of open sockets.

```
  op communicator : Address -> Oid [ctor] .
  op openSockets :_ : AddressList -> Attribute [ctor] .
```

The object accepts incoming sockets and creates additional objects to handle the communication. Regarding outgoing communication, the object keeps track of open sockets and only opens a socket with a destination host, if no socket communication with that host exists.

This way the overhead of creating a new socket for each message that is sent to a destination host can be reduced. The object further closes sockets if they are unused.

The TCP protocol does not preserve message boundaries. Our specifications of sockets therefore relies on a buffering mechanism to provide reliable communication. As we allow multiple messages to be sent through a single socket, we use `"$#"` as a message delimiting sequence. The sequence `"%"` indicates that all messages have been transfered through a socket and that the socket is ready to be closed. Two auxiliary objects are introduced, one on the client side and one on the server side. Both objects are constructed using a simple constructor for objects.

```
op (_::_) : Oid AttributeSet -> Object [ctor object] .
```

On the client side, for each socket, an object keeps track of waiting messages and the current message that is sent through the socket.

```
op waiting :_ : MsgList -> Attribute [ctor] .
op current :_ : Msg -> Attribute [ctor] .
```

On the server side, for each incoming socket connection, a buffer object buffers the incoming data and extracts messages from its buffer.

```
op buffer_ : Oid -> Oid [ctor] .
op buffer :_ : String -> Attribute [ctor] .
```

To describe the behavior of the socket communcation in the D-KLAIM extension, the variables

```
vars LISTENER CLIENT SOCKET : Oid .
var ID : Qid .
vars IP IP1 IP2 DATA S S1 S2 : String .
vars PORT PORT1 PORT2 : Nat .
vars M M1 M2 : Msg .
var ML : MsgList .
var MC : MsgContents .
var N : Nat .
var ATTS : AttributeSet .
var AL : AddressList .
```

are used.

The `startCommunicator` object is a helper object which takes an IP address and a port as arguments. Using this information, it creates the `communicator` object, starts the server-side TCP socket to listen for incoming connections on the specified port, and adds a portal to the configuration. The operator `portal` is described in more detail in Section 3.6.4.

```
op startCommunicator : String Nat -> Object [ctor] .
eq [startCommunicator] : startCommunicator(IP, PORT) =
  (communicator(IP : PORT) :: openSockets : emptyAddressList)
  || createServerTcpSocket(socketManager,
      communicator(IP : PORT), PORT, 5)
  || portal(IP, PORT) .
```

When the `communicator` object is informed that the server-side socket has been created, it starts accepting clients.

```
rl [createdSocket-incoming] :
  (communicator(IP : PORT) :: openSockets : AL, ATTS)
  || createdSocket(communicator(IP : PORT), socketManager, LISTENER)
```

```
=>
  (communicator(IP : PORT) :: openSockets : AL, ATTS)
  || acceptClient(LISTENER, communicator(IP : PORT)) .
```

When the `communicator` object is informed that a client has been accepted, a buffer object for the incoming connection is created. The `communicator` object furthermore starts requesting incoming data from the socket and accepts new incoming client connections.

```
rl [acceptedClient] :
  (communicator(IP1 : PORT1) :: openSockets : AL, ATTS)
  || acceptedClient(communicator(IP1 : PORT1), LISTENER, IP2, CLIENT)
=>
  (communicator(IP1 : PORT1) :: openSockets : AL, ATTS)
  || (buffer(CLIENT) :: buffer : "")
  || receive(CLIENT, communicator(IP1 : PORT1))
  || acceptClient(LISTENER, communicator(IP1 : PORT1)) .
```

Once data is received, the buffer object for the specific incoming connection adds the data to its buffer and, if the sequence `\%`, which indicates that the end of messages, is not found in the data requests more data from the socket.

```
crl [received-message] :
  (buffer(CLIENT) :: buffer : S)
  || received(communicator(IP : PORT), CLIENT, DATA)
=>
  (buffer(CLIENT) :: buffer : (S + DATA))
  || receive(CLIENT, communicator(IP : PORT))
if find(DATA, "%", 0) == notFound .
```

If the terminal symbol `"%"` is found in the incoming data, the data without the terminal symbol is added to the buffer and the socket is closed.

```
crl [received-terminal] :
  (buffer(CLIENT) :: buffer : S)
  || received(communicator(IP : PORT), CLIENT, DATA)
=>
  (buffer(CLIENT) :: buffer : (S + substr(DATA, 0, N)))
  || closeSocket(CLIENT, communicator(IP : PORT))
if find(DATA, "%", 0) =/= notFound
  /\ N := find(DATA, "%", 0) .
```

Buffer objects extract messages from its buffer if the message delimiting sequence is found in the buffer.

```
crl [extract-message] :
  (buffer(CLIENT) :: buffer : S1)
=>
  (buffer(CLIENT) :: buffer : S2)
  || string2msg(substr(S1, 0, N))
if find(S1, "$#", 0) =/= notFound
  /\ N := find(S1, "$#", 0)
  /\ S2 := substr(S1, s(s(N)), length(S1)) .
```

When a buffer object has an empty buffer, i.e., when all messages have been extracted and put into the configuration, and the socket is closed, the `communicator` object removes the buffer from the configuration.

```
rl [closedSocket-incoming] :
```

```
  (communicator(IP : PORT) :: openSockets : AL, ATTS)
  || (buffer(CLIENT) :: buffer : "")
  || closedSocket(communicator(IP : PORT), CLIENT, S)
=>
  (communicator(IP : PORT) :: openSockets : AL, ATTS) .
```

Regarding outgoing communication, the `communicator` object opens a new socket for messages that are designated for a remote host if no socket communication with the remote host already exists. If a new socket is created, a socket helper object is created and put into the configuration.

```
crl [socket-notopen] :
  (communicator(IP1 : PORT1) :: openSockets : AL, ATTS)
  || msg(IP2 : PORT2 # ID, MC)
=>
  (communicator(IP1 : PORT1) ::
    openSockets : ((IP2 : PORT2) // AL), ATTS)
  || (IP2 : PORT2 :: waiting : msg(IP2 : PORT2 # ID, MC))
  || createClientTcpSocket(socketManager, IP2 : PORT2, IP2, PORT2)
if not(contains(IP2 : PORT2, AL))
  /\ not(IP1 == IP2 and PORT1 == PORT2) .
```

A message for a remote host is added to the list of messages waiting to be sent to the host, if a helper object for a socket connection with that host exists in the configuration.

```
crl [socket-open] :
  (IP : PORT :: waiting : ML)
  || msg(IP : PORT # ID, MC)
=>
  (IP : PORT :: waiting : (msg(IP : PORT # ID, MC), ML))
if ML =/= emptyMsgList .
```

When a socket connection with a remote host has been established, the first message of the list of waiting messages to be sent to that host is sent through the socket.

```
rl [createdSocket-outgoing] :
  (IP : PORT :: waiting : (M, ML))
  || createdSocket(IP : PORT, socketManager, SOCKET)
=>
  (IP : PORT :: waiting : ML, current : M)
  || send(SOCKET, IP : PORT, msg2string(M) + "$#") .
```

When a message has been sent through a socket and another message is waiting to be sent to the same host, the next message is sent through the socket.

```
rl [send-next] :
  (IP : PORT :: waiting : (M1, ML), current : M2)
  || sent(IP : PORT, SOCKET)
=>
  (IP : PORT :: waiting : ML, current : M1)
  || send(SOCKET, IP : PORT, msg2string(M1) + "$#") .
```

When a message has been sent through a socket and no more messages are waiting to be sent through the socket, the termination character is sent to indicate that all messages have been sent.

```
op terminal : -> Msg [ctor] .
```

```
crl [send-last] :
  (IP : PORT :: waiting : emptyMsgList, current : M)
  || sent(IP : PORT, SOCKET)
=>
  (IP : PORT :: waiting : emptyMsgList, current : terminal)
  || send(SOCKET, IP : PORT, "%")
if M =/= terminal .
```

When the termination character has been sent, the `communicator` waits for the socket to be closed by the other side.

```
rl [sent-last] :
  (communicator(IP1 : PORT1) :: openSockets : AL, ATTS)
  || (IP2 : PORT2 :: waiting : emptyMsgList, current : terminal)
  || sent(IP2 : PORT2, SOCKET)
=>
  (communicator(IP1 : PORT1) :: openSockets : AL, ATTS)
  || (IP2 : PORT2 :: waiting : emptyMsgList) .
```

When the socket has been closed, the `communicator` knows that the remote host has received all data. It then removes the helper object for the outgoing communication from the configuration and the object's identifier from the list of open sockets. It is of note that in the time between when the termination character has been sent and the socket is closed, no messages can be enqueued in the helper object.

```
rl [closed-socket] :
  (communicator(IP1 : PORT1) :: openSockets : AL, ATTS)
  || (IP2 : PORT2 :: waiting : emptyMsgList)
  || closedSocket(IP2 : PORT2, SOCKET, S)
=>
  (communicator(IP1 : PORT1) ::
    openSockets : remove(IP2 : PORT2, AL), ATTS) .
```

### 3.6.4. The socket interface

The system theory *SOCKET-INTERFACE* declares a module interface for socket communication. The theory defines an operator `portal` which takes a term of sort `String` and a term of sort `Nat` as arguments. The two arguments represent the IP address and the port of the portal's physical location.

```
th SOCKET-INTERFACE is
  protecting STRING .
  protecting CONFIGURATION * (op __ to _||_) .

  op portal : String Nat -> Configuration [ctor] .
endth
```

In D-KLAIM, there are two modules that fulfill the *SOCKET-INTERFACE* theory, *SOCKET-PORTAL* and *SOCKET-ABSTRACTION*. The *D-KLAIM-COMMUNICATION* module requires a parameter that fulfils the *SOCKET-INTERFACE* theory and can be instantiated with the two aforementioned modules. While the *SOCKET-PORTAL* module provides the socket capabilities that come with the Maude distribution and allow for a specification to be executed in a distributed environment, the *SOCKET-ABSTRACTION* module gives an abstraction of Maude's socket behavior and allows for specifications to be

model checked. It is of note that model checking is not possible using Maude's built-in socket capabilities.

### Execution of specifications in a distributed environment

To use Maude's built-in socket capabilities, the module *SOCKET-PORTAL* defines the portal to be the operator `<>` which is part of the predefined module *CONFIGURATION*. Thereby, the IP and port arguments of the `portal` operator are not needed.

```
mod SOCKET-PORTAL is
  protecting STRING .
  protecting CONFIGURATION * (op __ to _||_) .

  var IP : String .
  var PORT : Nat .

  op portal : String Nat -> Configuration .
  eq [portal] : portal(IP, PORT) = <> .
endm

view Socket-Portal from SOCKET-INTERFACE to SOCKET-PORTAL is
  op portal to portal .
endv
```

### The D-KLAIM socket abstraction

As Maude's built-in socket capabilities do not allow for distributed specifications to be model checked, we developed a socket abstraction that captures the behavior of Maude's socket capabilities inside a Maude specification. Using the socket abstraction, distributed specifications of systems that rely on D-KLAIM's socket communication can be specified in a unified specification. The unified specification can then also be model checked. The module *SOCKET-PORTAL-ABSTRACTION* defines the operator `abstractPortal`, which creates an abstract portal that is used by the socket abstraction.

```
mod SOCKET-PORTAL-ABSTRACTION is
  protecting STRING .
  protecting OO-KLAIM-SOCKET-ABSTRACTION .

  var IP : String .
  var PORT : Nat .

  op abstractPortal : String Nat -> Configuration .
  eq [abstract-portal] : abstractPortal(IP, PORT) =
    < socketManager :: ip : IP, port : PORT, state : initialized > .
endm

view Socket-Abstraction from SOCKET-INTERFACE to SOCKET-PORTAL-ABSTRACTION is
  op portal to abstractPortal .
endv
```

In the following, we give an in-depth description of the D-KLAIM socket abstraction. To syntactically reflect a network of individual OO-KLAIM configurations in a single term, the sort `NetworkConfiguration` is defined.

```
sort NetworkConfiguration .
op [_] : Configuration -> NetworkConfiguration [ctor] .
op _||_ : NetworkConfiguration NetworkConfiguration -> NetworkConfiguration
  [ctor config assoc comm] .
```

Figure 3.9 gives an overview of a network configuration in the D-KLAIM socket abstraction.

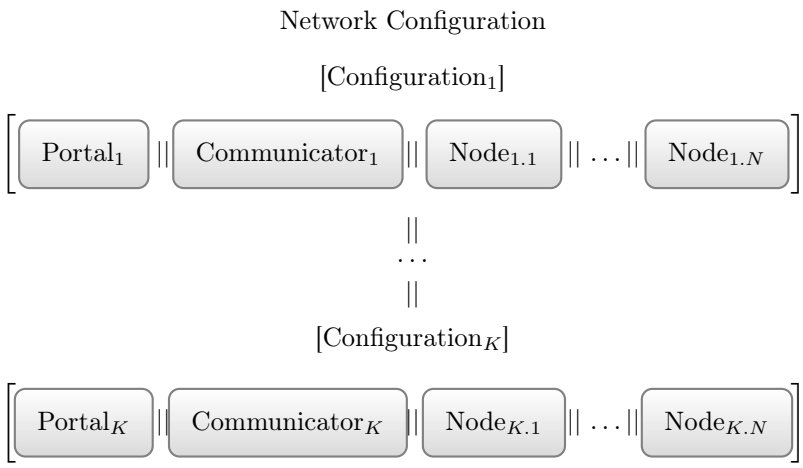



**Figure 3.9.:** Overview of a network configuration in the D-KLAIM socket abstraction

The syntax of the abstract portal resembles the syntax of the portal in the predefined *CONFIGURATION* module. It adds a set of attributes which include the IP address and port of the physical location. The object identifier of the abstract portal is the `socketManager` object identifier which is defined in Maude's *SOCKET* module. The external object with that identifier that usually handles the creation of sockets is thereby made an internal object in the socket abstraction.

```
op <_::_> : Oid AttributeSet -> Configuration [ctor object] .
op acceptor :_ : Oid -> Attribute [ctor gather(&)] .
op ip :_ : String -> Attribute [ctor gather(&)] .
op port :_ : Nat -> Attribute [ctor gather(&)] .
```

In the socket abstraction, sockets are objects that are part of the network configuration. The constructor

```
op (_::_) : Oid AttributeSet -> Configuration [ctor object] .
```

adds a simple constructor for socket objects. Socket objects are identified by socket identifiers which are constructed by the identifiers of the socket's endpoints. The socket abstraction assumes that only one socket between two endpoints exists. Hence the socket identifier is unique.

```
sort Endpoint .
subsort Endpoint < Oid .
op e : String Nat -> Endpoint [ctor] .

sort SocketIdentifier .
subsort SocketIdentifier < Nat .
op id : Endpoint Endpoint -> SocketIdentifier [ctor] .
```

Socket objects store the contents that are sent through the socket, i.e., the contents that are just being transferred, keep track of the server and client endpoints and the object that created the socket.

```
op contents :_ : String -> Attribute [ctor gather(&)] .
op serverEndpoint :_ : Endpoint -> Attribute [ctor gather(&)] .
op clientEndpoint :_ : Endpoint -> Attribute [ctor gather(&)] .
op creator :_ : Oid -> Attribute [ctor gather(&)] .
```

States are used by the socket objects and the abstract portal. The abstract portal can be in the states: initialized, listening, or accepting. Sockets can be in the sates: initialized, receiving, or idle.

```
sort State .
ops initialized listening accepting idle receiving : -> State .
op state :_ : State -> Attribute [ctor gather(&)] .
```

The variables

```
vars C C1 C2 : Configuration .
vars IP IP1 IP2 DATA REASON CONTENTS : String .
vars PORT PORT1 PORT2 BACKLOG : Nat .
var ID : SocketIdentifier .
vars ATTS ATTS1 ATTS2 : AttributeSet .
vars O O1 O2 : Oid .
vars NC NC' : NetworkConfiguration .
var STATE: State .
```

are used for the description of the behavior of the socket abstraction.

External rewrites in Maude happen only if no internal rewrites are possible. To reflect this in the specification of the socket abstraction, the meta-level is used to define the operator `noInternalTransitions`, which checks if rewrites internal to one of the configurations in a network configuration are possible. For each configuration in the network configuration, the operator calls the operator `metaSearch` that takes the meta-representation of the D-KLAIM socket communication semantics module and the meta-representation of the configuration term as arguments. The additional parameters define the search pattern. In our example, `metaSearch` searches if the configuration can be rewritten to another configuration in at least one rewriting step.

```
op noInternalTransitions : NetworkConfiguration -> Bool .
eq noInternalTransitions([C]) =
  metaSearch(upModule('D-KLAIM-COMMUNICATION, false),
    upTerm(C), 'R:Configuration, nil, '+, 1, 0) == failure .
eq noInternalTransitions([C] || NC) =
  if metaSearch(upModule('D-KLAIM-COMMUNICATION, false),
      upTerm(C), 'R:Configuration, nil, '+, 1, 0) == failure
  then noInternalTransitions(NC)
  else false fi .
```

In the following, all rules of the socket abstraction are only applicable, if no internal rewrites are possible.

When one of the participating configurations creates a server-side socket, the abstract portal changes its state from initialized to listening.

```
crl [createServerTcpSocket] :
  [C || createServerTcpSocket(socketManager, O, PORT, BACKLOG)
```

```
    || < socketManager :: ip : IP, port : PORT, state : initialized >]
=>
  [C || createdSocket(O, socketManager, e(IP, PORT))
    || < socketManager :: ip : IP, port : PORT, state : listening >]
if noInternalTransitions([C]) .
```

In case the abstract portal is already in the listening state, a `socketError` message saying that the address is already in use is created.

```
crl [createServerTcpSocket-error] :
  [C || createServerTcpSocket(socketManager, O, PORT, BACKLOG)
    || < socketManager :: ip : IP, port : PORT, state : listening, ATTS >]
=>
  [C || socketError(O, socketManager, "Address already in use")
    || < socketManager :: ip : IP, port : PORT, state : listening, ATTS >]
if noInternalTransitions([C]) .
```

When the configuration accepts a client, the abstract portal changes its state from listening to accepting.

```
crl [acceptClient] :
  [C || acceptClient(e(IP, PORT), O)
    || < socketManager :: ip : IP, port : PORT, state : listening >]
=>
  [C || < socketManager :: ip : IP, port : PORT, state : accepting,
   acceptor : O >]
if noInternalTransitions([C]) .
```

When a client socket is created by one of the configurations and the destination configuration's abstract portal is accepting incoming sockets, the destination configuration is informed that a client has been accepted. Thereby the state of the destination's abstract portal is changed from accepting to listening. Additionally, a socket object is created. The socket's identifier contains the information about the endpoints it connects.

```
crl [createClientTcpSocket] :
  [C1 || createClientTcpSocket(socketManager, O1, IP2, PORT2)
  || < socketManager :: ip : IP1, port : PORT1, ATTS >]
  || [C2 || < socketManager :: ip : IP2, port : PORT2, state : accepting,
       acceptor : O2 >]
=>
  [C1 || createdSocket(O1, socketManager, socket(ID))
  || < socketManager :: ip : IP1, port : PORT1, ATTS >]
  || [C2 || acceptedClient(O2, e(IP2, PORT2), IP1, socket(ID))
     || < socketManager :: ip : IP2, port : PORT2, state : listening >]
  || [socket(ID) :: state : initialized, creator : O1, contents : "",
       clientEndpoint : e(IP1, PORT1), serverEndpoint : e(IP2, PORT2)]
if ID := id(e(IP1, PORT1), e(IP2, PORT2))
  /\ noInternalTransitions([C1] || [C2]) .
```

When a configuration that established a client socket connection tells the socket that it is ready to receive data, the socket's state changes from initialized to receiving.

```
crl [receive-initialized] :
  [C || receive(socket(ID), O)]
  || [socket(ID) :: state : initialized, ATTS]
=>
  [C]
  || [socket(ID) :: state : receiving, ATTS]
```

```
if noInternalTransitions([C]) .
```

Similarly, if a configuration tells a socket that it is ready to receive data and the socket is in the idle state, the socket changes its state to receiving.

```
crl [receive-idle] :
  [C || receive(socket(ID), O)]
  || [socket(ID) :: state : idle, ATTS]
=>
  [C]
  || [socket(ID) :: state : receiving, ATTS]
if noInternalTransitions([C]) .
```

When a configuration sends data to a socket and the socket is in the receiving state, the socket adds this data to its contents.

```
crl [send] :
  [C || send(socket(ID), O, DATA)]
  || [socket(ID) :: state : receiving, contents : CONTENTS, ATTS]
=>
  [C || sent(O, socket(ID))]
  || [socket(ID) :: state : receiving, contents : (CONTENTS + DATA), ATTS]
if noInternalTransitions([C]) .
```

If a socket's contents is non-empty and the socket's state is receiving, the socket passes its contents on to the receiving configuration, i.e., the server endpoint's configuration. The state of the socket thereby changes from receiving to idle.

```
crl [received] :
  [C || < socketManager :: ip : IP2, port : PORT2, acceptor : O, ATTS1 >]
  || [socket(ID) :: state : receiving, contents : CONTENTS,
       serverEndpoint : e(IP2, PORT2), ATTS2]
=>
  [C || < socketManager :: ip : IP2, port : PORT2, acceptor : O, ATTS1 >
  || received(O, socket(ID), CONTENTS)]
  || [socket(ID) :: state : idle, contents : "",
       serverEndpoint : e(IP2, PORT2), ATTS2]
if CONTENTS =/= ""
  /\ noInternalTransitions([C]) .
```

When a configuration closes a socket, the client and the server endpoints' configurations are informed about the closed socket and the socket is removed from the network configuration.

```
crl [closeSocket] : [C2 || closeSocket(socket(ID), O1)]
  || [C1 || < socketManager :: ip : IP1, port : PORT1, ATTS >]
  || [socket(ID) :: state : STATE, creator : O2,
       clientEndpoint : e(IP1, PORT1), serverEndpoint : e(IP2, PORT2), ATTS1]
=>
  [C2 || closedSocket(O1, socket(ID), "")]
  || [C1 || < socketManager :: ip : IP1, port : PORT1, ATTS >
     || closedSocket(O2, socket(ID), "")]
if noInternalTransitions([C1] || [C2]) .
```

### 3.6.5. Example of a Cloud-based architecture specification based on D-KLAIM

In this Section, we show how a Cloud-based architecture can be specified based on D-KLAIM. We consider the following example: a service developer develops a Fibonacci service. The service should provide high scalability and availability and is started in the Cloud. A consumer calls the service in the Cloud and stores the incoming Fibonacci numbers in a local storage.

In the D-KLAIM-based specification of this example, each participating entity, the developer, the consumer, and the server in the Cloud are modeled as KLAIM nets. Initially, each net contains a KLAIM node, a communicator object, and a portal. Figure 3.10 provides an overview of the Fibonacci Cloud service architecture.

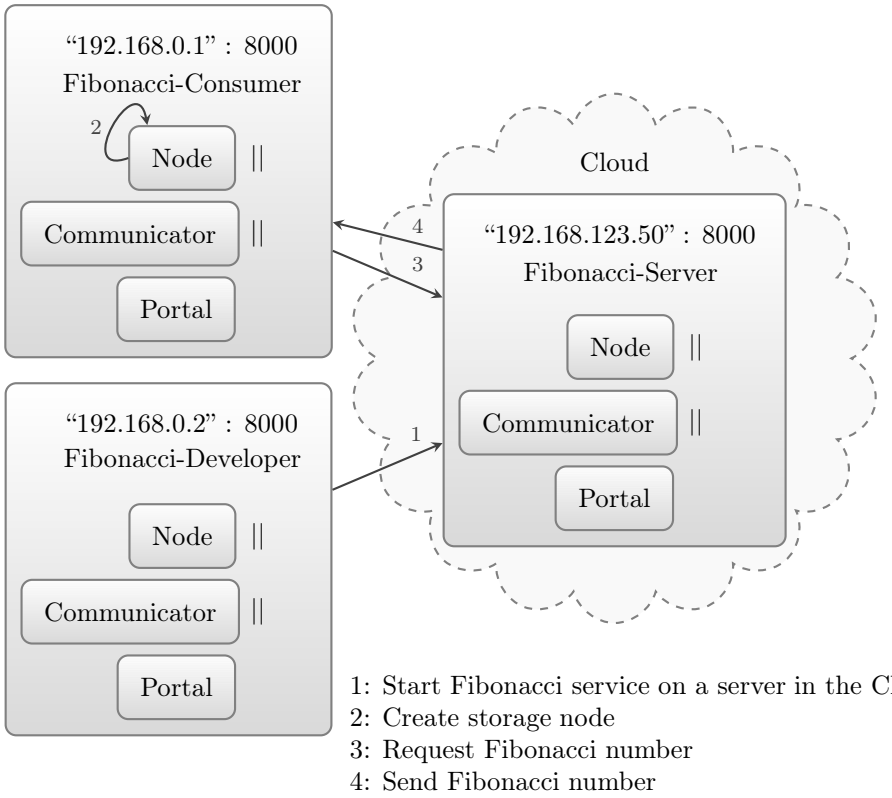



**Figure 3.10.:** Overview of the Fibonacci Cloud service architecture

We specify the behavior of the server and the consumer in two separate process contexts. The process context of the server provides the process definitions `'Fib` and `'FibRec`, which produce a continuous series of Fibonacci numbers upon client requests sending the produced numbers back to the clients.

```
eq Context =
  'Fib (nilProcessVarNameSeq, nilLocalityVarNameSeq, nilVarNameSeq)
    =def in(! u 'Client)@ self
         . out([0])@ u 'Client{0}
         . out([0], [1])@ self
```

```
        . 'FibRec < nilProcessSeq, nilLocalitySeq, nilExpressionSeq > &
  'FibRec (nilProcessVarNameSeq, nilLocalityVarNameSeq, nilVarNameSeq)
    =def in(! u 'Client)@ self
        . in(! x 'f1, ! x 'f2)@ self
        . out(x 'f1{0} +e x 'f2{0})@ u 'Client{0}
        . out(x 'f1{0} +e x 'f2{0}, x 'f1{0})@ self
        . 'FibRec < nilProcessSeq, nilLocalitySeq, nilExpressionSeq > .
```

The process context of the consumer provides the `'ConsumeFibRec` process definition, which consumes an incoming Fibonacci number by storing it in a local storage. After that, it requests the next number from the server.

```
eq Context =
  'ConsumeFibRec (nilProcessVarNameSeq, (u 'Store ; u 'Cloud), nilVarNameSeq)
    =def in(! x 'Fib)@ self . out(x 'Fib{0})@ u 'Store{0}
         . out(self)@ u 'Cloud{0}
         . ('ConsumeFibRec < nilProcessSeq, (u 'Store{0} ; u 'Cloud{0}),
            nilExpressionSeq >) .
```

The initial configuration of the server consists of no more than a plain KLAIM node with the `nil` process. This resembles a typical situation in Cloud Computing where new resources are often virtual machines that provide no more than an operating system.

```
startCommunicator("192.168.123.50", 8000)
|| ("192.168.123.50" : 8000 / '0 {0}::
    {["192.168.123.50" : 8000 / '0 / self]} nil)
```

The initial configuration of the developer consists of a KLAIM node with a process that starts the Fibonacci service on the server in the Cloud.

```
startCommunicator("192.168.0.2", 8000)
|| ("192.168.0.2" : 8000 / '0 {0}::
    {["192.168.0.2" : 8000 / '0 / self]}
    eval(in(! x 'Start)@ self
    . ('Fib < nilProcessSeq, nilLocalitySeq, nilExpressionSeq >))@
       ("192.168.123.50" : 8000 / '0) . nil |
    out([0])@ ("192.168.123.50" : 8000 / '0) . nil)
```

Finally, the initial configuration of the consumer consists of a KLAIM node with a process that first creates another node as a storage node and then starts requesting Fibonacci numbers from the server in the Cloud.

```
startCommunicator("192.168.0.1", 8000)
|| ("192.168.0.1" : 8000 / '0 {0}::
    {["192.168.0.1" : 8000 / '0 / self]}
    newloc(u 'Store)
    . out(self)@ ("192.168.123.50" : 8000 / '0)
    . ('ConsumeFibRec < nilProcessSeq,
       (u 'Store{0} ; ("192.168.123.50" : 8000/ '0)),
       nilExpressionSeq >))
```

The specification can now be executed on distributed machines using three Maude instances and the `erew` command.

The example of the Cloud-based Fibonacci service shows that a Cloud-based architecture can easily be specified at a high level based on D-KLAIM. Furthermore, the specification can be executed in a distributed environment. This opens the possibility of using D-KLAIM and the Maude system as a rapid prototyping environment for Cloud-based architectures.

## 3.7. Maude-based formal analysis of *-KLAIM

In the following we demonstrate how specifications based on *-KLAIM (M-KLAIM, OO-KLAIM, and D-KLAIM) can be formally analyzed using the Maude LTL model checker and the Maude search command.

### 3.7.1. Maude LTL model checking

Maude supports on-the-fly explicit state linear temporal logic (LTL) model checking of concurrent systems [76, 33]. Both, the system specification and the property specification are given in Maude. The Maude LTL model checker can be used to prove properties such as safety properties (something bad never happens) and liveness properties (something good will eventually happen) when the set of states that are reachable from the initial state of a system module is finite. The operators needed for the specification of model checking in Maude are specified in the predefined module *MODEL-CHECKER*.

### 3.7.2. A *-KLAIM-based token-based mutual exclusion algorithm

This example demonstrates how a token-based mutual exclusion algorithm similar to the synchronization model proposed in Example 3.1 can be specified and analyzed in *-KLAIM. The goal is to give an executable specification of the algorithm and to model check if the algorithm fulfills the mutual exclusion property and provides strong liveness guarantees.

The KLAIM net consists of three KLAIM nodes: one token server and two consumers. A token exists in the net and is represented by the value `[0]`. It is, if available for consumption, present as a tuple in the tuple space of the token server. The consumers can bid for the token by requesting it from the token server. In case the token is available in the tuple space of a consumer, i.e., the token was transferred from the token server to the consumer, the consumer enters its critical section by changing the value of the token. A consumer is defined to be in a critical section if it holds the tuple `[1]` in its tuple space. A consumer leaves the critical section when it consumes the tuple `[1]` and puts back the token tuple `[0]` into the tuple space of the token server. In the following, we specify the process context, which includes the process definitions to request a token from the token server (`'Request`) and to enter and exit a critical section (`'Enter` and `'Exit`).

```
ops tokenServer consumer1 consumer2 : -> Site [ctor] .
eq context =
  ('Request (nilProcessVarNameSeq, nilLocalityVarNameSeq, nilVarNameSeq)
    =def in(! x 'Token)@ tokenServer
      . out(x 'Token{0})@ self
      . 'Enter < nilProcessSeq, nilLocalitySeq, nilExpressionSeq >) &
  ('Enter (nilProcessVarNameSeq, nilLocalityVarNameSeq, nilVarNameSeq)
    =def in(! x 'Token)@ self
      . out([1])@ self
      . 'Exit < nilProcessSeq, nilLocalitySeq, nilExpressionSeq >) &
  ('Exit (nilProcessVarNameSeq, nilLocalityVarNameSeq, nilVarNameSeq)
    =def in(! x 'Token)@ self
      . out([0])@ tokenServer
      . 'Request < nilProcessSeq, nilLocalitySeq, nilExpressionSeq >) .
```

The specification of the process context is shared by the specifications of *-KLAIM. Thus, the aforementioned defining equation of the process context for the mutual exclusion algorithm example is valid for all the specifications of the algorithm that we will present in this section.

### Defining mutual exclusion and strong liveness

Mutual exclusion and strong liveness are two desirable properties of a mutual exclusion algorithm. In our example, mutual exclusion means that the two consumers are not in their critical sections simultaneously. Strong liveness means that if a consumer requests the token at certain point in time, the consumer eventually gets the token in order to enter its critical section. We first define two auxiliary properties, `critical` and `requesting`, which, for a given site, state if the node at the specified site is in its critical section or is requesting the token from the token server. Mutual exclusion is then defined by the LTL formula

```
[] ~(critical(consumer1) /\ critical(consumer2))
```

and strong liveness of the consumers is defined by the LTL formulas

```
([]<> requesting(consumer1)) -> ([]<> critical(consumer1))
```

and

```
([]<> requesting(consumer2)) -> ([]<> critical(consumer2)) .
```

### M-KLAIM-based model checking

In the specification based on M-KLAIM, model checking states are of sort `Net`.

```
subsort Net < State .
```

We first specify the properties `critical` and `requesting` based on M-KLAIM.

```
ops critical requesting : Site -> Prop .

var S : Site .
var N : Net .
var AE : AllocationEnvironment .
var P : Process .
var PR : Prop .

eq (S {0}::{AE} out([1]) | P) || N
  |= critical(S) = true .
eq (S {0}::{AE}
    'Request < nilProcessSeq, nilLocalitySeq, nilExpressionSeq >) || N
  |= requesting(S) = true .
eq N |= PR = false [owise] .
```

Next, we specify the initial state for the model checking of the mutual exclusion algorithm based on M-KLAIM.

```
eq tokenServer = site 'TokenServer .
eq consumer1 = site 'Consumer1 .
eq consumer2 = site 'Consumer2 .

op initial : -> Net .
```

```
eq initial =
  (tokenServer {0}::{[tokenServer / self]} out([0])) ||
  (consumer1 {0}::{[consumer1 / self] * [tokenServer / 'TokenServer]}
    'Request < nilProcessSeq, nilLocalitySeq, nilExpressionSeq >) ||
  (consumer2 {0}::{[consumer1 / self] * [tokenServer / 'TokenServer]}
    'Request < nilProcessSeq, nilLocalitySeq, nilExpressionSeq >) .
```

Model checking of the mutual exclusion property of the algorithm based on M-KLAIM is achieved by giving the command

```
Maude> red
  modelCheck(initial, []~ (critical(consumer1) /\ critical(consumer2))) .
reduce in KLAIM-MUTEX-CHECK :
  modelCheck(initial, []~ (critical(consumer1) /\ critical(consumer2))) .
rewrites: 1016 in 0ms cpu (3ms real) (1395604 rewrites/second)
result Bool: true
```

which shows that the property holds. Model checking of the strong liveness properties is achieved by giving the commands

```
Maude> red
  modelCheck(initial, []<> requesting(consumer1) -> []<> critical(consumer1)) .
reduce in KLAIM-MUTEX-CHECK :
  modelCheck(initial, []<> requesting(consumer1) -> []<> critical(consumer1)) .
rewrites: 723 in 2ms cpu (2ms real) (263100 rewrites/second)
result [ModelCheckResult]: counterexample(...)
```

and

```
Maude> red
  modelCheck(initial, []<> requesting(consumer2) -> []<> critical(consumer2)) .
reduce in KLAIM-MUTEX-CHECK :
  modelCheck(initial, []<> requesting(consumer2) -> []<> critical(consumer2)) .
rewrites: 896 in 3ms cpu (4ms real) (288566 rewrites/second)
result [ModelCheckResult]: counterexample(...)
```

which show that the liveness properties do not hold. The counterexamples, which are omitted for reasons of brevity, show that each one of the consumers can starve, i.e., for each consumer a looping path of transitions exists where in each intermediate state the property `critical` does not hold for the consumer. Figure A.2 shows a graphical representation of the counterexample for the liveness of `consumer2`.

### OO-KLAIM-based model checking

In the specification based on OO-KLAIM, model checking states are of sort `Configuration`.

```
subsort Configuration < State .
```

We first specify the auxiliary properties `critical` and `requesting` based on OO-KLAIM.

```
ops critical requesting : Site -> Prop .

var S : Site .
var C : Configuration .
var AE : AllocationEnvironment .
var P : Process .
var PR : Prop .
```

```
eq (S {0}::{AE} out([1]) | P) || C
  |= critical(S) = true .
eq (S {0}::{AE}
     'Request < nilProcessSeq, nilLocalitySeq, nilExpressionSeq >) || C
  |= requesting(S) = true .
eq C |= PR = false [owise] .
```

What follows is the definition of the initial state for the model checking of the mutual exclusion algorithm based on OO-KLAIM.

```
eq tokenServer = "127.0.0.1" : 8000 # '0 .
eq consumer1 = "127.0.0.1" : 8000 # '1 .
eq consumer2 = "127.0.0.1" : 8000 # '2 .

op initial : -> Configuration .
eq initial =
  (tokenServer {0}::{[tokenServer / self]} out([0])) ||
  (consumer1 {0}::{[consumer1 / self] * [tokenServer / 'TokenServer]}
    'Request < nilProcessSeq, nilLocalitySeq, nilExpressionSeq >) ||
  (consumer2 {0}::{[consumer1 / self] * [tokenServer / 'TokenServer]}
    'Request < nilProcessSeq, nilLocalitySeq, nilExpressionSeq >) .
```

Model checking of the mutual exclusion and liveness properties of the algorithm based on OO-KLAIM works as shown for the algorithm based on M-KLAIM. The results are the same, i.e., the mutual exclusion property holds and the strong liveness properties for the consumers do not hold.

### D-KLAIM-based model checking

In order to be able to model check the specification of the algorithm based on D-KLAIM, we use the socket abstraction introduced in Section 3.6.4. Model checking states are thereby of sort `NetworkConfiguration`.

```
subsort NetworkConfiguration < State .
```

We first define of the auxiliary properties `critical` and `requesting` based on D-KLAIM and the socket abstraction.

```
ops critical requesting : Site -> Prop .

var S : Site .
var C : Configuration .
var NC : NetworkConfiguration .
var AE : AllocationEnvironment .
var P : Process .
var PR : Prop .

eq [(S {0}::{AE} out([1]) | P) || C] || NC
  |= critical(S) = true .
eq [(S {0}::{AE}
     'Request < nilProcessSeq, nilLocalitySeq, nilExpressionSeq >) || C]
   || NC
  |= requesting(S) = true .
eq C |= PR = false [owise] .
```

What follows is a definition of the initial state for the model checking of the mutual exclusion algorithm based on D-KLAIM and the socket abstraction.

```
eq tokenServer = "192.168.123.1" : 8000 # '0 .
eq consumer1 = "192.168.123.2" : 8000 # '0 .
eq consumer2 = "192.168.123.3" : 8000 # '0 .

op initial : -> NetworkConfiguration .
eq initial =
   [startCommunicator("192.168.123.1", 8000, none) ||
     (tokenServer {0}::{[tokenServer / self]} out([0]))] ||
   [startCommunicator("192.168.123.2", 8000, none) ||
     (consumer1 {0}::{[consumer1 / self] * [tokenServer / 'TokenServer]}
       'Request < nilProcessSeq, nilLocalitySeq, nilExpressionSeq >)] ||
   [startCommunicator("192.168.123.3", 8000, none) ||
     (consumer2 {0}::{[consumer1 / self] * [tokenServer / 'TokenServer]}
       'Request < nilProcessSeq, nilLocalitySeq, nilExpressionSeq >)] .
```

Model checking of the mutual exclusion and strong liveness properties of the specification based on D-KLAIM and the socket abstraction works just as with the specifications based on M-KLAIM and OO-KLAIM. As additional intermediate states are introduced by the socket abstraction, the model checker now works on a much bigger state space. As a result, the model checking of the individual properties took up to 13 hours, whereas model checking of the same properties on the specifications based on M-KLAIM and OO-KLAIM took seconds on the same machine.

### 3.7.3. Model checking using the Maude search command

Maude's `search` command explores the reachable state space from an initial state for a pattern that has to be reached, possibly subjected to a user-specified semantic condition. The search can be further restricted by the form of the rewriting proof from the initial to the final term. Possible forms are,

- `=>1`, which means that a rewriting proof consisting of exactly one step is searched for.
- `=>+`, which means that a rewriting proof consisting of one or more steps is searched for.
- `=>*`, which means that a proof of none, one, or more steps is searched for.
- `=>!`, which indicates that only canonical final states, i.e., states were no further rewrites are possible, are searched for.

To model check invariants, i.e., predicates that define a set of states that contain all the states reachable from an initial state, with the `search` command, the optional semantic condition, corresponding to the violation of the invariant, is specified. Under some assumptions, which are omitted here for reasons of simplicity, for any invariant $I(x : k)$ and an initial state *init*, $I$ holds if and only if the command

`search` *init* `=>* x:k such that not` $I(x : k)$ `.`

returns no solutions ($k$ is the kind of the term *init*). For an in-depth description of the `search` command we refer to [33].

### 3.7.4. A D-KLAIM-based load balancer

In this example we show how a simple load balancer based on D-KLAIM can be specified. We then analyze the load balancer using the Maude search command.

Four nodes, a producer, a load balancer, and two consumers, form a KLAIM net. Initially, the producer has four tuples in its tuple space. These tuples can be seen as abstractions of work tasks. The producer then puts each tuple into the tuple space of the load balancer. The load balancer consumes tuples in its tuple space and distributes the tuples across the tuple spaces of the consumers in an alternating order. The expected outcome of the scenario is that each consumer is assigned two work tasks, i.e., each consumer ends up with two tuples in its tuple space. Figure 3.11 provides a schematic overview of the load balancer example.

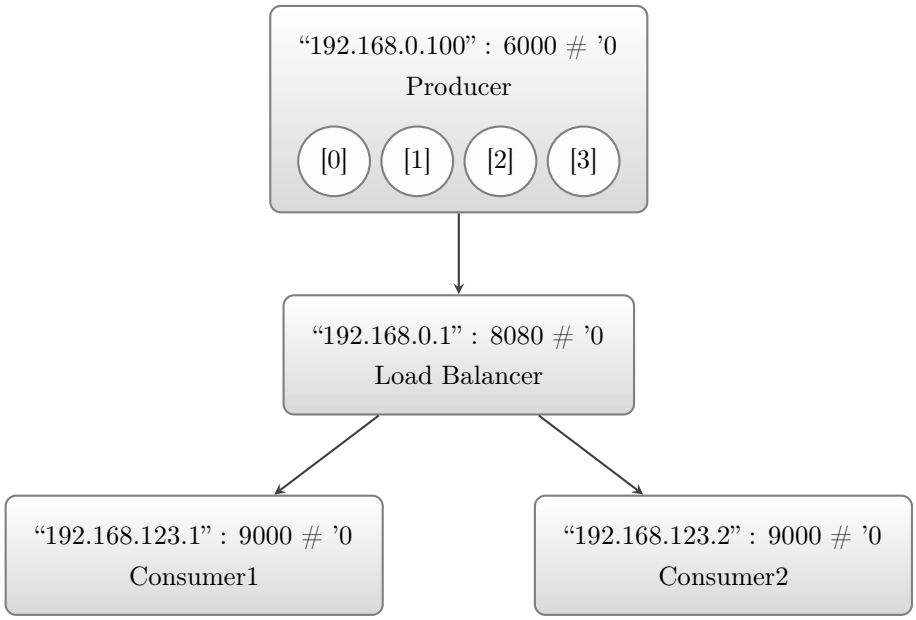



**Figure 3.11.:** Schematic overview of the load balancer example

We first define the initial configuration of the simple load balancer example based on D-KLAIM and the socket abstraction.

```
ops producer loadBalancer consumer1 consumer2 : -> Site .
eq producer = "192.168.0.100" : 6000 # ’0 .
eq loadBalancer = "192.168.0.1" : 8080 # ’0 .
eq consumer1 = "192.168.123.1" : 9000 # ’0 .
eq consumer2 = "192.168.123.2" : 9000 # ’0 .

op loadBalancerExample : -> NetworkConfiguration .
eq loadBalancerExample =
  [startCommunicator("192.168.0.100", 6000, none) ||
    (producer {0}::{[producer / self] * [loadBalancer / ’WorkBalancer]}
      out([0]) | out([1]) | out([2]) | out([3]) |
      in(! x ’X)@ self . out(x ’X {0})@ ’WorkBalancer .
      in(! x ’X)@ self . out(x ’X {0})@ ’WorkBalancer .
      in(! x ’X)@ self . out(x ’X {0})@ ’WorkBalancer .
      in(! x ’X)@ self . out(x ’X {0})@ ’WorkBalancer . nil)] ||
  [startCommunicator("192.168.0.1", 8080, none) ||
```

```
  (loadBalancer {0}::{[loadBalancer / self] *
    [consumer1 / 'Consumer1] * [consumer2 / 'Consumer2]}
    in(! x 'X)@ self . out(x 'X {0})@ 'Consumer1 .
    in(! x 'X)@ self . out(x 'X {0})@ 'Consumer2 .
    in(! x 'X)@ self . out(x 'X {0})@ 'Consumer1 .
    in(! x 'X)@ self . out(x 'X {0})@ 'Consumer2 . nil)] ||
[startCommunicator("192.168.123.1", 9000, none) ||
  (consumer1 {0}::{[consumer1 / self]} nil)] ||
[startCommunicator("192.168.123.2", 9000, none) ||
  (consumer2 {0}::{[consumer2 / self]} nil)] .
```

### Searching for possible final states

To determine all possible final states, i.e., all states in which no more rewrites are possible, the following Maude search command is used:

```
search in D-KLAIM-ABSTRACTION :
  loadBalancerExample =>! NC:NetworkConfiguration .
```

The search yields six possible final states, which correspond to the possible distribution of tuples across the consumers' tuple spaces. The following table shows the possible final configurations of the tuple spaces. Note that a tuple space is a commutative collection of tuples. E.g., `out([0])| out([1])` and `out([1])| out([0])` describe the same tuple space. In all cases each consumer ends up with two tuples in its tuple space, as conjectured.

| Consumer 1's tuple space | Consumer 2's tuple space |
|---|---|
| out([0]) \| out([1]) | out([2]) \| out([3]) |
| out([0]) \| out([2]) | out([1]) \| out([3]) |
| out([0]) \| out([3]) | out([1]) \| out([2]) |
| out([1]) \| out([2]) | out([0]) \| out([3]) |
| out([1]) \| out([3]) | out([0]) \| out([2]) |
| out([2]) \| out([3]) | out([0]) \| out([1]) |

### Model checking of an invariant

An invariant that our simple load balancer example should fulfill is that at no point in time a consumer has more than two tuples in its tuple space. In the following, the variables

```
var NC : NetworkConfiguration .
var C : Configuration .
var A : Site .
var AE : AllocationEnvironment .
var P : Process .
var AP : AuxiliaryProcess .
var SP : SyntacticProcess .
```

are used.

We first define the auxiliary property

```
op lessEqThanTwo : NetworkConfiguration -> Bool .
```

which takes a network configuration as an argument and determines if the two consumers in the network configuration each have less or equal than two tuples in their tuple spaces.

```
op count : Process -> Nat .
eq lessEqThanTwo([C || (A {0}::{AE} P)])
  = if A == consumer1 or A == consumer2 then
      count(P) <= 2
    else false fi .
eq lessEqThanTwo([C] || NC) =
  lessEqThanTwo([C]) or lessEqThanTwo(NC) .
eq count(SP) = 0 .
eq count(AP | P) = s(count(P)) .
```

We then use the command

```
Maude> search loadBalancerExample =>* NC:NetworkConfiguration
  such that not lessEqThanTwo(NC) .
```

to model check the invariant. The result is, that the invariant holds as no solutions are found.

# 4 Chapter

# A Modularized Actor Model for Statistical Model Checking

In this chapter, we introduce the standard *actor model of computation* and its Maude-based implementations, which can be used as foundation for the specification and statistical model checking of distributed systems. We further present an extension of the actor model which incorporates the Russian dolls model (modularity) and fulfills the requirements for statistical model checking. As the absence of un-quantified non-determinism is required to perform statistical model checking, we provide the description of a multi-level scheduling approach for our actor model which fulfills this requirement.

In the following, we

1. give an overview of the *actor model of computation* (Section 4.1),
2. provide an introduction to statistical model checking (Section 4.2),
3. introduce the *Reflective Russian Dolls* model (Section 4.3),
4. extend the standard actor model to support Russian dolls models (Section 4.4),
5. and finally show how the multi-level scheduling approach is specified and how it assures the absence of un-quantified non-determinism for the extended actor model (Section 4.5).

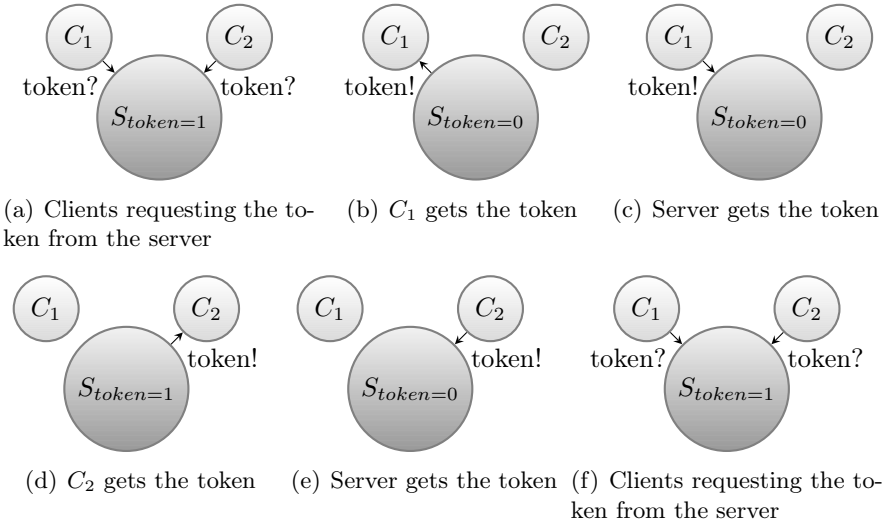
Modular, distributed, and concurrent systems can naturally be modeled and statistically model checked using the modularized actor model. The modular actor model is later used as a foundation of the specifications in the Chapters **??**, **??**, and **??**.

## 4.1. Introduction to the *Actor Model of Computation*

The *actor model of computation* [48, 47, 15] is a mathematical model of concurrent computation in distributed systems. The main building blocks of a distributed system in the actor model are, as its name suggests, *actors*. Similar to the *object-oriented programming paradigm*, in which the philosophy that *everything is on object* is adopted, the *actor model* follows the paradigm that *everything is an actor*. An *actor* is a concurrent object that encapsulates a state and can be addressed using a unique name. Actors can communicate with each other using asynchronous messages. Upon receiving a message, an actor can change its state and can send messages to other actors. Actors can be used to model and reason about distributed and concurrent systems in a natural way.

The following example demonstrates the usefulness of the *actor model* in modelling distributed and concurrent systems.

**Example 4.1: Mutual Exclusion using a Tuple Space like Server**
In this example we specify a simple mutual exclusion algorithm based on a token requesting mechanism. Thereby, the token is stored in a server in a structure which is comparable to a tuple space. Clients are required to possess the token to enter their critical sections. We model both — the clients and the server as actors. The server has a unique address and its state consists of a boolean flag, which indicates whether it possesses the token or not. As with the server, each of the clients has a unique address and enters a competition to get the token. After leaving its critical section (which, in our model, happens instantaneously after having entered), the clients send the token back to the server and repeatedly try to enter their critical sections. Figure 4.1 gives an overview of the interaction between the server and the clients. In the figure, actors are represented by circles and messages are represented by directed arrows.




(a) Clients requesting the token from the server (b) $C_1$ gets the token (c) Server gets the token

(d) $C_2$ gets the token (e) Server gets the token (f) Clients requesting the token from the server

**Figure 4.1.:** Mutual exclusion with a tuple-space like server

The two types of *actors* in our example are represented by the sorts

```
sort Client .
```

and

```
sort Server .
```

Additionally, the sorts and the operator

```
sort Contents
sort Message .

op [_<-_] : Nat Contents -> Message .
```

declare the contents of a message and a message itself. Terms of the sort `Message` are constructed using an identifier (the identifier of the receiver of the message) and a term of the sort `Contents` (the contents or body of the message). The operators

```
op request : -> Contents .
op token? : Nat -> Contents .
op token! : -> Contents .
```

are used as message contents within this model. The message `[A <- request]`, which a client sends to itself, triggers the client to request the token from the server `S` by sending the message `[S <- token?(A)]` to the server. The server (with identifier `S`) replies with a message of the form `[A <- token!]`. The token is sent back from a client by sending a `[S <- token!]` message to the server. Both, actors and messages are part of an associative and commutative multiset of the sort `Config` called a configuration. Such a configuration can be thought of as a soup with the actors and messages being its ingredients.

```
sort Config .
subsorts Client Server Message < Config .

op __ : Config Config -> Config [assoc comm] .
```

Clients are constructed using the operator

```
op <_|server:_> : Nat Nat -> Client .
```

which takes a unique identifier and the unique identifier of the server as arguments. In our example, we represent these unique identifiers with terms of the sort `Nat`. Similarly, a server is constructed with the operator

```
op <_|token:_> : Nat Bool -> Server .
```

which takes a unique identifier and a boolean flag, which indicates whether it possesses the token, as argument. Finally, the operator

```
op initState : -> Config .
eq initState =
  < 0 | token: true > < 1 | server: 0 > < 2 | server: 0 >
  [1 <- request] [2 <- request] .
```

represents the initial state of the system. Initially, the system consists of one server and two clients. Additionally, the two messages `[1 <- request]` and `[2 <- request]` are present in the system. The messages trigger the two clients to start their specified behavior. The dynamic aspects of the system are declared by the following rewrite rules. For their declaration, the variables

```
vars A S : Nat .
```

are used. `A` is used as the unique identifier of the client and `S` as the unique identifier of the server. The rewrite rule

```
rl [Client-sends-request] :
  < A | server:S >
  [A <- request]
=>
  < A | server:S >
  [S <- token?(A)] .
```

specifies the initial behavior of a client. Upon receiving a message of the form `[A <- request]`, a client sends the message `[S <- token?(A)]`, which contains its identifier, to the server. The rewrite rule

```
rl [Client-receives-token] :
  < A | server:S >
  [A <- token!]
=>
  < A | server:S >
  [S <- token!]
  [A <- request] .
```

specifies that when a client receives the token from the server, it directly sends the token back to the server. Additionally, it sends a message to itself. The self-addressed message causes the client to repeatedly request the token from the server. The two rewrite rules

```
rl [Server-receives-token-request] :
  < S | token: true >
  [S <- token?(A)]
=>
  < S | token: false >
  [A <- token!] .
```

and

```
rl [Server-get-the-token-back] :
  < S | token: false >
  [S <- token!]
=>
  < S | token: true > .
```

specify the behavior of the server. If the server has the token (the `token` flag is `true`) and receives a `[S <- token?(A)]` message, it sends a `[A <- token!]` message to the client and changes its `token` flag to `false`. When a server, by receiving a `[S <- token!]` message, gets the token back from a client, it sets its `token` flag to `true`.

The example above shows that distributed and concurrent systems can naturally be modelled using a model based on actors and message-passing communication. In the following, we introduce a Maude-based specification of the actor model that has originally been presented in [19].

### 4.1.1. A Maude-based Specification of the *Actor Model*

The main entities of the *actor model* — actors and messages — are represented by the sorts

```
sort Actor .
sort Msg .
```

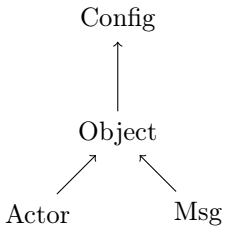



**Figure 4.2.:** Subsort Hierarchy of the Actor Model

and float around in a flat soup of the sort

```
sort Config .
```

The sort

```
sort Object .
```

is a supersort for single terms of the sorts `Actor` and `Msg`, which in turn are a subsort of the sort `Config`. This subsort-relationship is expressed by

```
subsorts Actor Msg < Object < Config .
```

and is depicted in Figure 4.2. Terms of the sort `Config` can be concatenated using the associative and commutative operator

```
op __ : Config Config -> Config [assoc comm id: nil] .
```

for which the constant operator `nil` acts as an identity. The sorts

```
sort ActorName .
sort Contents .
```

represent unique identifiers for actors (of the sort `ActorName`) and contents of messages (of the sort `Contents`). Messages are created using the constructor

```
op _<-_ : ActorName Contents -> Msg [ctor] .
```

which takes the name of the actor to whom the message is sent and contents as arguments. Actors are constructed using the operator

```
op <name:_|_> : ActorName AttributeSet -> Actor [ctor] .
```

which takes the unique name of the actor and a term of the sort `AttributeSet` as arguments. The sorts

```
sort Attribute .
```

and

```
sort AttributeSet .
```

are used to encode the internal state of an actor in a flexible way. Terms of sort `AttributeSet` can be constructed using the associative and commutative concatenation operator

```
op _,_ : AttributeSet AttributeSet -> AttributeSet [assoc comm] .
```

The operators for the the sort `Attribute` are user-definable and thereby allow for a flexible way of encoding the state of the actor.

## 4.2. Introduction to Statistical Model Checking

At a high level, statistical model checking records the evaluations on several runs of an executable model with respect to some property and uses the recordings to obtain an overall estimate of such a property. The recordings of the property evaluations on the system runs are thereby often referred to as samples. In the following, we first introduce probabilistic rewrite theories that allow for the specification of probabilistic models. Next, we present and discuss actor models based on a Maude-based implementation of probabilistic rewrite theories. Finally, we show how the model checker PVeStA [18], a version of VeStA [73] which heavily exploits parallelism, can be used to statistically analyze Maude models that are based on the aforementioned actor models.

### 4.2.1. Probabilistic Rewrite Theories

Many realistic systems possess a probabilistic nature. To specify the indeterminacy of such systems, rewrite theories have been generalized to *probabilistic rewrite theories* [54, 55, 16]. For example, in distributed and parallel systems, the exact time duration of an action highly depends on various factors (e.g. scheduling, network delays, processing times, etc.) and can be modeled as a stochastic process. In contrast to standard non-deterministic model-checkers which always provide absolute guarantees about a property, properties that are statistically model checked in a probabilistic system are checked up to a certain level of statistical confidence (which does not necessarily have to be equal to 1). This sacrifice in confidence is compensated for by the higher scalability of analyzable models and the ability to analyse quantitative properties (e.g. availability) of a system.

Rules in probabilistic rewrite theories are called *probabilistic rewrite rules* and are of the form:

$$l : t(\vec{x}) \rightarrow t'(\vec{x}, \vec{y}) \text{ if } cond(\vec{x}) \text{ with probability } \vec{y} := \pi(\vec{x})$$

Intuitively, a *probabilistic rewrite rule* of this form behaves like a *conditional rewrite rule*, with the difference being, that the next state is not *uniquely* determined. It depends on the choice of an additional substitution $\rho$ for the variables $\vec{y}$. $\rho$ is chosen according to the family of probability functions $\pi(\theta)$: one function for each matching substitution $\theta$ of the variables $\vec{x}$.

In [16], Agha et al. introduced PMaude, a specification language for modelling probabilistic, concurrent, and distributed systems. Additionally, tool support to run discrete-event simulations of PMaude models and for the statistical analysis of resulting samples of the simulations is provided. Example 4.2 illustrates the use of probabilistic rewrite rules by means of a simple example in PMaude. In the example, a client is connected to a server via an unreliable channel.

**Example 4.2: Modelling a lossy channel**
In this example, two entities, a server and a client, communicate via an unreliable channel. The channel is lossy and drops packets at a rate that is governed by a Bernoulli distribution.

The three operators

```
op S : -> ActorName .
op C : -> ActorName .
op CH : -> ActorName .
```

define the actor names of the server, the client, and the channel. The operators

```
op generate : -> Contents .
op msg : -> Contents .
```

are used as contents of the messages that are being sent in the system. Messages of the form `(C <- generate)` are sent by the client to itself to repeatedly generate messages of the form `(CH <- msg)`. These generated messages are being sent to the server by the client. The attribute

```
op cnt:_ : Nat -> Attribute .
```

is used to count the sent messages on the client side, and to count the arrived messages on the server side. The dynamic behavior of the system is defined by the following rewrite rules in which the variables

```
var N : Nat .
var B : Bool .
```

are used. The variable `N` is used by the `cnt:_` attribute and the variable `B` is used by PMAUDE conditions. The rule

```
rl [Client-generating-messages] :
  <name: C | cnt: N >
  (C <- generate)
=>
  <name: C | cnt: N+1 >
  (CH <- msg)
  (C <- generate) .
```

generates messages at the client side. The rule

```
rl [Server-receiving-messages] :
  <name: S | cnt: N >
  (S <- msg)
=>
  <name: S | cnt: N + 1 > .
```

consumes the arrived messages at the server side. Finally, the rule

```
rl [Channel-forwarding-msg-or-drop] :
  <name: CH | cnt: N >
  (CH <- msg)
=>
  <name: CH | cnt: N + 1 >
  if B then
    (S <- msg)
  else
    --- drop packet but keep the channel.
  fi
with probability B := BERNOULLI(N/1000) .
```

specifies the behavior of the channel, which drops or forwards the messages from the client to the server. Whether a message is forwarded or dropped depends on the value of the variable `B`. The binary variable `B` is distributed according to a Bernoulli distribution with mean $\frac{N}{1000}$. Hence on the long run, the proportion of times `B` is true and a message is forwarded relative to the total number of packets sent by clients will approximate $\frac{N}{1000}$.

The execution of a PMaude module such as the one defined above requires the module to be transformed into a corresponding Maude module, which then simulates its behavior. In [16], this transformation is provided by a PMaude specification using the actor model. The actor model was chosen for the specification, because it allows for an intuitive way to avoid unquantified non-determinism. The absence of (un-quantified) non-determinism is the key requirement for statistical model checking of such systems [72, 71].

### 4.2.2. Maude specification of Actor PMaude

As mentioned before, the key requirement for the proposed statistical model checking is the absence of un-quantified non-determinism. In PMaude, Agha et al. present an approach to avoid un-quantified non-determinism. In [19], AlTurki et al. present a slightly different approach to achieve the same goal. In the following we present and compare the two approaches.

#### Approach 1: (Original PMaude)

In addition to the basic actor model, a notion of stochastic real-time is introduced to capture the dynamics of various elements of a system. For example, message passing and computations that are triggered by a message may take some positive real-valued time. To model stochastic real-time associated with message passing delay or actor computation, an actor or a message can be made inactive up to a given global time by enclosing it in square brackets. The sort

```
sort ScheduleObject .
```

represents an inactive object that is waiting to become active. A scheduled object is constructed by the operator

```
op [_,_] : PosReal Object -> ScheduleObject .
```

that makes an object inactive until the global time has advanced to the specified activation time. The subsort relationship

```
subsorts PosReal ScheduleObject < Config .
```

enables terms of the sorts `ScheduleObject` and `PosReal` to float in the global soup of the sort `Config`. The latter is used to have one term in the global soup that represents the global time.

The global state of a system is represented by a term of the sort `Config`, which contains objects, scheduled objects, and a global time (a term of the sort `PosReal`).

#### Approach 2: (Scheduler-based)

In [19], AlTurki et al. developed an approach based on a scheduler. The scheduler contains a list of messages and is part of the configuration. The sorts

```
sort Scheduler .
sort ScheduleElem .
sort ScheduleList .
```

define the scheduler, elements that are stored within the scheduler, and a list of such elements. The subsort relationships

```
subsort Scheduler < Config .
subsort ScheduleElem < ScheduleList .
```

state that the scheduler is part of the global configuration, and that a list of schedule elements consists of single terms of the sort `ScheduleElem`. As with the previous approach, a notion of stochastic real-time is introduced. Thus, a term of the sort `ScheduleElem` is constructed using the operator

```
op [_,_,_] : Float Msg Nat -> ScheduleElem .
```

which takes three arguments: a timestamp that indicates the time when the message should be made active in the system, the message itself, and an additional flag which is used to provide a notion for lossy channels. The associative operator

```
op _;_ : ScheduleList ScheduleList -> ScheduleList [assoc id: nil] .
```

is used to concatenate terms of the sort `ScheduleList` with the constant operator

```
op nil : -> ScheduleList .
```

acting as the identity. A scheduler is a term that is built using the operator

```
op {_|_} : Float ScheduleList -> Scheduler .
```

which takes a term of sort `Float`, the global time of the system, and a term of sort `ScheduleList` , the list of scheduled messages, as arguments. Messages in the scheduler's list are ordered according to their scheduled time of activation. The operator

```
op insert : Scheduler ScheduleElem -> Scheduler .
```

inserts a given scheduled message into the list of scheduled messages and preserves the timely order of the list items. The operator

```
op mytick : Scheduler -> Config .
```

removes the first message, i.e., the message which is to be activated next, from the scheduler, updates the global time of the scheduler, and returns the message together with the updated scheduler.

The global state is similar to the one in the first approach. The state is represented by a term of sort `Config`, which contains objects and one term of the sort `Scheduler`.

### Similarities

In [16], a special tick rule is defined and is used in both approaches. A *one-step computation* rule of an actor in the PMaude model is defined as a transition of the form

$$[u]_A \xrightarrow{\neg tick}{}^{*} [v]_A \xrightarrow{tick} [w]_A$$

where

(i) $[u]_A$ is a canonical term of sort `Config`, representing the global state of a system.

(ii) $[v]_A$ is a term obtained after a sequence (zero or more) of one-step rewrites such that

- in none of those rewrites is the `tick` rule applied, and
- $[v]_A$ cannot be further rewritten by applying any rule except the `tick` rule.

(iii) $[w]_A$ is obtained after a one-step rewrite of $[v]_A$ by applying the `tick` rule, which does the following

- finds and removes the scheduled object, if one existst, with the smallest global time, say `[T', Obj]`, from the term $[v]_A$ to a term, say $[v']_A$,
- adds the term `Obj` to $[v']_A$ through multiset union to get the term $[v'']_A$, and
- replaces the global time of the term $[v'']_A$ with `T'` to get the final term $[w]_A$.

**The absence of un-quantified non-determinism**

In the original PMaude paper [16], sufficient requirements for the absence of unquantified non-determinism in an actor PMaude specification are presented. A PMaude specification fulfills the requirement, if

1. the initial global state of the system or the initial configuration can have at most one non scheduled message.
2. the computation performed by any actor after receiving a message must have no unquantified non-determinism; however, there may be probabilistic choices.
3. the messages produced by an actor in a particular computation (i.e., when receiving a message) can have at most one non scheduled message.
4. no two scheduled objects can become active at the same global time.

The last requirement is the one that should be ensured by the actor model. The actor *PMaude* model ensures this by associating continuous probability distributions with message delays and computation time. This approach relies on the fact, that for continuous distributions the probability of sampling the same real number twice is zero. The second approach ensures the last requirement by transferring the control on when a message becomes active to the scheduler. The scheduler emits the messages in the right order, which is deterministic for a fixed probability distribution, and ensures that only one message is active in the system at any point in time.

**Comparison**

The correctness of the first approach relies on the fact that no two scheduled objects `[R1, O1]` and `[R2, O2]` are scheduled to become active at the same time, i.e., for any to times `R1`, `R2` in the scheduler the equation `R1` $\neq$ `R2` always holds. This is achieved by associating continuous probability distributions with the scheduling times of objects in the scheduler. However, even though this assumption might hold for the real world, an infinite precision for time measurement and time representation is unachievable in a computerized model.

The second approach eliminates this shortcoming by using a scheduler. A message can be inserted into a schedule in order to become active after a fixed or random amount of time.

Example 4.3 illustrates the difference between the two approaches using the client server setting from Example 4.2.

**Example 4.3: Practical differences between the two approaches**
In both approaches, the operators

```
op S : -> ActorName .
op C : -> ActorName .
op CH : -> ActorName .

op generate : -> Contents .
op msg : -> Contents .

op cnt:_ : Nat -> Attribute .
```

are used. In Example 4.2, these operators are explained in more detail. The subtle difference between the aforementioned approaches lies in the way how messages are emitted.

**Approach 1: (Original PMaude solution)** The first approach relies on the fact that two scheduled object `[`$T_1$`,` $O_1$`]` and `[`$T_2$`,` $O_2$`]` with $T_1 = T_2$ are never present in the global state of the system at any point in time. To emit new messages, the times for the scheduled objects have to be chosen randomly. Following this approach, the resulting model has no un-quantified non-determinism, since it meets the conditions given in Section 4.2.2.

In the following, the variables

```
var N : Nat .
var B : Bool .
var T : PosReal .
```

are used, where the variable `N` is used together with the `cnt:_` attribute and the variable `T` is used to represent the global time of the system.

The rules

```
rl [Client-generating-messages] :
  <name: C | cnt: N >
  (C <- generate)
  T
=>
  <name: C | cnt: N+1 >
  [T + EXPONENTIAL(0.2), CH <- msg ]
  [T + EXPONENTIAL(0.2), C <- generate]
  T .

rl [Server-receiving-messages] :
  <name: S | cnt: N >
  (S <- msg)
  T
=>
  <name: S | cnt: N + 1 > T .

rl [Channel-forwarding-msg-or-drop] :
  <name: CH | cnt: N >
  (CH <- msg)
  T
=>
  <name: CH | cnt: N + 1 >
  if BERNOULLI(N/1000) then
    [T + EXPONENTIAL(0.2), S <- msg]
  else
    --- drop packet!
  fi
  T .
```

specify the dynamic aspects of the system. Messages are emitted as terms of the sort `ScheduleObject` and the term `EXPONENTIAL(0.2)` rewrites to a randomly chosen real number sampled from the exponential distribution with parameter 0.2.

**Approach 2: (Scheduler-based solution)** The second approach ensures the absence of un-quantified non-determinism by using a scheduler. All messages that are emitted have to be inserted into the scheduler. In contrast to the first approach, the time when a message should become active can be chosen without any restrictions. As thus, no additional randomness is introduced in the model.

In the following, the variables

```
var N : Nat .
var gt : Float .
var SL : ScheduleList .
```

are used, where the variable `N` is used together with the `cnt:_` attribute, the variable `gt` represents the global time, and the variable `SL` represents the list of scheduled messages of the scheduler. The rules

```
rl [Client-generating-messages] :
  <name: C | cnt: N >
  (C <- generate)
  { gt | SL }
=>
  <name: C | cnt: N+1 >
  insert(insert({ gt | SL }, [gt + 0.1, CH <- msg]), [gt + 0.1, C <- generate]) .

rl [Server-receiving-messages] :
  <name: S | cnt: N >
  (S <- msg)
=>
  <name: S | cnt: N + 1 > .

rl [Channel-forwarding-msg-or-drop] :
  <name: CH | cnt: N >
  (CH <- msg)
  { gt | SL }
=>
  <name: CH | cnt: N + 1 >
  if BERNOULLI(N/1000) then
    insert({ gt | SL }, [gt + 0.1, S <- msg])
  else
    --- drop packet!
  { gt | SL }
  fi .
```

define the dynamic aspects of the system, whereby new messages are inserted in the scheduler using the `insert` operator.

### 4.2.3. Statistical Analysis using the PVeStA model checker

In [72], Sen et. al. describe an algorithm for statistical model checking based on simple hypothesis testing. Formulas in *Probabilistic CTL* (PCTL) [43] and *Continuous Stochastic Logic* (CSL) [23, 24] can be model checked using this algorithm. PCTL is an extension

$$
\begin{aligned}
Q &::= D \textbf{ eval } E[PExp]; \\
D &::= set\ of\ Defn \\
Defn &::= N(x_1, \ldots, x_m) = PExp; \\
SExp &::= c \mid f \mid F(SExp_1, \ldots, SExp_k) \mid x_i \\
PExp &::= SExp \mid \bigcirc N(SExp_1, \ldots, SExp_n) \\
&\quad \mid \textbf{ if } SExp \textbf{ then } PExp_1 \textbf{ else } PExp_2 \textbf{ fi}
\end{aligned}
$$

**Figure 4.3.:** Syntax of QuaTEx

of standard CTL, which associates probability measure to computation paths and qualifies temporal logic formulas with probability bounds. CSL further extends PCTL by continuous timing and qualifies temporal logic operators by time bounds. In [16], Agha et. al. generalize PCTL and CSL to *Quantitative Temporal Expressions* (QuaTEx) in order to be able to express quantitative properties, as for instance the latency of a system. In QuaTEx, state formulas and path formulas are extended to real-valued state expressions and path expressions.

### QuaTEx

The syntax of QuaTEx is shown in Figure 4.3. A QuaTEx query $Q$ consists of a set of definitions $D$, followed by a query about the *expected value* of a path expression $PExp$. A definition $Defn \in D$ defines a temporal operator with name $N$, and a set of formal parameters on the left-hand side. The right-hand side consists of a path expression. If a temporal operator is used in a path expression, the formal parameters are replaced by state expressions. A state expression $SExp$ can be a constant $c$, a function $f$ which maps a state to a concrete value, a function $F$, that maps $k$ state expressions to a state expression, or a formal parameter. A path expression $PExp$ is either a state expression, a next operator followed by the application of a temporal operator $N$, that is defined in $D$, or a conditional expression.

We omit the specification of the semantics of QuaTEx for the sake of brevity here, but explain the use of QuaTEx by the following example. We refer the interested reader to [16].

**Example 4.4: QuaTEx by example**
This example is based on the aforementioned example, in which a server and a client are connected via a lossy channel. The statistical property we are interested in is *the probability that along a random path from a given state, the client C sends a message (that is not dropped) to the server S within the first 1.0 time units.*

This can be expressed by the following QuaTEx expression:

```
IfReceivedInTime(t) =
  if t > time() then
    0
  else
    if msgReceived() then
```

```
      1
    else
      ◯ (IfReceivedInTime(t))
    fi
  fi;

eval E[IfReceivedInTime(time() + 1.0)];
```

First, the temporal operator `IfReceivedInTime` is defined, which returns 1, if the server received a message (the state function `msgReceived` returns `true` on a state) along an execution path within time `t`. Otherwise, it returns 0. Here, the state function `time()` returns the global time associated with the state. Finally, the state query `eval E[IfReceivedInTime(time ()+ 1.0)]` returns the expected number of times a message is received at the server $S$ within the first 1.0 time units. This number lies in $[0, 1]$, since along a random path the temporal operator `IfReceivedInTime` either returns 0 or 1. The expected value is in fact equal to the probability that the server receives a message along a random path from the given state within the first 1.0 time units.

#### Statistical evaluation of QuaTEx expressions (VeStA/PVeStA)

A QuaTEx expression is evaluated on the initial state of a model. The expected value of the expression is statistically evaluated by the approximation of the value by the mean of $n$ samples such that the size of the confidence interval $(1 - \alpha)$ [51] for the expected value is bounded by $\delta$.

In [16], Agha et. al. implemented the evaluator VeStA for QuaTEx properties in Java. VeStA takes an actor PMaude model, an initial actor PMaude term that represents the initial configuration of the system, and a QuaTEx expression (with the two parameters $\alpha$ and $\delta$) as arguments.

The Maude interpreter is used to perform discrete-event simulations. VeStA maintains the current configuration of the system as a Java string and VeStA passes the current configuration to the Maude interpreter that performs a one-step computation at every simulation step. The result is stored as the next configuration. Using this approach, QuaTEx expressions can be evaluated using the Maude one-step computation as the next operator.

In [18], AlTurki and Meseguer present PVeStA, an extension and parallelization of the VeStA statistical model checking tool. It supports statistical model checking of discrete or continuous Markov Chains or of probabilistic rewrite theories in Maude. Properties can thereby be expressed in either PCTL/CSL or QuaTEx. PVeStA also provides a scalable performance through a parallelized generation of samples. In Section 4.6, we explain how PVeStA can be used for statistical model checking and quantitative analysis of QuaTEx properties of models based on the extended actor model, which is presented in the following.

## 4.3. Introduction to the *Reflective Russian Dolls* Model

In some situations, the state of a distributed system can be thought of as a *flat configuration* which contains objects and messages. Such a *flat configuration* can be modelled as a *flat soup* that consists of actors and messages. The actors in the soup communicate via asynchronous message passing or synchronous interactions.

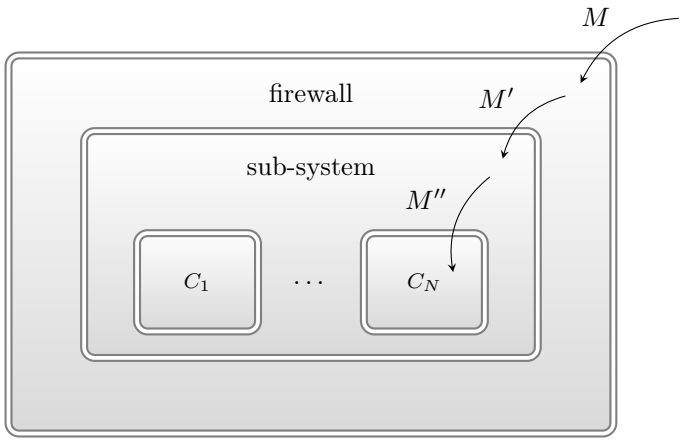



**Figure 4.4.:** Example of a Russian doll model of a system with boundaries

As a distributed system becomes more complex, hierarchies may have to be introduced to represent the structure of the system and its communication patterns. Furthermore, if hierarchies are not modeled, the distance between the system and a model of it might become bigger. For example, the Internet, the most widely used distributed system, inherently is a hierarchical system of nested systems. Trying to model the Internet as a flat system will not reflect its characteristics. Consequently, the distance between the model and the real system becomes a major issue. Additionally, a flat model does not reflect boundaries of systems. In a flat model, every participant can communicate with everybody else. However, some concepts, like a firewall, rely on the existence of physical boundaries that messages from the outside have to cross in order to reach destinations within a border.

In [62], Meseguer and Talcott present the *Reflective Russian Dolls* (RRD) model which extends and formalizes previous work on actor reflection and provides a generic formal model of distributed object reflection. The rewriting-logic based model combines logical reflection and hierarchical structuring. In their model, the state of a distributed system is not represented by a *flat soup*, but rather as a *soup of soups*, each enclosed within specific boundaries. As with traditional Russian dolls, soups can be nested up to an arbitrary depth.

Figure 4.4 illustrates the basic idea using a system that is guarded by a firewall. Each of the boxes represents a system. The firewall consists of a subsystem which itself is composed of several components $C_1 \ldots C_N$. Message $M$ is addressed to the innermost component $C_1$ and as such has to pass the boundary of the firewall. The firewall possibly transforms the message to $M'$ (e.g. tags a message with a security clearance). After that, the boundary of the sub-system has to be crossed which, respectively, can also alter the message to $M''$.

Mathematically, this can be modelled by boundary operators of the form

$$b : s_1, \ldots, s_n, Configuration \rightarrow Configuration$$

where $s_1, \ldots, s_n$ are additional sorts. These sorts are called the *parameters* of the boundary operator. Boundary operators encapsulate a configuration together with several parameters, and as with Russian dolls, they can be nested arbitrarily.

Using the Russian Dolls model, sophisticated distributed systems, that rely on system boundaries, can be modeled [62].

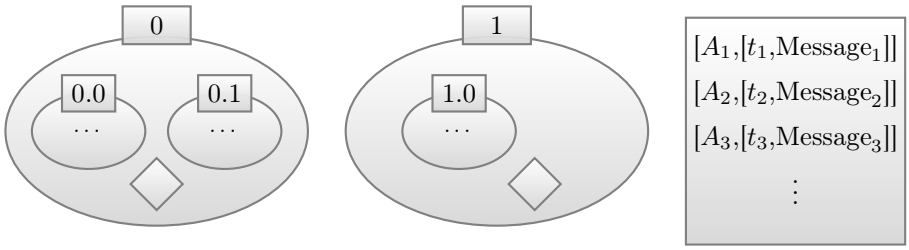



**Figure 4.5.:** Basic overview of the modularized actor model

## 4.4. The Modularized Actor Model

Previous approaches to statistical model checking of actor systems rely on a flat model. Neither of the two models presented in Section 4.2 can handle models based on the RRD. In the following, an extension to the actor model of Section 4.1 is presented, which incorporates the Russian dolls model. Additionally, the scheduling approach by AlTurki et al. is enhanced with support for multiple levels of Russian dolls actors in a way which preserves the guarantee for the absence of unquantified non-determinism. Modularity is intrinsically supported, since rewrite rules in specifications that use the modularized actor model remain local, i.e., they remain without any knowledge of the outside environment or of how they are used. Chapters 5 and 6 use the modularized actor model as a basic building block.

The actor model can easily be extended to support the RRD model by allowing an actor to contain a soup of objects. Similarly to the approach of AlTurki et al., a scheduler at the highest level of the actor-hierarchy is used to guarantee the absence of unquantified non-determinism. Furthermore, a hierarchical naming scheme is introduced, which allows for the automatic generation of fresh names. Messages can be emitted at any level and are automatically inserted into the scheduler. For messages that are scheduled to become active, the approach automatically inserts the messages in the configuration. The messages are thereby put into the subconfiguration were they have emitted. Messages only cross boundaries through *boundary crossing rewrite rules*.

Figure 4.5 illustrates the extended actor model. The top-level configuration consists of two Russian dolls actors and the scheduler. The actors have the unique addresses 0 and 1. Both actors themselves contain actors and a name-generator (symbolized as a diamond) in their sub-configurations. The scheduler contains three messages, $\text{Message}_1$, $\text{Message}_2$, and $\text{Message}_3$ with timestamps $t_1$, $t_2$, and $t_3$. Messages in the scheduler's list are ordered according to their scheduled time of activation, thus $t_1 < t_2 < t_3$.

### 4.4.1. The Hierarchical Addressing Scheme

Figure 4.6 shows an example of how the hierarchical naming scheme is used. It builds a naming tree, in which childrens' addresses are composed of their parent's address and a number. The number is chosen according to the order in which the children are created. The top of the tree has the constant address `toplevel`, which is omitted in its childrens' addresses.

Hierarchical addresses are terms of the sort

```
sort Address .
```

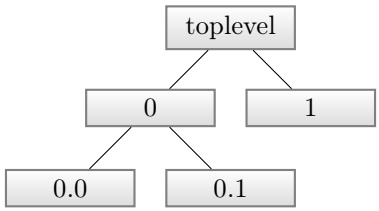



**Figure 4.6.:** Exemplary usage of the hierarchical naming scheme

while first-level addresses are represented by terms of the sort `Nat`. The subsort relationship

```
subsort Nat < Address .
```

states that a single natural number represents a term of the sort `Address`. The associative operator

```
op _._ : Address Address -> Address [assoc prec 10] .
```

concatenates terms of the sort `Address`. The two operators

```
op _<_ : Address Address -> Bool [ditto] .
```

and

```
op |_| : Address -> Nat .
```

define, respectively, a lexicographic ordering over the addresses and a length operator and have fairly obvious defining equations. Additionally, the constant operator

```
op toplevel : -> Address .
```

is defined to represent the root of the address-tree.

### 4.4.2. The Actor Model and the Name Generator

Terms of sort

```
sort Contents .
```

represent the contents of a message that is being sent within the system. The modularized actor model defines five different types of messages:

- Terms of sort `Msg` consist of a term of the sort `Contents` that is sent to a specific address.

  ```
  sort Msg .
  op _<-_ : Address Contents -> Msg .
  ```

- Terms of sort `ActiveMsg` represent *active messages* in the system. An *active message* is a message that can be consumed by an actor using a rewrite rule. Active messages are constructed using the operator `{_,_}` which takes the current global time and an actual message as arguments.

  ```
  sort ActiveMsg .
  subsort ActiveMsg < Config .
  op {_,_} : Float Msg -> ActiveMsg .
  ```

- Terms of sort `ScheduleMsg` represent *scheduled messages*, i.e., messages which are emitted by a rewrite rule and will be inserted into the scheduler. Scheduled messages contain the global time at which the message is made active (which has to be greater or equal than the current global time) and an actual message. Terms of sort `ScheduleMsg` are the only messages that an actor is allowed to emit.

```
sort ScheduleMsg .
subsort ScheduleMsg < Config .
op [_,_] : Float Msg -> ScheduleMsg .
```

- Terms of sort `LocActiveMsg` enclose an *active message* and the address of the configuration where the active message is inserted when it is made active. This is an intermediary message type that is used to internally push active messages down to their respective destinations.

```
sort LocActiveMsg .
subsort LocActiveMsg < Config .
op {_,_} : Address ActiveMsg -> LocActiveMsg .
```

- Similar to terms of sort `LocActiveMsg`, terms of sort `LocScheduleMsg` enclose *scheduled messages* and the address of the configuration where they have originally been emitted. This is an intermediary message type which is used to pull scheduled messages up to the scheduler and store them until they are made active.

```
sort LocScheduleMsg .
subsort LocScheduleMsg < Config .
op [_,_] : Address ScheduleMsg -> LocScheduleMsg .
```

The sorts

```
sort Actor .
sort ActorType .
sort AttributeSet .
```

declare actors, the type of an actor, and a set of attributes. The type of the actor can be thought of as the type of an object in object-oriented programming. A term of the sort actor is created using the operator

```
op <_:_|_> : Address ActorType AttributeSet -> Actor .
```

which takes a unique address, the type (that is the class) of the actor, and a set of attributes representing the internal state of the actor as arguments. The state of the actor is encoded in a set of attributes in the same way as in the actor model. Russian doll actors — actors that contain a soup themselves — are of sort `Actor` and furthermore contain the attribute

```
op config:_ : Config -> Attribute [gather(&)] .
```

in their set of attributes. The inner configuration of a Russian dolls actor (a term of the sort `Config`) is stored within this attribute. Terms of sort

```
sort NameGenerator .
```

specify name generators. The operator

```
op <_> : Address -> NameGenerator .
```

constructs a name generator. A name generator contains a new fresh address as an argument. This address can be extracted from the name generator using the operator

```
op _.new : NameGenerator -> Address .
```

The operator

```
op _.next : NameGenerator -> NameGenerator .
```

creates a fresh name for a name generator. After having created a new address using the operator `_.new`, it is necessary to replace the name generator with the name generator that is returned by the operator `_.next` in order to get a new address again. Terms of sort

```
sort Config .
```

constitute a configuration which may contain all kinds of messages, flat actors, Russian doll actors, and at most one name generator at its top level. The scheduling algorithm requires that all *scheduled messages* are inserted in the scheduler, and that all active messages are consumed by rewrite rules before a new message is made active. Thus, a mechanism to differentiate between configurations, that contain *active* or *scheduled* messages, and configurations that do not contain such messages, is needed. The sorts

```
sort InertActor .
sort ActorConfig .
```

serve this purpose. A term of sort `ActorConfig` is a configuration that contains no *active* or *scheduled* messages. A term of sort `InertActor` is an actor that is either flat, or its configuration is of sort `ActorConfig`. The subsort relationships

```
subsorts Actor ActorConfig < Config .
subsort InertActor < Actor .
subsort NameGenerator InertActor < ActorConfig .
subsort Attribute < AttributeSet .
```

are illustrated in Figure 4.7. Terms of the sorts `ActiveMsg`, `ScheduleMsg`, `LocActiveMsg`, and `LocScheduleMsg` are subsorts of sort `Config`. Terms of the sort `ActorConfig` are a specialization of the sort `Config`. Thus, `ActorConfig` is a subsort of `Config`. `InertActor` is a subsort of both, the sort `Actor` and the sort `ActorConfig`. The sort `Actor` is a subsort of `Config`, since it is possibly a Russian doll actor which may contain an *active* or *scheduled* message within its configuration. The conditional membership

```
cmb ACT : InertActor if flatActor(ACT).
```

states that an actor is of the sort `InertActor`, if it is a flat actor. The operator

```
op flatActor : Actor -> Bool .
```

determines if an actor is a flat actor. In the following definition, the variables

```
var ACT : Actor .
var A : Address .
var T : ActorType .
var C : Config .
var AS : AttributeSet.
```

are used. The operator is defined by the equations

```
eq flatActor(< A : T | config: C, AS >) = false .
eq flatActor(ACT) = true [owise] .
```

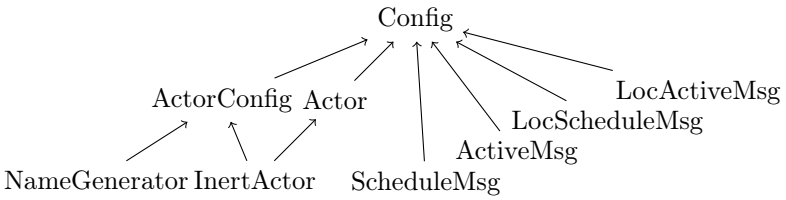



**Figure 4.7.:** Subsort Hierarchy of the Extended Actor Model

which specify that a flat actor is an actor that does not contain a `config:_` attribute. The membership[1]

```
mb < A : T | config: AC, AS > : InertActor .
```

states that if the configuration of an actor is of sort `ActorConfig`, then the actor is also of sort `ActorConfig`. Finally, the operators

```
op null : -> ActorConfig .
op __ : ActorConfig ActorConfig -> ActorConfig [assoc comm id: null] .
op __ : Config Config -> Config [assoc comm id: null] .
```

define the associative and commutative composition of terms of sort `ActorConfig` and of sort `Config`.

## 4.5. Multi-level scheduling for the Modularized Actor Model

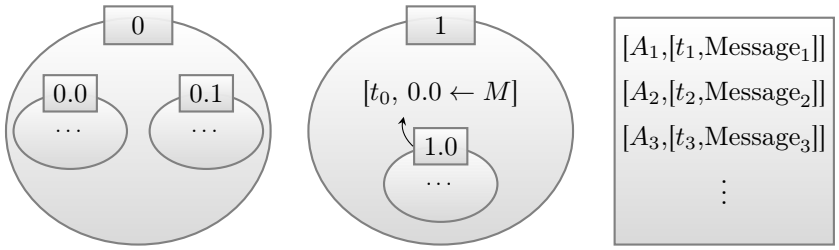
Figure 4.8 shows a schematic overview of the scheduling approach. The scheduling approach for a *scheduled messages* consists of two phases:

1. When a *scheduled message* is emitted at a specific level in the actor hierarchy, it is pulled up to the top-most level and is inserted into the scheduler. Internally, the *scheduled message* is thereby first wrapped by a term of the sort `LocScheduleMsg` which, in addition to the message, stores the address of the actor in whose configuration the message was emitted. Then, the message is pulled up until it is located at the top-most level. Finally, it is inserted into the scheduler at the correct position.

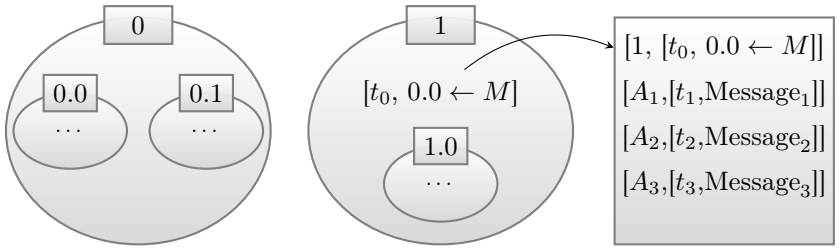
2. When a *scheduled message* is scheduled to be active, it is pushed down to the configuration where it was emitted and is inserted there as an *active message*. Internally, the *scheduled message* is thereby removed from the scheduler and inserted in the top-most configuration as a term of the sort `LocActiveMsg`. Since the address of the actor, in whose configuration the message was emitted, is known, and the hierarchical addressing scheme is used, the message can be pushed down to its destination easily [2]. When the destination configuration is reached, the wrapped term of the sort `ActiveMsg` is emitted in the configuration.

[1] For simplicity we assume that: the only attribute whose value is a configuration is the `config` attribute, so that no other attributes of an object can contain configurations as their values.

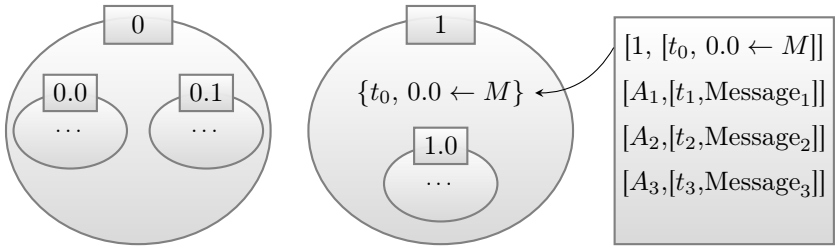
[2] There is a solution which allows for any naming scheme that prevents the collision of names to be used: While pulling the message up, the names of the actors on the path up to the top-level are collected in an ordered list that represents the path. In order to put a message back into the configuration in which it was emitted, this path is traversed in the reverse order.



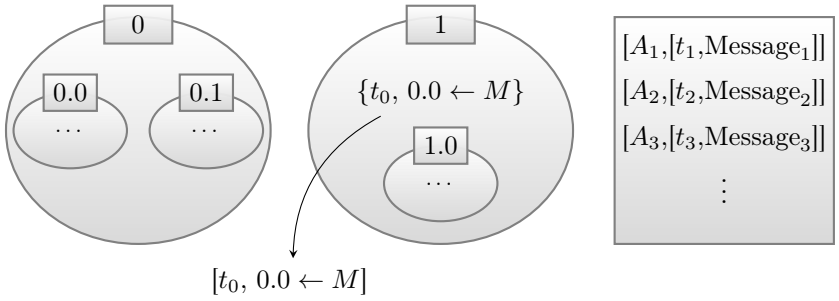

(a) The actor with address 1.0 is sending a message to the address 0.0



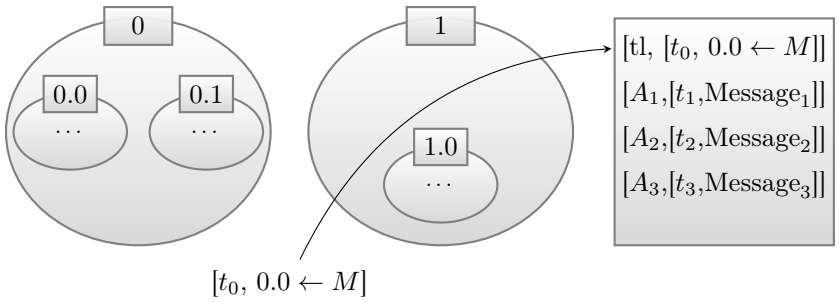

(b) The message is inserted in the scheduler with a reference to the containing actor's address




(c) Once the message becomes active, the message is put back in the soup




(d) A boundary-crossing rewrite rule pushed the message one level up




(e) The message is inserted in the scheduler with the reference to the topmost level

**Figure 4.8.:** Schematic overview of the scheduling approach

In the following, we present a Maude-based specification of the multi-level scheduling approach. The following variables

```
vars gt t1 t2 LIMIT : Float .
vars SL SL' : ScheduleList .
vars M1 M2 : Msg .
var S : Scheduler .
var C : Config .
vars A A' A1 A2 : Address .
var T : ActorType .
var AS : AttributeSet .
var CO : Contents .
var AC : ActorConfig .
var SM : ScheduleMsg .
var AM : ActiveMsg .
var LSM : LocScheduleMsg .
```

are used in what follows.

A term of sort

```
sort Scheduler .
```

specifies the state of a scheduler in the system. The sort `Scheduler` is specified as a subsort of `Config`

```
subsort Scheduler < Config .
```

Thus, a scheduler can be part of a configuration. The scheduler stores an ordered list of messages of sort

```
sort ScheduleList .
```

Terms of the sort `LocScheduleMsg` are constructed using the constant operator

```
op nil : -> ScheduleList [ctor] .
```

and the associative concatenation operator

```
op _;_ : ScheduleList ScheduleList -> ScheduleList
  [ctor assoc id: nil] .
```

for which the term `nil` acts as an identity. Since terms of the sort `LocScheduleMsg` are a subsort of `ScheduleList`

```
subsort LocScheduleMsg < ScheduleList .
```

the list of messages in the scheduler consist of terms of sort `LocScheduleMsg`. A scheduler is constructed by the operator

```
op {_|_} : Float ScheduleList -> Scheduler .
```

which takes the global time and a list of scheduled messages as arguments. The list items are ordered according to their scheduled time of activation. The operators

```
op insert : Scheduler LocScheduleMsg -> Scheduler .
op insert : ScheduleList LocScheduleMsg -> ScheduleList .
op insertList : Scheduler ScheduleList -> Scheduler .
op insertList : ScheduleList ScheduleList -> ScheduleList .
```

insert *scheduled messages* or lists of *scheduled messages* into the scheduler in the correct order. In order to assure the absence of non-determinism, the order of the messages must be deterministic (no matter in which order messages are inserted in the scheduler). The equations

```
eq insert({gt | SL }, LSM) = {gt | insert(SL,LSM) } .
eq insert([ A1, [ t1 , M1 ] ] ; SL , [ A2, [ t2 , M2 ] ]) =
  if (t1 < t2) or ((t1 == t2)) and lt(M1,M2) then
    [ A1, [ t1 , M1 ] ] ; insert(SL, [ A2, [ t2 , M2 ] ])
  else
     ([ A2, [ t2 , M2 ] ] ; [ A1, [ t1 , M1 ] ] ; SL)
  fi .
eq insert( nil , [A2, [ t2 , M2 ]]) = [A2, [ t2 , M2 ]] .
eq insertList({gt | SL }, SL') = {gt | insertList(SL, SL') } .
eq insertList(SL , [ A2, [ t2 , M2 ] ] ; SL') =
  insertList( insert(SL, [ A2, [ t2, M2 ] ]), SL' ) .
eq insertList(SL , nil ) = SL .
```

define the behavior of the insertion operators. Messages are inserted in the order of their scheduled time of activation. If two messages have an equal time of activation, the total term order which is provided by Maude as a meta-level function is used to create a total order on the two messages.

The equations

```
eq [create-loc-scheduled-msg1] :
  < A : T | config: SM C, AS > = [ A, SM ] < A : T| config: C, AS > .
eq [create-loc-scheduled-msg2] :
  SM C S = [ toplevel, SM ] C S .

eq [pull-up] :
  < A : T | config: LSM C, AS > = LSM < A : T | config: C, AS > .
eq [insert-in-scheduler] :
  LSM S = insert(S, LSM) .
```

define the semantics of the first phase of the scheduling algorithm. The equation `create-loc-scheduled-msg1` and `create-loc-scheduled-msg2` enclose a term of sort `ScheduleMsg` in a term of the sort `LocScheduleMsg`. The term is then pulled up by the equation `pull-up`. Finally, when the term reaches the top-level, it is inserted into the scheduler by the equation `insert-in-scheduler`.

The second phase of the scheduling algorithm is defined by the equations

```
eq [push-down] :
  < A : T | config: C, AS > {A . A', AM} = < A : T | config: C {A . A', AM}, AS >
      .
eq [insert-in-configuration] :
  < A : T | config: C, AS > {A , AM} = < A : T | config: C AM, AS > .
eq [insert-top-level] :
  {toplevel, AM } S = AM S .
```

When an *active message* enclosed in a term of sort `LocActiveMsg` is emitted, it is pushed down to its originating configuration by the equation `push-down`. The equations `insert-in-configuration` and `insert-top-level` insert the enclosed active message in the inner configuration of an actor, or at the top-level.

Similar to the special tick rule that is defined in [16], a one-step computation of a model written in the modularized actor model is defined by a transition of the form

$$[u]_A \xrightarrow{step} [v]_A \rightarrow * [w]_A$$

where

(i) $[u]_A$ is a canonical term of sort `ActorConfig`, which represents the global state of a system (and of all of its sub-systems). Since $[u]_A$ is of sort `ActorConfig`, there is no *active* or *scheduled* message in the system.

(ii) $[v]_A$ is a term obtained by removing the next message from the scheduler, say `[A,[T, M]]`, and by inserting it into the configuration of the object at address `A`. Additionally, the global time in the scheduler is changed to T.

(iii) $[w]_A$ is a term obtained after a sequence (zero or more) of one-step rewrites, until no more *active* or *scheduled* messages are in the system.

The operator

```
op step : Config -> Config [iter] .
```

is defined by the (partial) equation

```
eq step(AC {gt | [ A1, [ t1 , M1 ] ] ; SL})
  = { A1, { t1 , M1 } } AC {t1 | SL} .
```

It takes the first message from the list of *scheduled messages* of the scheduler, sets the global time of the scheduler to the activation time of the that message, and returns a configuration that contains the unmodified configuration together with the updated scheduler and the message. The operator

```
op run : Config Float -> Config .
```

is defined by the equation

```
eq run(AC {gt | SL}, LIMIT) =
  if (gt <= LIMIT) then
    run(step(AC {gt | SL}), LIMIT)
  else
    AC {gt | SL}
  fi .
```

It repeatedly calls the `run` operator until a specified amount (denoted by the variable `LIMIT`) of global time has passed.

The operator `step` is only defined on terms that are built using a term of sort `ActorConfig` and a scheduler. Thus, a new *active message* is only emitted in a configuration that contains no more *active* or *scheduled* messages. Hence, every inserted message has to be consumed, or the `step` operator is not defined (i.e., it cannot proceed). This behavior is ensured by the following steps which are taken after a message has been emitted:

- The new *active message* is inserted in the correct configuration. This is ensured due to the hierarchical addressing scheme.

- There is at most one rule that consumes the *active message* and possibly emits new *scheduled messages*.

- All *scheduled messages* are pulled up to the top level and are inserted into the scheduler. It is important to notice that the insertion of messages keeps a total order in the list of the scheduler. Thereby, the sequence of insertions always results in the same list in the scheduler (i.e., the equational system is confluent).

- The `step` rule emits one single new message in the top-level configuration.

In case a message cannot be consumed in step 4.5, the system reaches a final state as no more rewrites are possible.

### 4.5.1. The Absence of unquantified non-determinism

The scheduler approach ensures that only one message is active at any given time. We provide a new, multi-level version of the sufficient requirements for the absence of unquantified non-determinism given in Section 4.2.2. The requirements:

1. If Russian doll actors are used, then the inner configuration is contained in the `config :_` attribute of the actor. This guarantees that the conditional membership works as specified and that the scheduler can insert the messages in the modularized actor model.

2. There is at most one term of sort `NameGenerator` contained in each subterm of the sort `Config`, i.e., each such subconfiguration has a single corresponding name generator. This ensures that new names can uniquely and deterministically be created.

3. Addresses of actors follow the hierarchical addressing scheme.

4. The initial global state of the system has at most one *active message*. Otherwise, a non-deterministic choice could be made as more than one rule could be active to consume the messages.

5. The computation performed by each actor after receiving a message must have no unquantified non-determinism; however, there may be probabilistic choices in the application of an actor rewrite rule.

6. The messages produced by an actor in a particular computation (e.g., upon receiving a message) are solely *scheduled messages*.

7. There is no non-determinism in the choice of rewrite rules, i.e., for each message there is at most one rule that can be applied.

are sufficient to ensure the absence of unquantified non-determinism. Therefore, distributed system specifications satisfying conditions (1)–(7) and having some probabilistic rewrite rules can be formally analyzed by statistical model checking methods.

## 4.6. Using PVeStA to Statistically Analyze Specifications based on the Modularized Actor Model

As mentioned in Section 4.2.3, PVeStA can be used for statistical model checking and quantitative analysis of probabilistic rewrite theories expressed in Maude. In this thesis, this method is used for the formal analysis of specifications based on the extended actor model and the *SAMPLER* module (see Appendix B.1).

### 4.6.1. The module *APMAUDE*

PVeStA expects the module *APMAUDE* in specifications it performs model checking on.

```
mod APMAUDE is
  protecting ACTOR-MODEL .
  protecting SCHEDULER .
  protecting NAT .
  protecting FLOAT .

  var C : Config .
  var gt : Float .
  var SL : ScheduleList .

  op initState : -> Config .
  op sat : Nat Config -> Bool .
  op val : Nat Config -> Float .
  op getTime : Config -> Float .
  eq getTime(C {gt | SL}) = gt .

  op limit : -> Float .
  op tick : Config -> Config .
  eq tick(C) = run(C, getTime(C) + limit) .
endm
```

The module protects a definition of the extended actor model (module *ACTOR-MODEL*) and a definition of the scheduler for the extended actor model (module *SCHEDULER*). It further defines the operators `initState`, which is a constant operator that represents the initial configuration, `sat` and `val`, which are used to define properties in Maude, and `getTime`, which returns the global time of a configuration. The `tick` operator takes a configuration and calls the `run` operator. The maximum time that should pass in the tick (i.e., the logical time that the global time should advance during the run of a sample) is thereby given by the constant `limit`.

### 4.6.2. Running PVeStA

PVeStA is a client-server application. To perform an analysis using the tool, first a server application needs to be started on each server. The server is started using the command

```
java -jar pvesta-server.jar [PORT]
```

It is possible to start multiple servers on a single physical machine by assigning a different port to each of the instances. The port is definable as an optional parameter. It is recommendable to only start as many servers on a physical machine as CPU cores should be used for the analysis on that machine.

The second step is to create a file that contains a list of server addresses. Addresses are of the form `IP:PORT` and should be provided as a space or line separated list in the file.

Finally, the client application is started using the command

```
java -jar pvesta-client.jar
-m [MODEL FILE]
-f [FORMULA FILE]
[-l [SERVER LIST FILE]]
[-k [LOAD FACTOR]]
[-d1 [DELTA1]]
[-d2 [DELTA2]]
[-a [ALPHA]]
[-b [BETA]]
[-ps [STOPPING PROBABILITY]]
[-pd [DISCOUNT PROBABILITY]]
[-cs [CACHE SIZE]]
[-s [MAXIMUM SAMPLE SIZE]]
[-arg [MODEL ARGUMENT]]
```

For the descriptions of the application parameters we refer to [73, 18, 72] and Section 4.2.3. If not otherwise mentioned, the results in this thesis rely on a 99% confidence interval of size at most 0.01.

Chapter 5

# Guaranteeing High Availability under Distributed Denial of Service Attacks

Availability is a key quality of service property of Cloud-based services. However, such services can easily be overwhelmed by a Distributed Denial of Service (DDoS) attack. In this chapter, we present solutions how Cloud-based services can be made resilient to DDoS attacks with minimum performance degradation. In the following, we:

1. give an introduction to Denial of Service attacks (Section 5.1),

2. describe the Adaptive Selective Verification (ASV) protocol (Section 5.2),

3. specify an ASV Wrapper meta-object in Maude and formally analyze a client-server system under attack using the ASV Wrapper (Section 5.3),

4. and, finally, extend the ASV protocol with a Server Replicator meta-object (ASV$^{++}$), which exploits the Cloud's capacity for provisioning more servers on demand, and then formally analyze the extended protocol under various settings (Section 5.4).

The Maude specifications shown in this chapter are based on the modularized actor model described in Chapter 4 and make use of the Russian Dolls pattern which makes the specifications, as meta-objects, highly reusable. Furthermore, the Maude specification of ASV is adapted from [17], and the result analyses in Section 5.3, although they consider a wider range of attack volumes than in [20], are in agreement with the model checking results in [20].

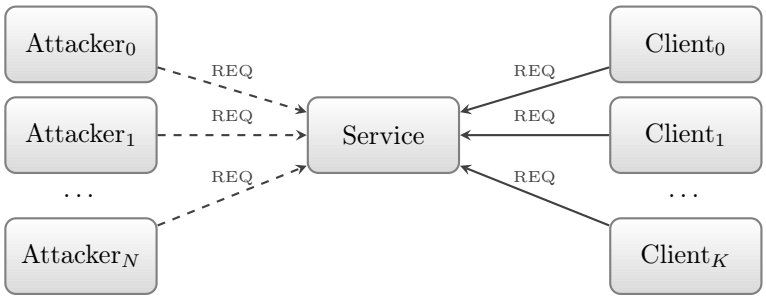



**Figure 5.1.:** Illustration of a service under a DDoS attack

## 5.1. Introduction to Denial of Service Attacks

The goal of a denial of service attack is to make a service temporarily or indefinitely unavailable to its consumers. A common method of attacking a service is to flood the service with requests which causes the computational resources to be overwhelmed. As a consequence, the system which runs the service experiences a slowdown or even crashes.

**D**enial **o**f **S**ervice (DoS) is one of the six categories defined in Micrsoft's STRIDE threat model [64] that defines a model for the security analysis of software-based systems: "Denial of service (DoS) attacks deny service to valid users — for example, by making a Web server temporarily unavailable or unusable. You must protect against certain types of DoS threats simply to improve system availability and reliability.".

We speak of a **D**istributed **D**enial **o**f **S**ervice attack (DDoS), when more than one system simultaneously attack a service. DDoS attacks are commonly run against web services and use computational resources that are connected to the Internet to start sending false requests to the service under attack. Usually attackers do not have direct access to hundreds or even thousands of machines that could be needed to overwhelm a service. Instead, attackers use so-called Botnets, which are collections of compromised computers that are connected to the Internet. Usually the systems in a Botnet have been infiltrated by a piece of software. This piece of software is often hidden to the user of the system and starts attacking a specific target after an initialization message is received. Figure 5.1 illustrates a service under a DDoS attack. Attackers send false requests that increase the workload on the system which the service is running on. Clients sending normal requests are then facing an increased time-to-service and may not receive a response at all.

#### DDoS attack on MasterCard.com

One of the more recent and prominent cases of a DDoS attack has been part of the "Operation Payback" campaign by a group called "Anonymous" [70]. In late 2010, the group orchestrated a DDoS attack against websites of financial institutions such as MasterCard.com and PayPal.com. On December 8, 2010 at 07:53 AM EDT, MasterCard issued a statement that "MasterCard is experiencing heavy traffic on its external corporate website — MasterCard.com. We are working to restore normal speed of service. There is no impact whatsoever on our cardholders ability to use their cards for secure transactions." [56]. In fact, by that time, the DDoS attack brought the website down and made their web presence unavailable for most costumers. The attack on the servers lasted for several hours. At 02:53

PM on December 8, 2010, MasterCard issued a second statement in which they reported that "MasterCard has made significant progress in restoring full-service to its corporate website. Our core processing capabilities have not been compromised and cardholder account data has not been placed at risk. While we have seen limited interruption in some web-based services, cardholders can continue to use their cards for secure transactions globally." [57].

The attack on the corporate website of MasterCard and the resulting downtime, which lasted for several hours, show that DDoS attacks pose a severe threat for web-based services. Even though, in this case, no core business critical services were affected, a scenario in which such services are neutralized can easily be imagined.

## 5.2. The ASV Protocol

In the *shared channel* attacker model [41], a legitimate sender and an attacker share a packet communication channel. The attacker does not have full control over the communication channel. Instead, both, the attacker and the sender, have each limited amounts of bandwidth that they can use at their disposal. In contrast, in the Dolev-Yao model, attackers are able to drop all packets of a legitimate sender (i.e. attackers are always able to perform a DoS attack) which makes the model unsuitable for DoS analysis.

The *Adaptive Selective Verification* (ASV) protocol [53] assumes the shared channel attacker model and is a cost-based DoS and DDoS resistant protocol in which bandwidth is used as the currency. The protocol is characterized by the application of two concepts: adaption and selection.

**Adaption.** Clients do not have access to explicit information about the nature and intensity of a current attack. They attempt to adapt to a current level of attack on a service by exponentially increasing the number of requests that they send within consecutive time windows. As client bandwidth is limited, the client increases the number of requests only up to a certain threshold. On the other side, the server adapts to the level of the attack by dropping packets, with a higher probability as the attack becomes more severe.

**Selection.** Servers collect a random sample of a bounded size among incoming requests and process them at their mean processing rate.

In the following we give a more precise definition of the ASV protocol's behavior. We assume a simple request-response (e.g. remote procedure call) message exchange pattern [81] between clients and a server. A client sends request packets ($REQ$) to the server. In response, the server sends response packets ($ACK$) back to the client. The server's mean processing rate (in $REQ$s per second) is denoted by $S$, while the clients' arrival rate is denoted by $\rho$, with $0 < \rho_{min} \leq \rho \leq \rho_{max} \leq 1$. Clients have a timeout window of $T$ seconds which is set to the expected worst case round-trip delay between the client and the server. $T$ is known to the clients as well as to the server. The timeout windows are denoted by $W_i$, with $i$ indicating the $i$th timeout window. In any given time window $W_i$, the system is under a DDoS attack which is flooding the server with $REQ$s at an attack rate which is denoted by $\alpha(W)$ with $0 \leq \alpha(W) \leq \alpha_{max}$. The attack aims to overwhelm the server by sending more $REQ$s per second (the attacker sends $\alpha(W) * S$ fake $REQ$s per second) than the server can

handle ($S$). Given that the channel capacity is not exceeded, it is assumed that no packets are lost during transmission. Thus, a server cannot guarantee that an individual $REQ$ will be processed, if $\rho_{max} + \alpha max > 1$. The clients' replication threshold, i.e., the maximum number of consecutive time windows a client tries to send $REQ$s to the server before it gives up, is denoted by $J$, with $J = \lceil log(\alpha_{max}/\rho_{min})/log(2) \rceil$. In the original specification of the ASV protocol, $J$ is also referred to as the *retrial span*.

**Behavior of the adaptive clients.** Clients join the system at the rate $\rho$ and send $REQ$s to the server. When no $ACK$ is received within the timeout window of $T$ seconds, the client adaptively increases the number of $REQ$s it sends in the succeeding time window up to a maximum. The client-side protocol proceeds as follows.

C1. Initialize $j \leftarrow 0$ and $J \leftarrow \lceil log(\alpha_{max}/\rho_{min})/log(2) \rceil$.

C2. Send $2^j$ $REQ$s to the server.

C3. If no $ACK$ is received within $T$ seconds, set $j \leftarrow j + 1$. Otherwise, if an $ACK$ is received within $T$ seconds, succeed and exit the system.

C4. If $j \leq J$, continue at C2. Otherwise, fail and exit the system.

**Behavior of the selective server.** The server keeps a bounded buffer of incoming $REQ$s and, after $T$ seconds, answers the $REQ$s stored in the buffer. If a server receives less $REQ$s than the buffer size, the server answers all incoming $REQ$s. Otherwise, the server probabilistically drops or replaces $REQ$s. The server-side protocol proceeds as follows.

S1. Initialize the window count $k \leftarrow 1$.

S2. In a window $W_k$, store the first $\lfloor S*T \rfloor$ $REQ$s in the buffer. If the timeout window ends and less than $\lfloor S * T \rfloor$ $REQ$s have been received, go to S4. Otherwise, set the packet count to $j \leftarrow \lfloor S * T \rfloor + 1$.

S3. The $j$th $REQ$ is accepted with probability $\lfloor S*T \rfloor / j$ and dropped with probability $1 - \lfloor S * T \rfloor / j$. If accepted, a uniformly distributed random $REQ$ in the buffer is replaced with the $j$th $REQ$. Then, increase the packet count ($j \leftarrow j + 1$) and repeat the step until the timeout window ends.

S4. Send $ACK$s for each of the $REQ$s in the buffer. Then, empty the buffer, set $k \leftarrow k + 1$ and continue at setp S2.

A manual analysis of the protocol described above is a demanding task [53]. In this chapter, we specify the ASV protocol as a wrapper meta-object in Maude adapting a similar wrapper specification in [17], and we formally analyze a ASV protected client-server system that is facing a DDoS attack. To take advantage of the Cloud's capacity for provisioning more servers on demand, we also specify a Server Replicator meta-object. We then combine both, the ASV and the Server Replicator meta-objects on top of the client-server system, and formally analyze the composed system.

## 5.3. Maude-based Analysis of the ASV Protocol

The Maude-based formal model of the ASV protocol is based on the modularized actor model and the Russian Dolls model that are introduced in Chapter **??**. We follow a mod-

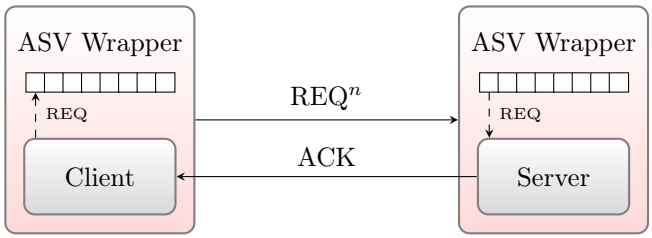



**Figure 5.2.:** Overview of a Cloud-based service setup using the ASV Wrapper

ular and adaptive approach in specifying the model by using distributed object reflection, based on highly reusable meta-object patterns. The ASV protocol is defined by a ASV Wrapper meta-objects for the client and the server (see Figure 5.2). This definition of a generic protocol wrapper can be applied to wide variety of client-server protocols that are originally not protected against DDoS attacks[1]. In the following, we describe the Maude-based specifications of the generic wrappers and their application to a simple client-server request-response system that is under a DDoS attack (see the scenario described in Section 5.2). We then formally analyze the formal model using statistical model checking.

### 5.3.1. Description of the ASV specification in Maude

Figure 5.3 gives an overview of the specification. In the following, we describe each module in more detail. The modules *ASV-SERVER* and *ASV-CLIENT* describe the ASV protocol behavior for the server and the client. Both are generic and take the theory *ASV-SERVER-INTERFACE* or *ASV-CLIENT-INTERFACE* as a parameter. The generic parameter defines the specific behavior of the wrapped server or client.

#### The module *SIMPLE-SERVER-COMMON*

The functional module *SIMPLE-SERVER-COMMON* specifies the simple server that is used in our setting. The server is modelled as a flat actor. The operator

```
op Server : -> ActorType .
```

defines the actor type of the simple server. The operators

```
op REQ : Address -> Contents .
op ACK : -> Contents .
```

define the messages that the simple server accepts from the outside. Clients can send a message of the form `S <- REQ(C)` with `S` being the address of the server and `C` the address of the client. The server answers this immediately with a message of the form `C <- ACK`. In practice, of course, request messages will be of the form `C <- REQ(q,C)`, with `q` an appropriate query, and answers from the server will be of the form `ACK(a)`, with `a` an appropriate answer; however, for purposes of analysing the effectiveness of the ASV defense under DDoS attack, the specific form that `q` and `a` can take in each underlying client-server system are immaterial and therefore ignored.

[1] In its current form, the specified generic ASV Wrapper does only support protocols that are based on a request-response message exchange pattern. The support of other message exchange patterns and more complex forms of orchestration are proposed as future work.

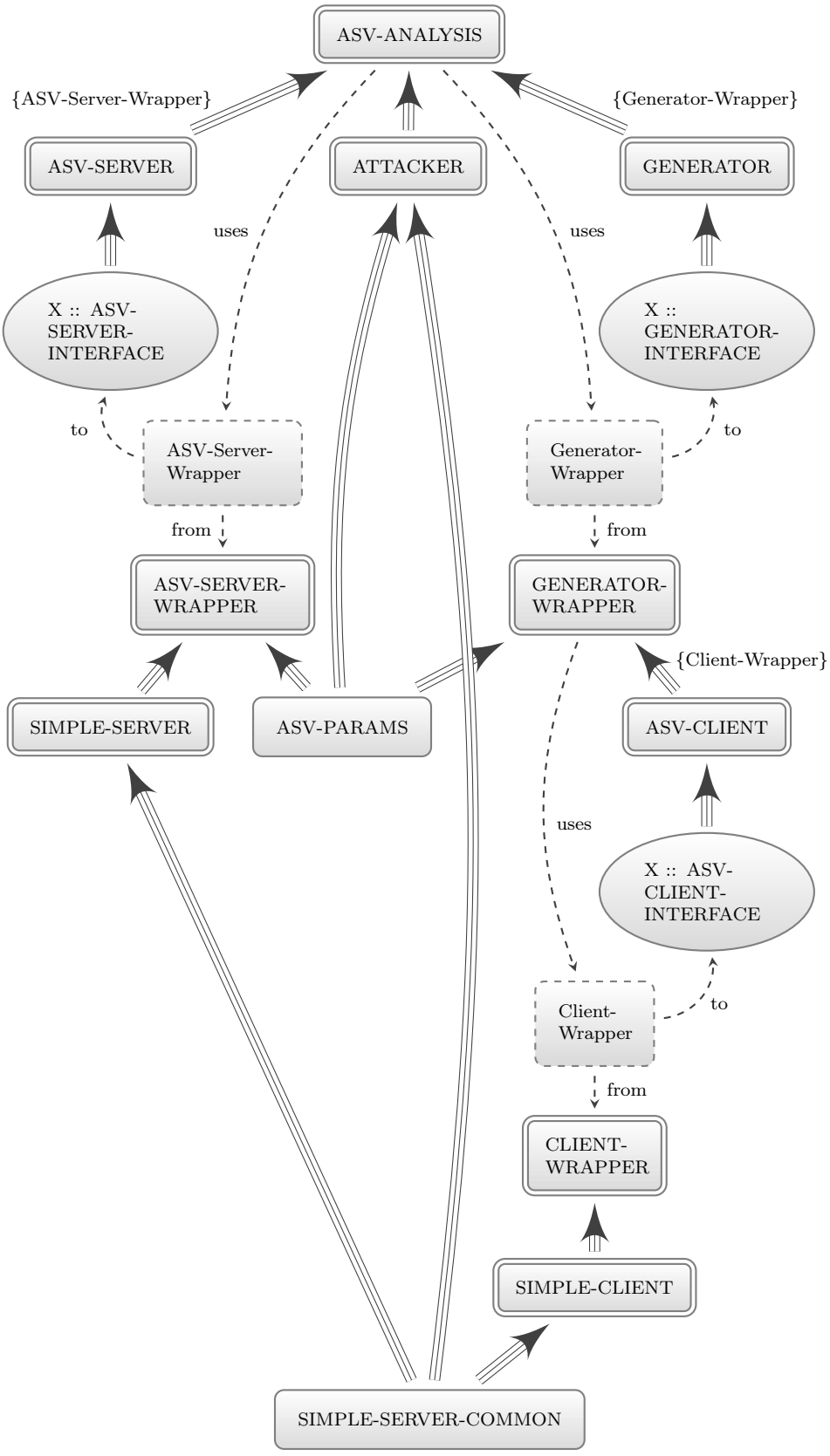



**Figure 5.3.:** Overview of the ASV analysis specification

### The module *SIMPLE-SERVER*

The system module *SIMPLE-SERVER* specifies the behavior of the simple server. The variables

```
vars SA CA : Address .
var AS : AttributeSet .
var gt : Float .
```

are used in the rewrite rule

```
rl [SIMPLE-SERVER-RECEIVE-REQ] :
  < SA : Server | AS >
  {gt , SA <- REQ(CA)}
=>
  < SA : Server | AS >
  [gt , CA <- ACK] .
```

in which the server answers a client with a message of the form `CA <- ACK` when it receives a message of the form `SA <- REQ(CA)`.

### The module *SIMPLE-CLIENT*

The system module *SIMPLE-CLIENT* specifies the client as a flat actor. The sort

```
sort Status .
```

and the operators

```
ops waiting connected : -> Status .
```

specify the state of the actor: The client is either waiting to get a response (`waiting`) from the server or has already been served by the server (`connected`). The actor type of the client is defined by the operator

```
op Client : -> ActorType .
```

and the internal state is specified by the operators

```
op status:_ : Status -> Attribute [gather(&)] .
op tts:_ : Float -> Attribute [gather(&)] .
```

which store the state of the actor and the time between the request of the actor and the answer from the server. The variables

```
var AC : Address .
var AS : AttributeSet .
vars gt tts : Float .
```

are used in the rewrite rule

```
rl [CLIENT-RECEIVE-ACK] :
  < AC : Client | status: waiting, tts: tts, AS >
  {gt, AC <- ACK}
=>
  < AC : Client | status: connected, tts: (gt - tts) ,AS > .
```

which specifies the behavior of a client upon receiving an answer from the server: It changes its state to `connected` and calculates the time between the request and the current time[2].

[2]The attribute `tts` is set to the current global time when the client sends the request to the server.

### The module *ASV-PARAMS*

The functional module *ASV-PARAMS* specifies the parameters that are used in subsequent specifications. The operators

```
op attacker-count : -> Nat .
op APS : -> Float .
```

specify the number of attackers and the number of attacks per second. The operator

```
op rho : -> Float .
```

specifies the client attack rate. The operator

```
op S : -> Float .
```

represents the number of packets a server can handle per server timeout. The operators

```
op Ts : -> Float .
op Tc : -> Float .
```

specify the timeout period of the server (`Ts`) and of the client (`Tc`). In the description of the ASV protocol in Section 5.2, the timeouts have the same value and are denoted by $T$. Finally, the operator

```
op J : -> Nat .
```

defines the retrial span.

### The module *ATTACKER*

The system module *ATTACKER* models the kind of DoS attacker used in our setting. The attacker is modeled as a flat actor. The operator

```
op Attacker : -> ActorType .
```

represents the actor type of the attacker. The operator

```
op attacker-period : -> Float .
```

which is defined by the equation

```
eq attacker-period = 1.0 / (APS * float(attacker-count)) .
```

defines the time periods in which the attacker periodically sends new requests to the server. Instead of modelling `attacker-count` DDoS attackers, we decided to model only one single DoS attacker that performs the attack for the intended number of attackers. Since an attacker sends `APS` requests per second, the single attacker actor we use attacks with `(APS * float(attacker-count)` requests per second. The attributes

```
op sua:_ : Address -> Attribute [gather(&)] .
op acount:_ : Nat -> Attribute [gather(&)] .
op success-cnt:_ : Nat -> Attribute [gather(&)] .
```

specify the attacker's state: The attribute `sua` contains the address of the server under attack, the attribute `acount` contains the count of requests that have already been sent, and the attribute `success-cnt` counts the number of requests that have actually been answered by the server. The operator

```
op attack! : -> Contents .
```

is used as the contents of a self-addressed message to periodically trigger an attack. In the following definitions of the behavioral rules, the variables

```
vars A SA : Address .
var N : Nat .
var gt : Float .
var AS : AttributeSet .
```

are used.

In the rewrite rule

```
rl [ATTACKER-ATTACK] :
< A : Attacker | sua: SA, acount: N, AS >
{gt, A <- attack!}
=>
< A : Attacker | sua: SA, acount: s(N), AS >
[gt, SA <- REQ(A)]
[gt + attacker-period, A <- attack!] .
```

the attacker reacts to a self-addressed `attack!` message by sending a request to the server under attack and scheduling a further `attack!` message. Finally, the rewrite rule

```
rl [ATTACKER-CONSUME-ACKS] :
< A : Attacker | success-cnt: N, AS >
{gt, A <- ACK}
=>
< A : Attacker | success-cnt: s(N), AS > .
```

specifies how the attacker consumes the answers from the server and increments the attribute `success-cnt`.

### The theory *ASV-SERVER-INTERFACE*

The theory *ASV-SERVER-INTERFACE* defines the interface that the ASV server wrapper needs to know about the server it wraps. The operator

```
op maxLoadPerServer : -> Float .
```

defines the maximum amount of packages a server can handle within a timeout period. The operator

```
op asv-server-timeoutperiod : -> Float .
```

defines the timeout period.

### The theory *ASV-CLIENT-INTERFACE*

Similar to the theory *ASV-SERVER-INTERFACE*, the theory *ASV-CLIENT-INTERFACE* specifies the interface an ASV client wrapper needs to know about the wrapped client. The operator

```
op uniqueMessageId : Contents -> Nat .
```

needs to be implemented by the client. For each message that a client sends, the operator returns a unique identifier that is used to differentiate between different request-reply message pairs. The operator

```
op asv-client-timeoutperiod : -> Float .
```

specifies the timeout period for the ASV client. Lastly, the operator

```
op asv-client-max-retry-count : -> Nat .
```

defines the maximum number of retries per message.

### The module *ASV-SERVER*

The system module *ASV-SERVER* is parametrized by the theory *ASV-SERVER-INTERFACE* and describes the generic ASV server wrapper. As already mentioned, the ASV server is specified as a Russian dolls actor that wraps the actual server in its configuration. The actor type

```
op ASV-Server : -> ActorType .
```

specifies the type of the ASV server. The internal state of the ASV server is represented by the attributes

```
op msg-buffer:_ : MsgList -> Attribute [gather(&)] .
op msg-count:_ : Float -> Attribute [gather(&)] .
op internal-addr:_ : Address -> Attribute [gather(&)] .
```

which contains the message buffer (`msg-buffer`), the overall count of the messages (`msg-count`), and the address of the internal server (`internal-addr`). The ASV server periodically sends a message with the contents

```
op asv-server-timeout : -> Contents .
```

to itself to trigger the buffered messages to be sent to the wrapped server after each timeout period. The variables

```
vars SA IA A : Address .
var I : Nat .
var L : MsgList .
var AS : AttributeSet .
var C : Config .
var gt : Float .
var REPLACE? : Bool .
var CO : Contents .
var CNT : Float .
```

are used in the definition of the following rewrite rules that specify the behavior of the ASV server.

If a message (other than the self-addressed message with the contents `asv-server-timeout`) arrives at the ASV server, the server performs the adaptation and selection as specified above and either drops the message or stores it in the buffer. The conditional rewrite rule

```
crl [ASV-SERVER-RECEIVE-MSG] :
  < SA : ASV-Server | msg-count: CNT, msg-buffer: L,
    internal-addr: IA, AS >
  {gt, SA <- CO}
=>
  if (float(L .size) >= floor(maxLoadPerServer)) then
    if (sampleBerWithP(floor(maxLoadPerServer) / (CNT + 1.0))) then
      < SA : ASV-Server | msg-count: CNT + 1.0,
        msg-buffer: L [ sampleUniWithInt(L .size) ]:= (IA <- CO),
        internal-addr: IA, AS >
```

```
    else
      < SA : ASV-Server | msg-count: CNT + 1.0, msg-buffer: L,
        internal-addr: IA, AS >
    fi
  else
    < SA : ASV-Server | msg-count: CNT + 1.0, msg-buffer: ((IA <- CO) ; L),
      internal-addr: IA, AS >
  fi
if
CO =/= asv-server-timeout .
```

defines this behavior. If the size of the buffer is already exceeded, a coin is tossed to decide whether the message should be dropped or not, i.e., it is randomly decided according to a Bernoulli distribution with success probability `floor(maxLoadPerServer)/ (CNT + 1.0)`. If the message is not dropped, a uniformly chosen request in the list is replaced by the incoming request. The second kind of messages a server has to handle is the periodical `asv-server-timeout` message. The rewrite rule

```
rl [ASV-SERVER-TIMEOUT] :
  < SA : ASV-Server | msg-count: CNT, msg-buffer: L, config: C, AS >
  {gt, SA <- asv-server-timeout}
=>
  < SA : ASV-Server | msg-count: 0.0, msg-buffer: mtMsgList,
    config: createMsgs(L, gt) C, AS >
  [gt + asv-server-timeoutperiod, SA <- asv-server-timeout] .
```

reacts to the timeout message and sends all buffered requests to the wrapped server using the `createMsgs` operator. Additionally, the periodical message is sent.

Since the wrapped server internally produces answers, the ASV server wrapper needs to take the internal messages out of its configuration and emit them in the configuration it is located in. The two rewrite rules

```
crl [ASV-SERVER-TAKE-MESSAGES-OUT1] :
  < SA : ASV-Server | config: {gt, A <- CO } C, AS >
=>
  < SA : ASV-Server | config: C, AS >
  [gt, A <- CO]
if | A | <= | SA | .

crl [ASV-SERVER-TAKE-MESSAGES-OUT2] :
  < SA : ASV-Server | config: {gt, A <- CO } C, AS >
=>
  < SA : ASV-Server | config: C, AS >
  [gt, A <- CO]
if | A | > | SA | /\ prefix(A, | SA |) =/= SA .
```

describe this behavior. There are two cases in which a message needs to be taken out of the internal configuration: (i) if the address of the receiver of the message is smaller (i.e., it is located at a higher level in the address hierarchy) than the ASV server's address, or (ii) if the address of the receiver is bigger (i.e., it is located at a lower level in the address hierarchy) but the ASV server's address is not contained as a prefix in the receiver's address (i.e., the message is addressed to another subtree of the address hierarchy).

### The module *ASV-CLIENT*

The system module *ASV-CLIENT* is parametrized by the theory *ASV-CLIENT-INTERFACE*. The module describes the behavior of the ASV client wrapper. The wrapper is modeled as a Russian dolls actor and contains the wrapped client in its configuration. Since the ASV client wrapper generally wraps any client with any kind of request-reply communication, the ASV client wrapper stores the requests the client sends to the server in a message buffer. The ASV client periodically sends a timeout message to itself and, upon receiving one, it replicates the individual requests according to the current ASV adaptation parameter. If the server answers a specific request, the request is removed from the buffer and the answer is sent to the client. In order to identify request-reply pairs, the client needs to implement the `uniqueMsgId` operator.

The operator

```
op ASV-Client : -> ActorType .
```

defines the actor type of the ASV client wrapper. The attributes

```
op message-buffer:_ : NatMsgNatSet -> Attribute [gather(&)] .
op client:_ : Address -> Attribute [gather(&)] .
op server:_ : Address -> Attribute [gather(&)] .
```

represent the internal state of the ASV client. The attribute `message-buffer` contains the buffered messages[3], the attribute `client` the address of the wrapped client, and the attribute `server` the address of the ASV server. The ASV client periodically sends a message with the contents

```
op asv-client-timeout : -> Contents .
```

to itself in order to duplicate the buffered messages according to the ASV client adaption strategy and to send the buffered messages to the server. The operators

```
op inc : NatMsgNatSet -> NatMsgNatSet .
op msgs : Float NatMsgNatSet -> Config .
op msgs : Float NatMsgNat -> Config .
```

are used to increment the replication counter for all messages in, and to create the replicated messages from a term of sort `NatMsgNatSet`. For the sake of brevity, the operators' behavior is not described here.

In the following, the variables

```
var A CA SA : Address .
var MB : NatMsgNatSet .
var ID : Nat .
var gt : Float .
var CO : Contents .
var N : Nat .
var C : Config .
var AS : AttributeSet .
var E : NatMsgNat .
var M : Msg .
```

[3] The sort `NatMsgNatSet` represents a mapping between a unique message id and a message together with a natural number that counts how often it has already been replicated. The sort is specified in the module *NATMSGNATSET*

are used. The conditional rewrite rule

```
crl [ASV-CLIENT-RECEIVING-REQUESTS-FROM-WRAPPED-CLIENT] :
  < A : ASV-Client | message-buffer: MB, server: SA,
    config: {gt, SA <- CO} C, AS >
=>
  < A : ASV-Client | message-buffer: (ID, SA <- CO, 1) MB, server: SA,
    config: C, AS > [gt, SA <- CO]
if ID := uniqueMessageId(CO) .
```

consumes a message that is sent from the wrapped client to the server. The message together with a replication count of 1 is added to the message buffer. Additonally, the message is directly sent. The conditional rewrite rule

```
crl [ASV-CLIENT-RECEIVE-ACK] :
  < A : ASV-Client | message-buffer: MB, client: CA, config: C, AS >
  {gt, CA <- CO}
=>
  if (ID in MB) then
    < A : ASV-Client | message-buffer: MB .remove(ID), client: CA,
    config: [gt, CA <- CO] C, AS >
  else
    < A : ASV-Client | message-buffer: MB, client: CA, config: C, AS >
  fi
if CO =/= asv-client-timeout .
```

handles the processing of answers from the server. If there is a matching request in the message buffer, the request is removed and the message is sent to the wrapped client. Otherwise, the message is dropped. The periodic timeouts are performed by the rewrite rule

```
rl [ASV-CLIENT-TIMEOUT] :
  < A : ASV-Client | message-buffer: MB, AS >
  {gt, A <- asv-client-timeout}
=>
  < A : ASV-Client | message-buffer: inc(MB), AS >
  msgs(gt, MB)
  [gt + asv-client-timeoutperiod, A <- asv-client-timeout] .
```

which creates the replicated messages from the message buffer using the operator `msgs` and increments the replication counter for all messages using the operator `inc`.

### The module *ASV-SERVER-WRAPPER*

The module *ASV-SERVER-WRAPPER* connects the module *ASV-SIMPLE-SERVER* with the theory *ASV-SERVER-INTERFACE*. The operators of the interface are specified as follows.

```
op maxLoadPerServer : -> Float .
eq maxLoadPerServer = Ts * S .

op asv-server-timeoutperiod : -> Float .
eq asv-server-timeoutperiod = Ts .
```

The view

```
view ASV-Server-Wrapper from ASV-SERVER-INTERFACE to ASV-SERVER-WRAPPER is
  op maxLoadPerServer to maxLoadPerServer .
  op asv-server-timeoutperiod to asv-server-timeoutperiod .
endv
```

finally connects the module with the theory and is later used to instantiate the *ASV-SERVER* module.

### The module *CLIENT-WRAPPER*

As with the module *ASV-SERVER-WRAPPER*, the module *CLIENT-WRAPPER* connects the module *SIMPLE-CLIENT* with the theory *ASV-CLIEN-INTERFACE*. The operators defined in the theory are specified and set to values according to the parameters specified in the module *ASV-PARAMS*.

```
op uniqueMessageId : Contents -> Nat .
op asv-client-timeoutperiod : -> Float .
op asv-client-max-retry-count : -> Nat .

eq asv-client-timeoutperiod = Tc .
eq asv-client-max-retry-count = J .

var A : Address .
var CO : Contents .
eq isRequest(REQ(A)) = true .
eq isRequest(CO) = false [owise] .
```

Finally, the view

```
view Client-Wrapper from ASV-CLIENT-INTERFACE to CLIENT-WRAPPER is
  op uniqueMessageId to uniqueMessageId .
  op asv-client-timeoutperiod to asv-client-timeoutperiod .
  op asv-client-max-retry-count to asv-client-max-retry-count .
endv
```

specifies a view from the theory *ASV-CLIENT-INTERFACE* to the module *CLIENT-WRAPPER*. This view is used to instantiate the *ASV-CLIENT* module.

### The module *GENERATOR-WRAPPER*

The module *GENERATOR-WRAPPER* specifies the generic parts of the generator that is used in our setting. The attribute

```
op server:_ : Address -> Attribute [gather(&)] .
```

represents an additional attribute of the generator that contains the address of the server. This attribute is needed to properly initialize the ASV clients. Additionally, the operators

```
op generator-spawn-period : -> Float .
op generator-create : AttributeSet Float Address -> Config .
```

specify the operators that are required by the theory *GENERATOR-INTERFACE*. The operator `generator-spawn-period` returns the period after which the generator periodically creates the configuration that is returned by reducing the operator `generator-create` with the generators attributes, the current global time, and a new address as arguments.

The variables

```
var AS : AttributeSet .
var A : Address .
var ASV : Address .
var gt : Float .
```

are used in the following equations. The equation

```
eq generator-spawn-period = 1.0 / (rho * S) .
```

specifies the spawn period of the server according to the parameters of the ASV protocol. The equation

```
eq generator-create((server: ASV, AS), gt, A) =
  < A : ASV-Client |
    config: < A . 1 > < A . 0 : Client | status: waiting, tts: gt >
    [gt, ASV <- REQ(A . 0)],
    message-buffer: mtNMNSet, server: ASV, client: A . 0 >
  [gt, A <- asv-client-timeout] .
```

defines the specific behavior of the operator `generator-create`. Thereby, a new ASV client wrapper is created that contains a simple client. The simple client directly sends a request and sets the request time to the current global time. Additionally, the periodic timeout message of the ASV client is sent.

Finally, the view

```
view Generator-Wrapper from GENERATOR-INTERFACE to GENERATOR-WRAPPER is
  op generator-spawn-period to generator-spawn-period .
  op generator-create to generator-create .
endv
```

connects the theory *GENERATOR-INTERFACE* and the *GENERATOR-WRAPPER*. The view is used to instantiate the module *GENERATOR*.

### The module *ASV-SR-INIT*

The system module *ASV-SR-INIT* connects the modules described before, defines the initial state of the system, and specifies operators which are used by the PVeStA tool. The initial state is defined by the equation

```
ceq initState =
  < SRA : ASV-Server |
    config:
      < SRA . 1 >
      < (SRA . 0) : Server | mt >,
    msg-count: 0.0, msg-buffer: mtMsgList, internal-addr: (SRA . 0) >
  < GA : Generator | count: 0, server: SRA, config: < GA . 0 > >
  < AA : Attacker | acount: 0, sua: SRA, success-cnt: 0 >
  {0.0 | nil}
  [0.0, AA <- attack!]
  [0.05 + generator-spawn-period, GA <- spawn]
  [asv-server-timeoutperiod, SRA <- asv-server-timeout]
if
  NG := < 0 > /\
  SRA := NG .new /\
  NG' := NG .next /\
  GA := NG' .new /\
  NG'' := NG' .next /\
  AA := NG'' .new .
```

which uses the variables

```
vars SRA GA AA : Address .
var NG NG' NG'' : NameGenerator .
var C : Config .
```

The initial state consists of the ASV server wrapper, which contains a simple server, the generator, the attacker, and the top-level scheduler. The periodic behaviors of the attacker, the generator, and the ASV server are initialized. Finally, the operators for the connection between Maude and the PVeStA tool are specified by the equations

```
eq val(0, C) = successRatio(C) .
eq val(1, C) = avgTTS(C) .
```

### 5.3.2. Statistical Model Checking Results

We use the specification of the ASV Wrapper meta-object together with the client-server setting to perform statistical model checking of QuaTEx formulas that analyse the behavior of ASV under DoS attacks using PVeStA.

The following QuaTEx formulas define the quantitative properties we want to analyze. The function *time*() denotes a state function that returns the global time value of the current configuration.

**Client success ratio.** The client success ratio defines the ratio of clients that receive an *ACK* from the server.

$$\begin{aligned} \mathit{successRatio}(t) = \textbf{if } \mathit{time}() > t \textbf{ then} \\ \frac{\mathit{countSuccessful}()}{\mathit{countClients}()} \\ \textbf{else } \bigcirc\,(\mathit{successRatio}(t)) \end{aligned}$$

with *countSuccessful*() being the result of equationally counting the number of clients whose status is *"connected"* (i.e., successful clients) and the total number of clients *countClients*() being equal to the client counter attribute (`count`) of the client generator object.

**Average TTS.** The average TTS is the average time it takes for a successful client to receive an *ACK* from the server.

$$\begin{aligned} \mathit{avgTTS}(t) = \textbf{if } \mathit{time}() > t \textbf{ then} \\ \frac{\mathit{sumTTS}()}{\mathit{countSuccessful}()} \\ \textbf{else } \bigcirc\,(\mathit{avgTTS}(t)) \end{aligned}$$

with *sumTTS*() being the result of adding up the times given by the successful clients' TTS attributes (`tts`). The number of successful clients *countSuccessful*() is computed as described above for the client success ratio.

**Number of client requests.** The number of client requests represents the number of *REQ*s sent by legitimate clients (not including *REQ*s sent by attackers).

$$\begin{aligned} \mathit{requests}(t) = \textbf{if } \mathit{time}() > t \textbf{ then } \mathit{countRequests}() \\ \textbf{else } \bigcirc(\mathit{requests}(t)) \end{aligned}$$

with *countRequests*() being equal to the client request counter attribute of the server object.

For the statistical model checking of the aforementioned properties, we fix the mean server processing rate $S$ to 600 packets per second, the timeout window $T$ to 0.4 seconds, the retrial span $J$ to 7, and the client arrival rate $\rho$ to 0.08. Additionally, an initial generation delay of 0.05 seconds is introduced and the duration of a simulation is set to 30 seconds. The values of the parameters correspond to the values chosen in [53, 20]. The properties are checked for various attack conditions represented by the constant $\alpha$ values 0.6666, 3.3333, 6.6666, 13.3333, 26.6666, 40.0, 53.3333, 66.6666, 80.0, 93.3333, 106.6666, 120.0, and 133.3333, which correspond to 1, 5, 10, 20, 40, 60, 80, 100, 120, 140, 160, 180, and 200 attackers (each attacker issues 400 fake *REQ*s per second). It is of note that already 1.5 attackers overwhelm the server. This represents a rather pessimistic setting (for the service provider) where the service is potentially highly resource-dependent.

To demonstrate the effectiveness of the ASV protocol and to compare the results of our modularized ASV model with the results in [53, 20], we check the aforementioned properties for two setups, using

a) the ASV protocol on the server- and client-side.

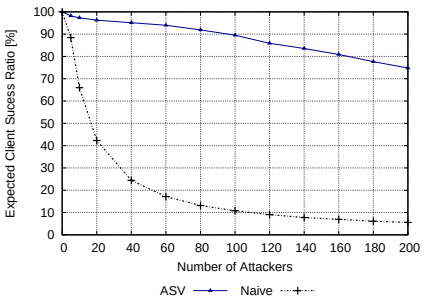
b) a non-adaptive client strategy, namely the *Naive* protocol, in which a client does not exponentially increase the number of *REQ*s after each time window but keeps sending a single *REQ*.

The parameters for each configuration that the properties are checked for are set in the module *ASV-PARAMS*. The initial configuration that is used for the statistical model checking using PVeStA is defined in the module *ASV-SR-INIT*.

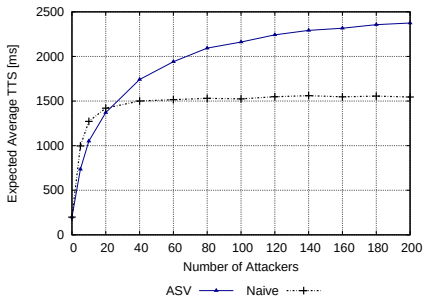
Figure 5.4 shows the results of the statistical model checking. The results clearly demonstrate the effectiveness of the ASV protocol compared to the non-adaptive client strategy. They also confirm the results in [53, 20]. Figure 5.4(a) shows that the ASV protocol can guarantee a high availability ($> 70\%$) of the system even when facing a heavy attack (200 attackers). ASV outperforms the Naive protocol at any level of attack regarding the client success ratio. As shown in Figures 5.4(b) and 5.4(c), the ASV protocol achieves this higher availability at the expense of an increased average TTS and an increased number of client requests (bandwidth), the only exception being that for less than 20 attackers the average TTS using the ASV protocol is slightly lower than the one using the Naive protocol.

## 5.4. ASV$^{++}$ — a 2-Dimensional Protection Mechanism against DDoS Attacks

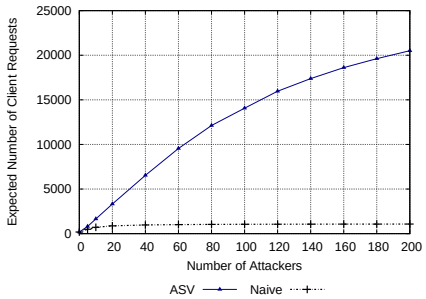
Cloud-based systems offer the possibility of provisioning resources on demand. For many applications, provisioning additional servers and replicating the service-providing application can alleviate a DDoS attack, because more requests per second can be handled. In




(a) Client success ratio




(b) Average TTS




(c) Number of client requests

**Figure 5.4.:** Performance of the ASV protocol compared to a non-adaptive client strategy

the following, we describe ASV$^{++}$, a two-dimensional protection mechanism against DDoS attacks, which uses two meta-objects, namely:

- the ASV Wrapper as an adaptive counter-measurement against DDoS attacks, which is applied by clients and the server, and
- a Server Replicator, which exploits the Cloud's capacity for provisioning more servers on demand.

### 5.4.1. The Server Replicator meta-object and the ASV$^{++}$ protocol

The Server Replicator (SR) is a meta-object that wraps around servers in a Cloud-based setup (see Figure 5.5). The servers are wrapped according to the Russian Dolls model. Thus, every message first has to pass through the meta-object (the Server Replicator) before a wrapped server can process it. In our abstraction, a Server Replicator distributes incoming messages among the wrapped servers (randomly according to a uniform distribution) and spawns new servers according to a server-side metric, i.e., replicates the service that is provided by the wrapped servers. In most cases, such a metric should be kept simple and easy to evaluate, because complex metrics can introduce a high additional workload on the system and can possibly increase the latency for the wrapped service. In the following, we use the number of incoming messages and a statically defined maximum load per server to define a simple metric. Thereby, the Server Replicator meta-object keeps track of the number of incoming messages and periodically checks if the result of a function of the two

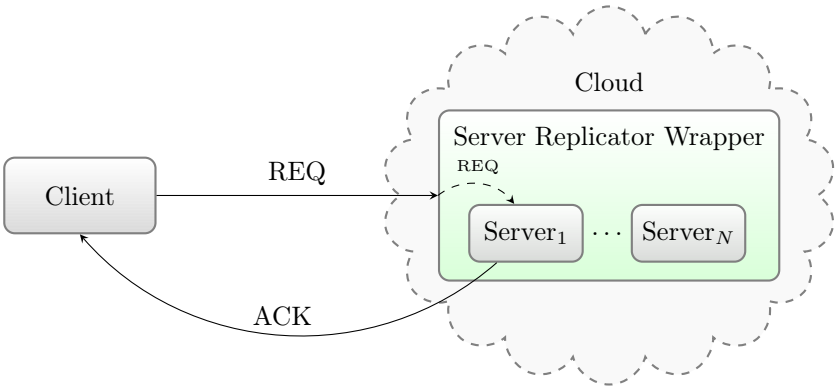



**Figure 5.5.:** Overview of a Cloud-based service setup using the Server Replicator Wrapper

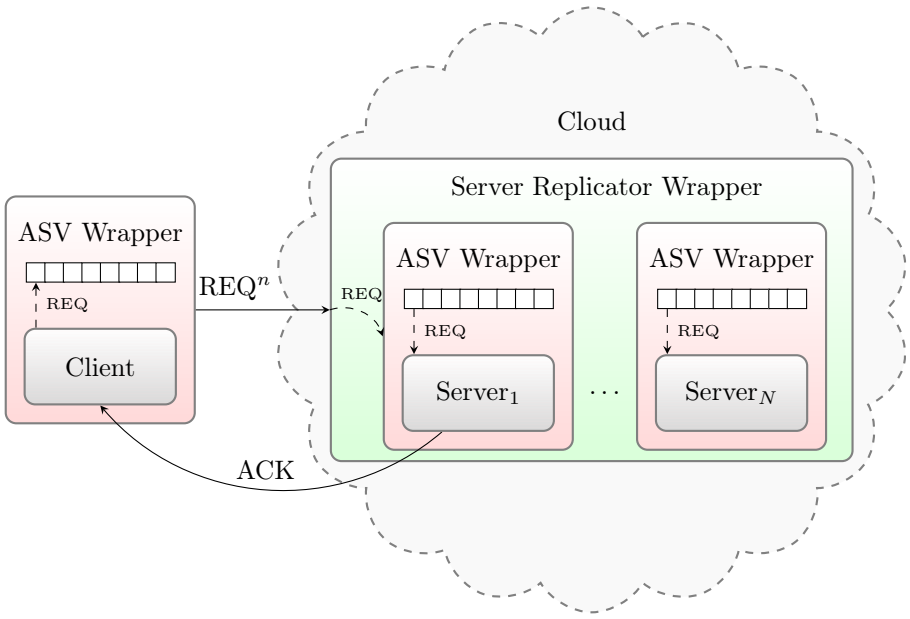



**Figure 5.6.:** Overview of a Cloud-based service setup using the ASV$^{++}$ protection

variables, number of incoming messages and maximum load per server, evaluates to an integer value that is greater than the number of wrapped servers and, if this is the case, the Server Replicator spawns a new server in its inner configuration.

The ASV$^{++}$ protocol combines the Server Replicator with the ASV Wrapper to achieve protection against DDoS attacks in two dimensions of adaptation: (i) adapting to increasingly more severe DoS attacks using the ASV mechanism; and (ii) adapting to the increasing need for server performance using the SR mechanism. An overview of a Cloud-based service setup that uses the ASV$^{++}$ protocol is shown in Figure 5.6. Clients and servers still adapt to a possible attack by exponentially increasing the number of requests on the client-side and by collecting a random sample of incoming requests on the server-side. However, the entry-point for all requests on the server-side is no longer a single server but the Server Replicator meta-object. The meta-object wraps around server instances which are themselves wrapped by the server-side ASV Wrapper. In the ASV$^{++}$ protocol, the maximum

load per server is equal to the product of its time out window size and the server's mean processing rate ($T * S$). For the replication metric, an additional parameter $k$, namely, the server overloading factor, is defined. The metric says that a new server is spawned by the Server Replicator, if the servers are overloaded by the overloading factor times their maximum load, e.g., for a factor of $k = 4$ and a maximum load of 10 *REQ*s per second per server, the Server Replicator spawns a new server if the wrapped servers have a load average that is greater than 40 *REQ*s per second per server. Thus, the factor $k$ defines by how much an ASV$^{++}$ protected system uses the selection mechanism of the server-side ASV Wrapper. An overloading factor of $k = 1$ means that the ASV protocol is nearly unused[4], an overloading factor of $k = \infty$ means that only the ASV protocol is used, because additional servers are never provisioned by the Server Replicator. We therefore propose an overloading factor $k$ with $1 < k < \infty$ to be used with the ASV$^{++}$ protocol.

### 5.4.2. Description of the ASV$^{++}$ specification in Maude

The modularity of the specification of the ASV protocol (see Section 5.3) allows for a simple extension of the model to include the Server Replicator meta-object. In the following, we describe the Maude modules that correspond to the specification of the Server Replicator meta-object. Figure 5.3 gives an overview of the specification which is used for the analysis of the ASV$^{++}$ protocol.

#### The theory *SERVER-REPLICATOR-INTERFACE*

The theory *SERVER-REPLICATOR-INTERFACE* defines the operators that the server replicator needs to use regarding the servers that are replicated. The operators

```
op maxLoadPerServer : Float -> Float .
op sr-check-period : -> Float .
```

specify the information that is needed for the server-side metric. The operator `maxLoadPerServer` represents the maximum load that one server can handle at a specific global time. The period after which the server replicator checks the server-side metric is specified by the operator `sr-check-period`. The constant operator

```
op sr-fwd-delay : -> Float .
```

specifies the delay that is introduced by forwarding a message from the server replicator to one of the replicated servers. If the server decides to spawn a new server, it calls the operators

```
op replicate : Address -> Actor .
op init : Address Float -> Config .
```

which create and initialize a new server. The operator `replicate` returns the new server and the operator `init` returns the messages that are needed to initialize the behavior of the wrapped server.

[4]The ASV protocol is only used, if the random distribution of incoming requests among the wrapped servers leads to the situation where a wrapped server is assigned with the processing of more requests than its buffer size in a specific time window.

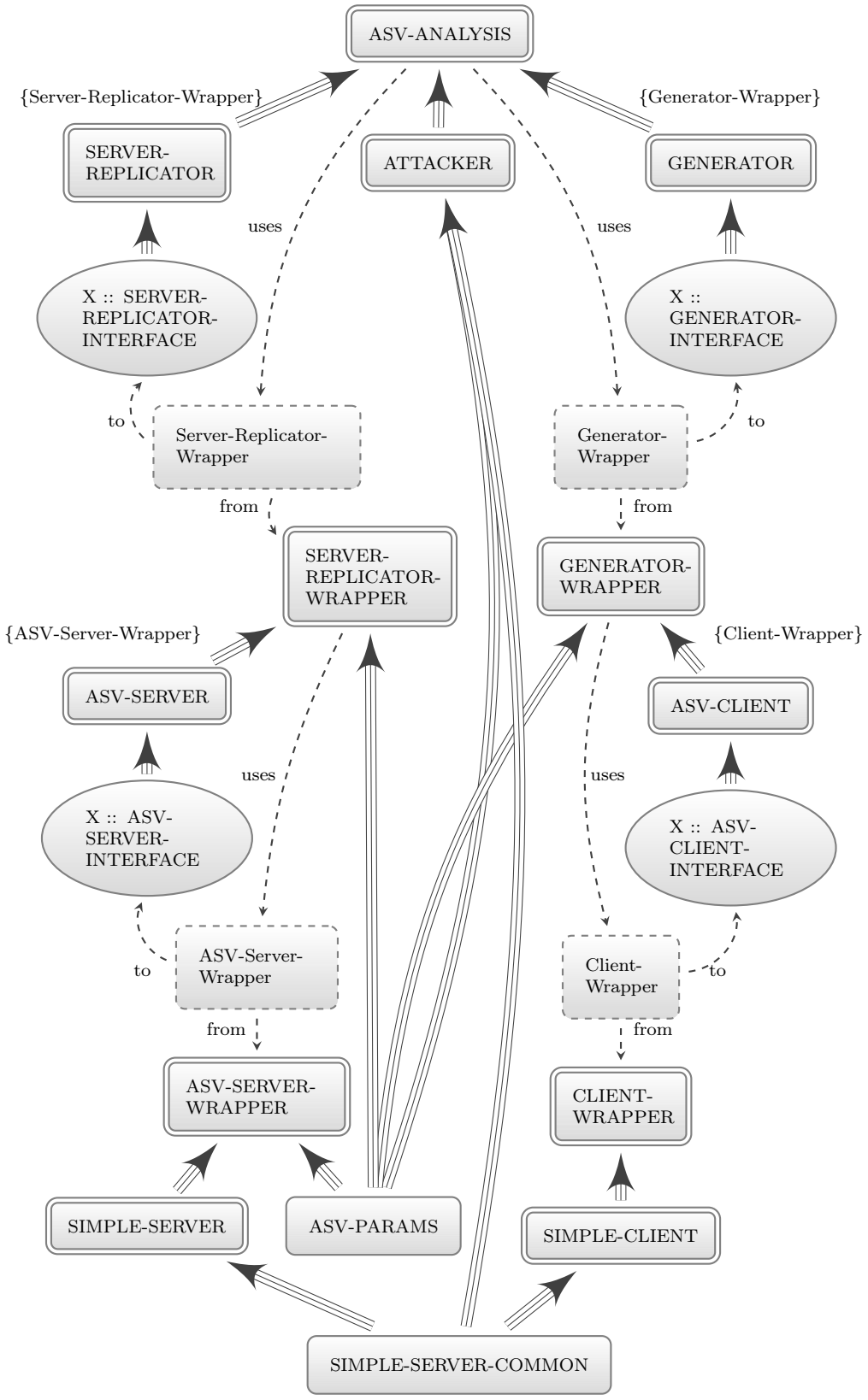



**Figure 5.7.:** Overview of the ASV$^{++}$ analysis specification

### The module *SERVER-REPLICATOR*

The system module *SERVER-REPLICATOR* specifies the behavior of the server replicator. The module is parametrized by the theory *SERVER-REPLICATOR-INTERFACE*, which specifies the internal behavior of a replicated server. As already mentioned before, the server replicator is specified as a Russian dolls actor. The operator

```
op ServerReplicator : -> ActorType .
```

defines the actor type of the server replicator. The internal state is represented by the attributes

```
op server-list:_ : AddressList -> Attribute [gather(&)] .
op msg-count:_ : Nat -> Attribute [gather(&)] .
```

which contain a list of addresses of the replicated servers and the total amount of messages that are forwarded to the replicated servers. The total amount of messages is used to decide when to replicate a new server. The server replicator periodically sends a message with contents

```
op check : -> Contents .
```

to itself to check whether a new server needs to be replicated. If a new server needs to be replicated, the server replicator sends a message with the message contents

```
op spawnServer : -> Contents .
```

to itself to trigger the replication. The operator

```
op pickRandom : AddressList -> Address .
```

takes a list of addresses as an argument and returns a uniformly randomly picked address from the list. The variables

```
var SL : AddressList .
var gt : Float .
var C : Config .
var N : Nat .
var AS : AttributeSet .
var CO : Contents .
var NG : NameGenerator .
vars A SA SRA : Address .
```

are used in the specification of the behavior of the server replicator. The equation

```
eq pickRandom(SL) = SL[sampleUniWithInt(SL .size)] .
```

defines the `pickRandom` operator which uniformly picks an address from a given list. The server-side metric is specified by the rewrite rule

```
rl [SERVER-REPLICATOR-CHECK] :
  < SRA : ServerReplicator | server-list: SL, msg-count: N, AS >
  {gt , SRA <- check }
=>
  < SRA : ServerReplicator | server-list: SL, msg-count: N, AS >
  if (max(float(N) / maxLoadPerServer(gt), 1.0) > float(SL .size)) then
    [gt, SRA <- spawnServer]
  else
    null
  fi
  [gt + sr-check-period, SRA <- check] .
```

which checks whether the servers are overloaded according to the operator `maxLoadPerServer`. If the servers are overloaded, a message with contents `spawnServer` is sent by the replicator to itself in order to trigger the replication of a server. Additionally, the next self-addressed check message is scheduled. The conditional rewrite rule

```
crl [SERVER-REPLICATOR-SPAWN-SERVER] :
  < SRA : ServerReplicator | config: NG C, server-list: SL, AS >
  {gt, SRA <- spawnServer }
=>
  < SRA : ServerReplicator |
    config: (NG .next) C replicate(SA) init(SA, gt),
    server-list: (SA ; SL), AS >
if SA := NG .new .
```

spawns a new server. A new name is generated using the name generator and the server is replicated and initialized with the new address. Finally, the name generator is updated and the address is added to the list of replicated servers.

If a message other than the self-addressed messages arrives at the server replicator, the replicator forwards the message to a uniformly chosen server. The rewrite rule

```
crl [SERVER-REPLICATOR-RECEIVE-MSG] :
  < SRA : ServerReplicator | server-list: SL, msg-count: N, config: C, AS >
  {gt , SRA <- CO }
=>
  < SRA : ServerReplicator | server-list: SL, msg-count: s(N),
    config: [gt + sr-fwd-delay, pickRandom(SL) <- CO] C, AS >
if
  CO =/= check /\ CO =/= spawnServer .
```

specifies this behavior. A random server is chosen using the `pickRandom` operator. As the replicated servers communicate with the outside, the server has to forward these messages to the outside. The rewrite rules

```
crl [SERVER-REPLICATOR-TAKE-MESSAGES-OUT1] :
  < SRA : ServerReplicator | config: {gt, A <- CO } C, AS >
=>
  < SRA : ServerReplicator | config: C, AS >
  [gt, A <- CO]
if | A | <= | SRA | .

crl [SERVER-REPLICATOR-TAKE-MESSAGES-OUT2] :
  < SRA : ServerReplicator | config: {gt, A <- CO } C, AS >
=>
  < SRA : ServerReplicator | config: C, AS >
  [gt, A <- CO]
if | A | > | SRA | /\ prefix(A, | SRA |) =/= SRA .
```

do the forwarding. The first rewrite rule forwards a message to the outside if the receiver's address is smaller than the server replicator's address. This is the fact if the receiver of the message is located at a higher level in the address hierarchy than the replicator. Otherwise, the message is forwarded if the receiver's address is longer but is not prefixed by the server replicator's address. This happens if the receiver is located at a lower level in another subtree of the address hierarchy[5].

[5]The addressing scheme that is introduced by the modularized actor model simplifies the decision whether a message is addressed to the outside or to the inside. One can judt check the length and the prefix of

### The module *SERVER-REPLICATOR-WRAPPER*

The system module *SERVER-REPLICATOR-WRAPPER* instantiates the theory *SERVER-REPLICATOR* interface. It specifies that the server replicator spawns new ASV servers. Basically, the operators and equations

```
op sr-fwd-delay : -> Float .
eq sr-fwd-delay = 0.0 .

op sr-check-period : -> Float .
eq sr-check-period = 0.01 .
```

define the constant operators that are specified in the theory. The operator

```
op maxLoadPerServer : Float -> Float .
```

is defined by the equation

```
eq maxLoadPerServer(t) =
  (floor(t / asv-server-timeoutperiod) + 1.0)
  * maxLoadPerServer * server-overload-factor .
```

and specifies the maximum load for a single server. In every timeout period, `maxLoadPerServer * server-overload-factor` packets can be handled by one server. Finally, the two operators

```
op replicate : Address -> Actor .
op init : Address Float -> Config .
```

specify the effect of a replicator replicating a server. The variables

```
var A : Address .
var t : Float .
```

are used in the following equations. The equation

```
eq replicate(A) =
  < A : ASV-Server |
    config:
      < A . 1 >
      < (A . 0) : Server | mt >,
    msg-count: 0.0, msg-buffer: mtMsgList, internal-addr: (A . 0) > .
```

creates an ASV server which contains one simple server. The equation

```
eq init(A, t) =
  [t + asv-server-timeoutperiod, A <- asv-server-timeout] .
```

initializes the replicated ASV server by emitting the ASV server timeout message.

Finally, the module *SERVER-REPLICATOR-WRAPPER* is connected to the theory *SERVER-REPLICATOR-INTERFACE* by the view

```
view Server-Replicator-Wrapper from SERVER-REPLICATOR-INTERFACE to SERVER-
    REPLICATOR-WRAPPER is
  op sr-fwd-delay to sr-fwd-delay .
  op sr-check-period to sr-check-period .
  op replicate to replicate .
  op init to init .
  op maxLoadPerServer to maxLoadPerServer .
endv
```

the address.

### The module *ASV-SERVER-REPLICATOR-INIT*

The system module *ASV-SERVER-REPLICATOR-INIT* connects the aforementioned modules, defines the initial configuration, and the connection to PVeStA. The initial state is defined by the equation

```
ceq initState =
  < SRA : ServerReplicator | config: < SRA . 0 >,
    server-list: mtAddressList, msg-count: 0 >
  < GA : Generator | count: 0, server: SRA, config: < GA . 0 > >
  < AA : Attacker | acount: 0, sua: SRA, success-cnt: 0 >
  {0.0 | nil}
  [0.0, SRA <- check]
  [0.05 + generator-spawn-period, GA <- spawn]
  [0.05, AA <- attack!]
if
  NG := < 0 > /\
  SRA := NG .new /\
  NG' := NG .next /\
  GA := NG' .new /\
  NG'' := NG' .next /\
  AA := NG'' .new .
```

which uses the variables

```
vars SRA GA AA : Address .
var NG NG' NG'' : NameGenerator .
var C : Config .
```

The initial state consists of the server replicator (which replicates the ASV server), the generator (which generates the ASV clients), the attacker, and the top level scheduler. Additionally, the replicator, generator and attacker are initialized. Finally, the equations

```
eq sat(0, C) = true .
eq val(0, C) = successRatio(C) .
eq val(1, C) = avgTTS(C) .
```

connect the operators `succerssRatio` and `avgTTS` to PVeStA.

## 5.4.3. Statistical Model Checking Results

As in Section 5.3.2, we use the specification of the ASV$^{++}$ protocol together with the client-server setting to perform statistical model checking of QuaTEx formulas analysing the behavior of ASV$^{++}$ under DoS attack using PVeStA.

We want to analyze the QuaTEx formulas defined in Section 5.3.2, but redefine the formula for the number of client requests due to the introduction of the Server Replicator meta-object and define a new formula to count the expected number of servers that are spawned by the Server Replicator. The function *time*() denotes a state function that returns the global time value of the current configuration.

**Number of client requests.** The number of client requests represents the number of *REQ*s sent by legitimate clients (not including *REQ*s sent by attackers).

$$\begin{aligned} \mathit{requestsReplication}(t) = \; & \textbf{if } \mathit{time}() > t \textbf{ then } \mathit{countRequestsReplication}() \\ & \textbf{else } \bigcirc\,(\mathit{requestsReplication}(t)) \end{aligned}$$

with *countRequestsReplication*() being equal to the client request counter attribute of the Server Replicator meta-object.

**Number of servers.** The number of servers represents the number of ASV server objects that are spawned by the Server Replicator meta-object.

$$servers(t) = \textbf{if } time() > t \textbf{ then } countServers() \\ \textbf{else } \bigcirc (servers(t))$$

with *countServers*() being equal to the size of the server list attribute of the Server Replicator meta-object.

As in Section 5.3.2, for the statistical model checking of the aforementioned properties, we again fix the mean server processing rate $S$ to 600 packets per second, the timeout window $T$ to 0.4 seconds, the retrial span $J$ to 7, and the client arrival rate $\rho$ to 0.08. Additionally, an initial generation delay of 0.05 seconds is introduced and the duration of a simulation is set to 30 seconds. The properties are checked for various attack conditions represented by the constant $\alpha$ values 0.6666, 3.3333, 6.6666, 13.3333, 26.6666, 40.0, 53.3333, 66.6666, 80.0, 93.3333, 106.6666, 120.0, and 133.3333, which correspond to 1, 5, 10, 20, 40, 60, 80, 100, 120, 140, 160, 180, and 200 attackers (each attacker issues 400 fake *REQ*s per second). The properties are further checked for a varying overloading factor $k$ (4, 8, 16, and 32) of the Server Replicator meta-object. The Server Replicator's check period is fixed to 0.01 seconds. A forward delay and a replication delay are not considered in our experiments. In the following, we will consider two general cases, in which the Server Replicator can provision

a) an unlimited number of servers.

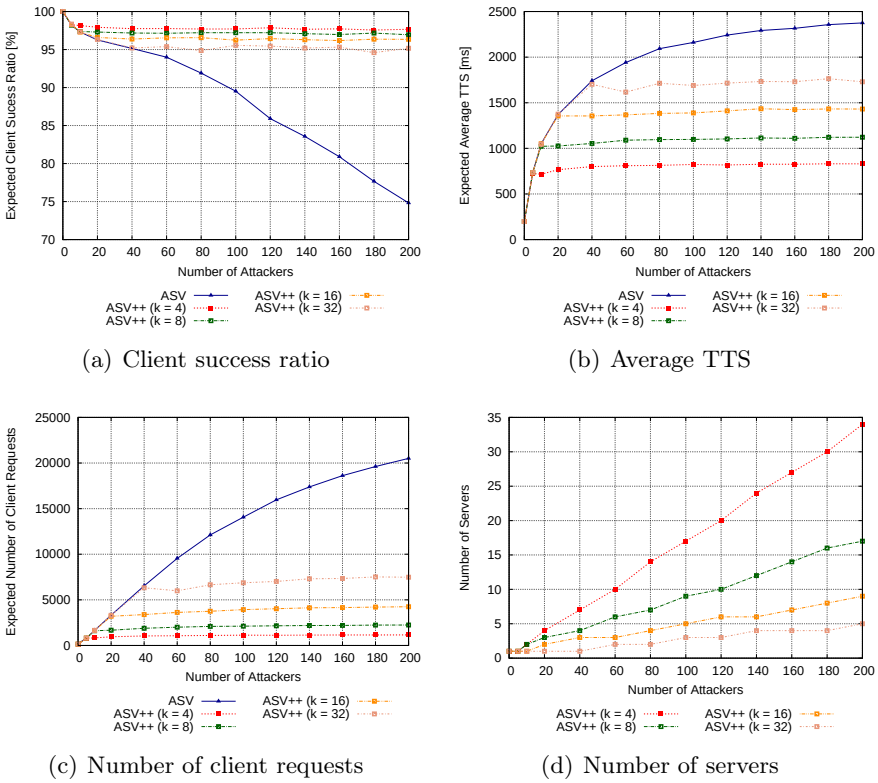
b) servers up to a limit $m$ of 5 and 10 servers, because, out of economical considerations and physical restrictions, it is not possible to assume an unlimited amount of resources.

The results in (a) will indicate how many servers are needed to provide certain service guarantees while the results in (b) will indicate what service guarantees can be given with limited resources. The properties for the case (b) are only checked for an overloading factor of $k = 4$, because we expect the results to be similar for other overloading factors.

The parameters for each configuration that the properties are checked for are set in the module *ASV-PARAMS*. The initial configuration that is used for the statistical model checking using PVeStA is defined in the module *ASV-SERVER-REPLICATOR-INIT*.

### Unlimited Resources

Figure 5.8 shows the model checking results for a varying load factor $k$ and no resource limits. As indicated by Figure 5.8(a), the ASV$^{++}$ protocol can sustain the expected client success ratio at a certain percentage. Even for an overloading factor of $k = 32$, a success ratio around 95% can be achieved. Figures 5.8(b) and 5.8(c) show that the same is true for the average TTS and the number of client requests that are sent to the server. Both values can be sustained at close to constant levels. ASV$^{++}$ outperforms the ASV protocol in all of the performance indicators. However, this is achieved at the cost of provisioning




(a) Client success ratio

(b) Average TTS

(c) Number of client requests

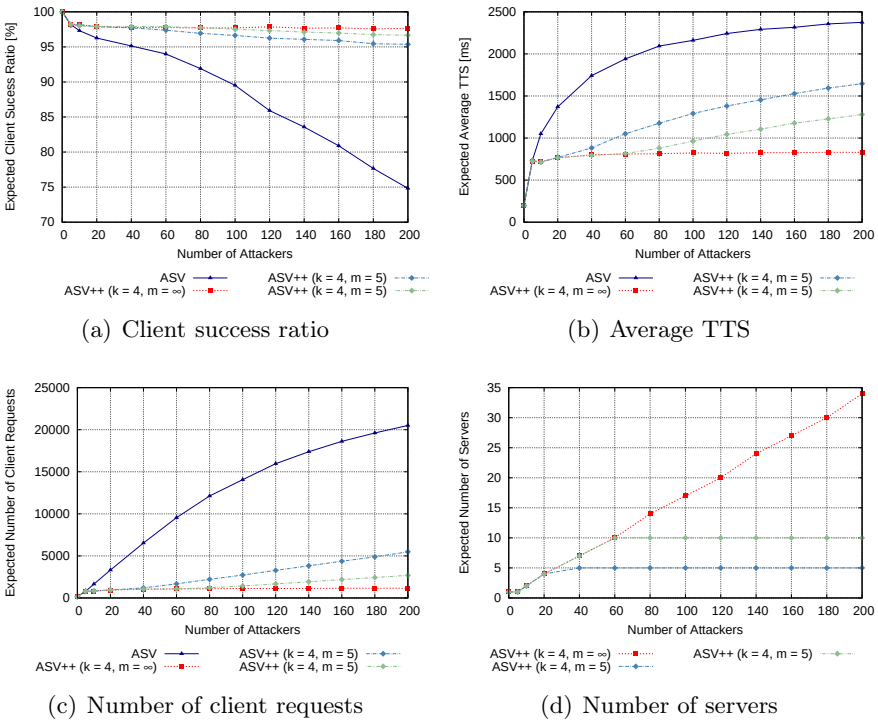
(d) Number of servers

**Figure 5.8.:** Performance of the ASV$^{+}$SR protocol with a varying load factor $k$ and no resource bounds

new servers. Figure 5.8(d) shows how many servers need to be provisioned to keep the performance indicators at their respective close to constant levels for the varying levels of attack. Not surprisingly, ASV$^{++}$ with an overloading factor of $k = 32$ requires significantly fewer resources than with an overloading factor of $k = 4$.

### Limited Resources

Figure 5.9 shows the model checking results for a load factor $k = 4$ and a limit $m$ of either 5 or 10 servers that the Server Replicator meta-object can provision. As indicated by Figure 5.9(a), the success ratio can still be kept at a high level under the assumption of limited resources. In fact, the protocol behaves just as in the case without limited resources up to the point where more servers than the limit would be needed to keep the success ratio close to the constant level. After that point, the protocol behaves like the original ASV protocol (but with the equivalent of a more powerful server) and the success ratio decreases. Nevertheless, it decreases more slowly since 5 and respectively 10 servers handle the incoming *REQ*s compared to the one server in the case of the original ASV protocol. Figures 5.9(b) and 5.9(c) show that the average TTS and the number of client requests behave in a way similar to that of the success ratio.




(a) Client success ratio

(b) Average TTS

(c) Number of client requests

(d) Number of servers

**Figure 5.9.:** Performance of the ASV$^{+}$SR protocol with a load factor of $k = 4$ and limited resources

## 5.5. Conclusion

In this chapter, we have shown that we can formally describe reflective Cloud-based architectures that are protected against DDoS attacks using meta-objects formally specified in rewriting logic. The meta-objects specified in this chapter, namely, the ASV Wrapper and the Server Replicator, are based on the modularized actor model which is introduced in Chapter **??**, and define a family of formal meta-object patterns.

We have further shown that the aforementioned specifications of architectures can be statistically model checked to analyze important qualitative properties. In this chapter, we have formally analyzed quantitative properties of systems under a DDoS attack, demonstrating that the ASV and ASV$^{++}$ protocols can guarantee high availability and good performance in hostile environments by adapting to the level of the DoS attack and by using the Cloud's capacity to provision new resources on demand.

# Chapter 6

# Specification and Analysis of a Group-Key Management System using the ZooKeeper Service

In this chapter, the goal is to develop a Maude specification of a group-key management system, that is based on the modularized actor model and uses the notification mechanism of the so-called ZooKeeper service, a centralized server that provides high reliability through replication. This specification is then used for statistical analysis of quantitative properties of the system. In the following, we:

1. give an introduction to the ZooKeeper service (Section 6.1),
2. describe the specific group-key management system built on top of the ZooKeeper service (Section 6.2),
3. model the group-key management approach which uses the ZooKeeper service in Maude using the modularized actor model (Section 6.3),
4. and, finally, statistically analyze the Maude specification using the PVeStA tool (Section 6.4).

## 6.1. The ZooKeeper Service

ZooKeeper™[1][77, 78] is an open-source centralized server that provides highly reliable distributed coordination. ZooKeeper aims to derive the essential parts of distributed services

[1]The name ZooKeeper is chosen "Because coordinating distributed systems is a Zoo"

like managing configuration information, naming, providing distributed synchronization, and providing group services into an interface to a centralized coordination service. The service itself is distributed and provides high reliability through replication. It is implemented in Java and provides a Java and C interface for its clients.

The ZooKeeper service allows distributed processes to communicate with each other through a shared hierarchical name space of data registers, the so-called *znodes*. Clients can interact with the ZooKeeper service through a simple programming interface which provides the following operations:

- **create**: creates a znode in the hierarchy.
- **delete**: deletes a znode from the hierarchy.
- **exists**: tests if a znode exists at a location.
- **get data**: reads the data from a znode.
- **set data**: writes data to a znode.
- **get children**: retrieves a list of children of a znode.
- **sync**: waits for data to be propagated.

The ZooKeeper service is distributed and replicated over a set of machines, called the *hosts*. The *hosts* must all know about each other and maintain an in-memory image of the state of the system. Each client connects to a single *host* and maintains a TCP connection for communication purposes. Since the internal state of the ZooKeeper service is replicated across the *hosts*, read requests are directly read from the memory of the *host*[2]. For processing a write request on a *znode*, the write request needs to be propagated to all *hosts*[3]. Thus, there is a unique *host*, called the *leader*, to which all write requests are forwarded. The *leader* forwards message proposals to the rest of the ZooKeeper *hosts*, called the *followers*, which then agree upon message delivery. The message layer takes care of replacing the *leader* on failures and of synchronizing the *followers* with the *leader*. As part of the leader election and agreement protocol, the ZooKeeper service remains available as long as a (strict) majority of the *hosts* are available. Because of the majority consensus protocol, an odd number of *hosts* is required in the system. Figure 6.1 gives a high-level overview of a ZooKeeper service with five *hosts* $H_1, \ldots, H_5$, where *host* $H_1$ is the leader. Furthermore, there are six client $C_1 \ldots C_6$ which are connected to the service.

The ZooKeeper service supports the concept of a *watch* — a client can set a watch on a set of *znodes*, and when one such *znode* changes, the watch is triggered and removed. Watches are maintained at the *host* the client is connected to and are not replicated. Using the concept of watches, ZooKeeper can be used as an event notification mechanism.

ZooKeeper aims to be the basis for the construction of complex services, thus it provides a set of guarantees:

- **Sequential Consistency** - Updates from a client will be applied in the order that they were sent.

[2]Since the image of the system is replicated over all *hosts*, there is no need for an agreement protocol on a read operation. Thus, a read operation is considered to be cheap.

[3]Since write operations involve a agreement protocol, they are considered to be expensive.

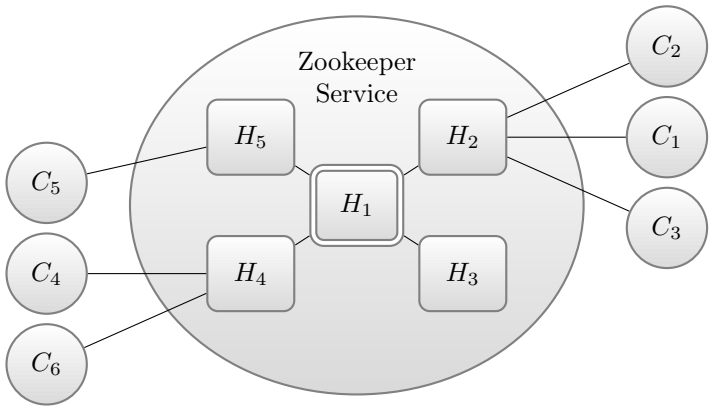



**Figure 6.1.:** High-level overview of the ZooKeeper service

- **Atomicity**: Updates either succeed or fail.
- **Single System Image**: A client will see the same view of the service no matter which server it is connected to.
- **Reliability**: Once an update has been applied, its changes will be reliably persisted in the system.
- **Timeliness**: The clients' view of the system is guaranteed to be up-to-date within a certain time bound.

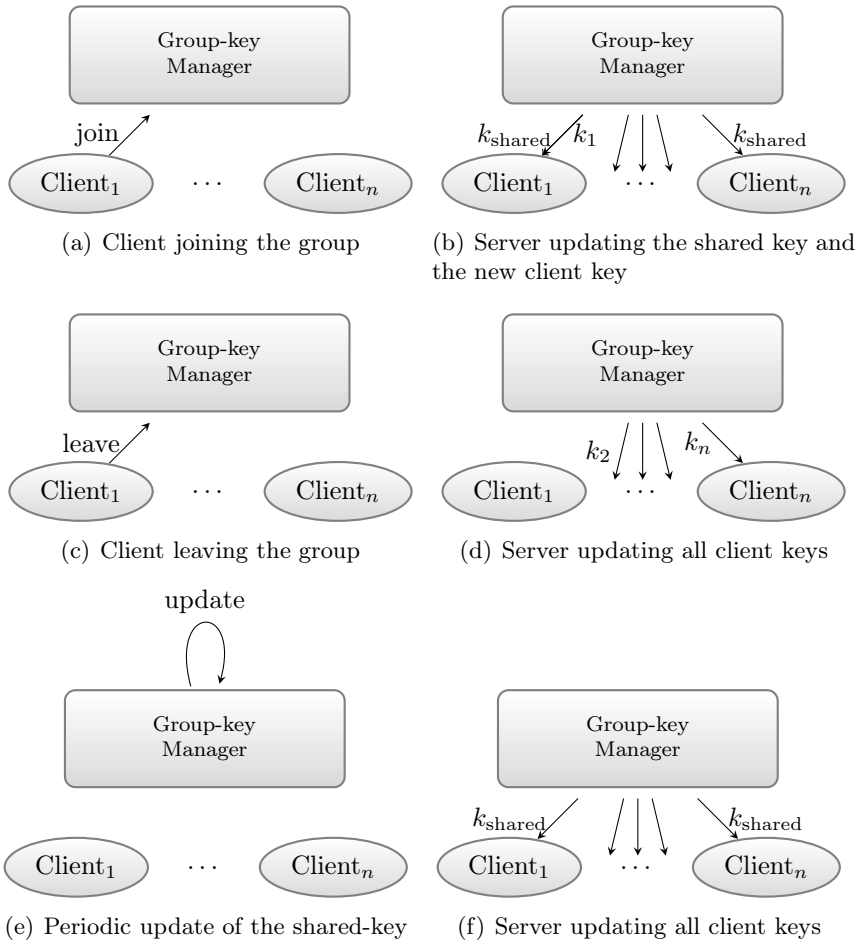
The group-key management approach that is presented in the following is based on the ZooKeeper event notification mechanism and the high reliability of the service.

## 6.2. Group-Key Management using ZooKeeper

The idea we present here is based on the work of Bobba et al. In [42, 29], Bobba et al. have implemented a centralized group-key management system where key updates are distributed based on the notification mechanism of ZooKeeper. Similarly to that, the group-key management we model in this chapter uses the notification mechanism of the ZooKeeper service for key distribution.

### 6.2.1. Abstraction of a Group-Key Management System

In the following, we use an abstraction of the **G**roup **D**omain **o**f **I**nterpretation (GDOI) protocol for group-key management. The GDOI [25] is a cryptographic protocol for group-key management which allows a group controller and key server to distribute keys to members of a group. The protocol is designed to be compatible with the use of mechanisms for efficiently distributing keys to group members, such as key hierarchies [32]. Nevertheless, no such mechanism is explicitly specified here. Some of these mechanisms use a method where many members use the same structure of keys — each member holds a different subset of keys. Similarly, we differentiate between a shared part of the key, the *shared key*, and the part of a key which is individually known to each member or client, the *client key*.




(a) Client joining the group

(b) Server updating the shared key and the new client key

(c) Client leaving the group

(d) Server updating all client keys

(e) Periodic update of the shared-key

(f) Server updating all client keys

**Figure 6.2.:** Interactions between the group-key manager and its clients.

Figure 6.2 gives an overview of the interaction between the clients of a group-key management service and the group-key manager. Clients can join the group by sending a *join* message to the manager (Figure 6.2(a)). As shown in Figure 6.2(b), after the client has joined the group, a new *shared key* $k_{\text{shared}}$ is generated and distributed to all clients, so that the new joining user is unable to decrypt previous communication data[4]. Additionally, the *client key* $k_1$ for the new member of the group is generated and sent to him. If a client decides to leave the group, it sends a *leave* message to the manager (Figure 6.2(c)). The manager subsequently generates new *client keys* $k_i$ for $i \in 2 \dots n$ and distributes them to the clients but the leaving client (Figure 6.2(d)). Thus, the leaving client is no longer able to decrypt the future communication data[5]. Additionally, the group-key manager periodically updates the *periodic* (shared) key (Figure 6.2(e)). The updated shared key is sent to all group members as shown in Figure6.2(f).

[4]This is called *forward security*.
[5]This is called *backward security*.

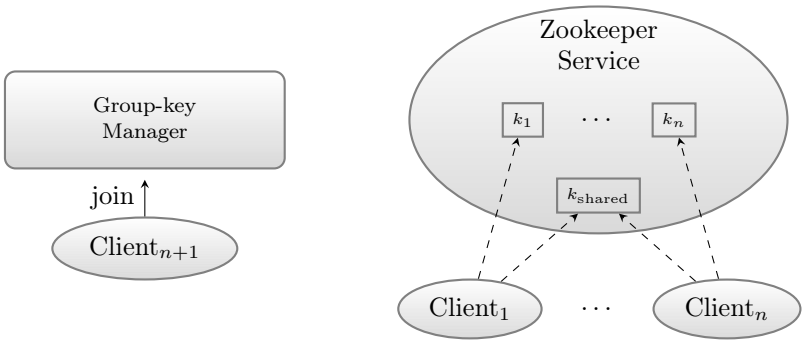



**Figure 6.3.:** Messages and message handling after a new client joins the group

### 6.2.2. Using ZooKeeper's Notification Mechanism for Key Distribution

The group-key management server uses ZooKeeper's notification mechanism for publishing key changes. Each client of the group watches its own *client key* as well as the *group key* in the ZooKeeper service. When the server needs to distribute a key update, it updates the specific value in the ZooKeeper service, which is then distributed to the watching clients[6]. This approach has the advantage that an existing and well-established event notification mechanism is used. Additionally, the group-key management can rely on the guarantees provided by ZooKeeper, such as timeliness and reliability. If the group-key management server fails, the service can be easily re-established, since there is no direct connection between the group-key management server and the clients.

Figures 6.3 to 6.7 give an overview of the messages that are sent if a new client joins the group. Figure 6.3 illustrates the situation when a new client joins the group. At this point, there are $n$ clients in the group and, as mentioned before, they are watching the *shared key* and their *client key* in the ZooKeeper service (depicted as a dashed arrow). After the client has joined the group, the group-key manager updates the *shared key* and adds a new *client key* $k_{n+1}$ by sending an *update* message to the ZooKeeper service (Figure 6.4). Additionally, it sends an accept message to the new client which contains a reference to the ZooKeeper service, the *client key* $k_{n+1}$, and the shared key $k_{\text{shared}}$. Figure 6.5 illustrates the notification mechanism of the ZooKeeper service: The ZooKeeper service sends notification messages to all the clients that are watching the *shared key*. Additionally, the new client Client$_{n+1}$ sends watching requests for the *shared key* and its *client key*. After the clients have received a *notify* message from the ZooKeeper service, they request the value and resend a watching request (Figure 6.6). Finally, Figure 6.7 shows the final situation, where the clients watch the *shared key* and their *client key*.

[6] We assume that the clients are only able to request/watch their *client key* and the *group key*. For example, this can be realized by using a PKI infrastructure and encrypting the *client keys* with the client public key, so that each client's private key is needed to decrypt its *client key*.

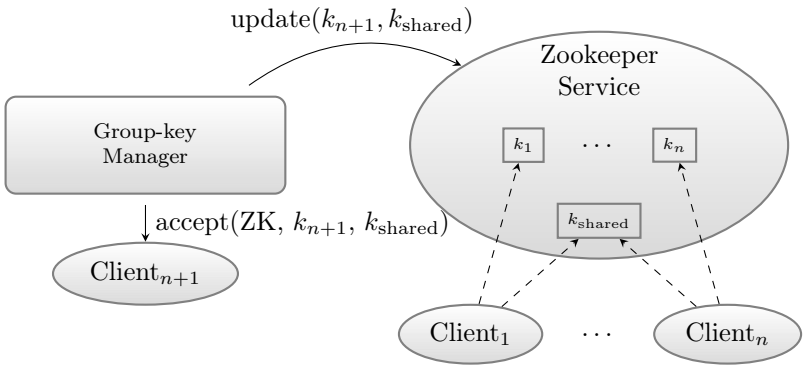



**Figure 6.4.:** Messages and message handling after a new client joins the group (II)

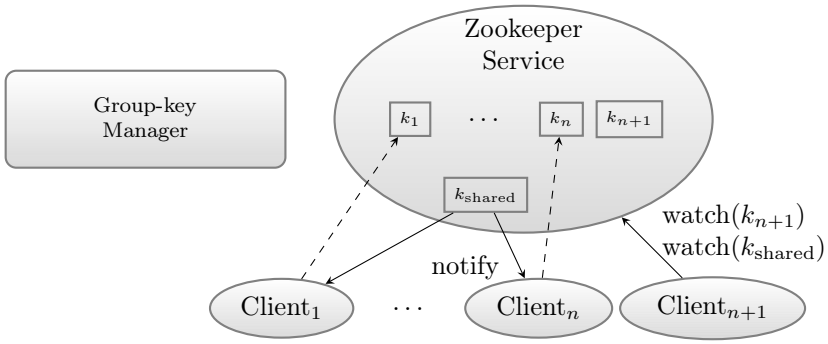



**Figure 6.5.:** Messages and message handling after a new client joins the group (III)

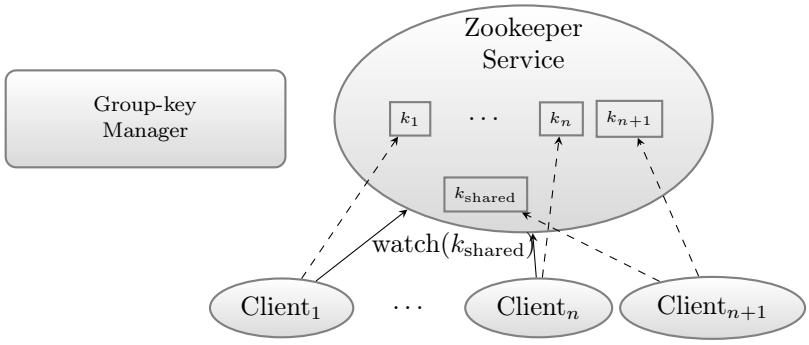



**Figure 6.6.:** Messages and message handling after a new client joins the group (IV)

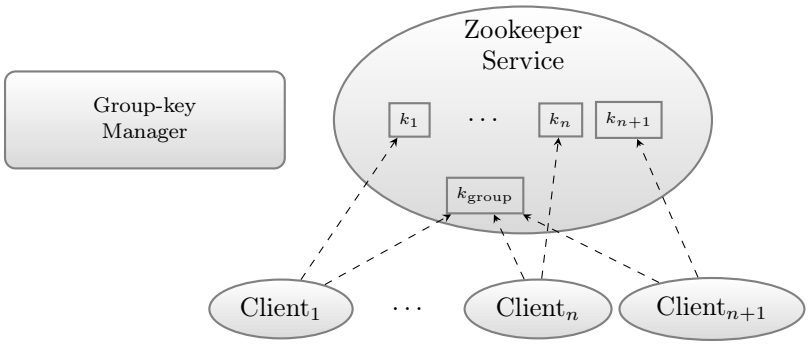



**Figure 6.7.:** Messages and message handling after a new client joins the group (V)

## 6.3. Maude specification of the Group-Key Management on top of ZooKeeper

In this section, the Maude specification of the group-key management that uses the ZooKeeper service is described. Since the ZooKeeper service provides an isolated service by itself, it is specified in a modular way. In the following, the Maude specification of the ZooKeeper service is described. Additionally, the Maude specification of the group-key management that uses the ZooKeeper service is given.

### 6.3.1. Maude specification of the ZooKeeper Service

To give a reasonable abstraction for our purposes of the ZooKeeper service, we modeled it without its leader election, and agreement protocol. As illustrated in Figure 6.8, the ZooKeeper service itself is modeled as a Russian dolls actor which contains a set of ZooKeeper nodes (which represent the *hosts* of the original model) in its configuration. Each ZooKeeper node is modelled as a flat actor which contains in its state a key-value store, representing the in-memory image of the hierarchical *znodes*, and a list of watching nodes. Each node has an associated probability of failure which is used to periodically check whether the node fails. To get closer to a real world scenario, we model message delays through links. Additionally, a processing time for requests and watching requests at the ZooKeeper nodes is introduced, i.e., if a request arrives at a ZooKeeper node, it is buffered and the ZooKeeper node processes the requests one after another within a specified processing time.

In our setting, the ZooKeeper nodes are distributed over Europe as illustrated in Figure 6.9. The red circle indicates the location of the ZooKeeper Service (Munich) and the yellow circles the location of the ZooKeeper nodes (Berlin, Rome, Madrid, Paris, and London). The ZooKeeper Service is connected with each of the ZooKeeper nodes via a direct link (with a fixed latency).

If a client joins the ZooKeeper service, it is connected to the ZooKeeper node to which the message delay is the shortest for the client. Upon a ZooKeeper node failure, the client is automatically reconnected to the closest available ZooKeeper node. Thus, the ZooKeeper service needs to know the link latencies from the clients to the ZooKeeper nodes.

After joining the ZooKeeper service, clients can interact with it through the following set of operations:

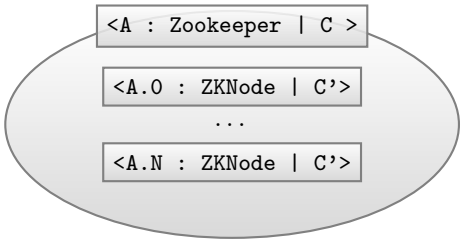



**Figure 6.8.:** Overview of the ZooKeeper specification

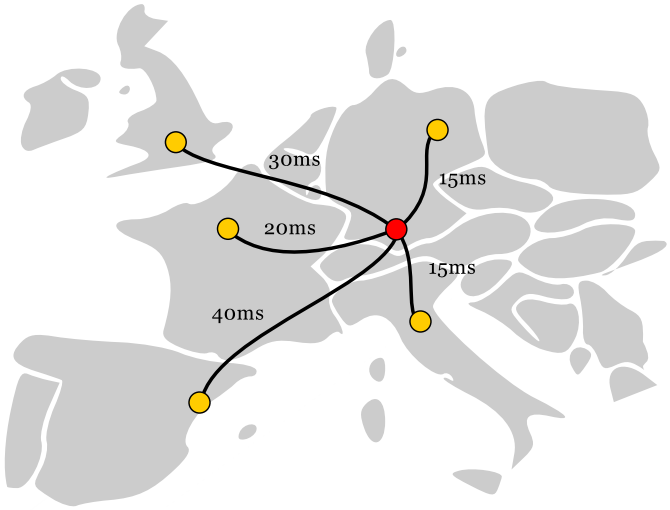



**Figure 6.9.:** Distribution of the ZooKeeper nodes in Europe

**Request:** The client can requests a value from the ZooKeeper service, which is forwarded to and answered by the ZooKeeper node the client is connected to.

**Update:** The client can update or create a new key-value pair in the ZooKeeper service. The ZooKeeper service itself acts as the *leader* and forwards the update to all ZooKeeper nodes in his configuration. We chose to abstract from a single ZooKeeper node to act as a *leader*, since we want to omit the leader election and agreement protocol. As long as the majority of ZooKeeper nodes are active, the update is sent to all ZooKeeper nodes. Otherwise, if the majority is inactive, the update is lost. This reflects the behavior of ZooKeeper at a higher level of abstraction.

**Watch:** The client can set a watch on a key in order to be notified when the corresponding value changes. The watch is removed after the client has been notified. The watches are only set at the closest ZooKeeper node, and are not replicated.

### Overview

Figure 6.10 gives an overview of the Maude modules and the structural dependencies that are used for the specification of the ZooKeeper service. Starting from the bottom of the figure, the modules describe the following:

- The functional module *ACTOR-MODEL* specifies the modularized actor model that is described in Chapter 4. The ZooKeeper service as well as the ZooKeeper nodes are modelled as actors that communicate through asynchronous message passing.
- The functional module *STORE* specifies a key-value store, which is used in each ZooKeeper node to store key-value pairs.
- The functional module *ADDRESS-LIST* defines a sort for lists of addresses, and several auxiliary operators.
- The functional module *ZK-PARAMS* defines parameters of the model, such as, e.g., the probability of a ZooKeeper node to experience a failure.
- The functional module *WATCH-LIST* specifies sorts and operators that are used for handling the watch-list behavior of a ZooKeeper node.
- The functional module *CONTENTS-LIST* describes a sort and auxiliary operator for a list of message contents.
- The functional module *LINK-GRAPH* specifies a graph of links using an adjacency list representation. Graphs of links are used to calculate the message delay that is introduced from one node to another.
- The functional module *ADR-MAPPING* describes a simple mapping between two addresses.
- The functional module *ZK-NODE* specifies the sorts and operators of ZooKeeper nodes. For instance, messages that are sent and received from ZooKeeper nodes, the states of the ZooKeeper nodes, etc., are specified in this module.

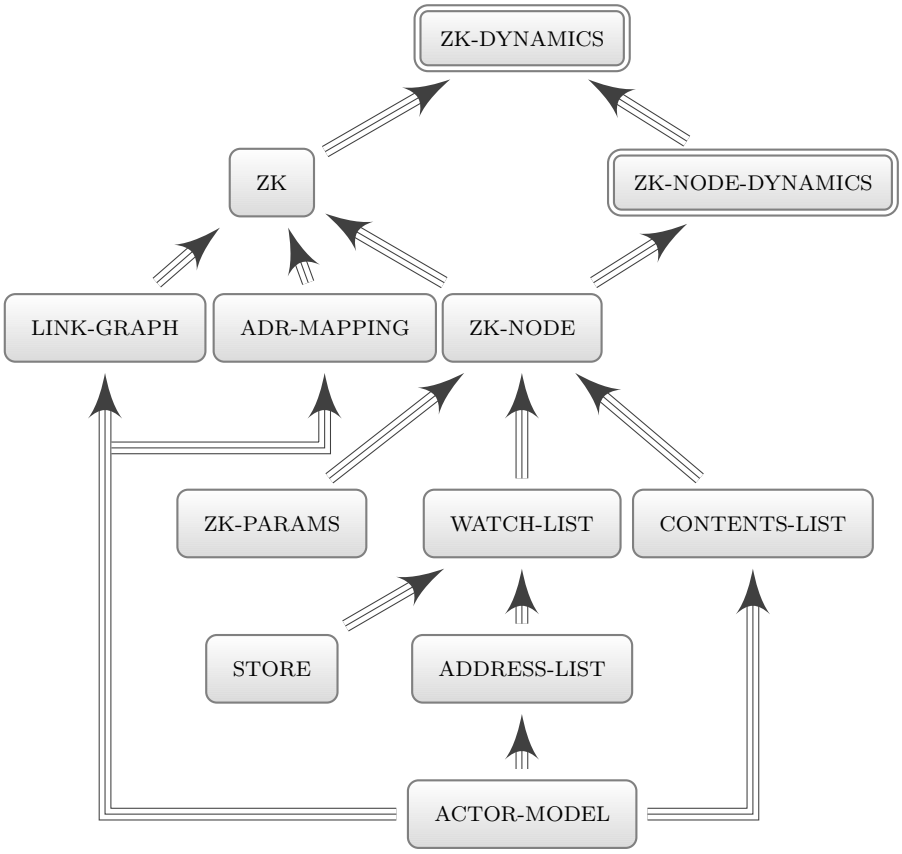



**Figure 6.10.:** Overview of Maude modules of the ZooKeeper specification

- The functional module *ZK* specifies the static parts of the ZooKeeper service. For example, an operator that initializes the ZooKeeper service with its internal nodes and links is specified here.

- The system module *ZK-NODE-DYNAMICS* described the dynamic aspects of a ZooKeeper node. This module describes how a ZooKeeper node reacts to messages that sent to it.

- Finally, the system module *ZK-DYNAMICS* describes the dynamics of the ZooKeeper system. The interplay between the internal nodes, the rewrite rules that need to cross boundaries, etc., are specified here.

At a high level, the functional modules *STORE, ADDRESS-LIST, ZK-PARAMS, WATCH-LIST, CONTENTS-LIST, LINK-GRAPH* and *ADR-MAPPING* define auxiliary sorts and operators that are used in the specification of the ZooKeeper nodes and the ZooKeeper service. The functional modules *ZK-NODE* and *ZK* define the static aspects of ZooKeeper nodes and the ZooKeeper service. Lastly, the system modules *ZK-NODE-DYNAMICS* and *ZK-DYNAMICS* define their dynamic behavior.

### Description of modules

The following modules show how the ZooKeeper service is specified using the modularized actor model. The functional module *ACTOR-MODEL* is described in Chapter 4.

**The *STORE* module.** The functional module *STORE* defines a store of id-value pairs, in which values can be stored, updated, and retrieved for a given id. The sorts

```
sort Id .
sort Value .
```

declare the sorts that represent ids (`Id`) and values (`Value`) (in the sense of key-value pairs). The name "Id" was chosen instead of "Key", since we want to differentiate between cryptographic keys, which are used in the group-key management and identifiers, which are used to retrieve a value in the key-value store. The constant operator

```
op undef : -> Value .
```

specifies a term of sort `Value` representing an undefined value. It is used to differentiate between values that are defined and those that are not. The sorts respectively

```
sort Store .
sort Entry .
```

specify the id-value store and a single id-value pair stored it. Terms of sort `Entry` can be constructed using the operator

```
op (_,_) : Id Value -> Entry .
```

which takes an id and a value as arguments. The auxiliary operators

```
op _.id : Entry -> Id .
op _.value : Entry -> Value .
```

return the id, respectively the value, for a given term of sort `Entry`. The subsort relationship

```
subsort Entry < Store .
```

states that a term of sort `Entry`, i.e., a single id-value pair, is a subsort of `Store`, and thus can be concatenated using the associative operator

```
op _:_ : Store Store -> Store [assoc id: mtStore] .
```

with the constant operator

```
op mtStore : -> Store .
```

acting as the identity. Additionally, the operator

```
op _[_] : Store Id -> Value .
```

returns for a given store and a given id the corresponding value, or the constant `undef` in case the given id is not defined in the store. The operator

```
op _[_->_] : Store Id Value -> Store .
```

takes a store, an id, and a value as arguments, and returns a store in which the value for the specified id has been updated.

The variables

```
vars I I1 I2 : Id .
vars V V1 V2: Value .
var S : Store .
var E : Entry .
```

are used in the following to define the behavior of these operators. The two equations

```
eq (I,V) .id = I .
eq (I,V) .value = V .
```

define the behavior of the two auxiliary operators on id-value pairs. The equations

```
eq mtStore[I] = undef .
eq (E : S) [I] =
  if E .id == I
  then
    E .value
  else
    S[I]
  fi .
```

define the `_[_]` operator. The first equation returns for an empty store the constant operator `undef`, while the second equation recursively decomposes the store if the id of the entry is not equal to the id given as argument. Otherwise, the value of the entry is returned. Finally, the equations

```
eq mtStore[I -> V] = (I, V) .
eq (E : S)[I -> V] =
  if E .id == I then
    (I, V) : S
  else
    E : (S[I -> V])
  fi .
```

update the value of an id in the store. If an empty store is updated, a store containing the new entry is returned. Otherwise, if the store has as least one entry, the value is replaced if the id of the entry is equal to the id that is provided as an argument. If the ids are different, the operator is recursively applied.

**The *ADDRESS-LIST* module.** The *ADDRESS-LIST* module defines an auxiliary list of addresses and additional operators. The sort

```
sort AddressList .
```

specifies a list that is built using the associative operator

```
op _;_ : AddressList AddressList -> AddressList
  [assoc id: mtAddressList] .
```

for which the constant operator

```
op mtAddressList : -> AddressList .
```

acts as the identity. Since the sort `AddressList` represent a list of addresses, the sort `Address` is defined as a subsort by the subsort relationship

```
subsort Address < AddressList .
```

Additionally, the auxiliary operators

```
op _[_] : [AddressList] [Nat] -> [Address] .
op _.size : AddressList -> Nat .
op _in_ : Address AddressList -> Bool .
```

specify auxiliary functions that work on a list of addresses. To access the i-th address of a list, the partial operator `_[_]` takes a list of addresses and a natural number, and returns the corresponding address at that position. This operator is only defined if the specified natural number is smaller than the size of the list. The operator `_.size` returns for a given list the length of the list. Additionally, the predicate `_in_` returns `true`, if the given address is contained in the list of addresses. The operators

```
op remove : AddressList Nat -> AddressList .
op remove : AddressList Address -> AddressList .
```

remove a specific address from the list. The first operator takes a list of addresses and a natural number as arguments, and returns a list of addresses in which the address at the specified position is removed. Similarly, the second operator takes a list of addresses and an address as arguments, and returns a list of addresses in which the specified address is removed. Finally, the operator

```
op createMsgs : AddressList Float Contents -> Config .
```

creates a list of messages that send the specified contents at the specified time to the specified list of addresses. In the following, the variables

```
var A A' : Address .
var L : AddressList .
var N : Nat .
var t : Float .
var C : Contents .
```

are used to define the behavior of these operators. The two equations

```
eq (A ; L) [0] = A .
eq (A ; L) [s(N)] = L [N] .
```

specify the partial indexing operator. The equation returns the first element of the list in case the list is indexed by `0`. Otherwise, if a successor of a natural number is given as index, the operator is recursively executed by the second equation. The behavior of the size operator is defined by the equations

```
eq mtAddressList .size = 0 .
eq (A ; L) .size = 1 + L .size .
```

which return `0` if applied to an empty address list and otherwise recursively decompose the list. Similarly, the equations

```
eq A in mtAddressList = false .
eq A' in (A ; L) = if (A' == A) then true else (A' in L) fi .
```

define the `in` predicate. If the predicate is applied to an empty list, `false` is returned. If the first address is equal to the specified one, `true` is returned. Otherwise, the list is recursively decomposed. The equations

```
eq remove(A ; L, 0) = L .
ceq remove(A ; L, s(N)) = A ; remove(L, N) if L =/= mtAddressList .

eq remove(mtAddressList, A) = mtAddressList .
eq remove(A' ; L, A) = if (A' == A) then L else A' ; remove(L, A) fi .
```

remove either an address at a specified location or a specific address from the list. The list is recursively decomposed and if the index is zero or the first address in the list is equal to the specified one, the element is removed. Finally, the equations

```
eq createMsgs(mtAddressList, t, C) = null .
eq createMsgs(A ; L, t, C) = [t, A <- C] createMsgs(L, t, C) .
```

return a list of messages: one message addressed to each address in the list of addresses. If the list of addresses is empty, the constant operator `null` is returned (which represents the identity of the concatenation operator `__` of sort `Config`). Otherwise, messages are created recursively — one message addressed to each address in the list. The time and the contents of the messages are specified as arguments of the operator.

**The *ZK-PARAMS* module.** The module *ZK-PARAMS* defines parameters for the ZooKeeper service. The operator

```
op request-processing-time : -> Float .
```

which is defined by the equation

```
eq request-processing-time = 0.01 .
```

specifies the time it takes for a ZooKeeper node to process a request. We set this parameter to be 10 ms in our analyses. The operator

```
op watch-processing-time : -> Float .
```

specifies the time it takes for a ZooKeeper node to process a watching request. In our analyses, we have fixed this value to 10 ms, which is specified by the equation

```
eq watch-processing-time = 0.01 .
```

The operator

```
op notify-creation-time : -> Float .
```

specifies the time it takes to create a message for one address in the list of watching clients in the ZooKeeper nodes. The equation

```
eq notify-creation-time = 0.01 .
```

fixes this value to 10 ms. The failure behavior of the ZooKeeper nodes is parametrized with the following two operators

```
op node-failure-prob : -> Float .
op node-failure-period : -> Float .
```

The operator `node-failure-prob` is defined by the equation

```
eq node-failure-prob = 0.01 .
```

and specifies the probability (in our analyses 1%) that a node fails within the time specified by the operator `node-failure-period`. Likewise, the equation

```
eq node-failure-period = 1.0 .
```

specifies the period for a node to fail to 1 second. Thus, every second, every ZooKeeper node can fail with probability of 1%. Finally, the operator and equation

```
op node-recover-time : -> Float .
eq node-recover-time = 2.0 .
```

define the time it takes a ZooKeeper node to recover to be 2 s. After a ZooKeeper node has failed, it takes two seconds for it to recover.

**The *WATCH-LIST* module.** The module *WATCH-LIST* specifies a mapping between an id and a list of addresses that is stored at every ZooKeeper node. The sorts

```
sort WatchList .
sort WLEntry .
```

specify the sort for the watch list (`WatchList`), and the sort for entries of the watch list (`WLEntry`). Additionally, the constant operator

```
op none : -> WLEntry .
```

specifies an undefined element of the watch-list. A term of sort `WLEntry` is constructed by the operator

```
op (_,_) : Id AddressList -> WLEntry .
```

which takes the id of the entry and the list of addresses that are watching for that id as arguments. The auxiliary operators

```
op _.id : WLEntry -> Id .
op _.list : WLEntry -> AddressList .
```

provide a conventient way to access the id or the list of addresses from a given term of sort `WLEntry`. The subsort relationship

```
subsort WLEntry < WatchList .
```

declares each term of sort `WLEntry` to also be of sort `Watchlist`. Thus, the list is constructed using the associative operator

```
op _;_ : WatchList WatchList -> WatchList [assoc id: mtWatchList] .
```

with the constant operator

```
op mtWatchList : -> WatchList .
```

acting as the identity. The operators

```
op addWatch : WatchList Id Address -> WatchList .
op removeWatching : WatchList Id -> WatchList .
```

respectively add a watching address for the specified id in the watch list and remove the watching addresses for the specified id from the watch list. The lookup operator

```
op _[_] : WatchList Id -> WLEntry .
```

takes a watch list and an id as arguments and returns the corresponding term of sort `WLEntry` from the watch list. If there is no entry for the specified id in the list, the constant operator `none` is returned. Finally, the operators

```
op getNotifications : WatchList Id Contents Float -> Config .
op getNotifications : AddressList Id Contents Float -> Config .
```

take a watch list (resp. a list of addresses) together with an id, the contents, and a time as arguments and return a list of messages.

In the following equations, the Maude on-the-fly variable declarations are used. The equations

```
eq (I:Id, L:AddressList).id = I:Id .
eq (I:Id, L:AddressList).list = L:AddressList .
```

respectively return the id and the list of addresses for a given term of sort `WLEntry`. The operator `addWatch` is defined through the following equations

```
eq addWatch(mtWatchList, I:Id, A:Address) = (I:Id, A:Address) .
eq addWatch(E:WLEntry ; L:WatchList, I:Id, A:Address) =
  if (E:WLEntry) .id == I:Id then
     (I:Id, ((E:WLEntry) .list ; (A:Address))) ; L:WatchList
  else
    E:WLEntry ; addWatch(L:WatchList, I:Id, A:Address)
  fi .
```

The second equation recursively decomposes the watch list and inserts the specified address into an entry, if the id of the entry is equal to the specified id. Otherwise, if there is no entry in the list for the id, a new entry is created by the first equation. Similarly, the two equations

```
eq removeWatching(mtWatchList, I:Id) = mtWatchList .
eq removeWatching(E:WLEntry ; L:WatchList, I:Id) =
  if (E:WLEntry) .id == I:Id then
    L:WatchList
  else
    (E:WLEntry); removeWatching(L:WatchList, I:Id)
  fi .
```

define the `removeWatching` operator. If the operator is applied to an empty watch list, the empty watch list is returned. Otherwise, the second equation recursively steps through the watch list, and if it finds an entry whose id is equal to the specified id, the entry is removed. The lookup operator is specified by the equations

```
eq (E:WLEntry ; L:WatchList)[I:Id] =
  if (E:WLEntry) .id == I:Id then
    E:WLEntry
  else
    (L:WatchList)[I:Id]
  fi .
eq (mtWatchList)[I:Id] = none .
```

which recursively walk through the list and, if the id of an entry equals the specified id, return it. The recursion stops if the empty watch list is reached. Finally, the equations

```
eq getNotifications(mtWatchList, I:Id, C:Contents, gt:Float) = null .
eq getNotifications(E:WLEntry ; L:WatchList, I:Id, C:Contents, gt:Float) =
  if (E:WLEntry) .id == I:Id then
     getNotifications((E:WLEntry).list, I:Id, C:Contents, gt:Float)
  else
    getNotifications(L:WatchList, I:Id, C:Contents, gt:Float)
  fi .

eq getNotifications(mtAddressList, I:Id, C:Contents, gt:Float) = null .
eq getNotifications((A:Address) ; L:AddressList, I:Id, C:Contents, gt:Float) =
  [gt:Float, A:Address <- C:Contents]
  getNotifications(L:AddressList, I:Id, C:Contents, gt:Float) .
```

define the behavior of the `getNotifications` operators. If a watch list is given as first argument, the watch list is recursively decomposed until the empty list is reached or an entry with the specified id is found. The list of addresses of this entry is then passed to the `getNotifications`

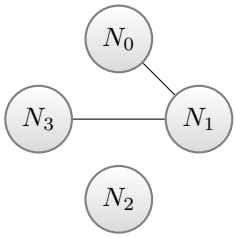



**Figure 6.11.:** Simple undirected graph

operator which takes an term of sort `AddressList` as first argument. A message is then created for every address in the list of messages.

**The *CONTENTS-LIST* module.** The module *CONTENTS-LIST* is a simple module that defines a sort representing lists of contents.

```
sort ContentsList .
```

Similar to the lists we defined before, the subsort relationship

```
subsort Contents < ContentsList .
```

states that terms of sort `Contents` are also of sort `ContentsList`. The constant operator

```
op mtContentsList : -> ContentsList .
```

represents an empty list of contents and is used as the identity for the associative operator

```
op __ : ContentsList ContentsList -> ContentsList
  [assoc id: mtContentsList] .
```

**The *LINK-GRAPH* module.** The Maude module *LINK-GRAPH* specifies a graph of links represented as an adjacency list. I.e., a list of links is stored for each node in the graph. For example, the simple undirected graph in Figure 6.11 can be represented by the following adjacency lists:

$$N_0 \rightarrow N_1$$
$$N_1 \rightarrow N_0, N_3$$
$$N_3 \rightarrow N_1$$

The sort

```
sort Link .
```

represents a link, that connects two actors with different addresses. Links are constructed using the operator

```
op (_->_:_) : Address Address Float -> Link [ctor] .
```

which takes two addresses and the delay of the link as parameters. Additionally, the constant operator

```
op noLink : -> Link .
```

specifies that there is no link between two addresses. The auxiliary operators

```
op _.delay : [Link] -> [Float] .
op _.src : [Link] -> [Address] .
op _.dst : [Link] -> [Address] .
```

take a link as argument, and respectively return the delay, the source address, and the destination address of that link. These three operators are partial, since they are undefined on the constant operator `noLink`. In order to store a list of links for each node, the sort

```
sort LinkList .
```

represents lists of links. The subsort relationship

```
subsort Link < LinkList .
```

states that a single link already is of sort `LinkList`, and thus, can be concatenated using the associative and commutative operator

```
op __ : LinkList LinkList -> LinkList [ctor assoc comm id: mtLinkList] .
```

with the constant operator

```
op mtLinkList : -> LinkList .
```

acting as its identity. Since nodes can only be connected via one link, the list is assumed to be duplicate-free. The absence of duplicates in the list guarantees that no non-determinism is introduced by using the `comm` attribute. In order to guarantee the absence of duplicates, the operator

```
op insert : LinkList Link -> LinkList .
```

takes a list of links and a link to insert as arguments and returns a list of links in which the link is inserted if it is not already contained in the list. Additionally, the lookup operator

```
op _[_] : LinkList Address -> Link .
```

takes a list of links and an address as arguments and returns the corresponding link to that address. If there is no link in the list whose destination is the specified address, the constant operator `noLink` is returned. The sort

```
sort AdjacentList .
```

specifies a sort for adjacency lists, the base building blocks of our graph representation. Adjacency lists can be constructed using the operator

```
op (_->_) : Address LinkList -> AdjacentList .
```

which takes the address of a node and the list of links that node is connected to as arguments. Finally, the sort

```
sort LinkGraph .
```

represents the whole link graph. Since the subsort relationship

```
subsort AdjacentList < LinkGraph .
```

states that terms of the sort `AdjacentList` have also sort `LinkGraph`, and therefore can be concatenated using the commutative and associative operator

```
op _;_ : LinkGraph LinkGraph -> LinkGraph [ctor assoc comm id: mtLinkGraph] .
```

which has the constant operator

```
op mtLinkGraph : -> LinkGraph .
```

as its identity. Since only one term of sort `AdjacentList` is stored for each address, no non-determinism is introduced by using the `comm` attribute. The operators

```
op insert : LinkGraph Link -> LinkGraph .
op insert : LinkGraph LinkList -> LinkGraph .
```

insert a link or a list of links in the specified link graph under the side condition that no two adjacency lists are created for the same address and no duplicate links are created. The lookup operator

```
op _[_] : LinkGraph Address -> LinkList .
```

returns for a given link graph and node address the corresponding list of links, the given node is connected to.

In the following equations, the variables

```
var L : Link .
var LL : LinkList .
var LG : LinkGraph .
vars A A' A'' : Address .
var t t' : Float .
```

are used. The equations

```
eq (A -> A' : t) .delay = t .
eq (A -> A' : t) .src = A .
eq (A -> A' : t) .dst = A' .
```

specify the behavior of the `_.delay`, `_.src`, and `_.dst` operators. They simply return the corresponding arguments of the constructor of sort `Link`. The equations

```
eq insert((A'' -> A' : t') LL, (A -> A' : t)) = (A'' -> A' : t') LL .
eq insert(LL, (A -> A' : t)) = (A -> A' : t) LL [owise] .
```

define the behavior of the `insert` operator. Since the concatenation operator of terms of sort `LinkList` is commutative, the equations do not need to recursively decompose the list, instead, Maude's associative-commutative term matching mechanism can be used. The first equation does not insert a link in the list if it is already a link contained in the list whose destination is equal to the destination of the link that is inserted. The second equation is specified using the `owise` attribute; therefore, if the first equation does not match, the second equation simply prepends the link in front of the set. The equations

```
eq ((A -> A' : t) LL) [A'] = (A -> A' : t) .
eq LL [A] = noLink [owise] .
```

also benefit from the associativity and commutativity of the operator: The first equation simply matches the indexing address with the destination of a link somewhere in the list and returns that link. Otherwise, if the first equation cannot be applied, the constant operator `noLink` is returned. The equation

```
eq insert(A -> LL, (A -> A' : t) )
  = ( A -> insert(LL, (A -> A' : t) )) .
```

for the insert operator, which takes an adjacency list and a link as arguments, simply uses the insert operator defined for lists of links. The equations

```
eq insert((A -> LL) ; LG, (A -> A' : t) ) =
  insert(A -> LL, (A -> A' : t)) ; LG .
eq insert(LG, (A -> A' : t) ) =
  (A -> (A -> A' : t)) ; LG [owise] .

ceq insert(LG, L LL) =
  insert(insert(LG, L), LL)
if LL =/= mtLinkList .
```

define the insert operator for links graphs. As with the insert operator for terms of sort `LinkList`, the equations benefit from the associativity and commutativity of the concatenation operator. The first equation inserts the link in the adjacency list of the node's source address, if it is contained in the graph. Otherwise, if there is no adjacency list contained for the source of the link, a new adjacency list is inserted in the graph and returned. The last equation recursively decomposes a list of links and inserts them one-by-one. Finally, the indexing operator for terms of sort `LinkGraph` is defined by the following equations:

```
eq ((A -> LL) ; LG) [A] = LL .
eq LG [A] = mtLinkList [owise] .
```

If an adjacency list can be found for that address, the contained list of links is returned; otherwise, the constant operator `mtLinkList` is returned.

The graph in Figure 6.11 is represented by the following term:

($N_0$ -> ($N_0$ -> $N_1$, 0.01));
($N_1$ -> ($N_1$ -> $N_0$, 0.01)($N_1$ -> $N_3$, 0.01));
($N_3$ -> ($N_3$ -> $N_1$, 0.01));

of sort `LinkGraph`. In what follows, we assume that the delay of the links is always 10 ms and that the addresses of the links are equal to their names.

**The *ADR-MAPPING* module.** The module *ADR-MAPPING* specifies a mapping between addresses, where a source address is uniquely mapped to a destination address. The sort

```
sort APair .
```

which is constructed by the operator

```
op (_,_) : Address Address -> APair [ctor] .
```

represents an address pair. Terms of sort `APair` are used to represent the mapping between two addresses. The first argument of the constructor represents the source address of the mapping and the second argument the destination address. The sort

```
sort AddressMapping .
```

represents a list of address pairs. Since the subsort relationship

```
subsort APair < AddressMapping .
```

relates terms of the sort `APair` with the sort `AddressMapping`, the associative and commutative operator

```
op __ : AddressMapping AddressMapping -> AddressMapping
  [ctor comm assoc id: mtAddressMapping] .
```

creates lists of address pairs. The constant operator

```
op mtAddressMapping : -> AddressMapping [ctor] .
```

acts as the identity of the operator __. The address mapping does not contain two address pairs with the same source address. In order to create an address mapping, the operator

```
op update : AddressMapping APair -> AddressMapping .
```

is used to update the given address mapping with the given address pair. Hence, no two pairs with the same source address will be in the set, and consequently no non-determinism is introduced. Additionally, the equations that are used to specify the behavior benefit from the associativity and commutativity of the concatenation operator. Lastly, the operator

```
op _[_] : AddressMapping Address -> Address .
```

returns the address that a given address maps to.

The variables

```
var AM : AddressMapping .
vars A A' A'' : Address .
```

are used in the specification of the behavior of the operators. The two equations

```
eq ((A , A') AM ) [A] = A' .
eq AM [A] = none [owise] .
```

define the behavior of the lookup operator, which returns the destination address of an address pair if there is a mapping of the specified address, and otherwise returns the constant address `none`. Finally, the equations

```
eq update((A, A'') AM, (A , A')) =
  (A, A') AM .
eq update(AM, (A, A')) = (A, A') AM [owise] .
```

define the update operator, which replaces an address pair with the same source address if it is contained in the list. Otherwise, the address pair is inserted in the mapping.

**The *ZK-NODE* module.** The functional module *ZK-NODE* specifies the sorts and operators that define ZooKeeper nodes. The specification of the ZooKeeper service builds upon the modularized actor model. A ZooKeeper node is represented as an actor. The operator

```
op ZKNode : -> ActorType .
```

specifies its type. The attributes

```
op zk:_ : Address -> Attribute [gather(&)] .
op watch-list:_ : WatchList -> Attribute [gather(&)] .
op store:_ : Store -> Attribute [gather(&)] .
op request-buffer:_ : ContentsList -> Attribute [gather(&)] .
op watch-buffer:_ : ContentsList -> Attribute [gather(&)] .
```

define the internal state of the actor. The attribute `zk:_` stores a reference to the main ZooKeeper actor. The watch list of a ZooKeeper node is stored in the attribute `watch-list :_`. The `store:_` attribute is used to store the id-value store of the ZooKeeper node, which represents the in-memory image of the ZooKeeper state. The watch list of the ZooKeeper node is local to the node in contrast to the store, which is replicated on all ZooKeeper nodes. The attributes `request-buffer` and `watch-buffer` store the requests and watch requests until they are processed. The operators

```
op request : Address Id -> Contents .
op value : Id Value -> Contents .
op update : Id Value -> Contents .
op watch : Address Id -> Contents .
op notify : Id -> Contents .
```

specify the contents of messages that are sent from clients of the ZooKeeper service to the ZooKeeper service and vice versa. If a client wants to retrieve a value of a specific id, it sends a message of the form `A <- request(C, I)` to the ZooKeeper with `A` being the address of the ZooKeeper service, `C` the address of the client, and `I` the id of the value the client wants to retrieve. The ZooKeeper service answers with a message of the form `C <- value(I, V)` with the value `V`. If a client wants to update a value of an id in the ZooKeeper service, it sends a message of the form `A <- update(I,V)`. Similarly, if a client wants to set a watch on a specific id, it sends a message of the form `A <- watch(C, I)`. If the value changes, the client is notified by the ZooKeeper service by sending a message of the form `C <- notify(I)`.

The operators

```
op processRequest : -> Contents [ctor] .
op processWatch : -> Contents [ctor] .
op die? : -> Contents [ctor] .
```

specify the contents of messages that nodes send to themselves in order to trigger periodic actions. In order to simulate the user-defined time it takes to process a request (respectively a watching request), after receiving a request the ZooKeeper node sends a message of the form `N <- processRequest` (respectively of the form `N <- processWatch`). Additionally, to specify the failure probability of a ZooKeeper node, each ZooKeeper node periodically sends to itself a message of the form `N <- die?`. Upon receiving this message, the node tosses a coin with a specified probability to determine whether it has failed or not. The operator

```
op die! : Address -> Contents [ctor] .
```

is then used to send a message of the form `A <- die!(C)` to the ZooKeeper service to indicate that the node with the specified address has died. The auxiliary operator

```
op createNotifications : AddressList Contents Float Float Nat -> Config .
```

takes an address list, the contents, two times $t_1$ and $t_2$, and a natural number $n$ as arguments and returns a list of messages. For each address in the address list, one message with the specified contents is created at time $t_1 + t_2 \cdot n$. The natural number $n$ is incremented every time a message is created. Lastly, the operator

```
op ZKNodeInit : Address WatchList Store Address Float -> Config .
```

initializes the ZooKeeper node with the specified address, watch list, address of the ZooKeeper service, and the current global time.

The variables

```
vars A ZK : Address .
var AL : AddressList .
vars gt t : Float .
var N : Nat .
var CO : Contents .
var WL : WatchList .
var ST : Store .
```

are used in the specification of the behavior of the above operators. The equation

```
eq createNotifications(A, CO, gt, t, N) = [gt + float(N) * t, A <- CO] .
ceq createNotifications(A ; AL, CO, gt, t, N) =
  createNotifications(A, CO, gt, t, N) createNotifications(AL, CO, gt, t, s(N))
if AL =/= mtAddressList .
endfm
```

creates a list of messages of the form `[gt + float(N)* t, A <- CO]` for each address `A` in the list of addresses. The natural number `N` is incremented each time a message is created. Finally, the equations

```
eq ZKNodeInit(A, WL, ST, ZK, gt) =
  < A : ZKNode | watch-list: WL, store: ST, zk: ZK,
  request-buffer: mtContentsList, watch-buffer: mtContentsList >
  [gt + node-failure-period, A <- die?] .
```

initialize the ZooKeeper node.

**The *ZK* module.** The functional module *ZK* specifies the static parts of the ZooKeeper service. The ZooKeeper service is represented by a Russian dolls actor, which contains the ZooKeeper nodes in its configuration. The operator

```
op Zookeeper : -> ActorType .
```

specifies the type of the ZooKeeper actor. The attributes

```
op activeNodes:_ : AddressList -> Attribute [ctor gather(&)] .
op deadNodes:_ : AddressList -> Attribute [ctor gather(&)] .
```

store references to the internal active and dead ZooKeeper nodes. The ZooKeeper service needs to know the link delays between the service itself and the ZooKeeper nodes, and between the clients and the ZooKeeper nodes. The attribute

```
op links:_ : LinkGraph -> Attribute [ctor gather(&)] .
```

stores this link-graph with link delays. The attribute

```
op node-mapping:_ : AddressMapping -> Attribute [ctor gather(&)] .
```

holds a mapping between a client and its closest ZooKeeper node, instead of computing the closest node every time a client sends a request or a watching request. The auxiliary attributes

```
op totalDeadNodes:_ : Nat -> Attribute [ctor gather(&)] .
op droppedMessages:_ : Nat -> Attribute [ctor gather(&)] .
op internalDropped:_ : Nat -> Attribute [ctor gather(&)] .
```

are used for statistical model-checking purposes. The attribute `totalDeadNodes:_` stores the total amount of ZooKeeper nodes that have died; the attribute `droppedMessages:_` indicates the number of messages that are dropped if there are less than a majority of nodes active; and the attribute `internalDropped` represents the number of messages that are sent to dead ZooKeeper nodes. The operator

```
op recover : Address -> Contents [ctor] .
```

creates the contents of a message that is sent from the ZooKeeper service to itself to trigger a ZooKeeper node to recover after it has failed. Finally, the operator

```
op ZKInit : Address Float -> Config .
```

takes the address of the ZooKeeper service and the global time at which it is initialized and returns a ZooKeeper service with five internal ZooKeeper nodes as illustrated in Figure 6.9.

The variables

```
var A : Address .
var C : Config .
var gt : Float .
var LG : LinkGraph .
var AN : AddressList .
```

are used in the following conditional equation, which defines the behavior of the `ZKInit` operator.

```
--- Initializes a specific zookeeper containing 5 nodes.
ceq ZKInit(A, gt) =
  < A : Zookeeper | config: C, links: LG, node-mapping: mtAddressMapping,
    activeNodes: A . 0 ; A . 1 ; A . 2 ; A . 3 ; A . 4,
    deadNodes: mtAddressList, totalDeadNodes: 0,
    droppedMessages: 0, internalDropped: 0 >
if
  C :=
    --- Create the five internal nodes.
    ZKNodeInit(A . 0, mtWatchList, mtStore, A, gt)
    ZKNodeInit(A . 1, mtWatchList, mtStore, A, gt)
    ZKNodeInit(A . 2, mtWatchList, mtStore, A, gt)
    ZKNodeInit(A . 3, mtWatchList, mtStore, A, gt)
    ZKNodeInit(A . 4, mtWatchList, mtStore, A, gt) /\
    --- Create the links
    LG := insert(mtLinkGraph,
      (A -> A . 0 : 0.015) --- Node in Berlin
      (A -> A . 1 : 0.015) --- Node in Rome
      (A -> A . 2 : 0.030) --- Node in London
      (A -> A . 3 : 0.020) --- Node in Paris
      (A -> A . 4 : 0.040) ) --- Node in Barcelona .
```

It creates the actor that represents the ZooKeeper service. The internal configuration contains the five ZooKeeper nodes, which are initialized using the `ZKNodeInit` operator with an empty list of watching clients, an empty store, the address of the ZooKeeper service, and the current global time. Additionally, the link graph is created according to Figure 6.9.

**The *ZK-NODE-DYNAMICS* module.** The system module *ZK-NODE-DYNAMICS* specifies the dynamic aspects of a ZooKeeper node. The variables

```
vars A A' : Address .
var CO : Contents .
var CL : ContentsList .
var AS : AttributeSet .
var gt : Float .
var I : Id .
var V : Value .
var ST : Store .
var WL : WatchList .
var WE : WLEntry .
```

are used in the following rewrite rules. The rule

```
rl [ZOOKEEPER-NODE-REQUEST] :
  < A : ZKNode | request-buffer: CL, AS >
  {gt, A <- request(A', I)}
=>
  < A : ZKNode | request-buffer: request(A', I) CL, AS >
  if (CL == mtContentsList) then
    [gt + request-processing-time, A <- processRequest]
  else
    null
  fi .
```

matches if a ZooKeeper node receives a message of the form `A <- request(A', I)` with the client address `A'` and id `I`. The request is added to the request buffer. If the buffer was previously empty, the node sends itself a message of the form `A <- processRequest` with an arrival delay computed using the constant operator `request-processing-time`. Similarly, the rule

```
rl [ZOOKEEPER-NODE-REQUEST] :
  < A : ZKNode | watch-buffer: CL, AS >
  {gt, A <- watch(A', I)}
=>
  < A : ZKNode | watch-buffer: watch(A', I) CL, AS >
  if (CL == mtContentsList) then
    [gt + watch-processing-time, A <- processWatch]
  else
    null
  fi .
```

adds the watching request to the watching request buffer. Despite requests and watching requests, updates are processed directly, since they already arrive after the delay between the ZooKeeper service and the specific ZooKeeper node. The rule

```
crl [ZOOKEEPER-NODE-UPDATE] :
  < A : ZKNode | store: ST, watch-list: WL, AS >
  {gt, A <- update(I, V)}
=>
  < A : ZKNode | store: ST[I -> V], watch-list: removeWatching(WL, I), AS >
  if (WE =/= none) then
    createNotifications( WE .list, notify(I), gt, notify-creation-time, 1)
  else
    null
  fi
if
  WE := WL[I] .
```

updates the value of the id in the store and notifies the watching clients. The rule

```
rl [ZOOKEEPER-NODE-PROCESS-REQUEST] :
  < A : ZKNode | request-buffer: request(A', I) CL, store: ST , AS >
  {gt, A <- processRequest}
=>
  < A : ZKNode | request-buffer: CL, store: ST, AS >
  [gt, A' <- value(I, ST[I])]
  if (CL == mtContentsList) then
    null
  else
    [gt + request-processing-time, A <- processRequest]
  fi .
```

processes the first request from the request buffer, and, if the remaining buffer is not empty, triggers the processing of the next request. Similarly, the rule

```
rl [ZOOKEEPER-NODE-PROCESS-WATCH] :
  < A : ZKNode | watch-buffer: watch(A', I) CL, watch-list: WL, AS >
  {gt, A <- processWatch}
=>
  < A : ZKNode | watch-buffer: CL, watch-list: addWatch(WL, I, A'), AS >
  if (CL == mtContentsList) then
    null
  else
    [gt + watch-processing-time, A <- processWatch]
  fi .
```

processes the first watching request from the buffer and, if the remaining buffer is not empty, triggers the processing of the next request. Finally, node failure is simulated by the rule

```
rl [ZOOKEEPER-NODE-DIE?] :
  < A : ZKNode | zk: A', AS >
  {gt, A <- die?}
=>
  if (sampleBerWithP(node-failure-prob)) then
    [gt, A' <- die!(A)]
  else
    < A : ZKNode | zk: A', AS >
    [gt + node-failure-period, A <- die?]
  fi .
```

which periodically tosses a coin by using the operator `sampleBerWithP` with the probability specified by the constant operator `node-failure-prob`. If the test succeeds, the node sends a message of the form `A' <- die!(A)` immediately to the ZooKeeper service to indicate that the node has died and removes itself from the configuration. Otherwise, a message of the form `A <- die?` is sent, with the appropriate delay, to the node itself to trigger again the periodic behavior.

**The *ZK-DYNAMICS* module.** The system module *ZK-DYNAMICS* specifies the dynamic aspects of the ZooKeeper service. The auxiliary operators

```
op randomAdjList : Address AddressList -> [LinkList] .
op rndAdjList : Address Address [Float] -> [Link] .
```

are used to create a list of random links to the list of ZooKeeper nodes when a new client sends a first request or a watching request to the ZooKeeper service. The operator

```
op closestLink : LinkGraph Link Address AddressList -> Link .
```

takes a link graph, a link, a client address, and the list of addresses of the active ZooKeeper nodes as arguments and returns the link that has the shortest delay between the client and a ZooKeeper node. The auxiliary operator

```
op createUpdateMessages : Float LinkGraph Address AddressList Contents -> Config .
```

creates the update messages for the ZooKeeper nodes according to the delays specified in the link graph. The first argument specifies the current global time and the second argument the link graph. The third argument specifies the address of the ZooKeeper service which is used to compute the message delays. The fourth arguments specifies the list of addresses of

the active ZooKeeper nodes, and, finally, the last argument specifies the contents that are sent. The variables

```
vars A A' A'' N : Address .
vars AS AS' : AttributeSet .
var NM : AddressMapping .
vars LG LG' : LinkGraph .
vars AN DN AL : AddressList .
var C : Config .
vars gt t t' : Float .
var R : Nat .
vars L L' : Link .
var AJL : LinkList .
var V : Value .
var I : Id .
var WL : WatchList .
var ST : Store .
var CO : Contents .
```

are used below to specify the dynamic behavior of the ZooKeeper service. The operator `randomAdjList` is defined by the equations

```
eq randomAdjList(A, A') = rndAdjList(A, A', genRandom(0.050, 0.25)) .
ceq randomAdjList(A, A' ; AL) = randomAdjList(A, A') randomAdjList(A, AL) if AL
    =/= mtAddressList .

eq rndAdjList(A, A', t) = (A -> A' : t) .
```

which recursively apply the `rndAdjList` operator with a randomly generated time between 50 and 250 ms to all addresses in the list of addresses. The closest link is computed by the equation

```
eq closestLink (LG, L, A, mtAddressList) = L .

ceq closestLink(LG, L, A, A' ; AL) =
  if (L .delay < L' .delay) then
    closestLink(LG, L, A, AL)
  else
    closestLink(LG, L', A, AL)
  fi
if L' := (LG[A])[A'] .
```

which recursively decomposes the list of addresses and compares the link between the address `A` and the current address with the link `L`. The link whose delay is smaller is used in the further recursive call. If the list of addresses is empty, the link `L` is returned. The equations

```
eq createUpdateMessages(gt, LG, A, mtAddressList, update(I, V)) = null .
eq createUpdateMessages(gt, LG, A, A' ; AN, update(I, V)) =
  [gt + ((LG[A])[A']) .delay, A' <- update(I, V)]
  createUpdateMessages(gt, LG, A, AN, update(I, V)) .
```

recursively create messages with a delay according to the link graph.

The following rewrite rules define the behavior of the ZooKeeper service. The rule

```
rl [ZOOKEEPER-UPDATE] :
  < A : Zookeeper | links: LG, activeNodes: AN, deadNodes: DN, config: C,
    droppedMessages: R, AS >
  {gt, A <- update(I, V)}
```

```
=>
  if (float(AN .size) / float(AN .size + DN .size) > 0.5) then
    < A : Zookeeper | links: LG, activeNodes: AN, deadNodes: DN,
      config: createUpdateMessages(gt, LG, A, AN, update(I, V)) C,
      droppedMessages: R, AS >
  else
    < A : Zookeeper | links: LG, activeNodes: AN, deadNodes: DN, config: C,
      droppedMessages: s(R), AS >
  fi .
```

reacts to an update message from a client. If a (strict) majority of ZooKeeper nodes are active, then the message is forwarded to the ZooKeeper nodes with a delay according to the link graph. Otherwise, if the majority of nodes is dead, the status attribute `droppedMessages:_` is incremented. The rule

```
rl [ZOOKEEPER-NODE-DEATH] :
  < A : Zookeeper | config: {gt, A <- die!(A')} C, activeNodes: AN,
    deadNodes: DN, totalDeadNodes: R, AS >
=>
  < A : Zookeeper | config: C, activeNodes: remove(AN, A'), deadNodes: A' ; DN,
    totalDeadNodes: s(R), AS >
  [gt + node-recover-time, A <- recover(A')] .
```

handles a message of the form `A <- die!(A')` which is sent by one of the internal ZooKeeper nodes to tell the ZooKeeper service that the node has died. The ZooKeeper service removes the address of the node from the list of active nodes, adds the address to the list of dead nodes, increments the value of the `totalDeadNodes:_` attribute, and sends a recover message to itself. The recover message triggers a node recovery after the specified time. The recovery behavior is specified by the following two rules:

```
crl [ZOOKEEPER-NODE-RECOVERY-COPY] :
  < A : Zookeeper | activeNodes: AN, deadNodes: DN,
    config: < A' : ZKNode | store: ST, AS' > C, AS >
  {gt, A <- recover(A'')}
=>
  < A : Zookeeper | activeNodes: A'' ; AN, deadNodes: remove(DN, A''),
    config: < A' : ZKNode | store: ST, AS' >
      ZKNodeInit(A'', mtWatchList, ST, A, gt) C, AS >
if AN =/= mtAddressList /\ A' in AN .

rl [ZOOKEEPER-NODE-RECOVERY-0] :
< A : Zookeeper | config: C, activeNodes: mtAddressList, deadNodes: DN, AS >
  {gt, A <- recover(A')}
=>
< A : Zookeeper | config: ZKNodeInit(A', mtWatchList, mtStore, A, gt) C,
  activeNodes: A', deadNodes: remove(DN, A'), AS > .
```

The first rule recovers a ZooKeeper node by copying the state of another active node. If no other node is active, an empty ZooKeeper node is created. Since a dead node removes itself from the internal configuration, internal messages that are addressed to dead nodes need to be consumed by the ZooKeeper service. The rule

```
crl [ZOOKEEPER-CONSUME-MESSAGES-FOR-DEAD-NODES] :
  < A : Zookeeper | config: {gt, A' <- CO} C, deadNodes: DN,
    internalDropped: R, AS >
=>
```

```
  < A : Zookeeper | config: C, deadNodes: DN, internalDropped: s(R), AS >
if A' in DN .
```

specifies this behavior. Additionally, the value of the status attribute `internalDropped:_` is incremented.

The ZooKeeper service is modeled as a Russian dolls actor, thus internal messages need to cross the boundary. The boundary-crossing rewrite rules

```
crl [ZOOKEEPER-VALUE-FWD] :
  < A : Zookeeper | config: {gt, A' <- value(I, V)} C,
    node-mapping: NM, links: LG, AS >
=>
  < A : Zookeeper | config: C, node-mapping: NM, links: LG, AS >
  [gt + L .delay, A' <- value(I, V)]
if L := (LG[A'])[NM[A']] .

crl [ZOOKEEPER-VALUE-FWD] :
  < A : Zookeeper | config: {gt, A' <- notify(I)} C,
    node-mapping: NM, links: LG, AS >
=>
  < A : Zookeeper | config: C, node-mapping: NM, links: LG, AS >
  [gt + L .delay, A' <- notify(I)]
if L := (LG[A'])[NM[A']] .
```

forward to the outside internal messages of the form `A' <- value(I,V)` and `A' <- notify(I)` with the delay specified in the link graph. On the other hand, if a request or a watching request is sent to the ZooKeeper service, it is handled by the following boundary-crossing rewrite rules

```
crl [ZOOKEEPER-REQUEST] :
  < A : Zookeeper | node-mapping: NM, links: LG,
    activeNodes: AN, deadNodes: DN,
    droppedMessages: R, config: C, AS >
  {gt, A <- request(A', I)}
=>
  if (AN .size > 0) then
    if (N == none) then
      < A : Zookeeper | node-mapping: update(NM, (A', A')),
        links: insert(LG, randomAdjList(A', AN ; DN)), activeNodes: AN,
        deadNodes: DN, config: C, droppedMessages: R, AS >
      [gt, A <- request(A', I)]
    else
      if (N in AN) then
        < A : Zookeeper | node-mapping: NM, links: LG, activeNodes: AN,
          deadNodes: DN, config: [gt + t, N <- request(A', I)] C,
          droppedMessages: R, AS >
      else
        < A : Zookeeper | node-mapping: update(NM, (A', L .dst)), links: LG,
          activeNodes: AN, deadNodes: DN,
          config: [gt + L .delay, L .dst <- request(A' , I) ] C,
          droppedMessages: R, AS >
      fi
    fi
  else
    < A : Zookeeper | node-mapping: NM, links: LG, activeNodes: AN, config: C,
      droppedMessages: s(R), AS >
```

```
    fi
  if N := NM[A'] /\
     t := ((LG[A'])[N]) .delay /\
     L := closestLink(LG, (A' -> A' : 1000000.0), A', AN).

  crl [ZOOKEEPER-WATCH] :
    < A : Zookeeper | node-mapping: NM, links: LG,
      activeNodes: AN, deadNodes: DN,
      config: C, droppedMessages: R, AS >
    {gt, A <- watch(A', I)}
  =>
    if (AN .size > 0) then
      if (N == none) then
        < A : Zookeeper | node-mapping: update(NM, (A', A')),
          links: insert(LG, randomAdjList(A', AN ; DN)), activeNodes: AN,
          deadNodes: DN, config: C, droppedMessages: R, AS >
        [gt, A <- watch(A', I)]
      else
        if (N in AN) then
          < A : Zookeeper | node-mapping: NM, links: LG, activeNodes: AN,
          deadNodes: DN, config: [gt + t, N <- watch(A', I)] C,
          droppedMessages: R, AS >
        else
          < A : Zookeeper | node-mapping: update(NM, (A', L .dst)), links: LG,
            activeNodes: AN, deadNodes: DN,
            config: [gt + L .delay, L .dst <- watch(A' , I) ] C,
            droppedMessages: R, AS >
        fi
      fi
    else
      < A : Zookeeper | node-mapping: NM, links: LG, activeNodes: AN, config: C,
        droppedMessages: s(R), AS >
    fi
  if N := NM[A'] /\
     t := ((LG[A'])[N]) .delay /\
     L := closestLink(LG, (A' -> A' : 1000000.0), A', AN).
```

The first rewrite rule handles the case when a client sends a request to the ZooKeeper service. If there is as least one active node, the request can be processed. Otherwise, if there is no active node, the request is dropped, and the value of the status attribute `droppedMessages:_` is incremented. The request can be processed according to the following three cases:

- If the client has not yet sent a request or a watching request to the Zookeeper service, an adjacency list with random link delays is created for the new client and is inserted into the link graph. Then, the request is immediately resent.
- If the client has already sent a request or watching request to the ZooKeeper service, the client and the closest ZooKeeper node are contained in the node mapping. If the closest node is still alive, the request is simply forwarded.
- If the client and the closest ZooKeeper node are contained in the node mapping, but the closest node is currently not active, a new closest ZooKeeper node is computed, the node mapping is updated, and the request is sent to the new node.

The second rewrite rule handles the case when a client sends a watching request to the ZooKeeper. Similarly, this rewrite rule processes the watching request.

### 6.3.2. Maude specification of the Group-Key Management Service

The group-key management service uses the ZooKeeper event notification mechanism for broadcasting the keys to the clients. At a high level, after a client has connected to the group-key management service, the group-key management service sends the address of the ZooKeeper service as well as the ids of the client's personal key and the shared key to the client. The client then sets watches on these ids. If the group-key management server performs a periodic key update or a client joins or leaves the group, the server sends updates of the corresponding ids to the ZooKeeper service. The ZooKeeper service, in turn, notifies the watching clients about the key updates.

The clients can join the group-key management service by sending a **join** message to the group-key management server. The server acknowledges the message by sending an **accepted** message to the client. The **accepted** message additionally contains the address of the ZooKeeper service, the ids of the shared and the client's key and the current value of the client's key. If a key update occurs — due to a periodic key update, or a client joining or leaving the group — the client is notified by the ZooKeeper service. If the client wants to leave the group-key management service, it sends a **leave** message to the group-key management server. The server acknowledges by answering with a **left** message.

In our specification, new clients are periodically generated by a generic generator. The clients directly join the group after being generated and leave the group after a randomly chosen time.

#### Overview

Figure 6.12 gives an overview of the Maude modules comprising the specification of the group-key management service. The already described modules *ZK* and *ZK-DYNAMICS* containing the specification of the ZooKeeper service appear at the bottom of the module hierarchy. The functional module *GKM* specifies the static aspects of the group-key management server, while the functional module *GKM-CLIENT* specifies the static aspects of the clients. The two system modules *GKM-DYNAMICS* and *GKM-CLIENT-DYNAMICS* specify the dynamic aspects of the group-key management server and its clients. For the periodic generation of clients that join and leave the group, the modularized specification of the generator is used. The description of the generator can be found in Appendix B.2. The functional module *GENERATOR-WRAPPER* specifies the required equations of the theory *GENERATOR-INTERFACE*. Lastly, the system module *GKM-INIT* puts all the parts together and instantiates the generator with the view *GeneratorWrapper*, which defines a view from the *GENERATOR-INTERFACE* to the *GENERATOR-WRAPPER*.

#### Description of modules

In the following, the specification of the group-key management service is described in-depth, following the above hierarchical module structure.

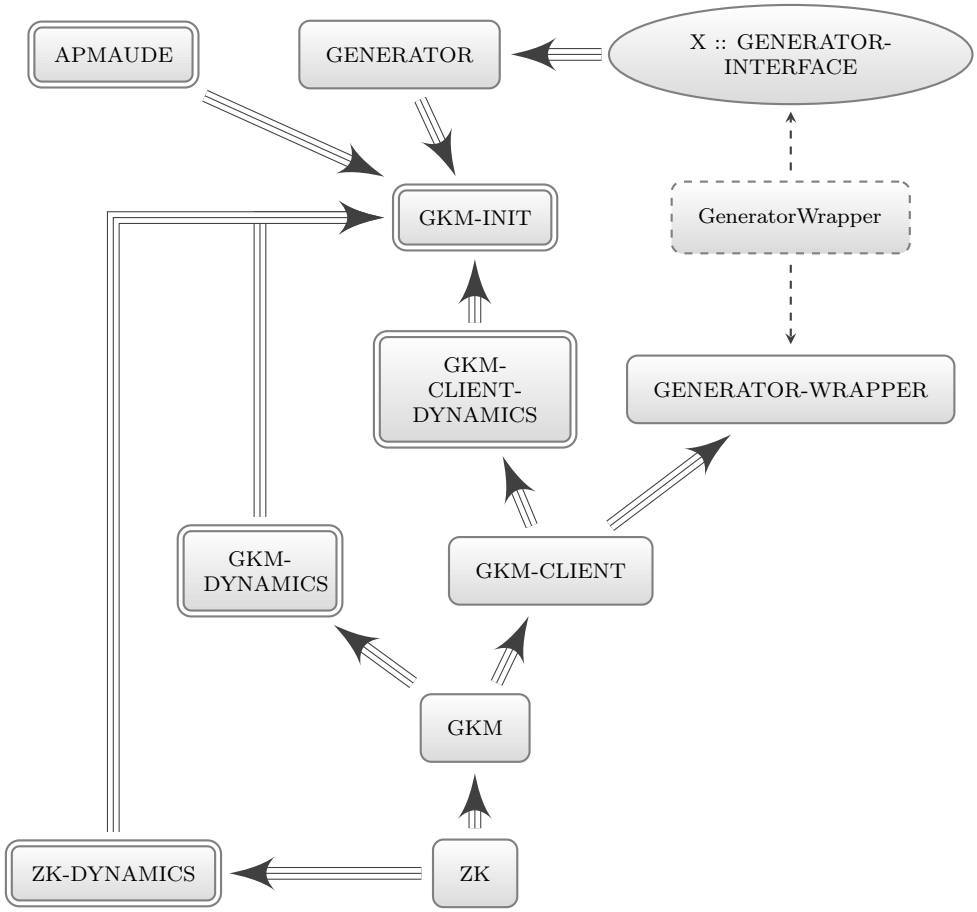



**Figure 6.12.:** Overview of Maude modules of the Group-Key Management specification

**The *GKM* module.** The functional module *GKM* specifies the static aspects of the group-key management server. The specification is based on the modularized actor model. Thus, the operator

```
op GKManager : -> ActorType .
```

specifies the type of the actor that models the group-key management server. The attributes

```
op clients:_ : AddressList -> Attribute [ctor gather(&)] .
op zookeeper:_ : Address -> Attribute [ctor gather(&)] .
op lastkey:_ : Nat -> Attribute [ctor gather(&)] .
```

define the state of the group-key management server. The attribute `clients:_` stores the addresses of the currently connected clients. The address of the ZooKeeper service is stored in the attribute `zookeeper:_`. The attribute `lastkey:_` is used to compute the next key. In this specification, a new key is created by incrementing the last key. The operators

```
op join : Address -> Contents [ctor] .
op accepted : Address Id Id Value -> Contents [ctor] .
op leave : Address -> Contents [ctor] .
op left : -> Contents [ctor] .
```

define the interface of the group-key management server, i.e., which messages are accepted and sent by the server. When a client with address `C` wants to join the group, it sends a message of the form `GKM <- join(C)` to the server with address `GKM`. The server answers with a message of the form `C <- accepted(ZKA,` $id_{k_{\text{group}}}$`,` $id_{k_C}$`,` $k_C$`)` with the address of the ZooKeeper service `ZKA`, the ids of the group key and the client key, and the value of the clients key. If a client with address `C` wants to leave the group, it sends a message of the form `GKM <- leave(C)` to the server. The server acknowledges this message by answering with a `C <- left` message. The operator

```
op periodic-key-upd : -> Contents [ctor] .
```

is used in periodic self-sent messages to trigger the periodic update of the *shared key*. The two operators

```
op sharedGKey : -> Id [ctor] .
op clientKey : Address -> Id [ctor] .
```

create the ids of the *shared key* and the *client keys*. The operator `clientKey` takes the address of the client as argument and returns the id for the particular client. The operator

```
op key : Nat -> Value .
```

takes a natural number and creates a term of sort `Value`. Since the group-key management server needs to update the particular *client keys* if a client joins or leaves the group, the auxiliary operator

```
op createUpdateKeyMsgs : Address AddressList Float Value -> Config .
```

takes the address of the ZooKeeper service, the addresses of the clients, the current global time and the new value for the keys as arguments, and returns the list of update messages. Finally, the operator

```
op GKMInit : Address Address Float -> Config .
```

takes the address of the group-key management server, the address of the ZooKeeper service, and the current global time as arguments and returns an initialized group-key management server.

The variables

```
vars A AN ZKA : Address .
vars t npu nhb nd : Float .
var V : Value .
var L : AddressList .
```

are used in the following equations. The two equations

```
eq createUpdateKeyMsgs(ZKA, mtAddressList, t, V) = null .
eq createUpdateKeyMsgs(ZKA, A ; L, t, V) =
  [t, ZKA <- update(clientKey(A), V) ] createUpdateKeyMsgs(ZKA, L, t, V) .
```

recursively create the update messages for the keys of the given client addresses. The update messages are sent to the ZooKeeper service, which notifies all clients that have set a watch on that id. The group-key management server is initialized with the following equation

```
eq GKMInit(A, ZKA, t) =
  < A : GKManager | clients: mtAddressList , zookeeper: ZKA, lastkey: 0 >
  [t, A <- periodic-key-upd] .
```

which sets the default values and starts the periodic updates of the *shared key*.

**The *GKM-CLIENT* module.** The functional module *GKM-CLIENT* specifies the static aspects of the clients of the group-key management service. As with the specification of the group-key management server, the clients are also specified as actors. The sort

```
sort Status .
```

which is created with the constant operators

```
ops created connected disconnected : -> Status .
```

specifies the state of a client: It is either just being created, is already connected (joined the group), or is disconnected (left the group). The operator

```
op GKClient : -> ActorType .
```

specifies the type of the group-key management client. Furthermore, the attributes

```
op state:_ : Status -> Attribute [gather(&)] .
op gkm:_ : Address -> Attribute [gather(&)] .
op zk:_ : Address -> Attribute [gather(&)] .
op groupKey:_ : Value -> Attribute [gather(&)] .
op clientKey:_ : Value -> Attribute [gather(&)] .
op groupKeyId:_ : Id -> Attribute [gather(&)] .
op clientKeyId:_ : Id -> Attribute [gather(&)] .
```

represent the internal state of the client. The attribute `state:_` contains the current state of the client: created, connected, or disconnected. The attributes `gkm:_` and `zk:_` store the addresses of the group-key management server and the ZooKeeper service. The values of the current keys are stored in the `groupKey:_` and `clientKey:_` attribute, while the ids of the keys are stored in the `groupKeyId:_` and `clientKeyId:_` attributes. Lastly, the operator

```
op clientInit : Address Address Float -> Config .
```

initializes the client with the address of the client itself, the address of the group-key server, and the current global time. The equation

```
eq clientInit(A, GMA, t) =
  < A : GKClient | state: created, gkm: GMA>
  [t, GMA <- join(A)]
  [t + genRandom(gk-client-min-disconnect-time, gk-client-max-disconnect-time),
    GMA <- leave(A)] .
```

which uses the variables

```
vars A, GMA CA : Address .
var t : Float .
```

initializes the client, sends a join message to the group-key management server, and triggers the client to leave after a uniformly chosen time between `t + gk-client-min-disconnect-time` and `t + gk-client-max-disconnect-time`.

**The *GENERATOR-WRAPPER* module.** The module *GENERATOR-WRAPPER* is used to define the view between the *GENERATOR-INTERFACE* theory and the *GROUPKEYMANAGER-CLIENT* module.

```
mod GENERATOR-WRAPPER is
  pr FLOAT .
  pr ACTOR-MODEL .
  pr GROUPKEYMANAGER-CLIENT .

  op generator-spawn-period : -> Float .
  op generator-create : AttributeSet Float Address -> Config .

  vars GMA A CA : Address .
  var AS : AttributeSet .
  var gt : Float .

  eq generator-spawn-period = gk-client-spawn-period .

  eq generator-create((gkm: GMA, AS), gt, A) =
    clientInit(A, GMA, gt) .
endm
```

**The *GKM-CLIENT-DYNAMICS* module.** The system module *GKM-CLIENT-DYNAMICS* specifies the dynamic behavior of a group-key management client. The variables

```
vars A GMA ZKA CA : Address .
vars gt t : Float .
var AS : AttributeSet .
vars SI CI I : Id .
vars SV CV SV' CV' V : Value .
var S : Status .
var N : Nat .
```

are used in the following rewrite rules. The client has to react to two different types of messages: Messages sent by the group-key management server, and messages sent by the ZooKeeper service. The rewrite rule

```
rl [GROUPKEYMANAGER-CLIENT-RECEIVES-ACCEPTED-MSG] :
  < A : GKClient | state: created, AS >
  {gt, A <- accepted(ZKA, SI, CI, V)}
=>
  < A : GKClient | state: connected, zk: ZKA,
    groupKeyId: SI, clientKeyId: CI, clientKey: V, groupKey: undef, AS >
  [gt, ZKA <- watch(A, SI)]
  [gt, ZKA <- watch(A, CI)]
  [gt, ZKA <- request(A, SI)]
  [gt, ZKA <- request(A, CI)] .
```

reacts to the `accepted` message sent by the group-key management server. The message contains the address of the ZooKeeper service, the ids of the keys, and the current value of the client's key. Thus, the client sets watches on the ids, requests the values of the ids, and changes its state to connected.

The rewrite rule

```
rl [GROUPKEYMANAGER-CLIENT-RECEIVES-LEFT-MSG] :
  < A : GKClient | state: connected, AS >
  {gt, A <- left}
=>
  < A : GKClient | state: disconnected, AS > .
```

reacts to the second type of message that can be sent by the group-key management server: The `left` message. The client simply changes its state from connected to disconnected. The three rewrite rules

```
rl [GROUPKEYMANAGER-CLIENT-RECEIVES-REQUESTED-VALUE-GROUPKEY] :
  < A : GKClient | groupKeyId: SI, groupKey: SV, AS >
  {gt, A <- value(SI, V )}
=>
  < A : GKClient | groupKeyId: SI, groupKey: V, AS > .

rl [GROUPKEYMANAGER-CLIENT-RECEIVES-REQUESTED-VALUE-CLIENTKEY] :
  < A : GKClient | clientKeyId: CI, clientKey: SV, AS >
  {gt, A <- value(CI, V )}
=>
  < A : GKClient | clientKeyId: CI, clientKey: V, AS > .

rl [GROUPKEYMANAGER-CLIENT-RECEIVES-NOTIFY] :
  < A : GKClient | zk: ZKA, state: S, AS >
  {gt, A <- notify(I)}
=>
  < A : GKClient | zk: ZKA, state: S, AS >
  [gt, ZKA <- request(A, I)]
  [gt, ZKA <- watch(A, I)] .
```

react to messages sent by the ZooKeeper service. The first two rules receive the value of an id and store it in the corresponding attribute. The last rewrite rule reacts to a notification of the ZooKeeper service. The client requests the value and sets the watch again.

**The *GKM-DYNAMICS* module.** The system module *GKM-DYNAMICS* specifies the behavior of the group-key management server: The server receives join and leave requests from the clients, and periodically updates the keys. The variables

```
vars GA ZKA CA AA : Address .
vars CL NL : AddressList .
vars K NK : Nat .
var AS : AttributeSet .
var gt : Float .
```

are used in the following rewrite rules. The rewrite rule

```
crl [GROUPKEYMANAGER-CLIENT-JOIN] :
  < GA : GKManager | clients: CL , zookeeper: ZKA, lastkey: K, AS >
  {gt, GA <- join(CA)}
=>
  < GA : GKManager | clients: (CA ; CL), zookeeper: ZKA, lastkey: NK, AS >
  [gt, ZKA <- update(sharedGKey, key(NK))]
  [gt, ZKA <- update(clientKey(CA), key(NK))]
  [gt, CA <- accepted(ZKA, sharedGKey, clientKey(CA), key(NK)) ]
if NK := s(K) .
```

specifies the behavior of the server when a client with address `CA` joins the group: First, the key is updated by simply incrementing its value. Then, update messages for the shared key and the client's key are sent to the ZooKeeper. Finally, the server sends an accepted message to the client. The conditional rewrite rule

```
crl [GROUPKEYMANAGER-CLIENT-LEAVE] :
  < GA : GKManager | clients: CL , zookeeper: ZKA, lastkey: K, AS >
  {gt, GA <- leave(CA)}
=>
  < GA : GKManager | clients: NL, zookeeper: ZKA, lastkey: NK, AS >
  [gt, ZKA <- update(sharedGKey, key(NK))]
  createUpdateKeyMsgs(ZKA, NL, gt, key(NK))
  [gt, CA <- left ]
if NK := s(K) /\
   NL := remove(CL, CA) .
```

specifies how the server reacts when a client leaves the group. First, a new key is created by simply incrementing the last key and the client is removed from the list of clients. Then, the shared key is updated, and all client keys but the key of the client that just left are likewise updated. Finally, a `left` message is sent to the client. The group-key management server needs to periodically create new shared keys and send them to its clients. The conditional rewrite rule

```
crl [GROUPKEYMANAGER-PERIODIC-KEY-UPD] :
  < GA : GKManager | zookeeper: ZKA, lastkey: K, AS >
  {gt, GA <- periodic-key-upd}
=>
  < GA : GKManager | zookeeper: ZKA, lastkey: NK, AS >
  [gt, ZKA <- update(sharedGKey, key(NK))]
  [gt + gk-shared-key-upd-period, GA <- periodic-key-upd]
if NK := s(K) .
```

reacts to the self-sent periodic message of the form `GA <- periodic-key-upd`. Upon receiving this message, the server sends an update message of the shared key to the ZooKeeper service, and reschedules the periodic message. The ZooKeeper service will then notify the watching clients.

**The *GKM-INIT* module.** Finally, the system module *GKM-INIT* puts all these parts together. The generic generator is instantiated with the view *GeneratorWrapper*, which defines a view from the *GENERATOR-INTERFACE* to the *GENERATOR-WRAPPER*. Basically, the constant operator `initState` of the *APMAUDE* module is specified. The variables

```
vars ZKA GMA GA CA : Address .
var C : Config .
```

are used in the equation

```
eq initState =
  GKMInit(0, 1, 0.0)
  ZKInit(1, 0.0)
  < 2 : Generator | count: 0, gkm: 0, config: < 2 . 0 > >
  [generator-spawn-period, 2 <- spawn]
  { 0.0 | nil} .
```

to define the initial state of the configuration, which consists of the group-key management server, the ZooKeeper service, a generator for the clients and the top-level scheduler. Additionally, the periodic generation of clients is started by sending a message of the form `2 <- spawn` to the generator.

## 6.4. Statistical Analysis of the Group-Key Management Service

In this section, we analyze the specification of the group-key management service built on top of the ZooKeeper by statistically model checking appropriate qualitative properties expressed as QuaTEx formulas using PVeStA. We analyzed the following two properties of the system:

1. The success ratio of the group-key management service, i.e., how many of the key updates the group-key management server performs actually arrive at the clients?

2. The average latency of the group-key management service, i.e., if a key update arrives at a client, how much time has passed on average since it was sent?

We evaluate these properties under an increasing number of participating clients of the group-key management system. In order to analyze these two properties, the model has to be instrumented so that the properties can be equationally computed from a configuration of the system. Thus, we introduce the flexible concept of an *analyzer*, which is an actor that receives status information from the participating entities. In particular, we introduce the module *GKM-ANALYSIS* and *GKM-CLIENT-ANALYSIS*, which specify actors that receive status messages from the group-key management server and the clients. Figure 6.13 shows how the two system modules are integrated in the module structure of the specification of the group-key management.

For the sake of brevity, the two modules are not described here in detail. Basically, both modules define a set of status messages that are sent by the group-key management server and client if specific actions happen:

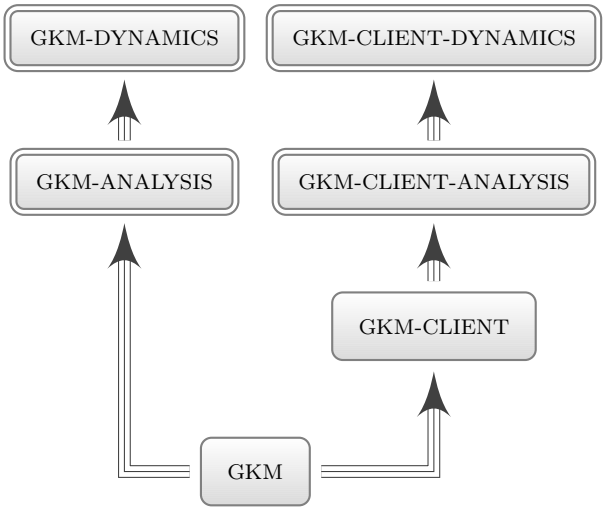



**Figure 6.13.:** Modified Maude module structure of the group-key management

- If a server updates the shared key or the key for the client with address `C` at the global time `t` with the value `V`, the server sends a message of the form `A <- groupKeyUpdated(V , t)` or `A <- clientKeyUpdated(C, V, t)` to the server analyzer with address `A`.
- Similarly, if a client with address `C` receives an updated key with value `V` at global time `t`, the client sends a message of the form `A <- clientKeyReceived(C, V, t)` or `A <- groupKeyReceived(C, V, t)` to the client analyzer with address `A`.
- Additionally, if a client joins or leaves the group at time t, it sends a message of the form `A <- clientConnected(C, t)` or `A <- clientDisconnected(C, t)` to the client analyzer.

The messages are stored at the analyzers for later analysis.

### 6.4.1. QuaTEx Formulas

The following QuaTEx formulas define the quantitative properties that we want to analyze. The function *time*() denotes a state function that returns the global time in the current configuration.

**Success ratio.** The success ratio defines the percentage of key updates that actually arrive at the clients.

$$\begin{aligned} successRatio(t) = \; & \textbf{\textit{if}}\ time() > t \ \ \textbf{\textit{then}} \\ & \frac{countKeysArrivedAtClients()}{countKeysSentToClients()} \\ & \textbf{\textit{else}}\ \bigcirc\,(successRatio(t)) \end{aligned}$$

with *countKeysArrivedAtClients*() being the result of equationally counting the key updates that arrived at the clients and the state function *countKeysSentToClients*() returns the result of equationally counting the total number of key updates that have

been performed by the server. The server and the client analyzer are used to count those key updates.

**Average latency.** The average latency is the average time it takes for a key update to arrive at the client (if it arrives).

$$avgLatency(t) = \textbf{\textit{if}}\ time() > t\ \ \textbf{\textit{then}}\ \frac{sumLatency()}{countKeysArrivedAtClients()}\ \textbf{\textit{else}}\ \bigcirc(avgLatency(t))$$

with *sumLatency*() being the result of summing up the time between a key update is sent and the key is arrived at the client and the state function *countKeysArrivedAtClients*() returns the result as described above.

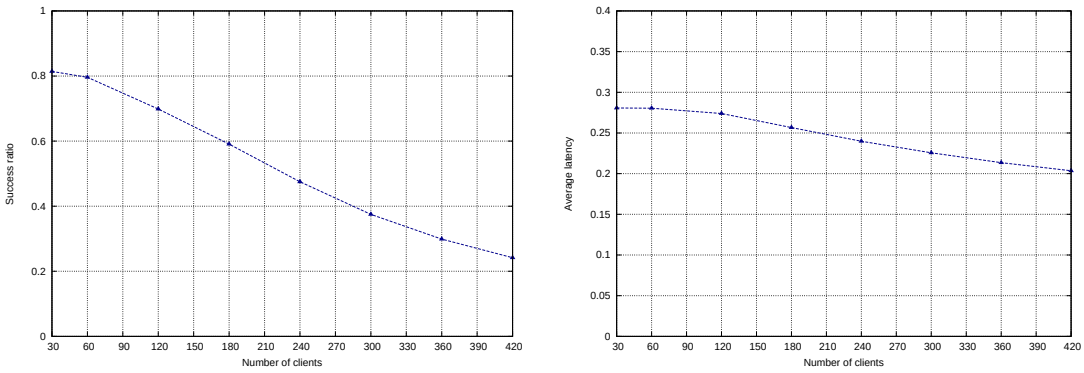
For the statistical analysis of the above properties we fix the processing time of a request and a watching request for a ZooKeeper node to 10 ms. Additionally, the time it takes for a ZooKeeper node to create a notification is also set to 10 ms. The individual ZooKeeper nodes' failure probability is set to 1 % every second. The time it takes for a ZooKeeper node to recover is fixed to two seconds. The period for the periodic key updates is set to two seconds, while the time a client is connected to the group-key management service is randomly chosen between five and 20 seconds. Lastly, the time frame in which we measured the properties is 30 seconds. The two properties are checked for an increasing number of clients, represented by the generator's spawn period: 1.0, $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{6}$, $\frac{1}{8}$, $\frac{1}{10}$, $\frac{1}{12}$ and $\frac{1}{14}$ which corresponds to the total number of clients: 30, 60, 120, 180, 240, 300, 360 and 420[7].

Figure 6.14 illustrates the results of the statistical analysis. The results of the statistical analysis of the success ratio are shown in Figure 6.14(a). As one can see, as more clients are present in the system, the success ratio goes down. With 30 clients in the system approximately 80 % of the key updates arrive at the clients. But as the number of clients increases, the success ratio decreases, until it reaches just above 21 %. In contrast to this, the results of the statistical analysis of the average latency slightly improve as more clients join the system. The results are shown in Figure 6.14(b): Initially, with 30 clients in the system, the average latency is approximately 280 ms, which improves to approximately 200 ms with 420 clients in the system. In the latter statistical analysis, the average latency is only computed for key updates that actually arrive at the client. Since the success ratio dramatically goes down with more clients in the system, the improving value can be explained.

### 6.4.2. Discussion and Analysis of the Results

The quantitative analysis of the success ratio uncovers serious issues about the group-key management service's design on top of ZooKeeper. A key property of a group-key management system is the freshness of the keys, i.e., key updates need to arrive at the members of the group. The results show that the freshness property of the group-key management

[7]The parameters of the ZooKeeper service and the group-key management service can be adjusted in the modules *ZK-PARAMS* and *GK-PARAMS*




(a) Success ratio (b) Average latency

**Figure 6.14.:** Results of the statistical analysis

system is only poorly satisfied. In a system with 420 clients just about 21% of the key updates arrive at the clients.

Initially, the idea of using the ZooKeeper service's notification mechanism for the distribution of the key updates seamed reasonable. But there are at least two main shortcomings with this approach.

**One-time watches.** The watches in the ZooKeeper service are removed after being triggered. Thus, in order to be continuously notified, users of the ZooKeeper service need to reset a watch after it has been triggered. This has one major disadvantage: In the timeframe between the update of an id and the arrival of the re-sent watching request for that id, the client is not notified about key updates of that id.

**Watches are not replicated.** The watches in the ZooKeeper service are local to the node a client is connected to and are not replicated across the nodes. This design decision is made since it "allows watches to be light weight to set, maintain, and dispatch". The major disadvantage of this approach is that all watches kept by a ZooKeeper node are lost when such a node goes down. In the group-key management service, this results in clients that will no longer receive key updates.

We propose two solutions to this problem:

- The duration of a watch in the ZooKeeper service needs to be changed from one-time to continuous, which would make the necessity of a client having to re-send the watch obsolete.
- Watches need to be replicated across all ZooKeeper nodes, similar to the replication of the id-value store.

The timeframe between the update of an id and the arrival of the re-sent watching request is no longer relevant if continuous watches are used. Additionally, if watches are replicated across all ZooKeeper nodes, a single node failure will not lower the quality of the service. Nevertheless, if more than the (strict) majority of ZooKeeper nodes are down, the clients would not receive key updates but the ZooKeeper service would not work either.

The specification of this redesigned system and the verification of its properties are left for future work.

## 6.5. Related Work

Katelman et al. use a similar approach in [52]. They perform a formal statistical analysis of the local minimum spanning tree (LMST) topology control protocol under realistic conditions and show that an important invariant is easily lost. As a result, they propose a redesign of the protocol with which the correct system design is reached.

The combination of PMaude together with (P)VeSta is used in multiple case studies, including the analysis of a Cloud-based Publish-Subscribe middleware [67], the analysis of a DoS resistant client-server request-response protocol [20] and two case studies on distributed object-based stochastic hybrid systems [61].

In [58], Menezes et al. define Group Key Management as the set of techniques and procedures supporting the establishment and maintenance of keying relationships between authorized parties that form a group. Harney and Muckenhirn describe the specification and architecture for a **G**roup **K**ey Management **P**rotocol (GKMP) in [44, 45] which provides the means to create and distribute keys within arbitrary-sized groups.

## 6.6. Conclusion

In this chapter we have formally modelled a centralized key-management service that uses a distributed notification mechanism for key updates. The specifications are based on the modularized actor model, is well-suited for modelling distributed and parallel systems.

In addition, we have shown that the specification of the centralized key-management service on top of ZooKeeper can be formally analyzed. We conducted statistical analyses of basic properties of the system, and, based on our results, have uncovered serious design issues in the proposed architecture of the group-key management service. Furthermore, we have proposed a solution to solve these issues involving only minimal changes to the original design.

# Chapter 7

# Conclusion & Future Work

## 7.1. Conclusion

In this thesis we have tackled the complexity of designing and analyzing secure and safe Cloud-based services. We have presented an general approach that is based on the rigorous formal semantics of rewriting logic together with the analytic power of (statistical) model-checking. Using this approach (i) models of Cloud-based services can be specified, (ii) security and safety properties of such services can be expressed, and (iii) quantitative as well as qualitative analyses of such properties can be performed. Additionally, we have shown that this approach can be applied to different Cloud-based systems and interesting properties can be analyzed.

First, we have specified formal languages, *-KLAIM, in which Cloud Computing architectures can be specified and analyzed. As an example of the expressiveness of *-KLAIM, we demonstrated how a token-based mutual exclusion algorithm can be specified and analyzed in *-KLAIM. We have analyzed the mutual exclusion property and strong liveness guarantees in this specification using the Maude LTL model checker and the Maude search command. We were able to show that Cloud-based systems can be designed and basic properties of the system can be analyzed using *-KLAIM. However, the use of exact model-checking algorithms puts a spoke in our wheel: The state-explosion problem puts a hard limitation on the size of the specified system.

In the second part of this work, we challenged this limitation: We developed a modularization of the actor model of computation that can be used for specifying modular Cloud-based systems. The use of the modularized actor model enabled us to perform quantitative analyses in hierarchical and modular systems by using the statistical model checker PVeStA. The use of statistical analyses instead of exact analyses made it possible for us to overcome the limitations on the size of the specified systems.

We were able to show with two case studies that even large systems can be specified and quantitatively analyzed. In the first case study we have specified the ASV protocol, a defense mechanism against DDoS attacks, and analyzed quantitative properties like for example the availability of a server under attack. In contrast to the small systems, we were analyzing in the first part of the thesis, we have analyzed a client-server system with up to 200 attackers that emitted 40 messages per second. Based on the results of the statistical analyses, we extended the specification by leveraging the scalability of the Cloud. We were able to show that the properties like, for example, the availability stays close to stable — even under heavy attack — in the extended specification. In the second case study we specified a group-key management system, that uses the notification mechanism of the so-called ZooKeeper service for the distribution of key updates. Based on a statistical analysis we uncovered serious design issues in the proposed architecture.

## 7.2. Future Work

We propose as one main result a formal approach for designing secure and safe Cloud-based applications: (i) Develop executable formal models of designs using the modularized actor model in rewriting logic, then (ii) perform statistical analyses using the PVeStA model-checker, and (iii) based on the results of the analyses, improve the original specification to tackle shortcomings of the original. We suggest to apply this approach on other Cloud-based systems to formally analyse interesting properties. Additionally, reoccurring pattern, like for example the ASV wrapper or the server replicator, may be identified and put on a more abstract level such that the pattern can be reused.

Both, the ASV wrapper and the server replicator require that the underlying system is a client-server request-reply system. The concept of protecting a service against denial of service attack through replication and the ASV protocol does not necessarily require this restriction. This can be examined in future work.

In the Chapter 6 we have uncovered serious design issues in the proposed architecture of the Group-Key management system. This is an interesting topic to work on, since we already uncovered design issues and now can improve the existing architecture.

# Appendices

# KLAIM Appendix

$$(1)\ \frac{P \xrightarrow[\rho']{s(t)@l} P' \qquad s = \rho' \bullet \rho(l) \qquad et = \mathcal{T}[\![t]\!]_{\rho' \bullet \rho}}{s ::_\rho P \rightarrowtail s ::_\rho P' \mid out(et)}$$

$$(2)\ \frac{P_1 \xrightarrow[\rho]{s(t)@l} P_1' \qquad s_2 = \rho \bullet \rho_1(l) \qquad et = \mathcal{T}[\![t]\!]_{\rho \bullet \rho_1}}{s_1 ::_{\rho_1} P_1 \parallel s_2 ::_{\rho_2} P_2 \rightarrowtail s_1 ::_{\rho_1} P_1' \parallel s_2 ::_{\rho_2} P_2 \mid out(et)}$$

$$(3)\ \frac{P \xrightarrow[\rho']{e(Q)@l} P' \qquad s = \rho' \bullet \rho(l)}{s ::_\rho P \rightarrowtail s ::_\rho Q \mid P'}$$

$$(4)\ \frac{P_1 \xrightarrow[\rho]{e(Q)@l} P_1' \qquad s_2 = \rho \bullet \rho_1(l)}{s_1 ::_{\rho_1} P_1 \parallel s_2 ::_{\rho_2} P_2 \rightarrowtail s_1 ::_{\rho_1} P_1' \parallel s_2 ::_{\rho_2} Q \mid P_2}$$

$$(5)\ \frac{P_1 \xrightarrow[\rho']{i(t)@l} P_1' \qquad s = \rho' \bullet \rho(l) \qquad P_2 \xrightarrow[\phi]{o(et)@self} P_2' \qquad match(\mathcal{T}[\![t]\!]_{\rho' \bullet \rho}, et)}{s ::_\rho P_1 \mid P_2 \rightarrowtail s ::_\rho P_1'[et/\mathcal{T}[\![t]\!]_{\rho \bullet \rho_1}] \mid P_2'}$$

$$(6)\ \frac{P_1 \xrightarrow[\rho]{i(t)@l} P_1' \qquad s_2 = \rho \bullet \rho_1(l) \qquad P_2 \xrightarrow[\phi]{o(et)@self} P_2' \qquad match(\mathcal{T}[\![t]\!]_{\rho \bullet \rho_1}, et)}{s_1 ::_{\rho_1} P_1 \parallel s_2 ::_{\rho_2} P_2 \rightarrowtail s_1 ::_{\rho_1} P_1'[et/\mathcal{T}[\![t]\!]_{\rho \bullet \rho_1}] \parallel s_2 ::_{\rho_2} P_2'}$$

$$(7)\ \frac{P_1 \xrightarrow[\rho']{r(t)@l} P_1' \qquad s = \rho' \bullet \rho(l) \qquad P_2 \xrightarrow[\phi]{o(et)@self} P_2' \qquad match(\mathcal{T}[\![t]\!]_{\rho' \bullet \rho}, et)}{s ::_\rho P_1 \mid P_2 \rightarrowtail s ::_\rho P_1'[et/\mathcal{T}[\![t]\!]_{\rho \bullet \rho_1}] \mid P_2}$$

$$(8)\ \frac{P_1 \xrightarrow[\rho]{r(t)@l} P_1' \qquad s_2 = \rho \bullet \rho_1(l) \qquad P_2 \xrightarrow[\phi]{o(et)@self} P_2' \qquad match(\mathcal{T}[\![t]\!]_{\rho \bullet \rho_1}, et)}{s_1 ::_{\rho_1} P_1 \parallel s_2 ::_{\rho_2} P_2 \rightarrowtail s_1 ::_{\rho_1} P_1'[et/\mathcal{T}[\![t]\!]_{\rho \bullet \rho_1}] \parallel s_2 ::_{\rho_2} P_2}$$

$$(9)\ \frac{s ::_\rho P_1 \rightarrowtail s ::_\rho P_1'}{s ::_\rho P_1 \mid P_2 \rightarrowtail s ::_\rho P_1' \mid P_2}$$

$$(10)\ \frac{P \xrightarrow[\rho']{n(u))@self} P' \qquad s' \neq s}{s ::_\rho P \rightarrowtail s ::_\rho P'[s'/u] \parallel s' ::_{[s'/self] \bullet \rho} nil}$$

$$(11)\ \frac{N_1 \rightarrowtail N_1' \qquad st(N_1') \cap st(N_2) = \emptyset}{N_1 \parallel N_2 \rightarrowtail N_1' \parallel N_2}$$

$$(12)\ \frac{N \equiv N_1 \qquad N_1 \rightarrowtail N_2' \qquad N_2 \equiv N'}{N \rightarrowtail N'}$$

**Figure A.1.:** KLAIM's reduction relation

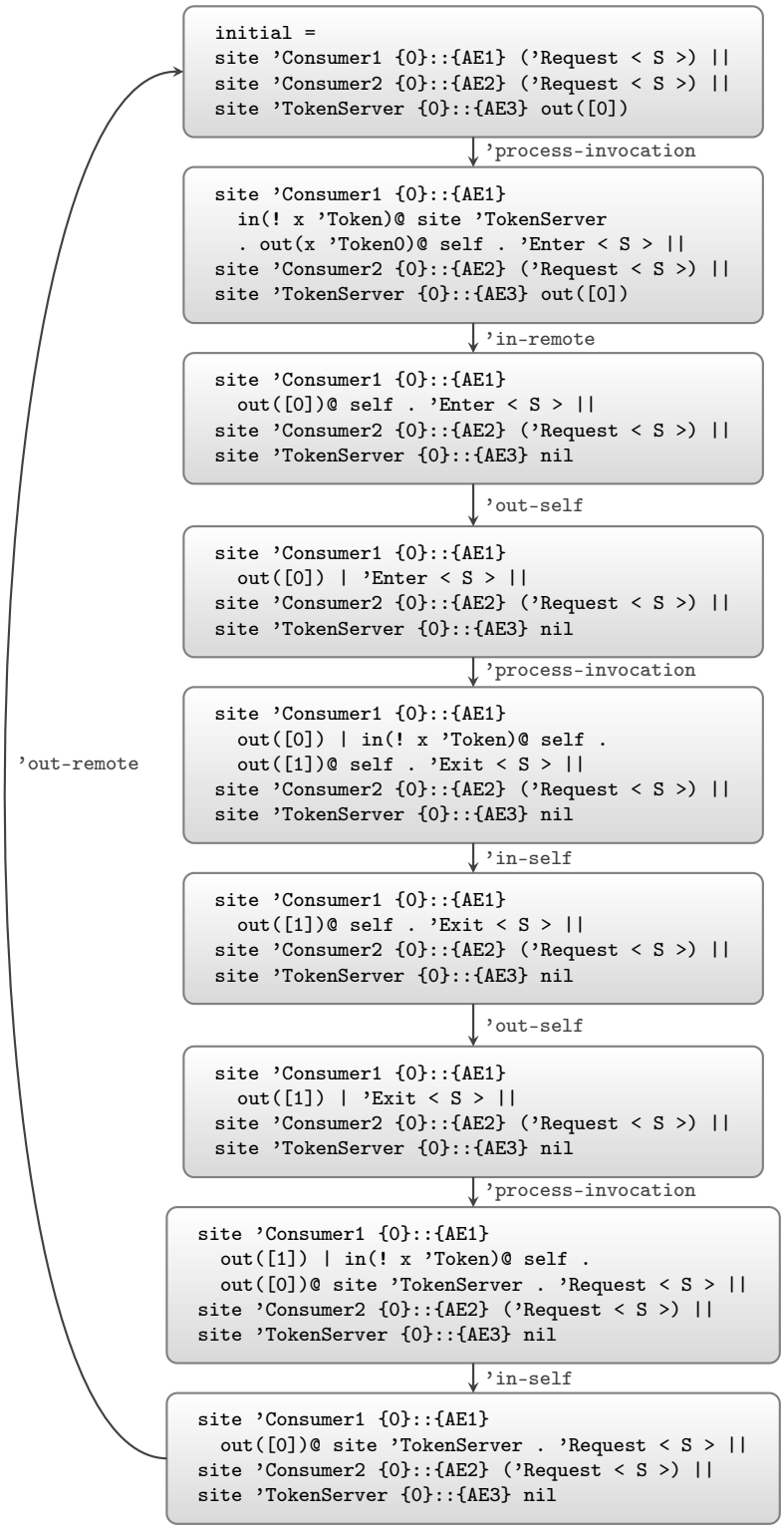



**Figure A.2.:** Counterexample for the liveness of `consumer2`
(`AE1 := [site 'Consumer1 / self] * [site 'TokenServer / 'TokenServer]`,
`AE2 := [site 'Consumer2 / self] * [site 'TokenServer / 'TokenServer]`,
`AE3 := [site 'TokenServer / self]`,
`S := nilProcessSeq,nilLocalitySeq,nilExpressionSeq`)

# Additional Maude Specifications

## B.1. The *SAMPLER* module

The module *SAMPLER* protects the predefined Maude modules *RANDOM* and *COUNTER*. It provides operators which return random values according to various probability distributions.

In the following, the variables

```
vars MAXNAT N RANK RND : Nat .
vars MIN MAX R MEAN STD-DEVIATION ALPHA F FREQ DICE : Float .
vars A B : Float .
```

are used.

The operators

```
op rand : -> [Float] .
eq rand = float(random(counter) / 4294967296) .
```

and

```
op rrand : -> [Rat] .
eq rrand = random(counter) / 4294967296 .
```

are used to create random floating point and rational numbers in the range $(0, 1]$[1].

A Bernoulli-distributed random boolean value is generated by the operator

```
op sampleBerWithP : Float -> [Bool] .
eq sampleBerWithP(R) = rand < R .
```

which takes a success probability as an argument.

Uniformly distributed natural numbers are generated by the operator

```
op sampleUniWithInt : Nat -> [Nat] .
eq sampleUniWithInt(MAXNAT) = floor(rrand * MAXNAT) .
```

which takes a maximum value (`MAXNAT`) as an argument. The returned natural number is in the range [0,`MAXNAT`].

Random floating point values between an upper (`MAX`, inclusive) and a lower (`MIN`, exclusive) bound are generated by the operator

[1] $4294967296 = 2^{32}$. The operator `random` of the built-in Maude module *RANDOM* returns terms of the sort `Nat` that are in the range $[0, 2^{32} - 1]$.

```
op genRandom : Float Float -> [Float] .
eq genRandom(MIN, MAX) = rand * (MAX - MIN) + MIN .
```

Random values according to a normal distribution are generated using the Box-Muller method [30]. The method generates (pairs of) independent standard normally distributed random numbers from a source of uniformly distributed random numbers. The operator

```
eq boxMullerValue(MEAN, STD-DEVIATION) =
  MEAN + STD-DEVIATION * sqrt(-2.0 * log(genRandom(0.0, 1.0)))
    * cos(2.0 * pi * genRandom(0.0, 1.0)) .
```

takes a mean (`MEAN`) and a standard deviation (`STD-DEVIATION`) as arguments and returns a random number according to $\mathcal{N}$(`MEAN`,`STD-DEVIATION`$^2$). A pair of normally distributed values can be generated more efficiently than a single value using the Box-Muller method. The operator `boxMullerPair`

```
sort Pair .
op {_,_} : Float Float -> Pair [ctor] .
op boxMullerPair : Float Float -> [Pair] .
op boxMullerExpr : Float [Float] [Float] -> [Pair] .
eq boxMullerExpr(MEAN, A, B) =
  { MEAN + A * cos(B), MEAN + A * sin(B) } .
eq boxMullerPair(MEAN, STD-DEVIATION) =
  boxMullerExpr(MEAN, STD-DEVIATION
    * sqrt(-2.0 * log(genRandom(0.0, 1.0))),
    2.0 * pi * genRandom(0.0, 1.0)) .
```

can be used to create a pair (term of sort `Pair`) of normally distributed random values according to $\mathcal{N}$(`MEAN`,`STD-DEVIATION`$^2$). Just as `boxMullerValue`, the operator takes a mean (`MEAN` ) and a standard deviation (`STD-DEVIATION`) as arguments.

Random Pareto values are generated by the operator

```
op paretoValue : -> [Float] .
eq paretoValue = (1.0 - genRandom(0.0, 1.0)) ^ -1.0 .
```

which returns a floating point number in the range $(1, \infty)$.

Random values according to a Zipf distribution with parameters `skew` and `maxZipf` can be generated using the operator

```
op zipfValue : -> [Nat] .
```

The parameters of the distribution are defined by the operators and equations

```
op maxZipf : -> Nat .
eq maxZipf = 1000 .

op skew : -> Float .
eq skew = 1.0 .
```

The helper functions

```
op bottom : -> Float [memo] .
op bottomRec : Nat Float -> Float .
eq bottom = bottomRec(1, 0.0) .
eq bottomRec(N, F) =
  if N <= maxZipf then
    bottomRec(s(N), F + 1.0 / (float(N) ^ skew))
  else
```

```
    F
  fi .

op probability : Nat -> [Float] [memo] .
eq probability(N) = (1.0 / (float(N) ^ skew)) / bottom .

op genRecExpression : Nat Float -> [Nat] .
eq genRecExpression(N, F) = zipfValueRec(N, probability(s(N)), F) .

op zipfValueRec : Nat Float Float -> [Nat] .
eq zipfValueRec(RANK, FREQ, DICE) =
  if DICE >= FREQ then
    genRecExpression(sampleUniWithInt(maxZipf), genRandom(0.0, 1.0))
  else
    s(RANK)
  fi .
eq zipfValueRec(0, 0.0, 0.0) =
  genRecExpression(sampleUniWithInt(maxZipf), genRandom(0.0, 1.0)) .
```

are used by the `zipfValue` operator. Finally, the equation

```
eq zipfValue = zipfValueRec(0, 0.0, 0.0) .
```

defines the value generating operator.

It is of note that all operators in the *SAMPLER* module do not have a sorts but the respective kind. This is due to the fact that all operators are based on calls to `random(counter)` and only a rewrite substitutes a random term of sort `Nat` in the range $[0, 2^{32} - 1]$ for the term `random(counter)` of kind `[Nat]`.

The operators presented in this Section are summarized in Table B.1.

| Operator | Argument(s) | Result |
|---|---|---|
| `rand` | - | `[Float]` $\in (0, 1]$ |
| `rrand` | - | `[Rat]` $\in (0, 1]$ |
| `sampleBerWithP` | success probability (`Float`) | `[Bool]` $\in \{$`true`, `false`$\}$ |
| `genRandom` | minimum (`MIN : Float`) | `[Float]` $\in$ (`MIN`,`MAX`] |
| | maximum(`MAX : Float`) | |
| `boxMullerValue` | mean (`Float`) | [Float] |
| | standard deviation (`Float`) | |
| `boxMullerPair` | mean (`Float`) | [Pair] |
| | standard deviation (`Float`) | |
| `paretoValue` | - | `[Float]` $\in (0, \infty)$ |
| `zipfValue` | -[2] | `[Nat]` $\in$ [1,`maxZipf`] |

**Figure B.1.:** Operators to generate random values

[2]Paramters are indirectly set by the equations that define the constant operators `maxZipf` and `skew`.

## B.2. Maude Specification of a Generic Actor Generator

The generic actor generator is specified as a Russian dolls actor that periodically creates and initializes new actors. The generated actors are kept within the generator's configuration, so that messages that are addressed to the internal actors and messages that are addressed to the outside are forwarded.

Since the generator generically wraps an actor, the theory

```
th GENERATOR-INTERFACE is
  pr FLOAT .
  pr ACTOR-MODEL .

  op generator-spawn-period : -> Float .
  op generator-create : AttributeSet Float Address -> Config .
endth
```

needs to be implemented. The operator `generator-spawn-period` specifies the period for generating new actors. The operator `generator-create` takes the attributes of the generator, the current global time, and a new address as arguments and returns the actor together with the messages needed for the initialization.

The system module *GENERATOR* specifies the behavior of the generator. It is generic with respect to the theory *GENERATOR-INTERFACE*. The actor type

```
op Generator : -> ActorType .
```

specifies the type of the generator. The internal state of the generator is represented by the attribute

```
op count:_ : Nat -> Attribute [gather(&)] .
```

which counts the generated actors. The generator periodically sends a message with the contents

```
op spawn : -> Contents .
```

to itself to trigger the generation of a new actor. The following rewrite rules make use of the variables

```
var A A' : Address .
var NG : NameGenerator .
var N : Nat .
var C : Config .
var gt : Float .
var AS : AttributeSet .
var CO : Contents .
```

The rewrite rule

```
rl [GENERATOR-SPAWN] :
  < A : Generator | count: N, config: C NG, AS >
  {gt, A <- spawn}
=>
  < A : Generator | count: s(N),
    config: generator-create(AS, gt, NG .new) C NG .next, AS >
    [gt + generator-spawn-period, A <- spawn] .
```

periodically generates a new actor by calling the operator `generator-create`. Finally, the rewrite rules

```
crl [GENERATOR-PASS-DOWN] :
  < A : Generator | config: C, AS >
  {gt, A . A' <- CO}
=>
  < A : Generator | config: [gt, A . A' <- CO] C, AS >
if
  CO =/= spawn .

crl [GENERATOR-PASS-UP1] :
  < A : Generator | config: {gt, A' <- CO } C, AS >
=>
  < A : Generator | config: C, AS >
  [gt, A' <- CO]
if
  | A' | <= | A | .

crl [GENERATOR-PASS-UP1] :
  < A : Generator | config: {gt, A' <- CO } C, AS >
=>
  < A : Generator | config: C, AS >
  [gt, A' <- CO]
if
  | A' | > | A | /\ prefix(A', | A |) =/= A .
```

forward messages into the generator's configuration and to the outside. The first rewrite rule consumes messages whose receiver's address is prefixed by the generator's address. This is the case if the receiver of the message is a child in the generator's address tree. The second and third rewrite rules consume a message from within the generator's configuration and pass it to the outside. The second rewrite rule consumes messages that are addressed to an actor that is located at a higher level in the address hierarchy. The last rewrite rule consumes messages whose receiver is located in another subtree of the address hierarchy.

# Bibliography


[1] Amazon Elastic Compute Cloud (Amazon EC2). See: http://aws.amazon.com/ec2/ (Visited: Sept., 2011).

[2] Flexiscale — Utility Computing on Demand. See: http://www.flexiscale.com/ (Visited: Sept., 2011).

[3] Force.com. See: http://www.force.com/ (Visited: Sept., 2011).

[4] GoGrid. See: http://www.gogrid.com/ (Visited: Sept., 2011).

[5] Google App Engine. See: http://code.google.com/appengine/ (Visited: Sept., 2011).

[6] Google Apps for Business. See: www.google.com/a/ (Visited: Sept., 2011).

[7] Kernal Based Virtual Machine. See: http://www.linux-kvm.org/ (Visited: Sept., 2011).

[8] Rackspace Hosting. See: http://www.rackspace.com/ (Visited: Sept., 2011).

[9] Salesforce.com. See: http://www.salesforce.com/ (Visited: Sept., 2011).

[10] SAP Business ByDesign. See: http://www.sap.com/solutions/products/sap-bydesign/index.epx (Visited: Sept., 2011).

[11] Twitter. See: http://twitter.com/ (Visited: Sept., 2011).

[12] VMWare ESX Server. See: www.vmware.com/products/esx (Visited: Sept., 2011).

[13] Windows Azure Platform. See: http://www.microsoft.com/windowsazure/ (Visited: Sept., 2011).

[14] XenSource Inc, Xen. See: http://www.xensource.com/ (Visited: Sept., 2011).

[15] G. Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.

[16] G. Agha, J. Meseguer, and K. Sen. Pmaude: Rewrite-based specification language for probabilistic object systems. *Electron. Notes Theor. Comput. Sci.*, 153:213–239, May 2006.

[17] M. AlTurki. *Rewriting-based formal modeling, analysis and implemenation of real-time distributed services*. PhD thesis, University of Illinois in Urabana-Champaign, 2011. `http://hdl.handle.net/2142/26231`.

[18] M. AlTurki and J. Meseguer. PVeStA: A Parallel Statistical Model Checking and Quantitative Analysis Tool. In *Algebra and Coalgebra in Computer Science*, volume 6859 of *Lecture Notes in Computer Science*, pages 386–392. Springer Berlin / Heidelberg, 2011.

[19] M. AlTurki, J. Meseguer, and C. A. Gunter. Probabilistic modeling and analysis of dos protection for the asv protocol. *Electron. Notes Theor. Comput. Sci.*, 234:3–18, March 2009.

[20] M. AlTurki, J. Meseguer, and C. A. Gunter. Probabilistic Modeling and Analysis of DoS Protection for the ASV Protocol. *Electronic Notes in Theoretical Computer Science*, 234:3–18, 2009.

[21] Amazon. Amazon EC2 Service Level Agreement. See: `http://aws.amazon.com/ec2-sla/` (Visited: Sept., 2011).

[22] Arash Ferdowsi. Yesterday's Authentication Bug. `http://blog.dropbox.com/?p=821` (Visited: Sept., 2011).

[23] A. Aziz, V. Singhal, F. Balarin, R. Brayton, and A. Sangiovanni-Vincentelli. It usually works: The temporal logic of stochastic systems. In P. Wolper, editor, *Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 155–165. Springer Berlin / Heidelberg, 1995.

[24] C. Baier, J.-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time markov chains. In *Proceedings of the 10th International Conference on Concurrency Theory*, CONCUR '99, pages 146–161, London, UK, 1999. Springer-Verlag.

[25] M. Baugher, T. Hardjono, H. Harney, and B. Weis. Group domain of interpretation for ISAKMP, 2001. See: `http://search.ietf.org/internet-drafts/draft-irtf-smug-gdoi-01.tx`
(Visited: Sep, 2011).

[26] BBC. Google takes on Facebook and Twitter with network site. `http://news.bbc.co.uk/2/hi/8506148.stm` (Visited: Sept., 2011).

[27] K. Berkling. A Symmetric Complement to the Lambda Calculus. Technical report, GMD, September 1976. ISF-76-7.

[28] K. Berkling and E. Fehr. A Consistent Extension of the Lambda Calculus as a Base for Functional Programming Languages. *Information and Control*, 55(1-3):89–101, 1982.

[29] R. Bobba and J. Gupta. Group-Key Management using the ZooKeeper Service — A Java Implementation. *TBA*, 2011.

[30] G. E. P. Box and M. E. Muller. A Note on the Generation of Random Normal Deviates. *The Annals of Mathematical Statistics*, 29(2):610–611, 1958.

[31] C. Braga and J. Meseguer. Modular Rewriting Semantics in Practice. *Electronic Notes in Theoretical Computer Science*, 117:393–416, Jan. 2005.

[32] R. Canetti, J. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast security: A taxonomy and some efficient constructions. In *Procceedings of INFOCOM'99*, volume 2, pages 708–716, March 1999.

[33] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, 2007.

[34] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude – A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.

[35] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Proc. Kninkl. Nederl. Akademie van Wetenschappen*, 75(5):381–392, 1972.

[36] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[37] R. Drucker and A. Frank. A C++/Linda Model for Distributed Objects. *iccsse*, page 30, 1996.

[38] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan. 1985.

[39] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96, Feb. 1992.

[40] Google. Google Apps Service Level Agreement. See: `http://www.google.com/apps/intl/en/terms/sla.html` (Visited: Sept., 2011).

[41] C. Gunter, S. Khanna, K. Tan, and S. Venkatesh. DoS Protection for Reliably Authenticated Broadcast. In *NDSS*, 2004.

[42] J. Gupta. Group-Key Management using the ZooKeeper Service — A Java Implementation. Master's thesis, University of Illinois in Urbana-Champaign, 2011 TBA.

[43] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6:512–535, 1994. 10.1007/BF01211866.

[44] H. Harney and C. Muckenhirn. Group Key Management Protocol (GKMP) Architecture. *IEFT draft gkmp-arch*, June 2006.

[45] H. Harney and C. Muckenhirn. Group Key Management Protocol (GKMP) Specification. *IEFT draft gkmp-spec*, June 2006.

[46] J. Heiser and M. Nicolett. Assessing the Security Risks of Cloud Computing. *Gartner Inc.*, June 2008.

[47] C. Hewitt and H. G. Baker. Laws for communicating parallel processes. In *IFIP Congress*, pages 987–992, 1977.

[48] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.

[49] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21:666–677, August 1978.

[50] M. D. Hogan, F. Liu, A. W. Sokol, and T. Jin. NIST-SP 500-291, NIST Cloud Computing Standards Roadmap, August 2011. See: `http://www.nist.gov/manuscript-publication-search.cfm?pub_id=909024` (Visited: Sept., 2011).

[51] R. V. Hogg, A. Craig, and J. W. Mckean. *Introduction to Mathematical Statistics*. Prentice Hall, 6th edition, June 2004.

[52] M. Katelman, J. Meseguer, and J. C. Hou. Redesign of the lmst wireless sensor protocol through formal modeling and statistical model checking. In *FMOODS*, pages 150–169, 2008.

[53] S. Khanna, S. Venkatesh, O. Fatemieh, F. Khan, and C. Gunter. Adaptive Selective Verification. In *The 27th Conference on Computer Communications*, pages 529–537, 2008.

[54] N. Kumar, K. Sen, J. Meseguer, and G. Agha. Probabilistic rewrite theories: Unifying models, logics and tools. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, 2003.

[55] N. Kumar, K. Sen, J. Meseguer, and G. Agha. A rewriting based model for probabilistic distributed object systems. In *In Proceedings of 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'03), volume 2884 of Lecture Notes in Computer Science*, pages 32–46. Springer, 2003.

[56] MasterCard. MasterCard Statement. See: `http://www.businesswire.com/news/home/20101208005866/en/MasterCard-Statement`
(Visited: September, 2011).

[57] MasterCard. MasterCard Statement.
See: `http://www.businesswire.com/news/home/20101208006660/en/MasterCard-Statement`
(Visited: September, 2011).

[58] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[59] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *TCS*, 96(1):73–155, 1992.

[60] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[61] J. Meseguer and R. Sharykin. Specification and analysis of distributed object-based stochastic hybrid systems. In *HSCC*, pages 460–475, 2006.

[62] J. Meseguer and C. L. Talcott. Semantic models for distributed object reflection. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 1–36, London, UK, 2002. Springer-Verlag.

[63] Microsoft. Service Level Agreement for Microsoft Online Services. See: `http://microsoftvolumelicensing.com/DocumentSearch.aspx?Mode=3&DocumentTypeId=37` (Visited: Sept., 2011).

[64] Microsoft. The STRIDE Threat Model. See: `http://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx` (Visited: September, 2011).

[65] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[66] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1st edition, June 1999.

[67] T. Mühlbauer. Formal Specification and Analysis of Cloud Computing Management. Master's thesis, Ludwig Maximilian University of Munich, 2011.

[68] R. D. Nicola, G. L. Ferrari, and R. Pugliese. Klaim: A kernel language for agents interaction and mobility. *IEEE Trans. Software Eng.*, 24(5):315–330, 1998.

[69] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *HOSC*, 20(1–2):161–196, 2007.

[70] Robert Mackey. 'Operation Payback' Attacks Target MasterCard and PayPal Sites to Avenge WikiLeaks. See: `http://thelede.blogs.nytimes.com/2010/12/08/operation-payback-targets-mastercard-and-paypal-sites-to-avenge-wikileaks/` (Visited: September, 2011).

[71] K. Sen, M. Viswanathan, and G. Agha. Statistical model checking of black-box probabilistic systems. In *Computer Aided Verification*, pages 202–215, 2004.

[72] K. Sen, M. Viswanathan, and G. Agha. On statistical model checking of stochastic systems. In K. Etessami and S. Rajamani, editors, *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 251–255. Springer Berlin / Heidelberg, 2005.

[73] K. Sen, M. Viswanathan, and G. Agha. VESTA: A Statistical Model-checker and Analyzer for Probabilistic Systems. In *QEST 2005*, pages 251–252, 2005.

[74] T.-F. Serbanuta, G. Rosu, and J. Meseguer. A rewriting logic approach to operational semantics. *Inf. Comput.*, 207(2):305–340, 2009.

[75] M. Stehr. CINNI - A Generic Calculus of Explicit Substitutions and its Application to $\lambda$- $\varsigma$- and $\pi$-Calculi. *Electronic Notes in Theoretical Computer Science*, 36:70–92, 2000.

[76] E. Steven, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In F. Gadducci and U. Montanari, editors, *Fourth Workshop on Rewriting Logic and its Applications, WRLA '02*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[77] The Apache Software Foundation. Apache Zookeeper. See: `http://zookeeper.apache.org/`
(Visited: Sep, 2011).

[78] The Apache Software Foundation. Apache Zookeeper Wiki. See: `https://cwiki.apache.org/confluence/display/ZOOKEEPER/ProjectDescription`
(Visited: Sep, 2011).

[79] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds : Towards a cloud definition. *Computer Communication Review*, 39(1):50–55, 2009.

[80] a. Verdejo and N. Martioliet. Executable structural operational semantics in Maude. *Journal of Logic and Algebraic Programming*, 67(1-2):226–293, Apr. 2006.

[81] W3C. Request-Response Message Exchange Pattern.
See: `http://www.w3.org/TR/2003/PR-soap12-part2-20030507/#singlereqrespmep`
(Visited: September, 2011).

[82] G. Wells. Coordination languages: Back to the future with linda. *New Issues on Coordination and Adaptation*, pages 1–12, 2005.

[83] G. C. Wells, a. G. Chalmers, and P. G. Clayton. Linda implementations in Java for concurrent systems. *Concurrency and Computation: Practice and Experience*, 16(10):1005–1022, Aug. 2004.

[84] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, May 2010.



View publication stats