Universität Augsburg
University

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

TECHNISCHE UNIVERSITÄT MÜNCHEN

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

Masterarbeit im Elitestudiengang Software Engineering

Formal Specification and Analysis of Cloud Computing Management

Tobias Johann Mühlbauer

Software Engineering
Elite Graduate Program

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
24 JAN 2012		**00-00-2012 to 00-00-2012**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Formal Specification And Analysis Of Cloud Computing Management	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Institute for Computer Science ,Teaching and research unit for programming and software engineering ,Oettingenstr. 67. D-80538 Munich, Germany, ,**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Cloud Computing-based systems (i) are safety- and security-critical systems which have strong qualitative and quantitative formal requirements, (ii) have equally important timecritical performance-based quality of service properties (e.g., availability), and (iii) need to dynamically adapt to changes in the potentially hostile (e.g., distributed denial of service attacks) and often probabilistic environment they operate in. These aspects make distributed and Cloud-based systems complex and hard to design, build, test, and verify; and in this context, Cloud Computing management has to deal with a multitude of obstacles for the growth and adoption of the Cloud Computing paradigm. In this thesis, we focus on three of these obstacles: bugs in large distributed systems, service availability, and performance unpredictability. To tackle these challenges and the aforementioned complexity, we propose solutions based on executable formal specifications and formal analysis, using an adequate semantic framework. We chose rewriting logic as the semantic framework and Maude, a language and system based on rewriting logic that offers the possibility of executing and formally analyzing specifications, as the foundation for our work. The main contributions of this thesis are ? The specification of formal languages for the design and analysis of Cloud-based architectures. In particular, the rewriting logic-based specification of formal languages based on the coordination language and mobile calculus KLAIM. ? The specification of a modularized actor model of computation which incorporates the Russian Dolls model and fulfills the requirements for statistical model checking; thus allowing the specification of hierarchically structured distributed systems and their quantitative and qualitative formal analysis. ? The formal specification and formal analysis of the denial of service (DoS) defense mechanism ASV+SR, which is a combination of the DoS defense mechanism ASV and the Cloud-based resource provisioning mechanism SR. We show that ASV+SR provides stable availability at a reasonable cost; where stable availability means that with very high probability service quality remains very close to a threshold, regardless of how bad the DoS attack can get. ? The formal specification of a Publish/Subscribe system that is used to (a) answer the question of how a Publish/Subscribe architecture can be enriched with Cloud-based dynamic resource provisioning mechanisms to better meet quality of service (QoS) requirements and (b) show that predictions about QoS properties can be made using statistical analysis.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**236**	

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

Masterarbeit im Elitestudiengang Software Engineering

Formal Specification and Analysis of Cloud Computing Management

Tobias Johann Mühlbauer

Matrikelnummer:	1110475
Erstgutachter:	Prof. Dr. Martin Wirsing
Zweitgutachter:	Prof. Dr. Alexander Knapp
Betreuer:	Prof. Dr. José Meseguer
Abgabe:	24. Januar 2012

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

Tobias Johann Mühlbauer Augsburg, den 24. Januar 2012

Acknowledgements

This thesis would not have been possible without helpful guidance and advice. I want to express my gratitude to everyone involved. My special thanks go to

Prof. Dr. José Meseguer,
for all his contributions of time, ideas, and guidance without which this thesis would not have been possible. I want to thank Prof. Meseguer for offering me the opportunity to work on this thesis at the University of Illinois at Urbana-Champaign and for the productive, inspiring, and fun time we spent in the "Moraira research institute". It has been my great honor to be your student.

Prof. Dr. Martin Wirsing,
for offering me this great thesis opportunity and for all his guidance and support throughout. His academic experience was invaluable during the work on this thesis.

Prof. Dr. Alexander Knapp,
for being a great teacher. Without the knowledge gained through his lectures I would have lacked the fundamentals to work on this thesis.

Prof. Dr. Santiago Escobar,
for all the inspiring conversations and fun I enjoyed with him, especially during coffee breaks.

Prof. Musab Al-Turki, Ph.D.,
for all the great help he provided with the ASV protocol, statistical model checking, and the Maude system in general.

**Kyungmin Bae, Prof. Dr. Francisco Durán, Michael Katelman, Ph.D.,
Camilo Rocha, Dr. Raúl Gutiérrez, and Ralf Sasse,**
my colleagues and friends, not only for being supportive during the work on this thesis but also for the fun times we spent together.

Prof. Klara Nahrstedt, Ph.D., and Guijun Wang, Ph.D.,
for all the advice they provided to the work on the formal specification and analysis of Publish/Subscribe systems.

Denise and Ron Mazza,
for giving me the possibility to stay in beautiful Cardiff-by-the-Sea where parts of this thesis originated.

Anita, Johann, and Verena Mühlbauer,

my parents and my sister, for all their love and encouragement, and their support in all my pursuits. Thank you.

Lastly, my very special thanks go to **Jonas Eckhardt**, who has been a great colleague and friend over the past several months. The work in this thesis originated in collaboration with him and would not have been possible without his contributions of ideas and inspiration. Thank you for our sincere friendship and the fun time we spent together in the United States, in Spain, and throughout our studies in Germany.

This work was funded in part by NSF Grant CCF 09-05584, AFOSR Grant FA8750-11-2-0084, the EU-funded projects FP7-257414 ASCENS and FP7-256980 NESSoS, the PROSA$^{\text{LMU}}$ scholarship for research stays abroad, and the Software Engineering Elite Graduate Program.

Abstract

Cloud Computing-based systems (i) are safety- and security-critical systems which have strong qualitative and quantitative formal requirements, (ii) have equally important time-critical performance-based quality of service properties (e.g., availability), and (iii) need to dynamically adapt to changes in the potentially hostile (e.g., distributed denial of service attacks) and often probabilistic environment they operate in. These aspects make distributed and Cloud-based systems complex and hard to design, build, test, and verify; and in this context, Cloud Computing management has to deal with a multitude of obstacles for the growth and adoption of the Cloud Computing paradigm. In this thesis, we focus on three of these obstacles: bugs in large distributed systems, service availability, and performance unpredictability. To tackle these challenges and the aforementioned complexity, we propose solutions based on executable formal specifications and formal analysis, using an adequate semantic framework. We chose rewriting logic as the semantic framework and Maude, a language and system based on rewriting logic that offers the possibility of executing and formally analyzing specifications, as the foundation for our work.

The main contributions of this thesis are:

- The specification of formal languages for the design and analysis of Cloud-based architectures. In particular, the rewriting logic-based specification of formal languages based on the coordination language and mobile calculus KLAIM.
- The specification of a modularized actor model of computation which incorporates the Russian Dolls model and fulfills the requirements for statistical model checking; thus allowing the specification of hierarchically structured distributed systems and their quantitative and qualitative formal analysis.
- The formal specification and formal analysis of the denial of service (DoS) defense mechanism ASV$^+$SR, which is a combination of the DoS defense mechanism ASV and the Cloud-based resource provisioning mechanism SR. We show that ASV$^+$SR provides stable availability at a reasonable cost; where stable availability means that with very high probability service quality remains very close to a threshold, regardless of how bad the DoS attack can get.
- The formal specification of a Publish/Subscribe system that is used to (a) answer the question of how a Publish/Subscribe architecture can be enriched with Cloud-based dynamic resource provisioning mechanisms to better meet quality of service (QoS) requirements and (b) show that predictions about QoS properties can be made using statistical analysis.

Contents

Chapter 1
Introduction

Cloud-based systems are complex and hard to design, build, test, and verify; and in this context, Cloud Computing management has to deal with a multitude of obstacles for the growth and adoption of the Cloud Computing paradigm. It is our goal to provide solutions for the challenges of Cloud Computing management using formal specifications and formal analysis. This chapter outlines the challenges of Cloud Computing management, states the main contributions of this work, and, finally, gives an outline of the thesis.

1.1. Motivation

On June 20, 2011, 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."*
>
> — Arash Ferdowsi, Dropbox [18]

During these nearly four hours, the broken authentication mechanism granted access to possibly private data stored on some accounts using any chosen password.

In late 2010, a denial of service (DoS) attack targeted 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."*
>
> — Mastercard press release [72]

In fact, by that time, the DoS attack brought the website down and made the web presence unavailable for most costumers. 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."*
>
> — Mastercard press release [73]

The attack and the resulting downtime lasted for several hours.

The Dropbox and MasterCard incidents are just two out of many recent of Cloud-based system incidents; incidents that Cloud Computing management has to deal with. In fact, Cloud Computing-based systems (i) are safety- and security-critical systems which have strong qualitative and quantitative formal requirements, (ii) have equally important time-critical performance-based quality of service properties (e.g., availability), and (iii) need to dynamically adapt to changes in the potentially hostile (e.g., distributed denial of service attacks) and often probabilistic environment they operate in. These aspects make distributed and Cloud-based systems complex and hard to design, build, test, and verify. To tackle these challenges, the solutions in this thesis are based on executable formal specifications and formal analysis, using an adequate semantic framework. We chose rewriting logic [75] as the semantic framework and Maude [35], a language and system based on rewriting logic that offers the possibility of executing and formally analyzing specifications, as the foundation for the work in this thesis.

1.2. Challenges of Cloud Computing Management

To better understand what Cloud Computing Management means, we first need to define the meaning of management. One of the first comprehensive statements of a theory of management was given by Henri Fayol. In his work "Administration industrielle et générale" [46] he states that the five primary functions of management are:

1. to forecast and plan,
2. to organize,
3. to command,
4. to coordinate, and
5. to control.

While the term management is generally understood to be centered around people, in this work we also consider automatic management performed by machines which includes the broad category of systems with self-* properties such as, e.g., self-healing, self-scaling, and self-organiztion. Bringing the definitions of Cloud Computing and management together, Cloud Computing management means the application of the aforementioned management

	Obstacle	Opportunity
1	Availability of Service	Use Multiple Cloud Providers; Use Elasticity to Prevent DDOS
2	Data Lock-In	Standardize APIs; Compatible SW to enable Surge Computing
3	Data Confidentiality and Auditability	Deploy Encryption, VLANs, Firewalls; Geographical Data Storage
4	Data Transfer Bottlenecks	FedExing Disks; Data Backup/Archival; Higher BW Switches
5	Performance Unpredictability	Improved VM Support; Flash Memory; Gang Schedule VMs
6	Scalable Storage	Invent Scalable Store
7	Bugs in Large Distributed Systems	Invent Debugger that relies on Distributed VMs
8	Scaling Quickly	Invent Auto-Scaler that relies on ML; Snapshots for Conservation
9	Reputation Fate Sharing	Offer reputation-guarding services like those for email
10	Software Licensing	Pay-for-use licenses; Bulk use sales

Figure 1.1.: The top 10 obstacles for the growth and adoption of Cloud Computing according to the Berkeley view on Cloud Computing [19]

functions on the different layers of Cloud Computing: the infrastructure, the platform, and the service layer.

Table 1.1 outlines the top 10 obstacles for the growth and adoption of Cloud Computing [19]. Although all of the mentioned obstacles are in some way related to management-related tasks, in this thesis we focus on three of these obstacles: bugs in large distributed systems, service availability, and performance unpredictability.

1.3. Main contributions

In this thesis, we contribute solutions for the following Cloud Computing management-related obstacles:

Bugs in large distributed systems. As mentioned before, distributed and Cloud-based systems are complex and hard to design, build, test, and verify. This may result in bugs that are hard to think about and thus prevent during system development and are equally hard to find and fix in a running system. We propose a solution based on formal specification and formal analysis that can identify flaws during early stages of development. In particular, we contribute a formal executable specification and extensions of the coordination language and mobile calculus KLAIM [38]; and further show that these KLAIM-based executable formal languages can be used to formally specify and qualitatively formally analyze models of distributed and Cloud-based systems. Our second main contribution in this area is a Maude-based specification of a modularized actor model of computation which incorporates the Russian Dolls model and fulfills the requirements for statistical model checking. The rewriting-logic based Russian Dolls model [78] combines logical reflection and hierarchical structuring which allows the simple expression of more complex distributed systems. Statistical model checking allows the formal analysis of quantitative and qualitative properties; and can,

up to a certain level of statistical confidence, check larger system models than original model checking techniques.

Service availability. Service availability is crucial for many businesses and end-users to be able to migrate from locally run software to software services in the Cloud. Availability of Internet-based services can be compromised by distributed denial of service (DoS) attacks. DoS defense mechanisms help maintaining availability. However, even when equipped with defense mechanisms, systems will typically show performance degradation. Therefore, one goal is to achieve stable availability, which means that with very high probability service quality remains very close to a threshold, regardless of how bad the DoS attack can get. Another goal is to achieve stable availability at an economically reasonable cost. We contribute the formal specification and formal analysis of the DoS defense mechanism ASV$^+$SR which provides stable availability at a reasonable cost. ASV$^+$SR is a combination of the defense mechanism ASV and the Cloud-based resource provisioning mechanism SR.

Performance unpredictability. Cloud Computing management needs to be able to predict, forecast, and adapt the performance of their system. In particular, we consider Publish/Subscribe architectures. However, several parameters of the system and the unreliability of best-effort networks, especially when deployed in a worldwide setting, make it difficult to analyze such services. One further source of uncertainty, and challenge in order to ensure quality of service (QoS) system requirements, is the variability in the number of users of the service. An interesting research question is how to enrich a Publish/Subscribe architecture with Cloud-based dynamic resource provisioning mechanisms to better meet QoS requirements. In this thesis, we contribute a formal specification of a Publish/Subscribe system and show that predictions about QoS properties of the specified systems can be made using statistical analysis.

1.4. Outline of this thesis

This thesis is structured as follows: Chapters 2 and 3 respectively introduce the concept of Cloud Computing and the prerequisites on rewriting logic and the Maude system. In Chapter 4 we present formal languages for the design and analysis of Cloud Computing systems; in particular, the formal specification of formal languages based on the coordination language and mobile calculus KLAIM. Chapter 5 introduces a modularized actor model of computation which allows the specification of hierarchical models of distributed systems that can be statistically model checked. In Chapter 6 we formally specify and analyze ASV$^+$SR, a Cloud-based strategy to prevent a common threat in distributed computing: distributed denial of service attacks. Chapter 7 addresses the problem of performance unpredictability in Cloud-based systems; in particular, we show how a formal specification and analysis of a Cloud-based Publish/Subscribe infrastructure can help predict service quality and how reliability and availability can be improved using dynamic scaling of resources. Finally, Chapter 8 summarizes this thesis and gives an outlook on future work.

Chapters 4, 5, and 6 originated in collaboration with Jonas Eckhardt who published these chapters in his thesis "A Formal Analysis of Security Properties in Cloud Computing" [43].

Chapter 2

Cloud Computing in a Nutshell

We begin this introduction to Cloud Computing with a famous quote by Larry Ellison:

> *"The interesting thing about cloud computing is that we've redefined cloud computing to include everything that we already do. I can't think of anything that isn't cloud computing with all of these announcements. The computer industry is the only industry that is more fashion-driven than women's fashion. Maybe I'm an idiot, but I have no idea what anyone is talking about. What is it? It's complete gibberish. It's insane. When is this idiocy going to stop? We'll make cloud computing announcements. I'm not going to fight this thing. But I don't understand what we would do differently in the light of cloud other than change the wording of some of our ads."*
>
> — Larry Ellison, Oracle CEO [106]

In view of this statement, we summarize the essential aspects of Cloud Computing, give a technical definition of the term, and compare it to other utility computing paradigms.

2.1. What is Cloud Computing?

Cloud Computing can be described as *"the long-held dream of computing as a utility"* [19]. This dream of providing and consuming computing as a service is not new and has already been predicted by Turing award winner John McCarthy in the early 1960s [49]: *"Computation may someday be organized as a public utility just as the telephone system is a public utility"*. Within this context, Cloud Computing is only one paradigm among many that follows the idea of utility computing. Other popular examples are Grid Computing and Peer-to-Peer Computing; and all of these examples have gained increasing popularity over the past decade. For Timothy Chou, a driver for this success is a shift in software business models. In his book "Introduction to Cloud Computing" [34], he presents the seven fundamental software business models (Figure 2.1):

	1 Traditional	2 Open Source	3 Outsourcing	4 Hybrid	5 Hybrid+	6 SaaS	7 Web
Software	\$4,000/user (one time)	\$0	\$4,000/user (one time)	\$4,000/user (one time)	\$300/user/month	<\$100/user/month	Ads, Transactions, Embedded (<\$10/user/month)
Support	\$800/user/ year	\$1,600/user (one time)	\$800/user/ year	\$800/user/ year			
Service			Bid <\$1,300/ user/month	\$150/user/ month			
			@H @C	@H @C	@H @C		

Figure 2.1.: The seven software business models according to Timothy Chou (prices are descriptive examples) [34]

Model 1: Traditional. In the traditional software business model, software is permanently licensed to a customer for a one-time fee. In order to receive update and upgrade rights as well as technical support, customers are repeatedly charged a certain percentage of the one-time fee. In addition to the one-time and support fees customers are faced with management costs which can be, according the Gartner Group, as high as four times the one-time fee per user per year.

Model 2: Open Source. In the Open Source software business model, software is usually distributed free of charge. Customers pay for support and software management.

Model 3: Outsourcing. In the Outsourcing software business model, customers buy software licenses and sign support contracts just as in the traditional model. However, software management is outsourced to a third party. The physical hardware can thereby be located in a data center owned by the third party (@H) or in a data center that the customer owns (@C).

Model 4: Hybrid. The Hybrid software business model differs from model 3 in that no third party is involved and that the software company itself services their software. Service costs are usually lower than in model 3 because the software company can cut costs through specialization and standardization.

Model 5: Hybrid+. In the Hybrid+ software business model software, support, service, and management are offered for an all-in-one per-user fee.

Model 6: SaaS. Chou argues that if software is engineered to be delivered as a service and only as a service, operational costs are far lower than in model 4 and 5.

Model 7: Web. Software on the web is often not sold as in models 1–6. Companies that follow this business model rather indirectly monetize their offerings. eBay for example charges for transactions, Google on the other hand shows ads. These applications are

highly specialized and distribution costs on the Internet are marginal which allows even cheaper operation.

In his summary of the seven software business models, Chou argues that *"the days of million-dollar software licenses are clearly over"*. In the future, more and more companies will move from Model 1 to business models that are more service and utility oriented. Even Microsoft, a company which had great success with the traditional software business model, is undergoing this transformation. In a talk on Cloud Computing in 2010, Steve Ballmer said:

> *When you buy a new crate and you put it in a data center, is that cloud computing? I can't even tell you the private cloud versus the next generation of server and enterprise computing. But, about 70 percent of our folks are doing things that are entirely cloud-based, or cloud inspired. And by a year from now that will be 90 percent.*
>
> — Steve Ballmer, Microsoft CEO [80]

2.2. Technical definition

We have seen that, from a business perspective, Cloud Computing has gained quite some importance. In the following, we define the term Cloud Computing from a more technical point of view. The main actors in Cloud Computing are service users and infrastructure, platform, and service providers. Service providers make services accessible to service users through Internet-based interfaces. Thereby services are running on infrastructure provided by an infrastructure provider. In some cases, service providers build their services on platforms provided by platform providers who not only offer infrastructure but a software platform with standardized APIs and tools as a service. Dependent on the type of resource that is offered as a service, the term Cloud Computing covers the following "as a service" scenarios:

Infrastructure as a service (IaaS). In the Infrastructure as a service (IaaS) delivery model, an infrastructure provider delivers virtualized computing resources, such as data storage and computing power (CPU and memory). Customers of these resources deploy software stacks that run their services on these resources. Amazon is an example of an IaaS provider and offers their products S3 [16] and EBS [12] for storage and EC2 [14] for computing power.

Platform as a service (PaaS). Instead of providing infrastructure resources, in the platform as a service (PaaS) delivery model, software platforms which introduce an additional level of abstraction are offered. Management tasks such as the scaling of physical hardware resources for the platform are often transparent to customers and the applications they run on the platform. Additionally, service developers building for the platform have access to platform-specific APIs offered by the PaaS provider. Examples for PaaS products include Google App Engine [53], force.com [95], and Microsoft Windows Azure [82].

Software as a service (SaaS). As an alternative to locally installed applications, in the software as a service (SaaS), applications are being made available as a service that runs in the Cloud and can be accessed on the Internet. The infrastructure resources, the

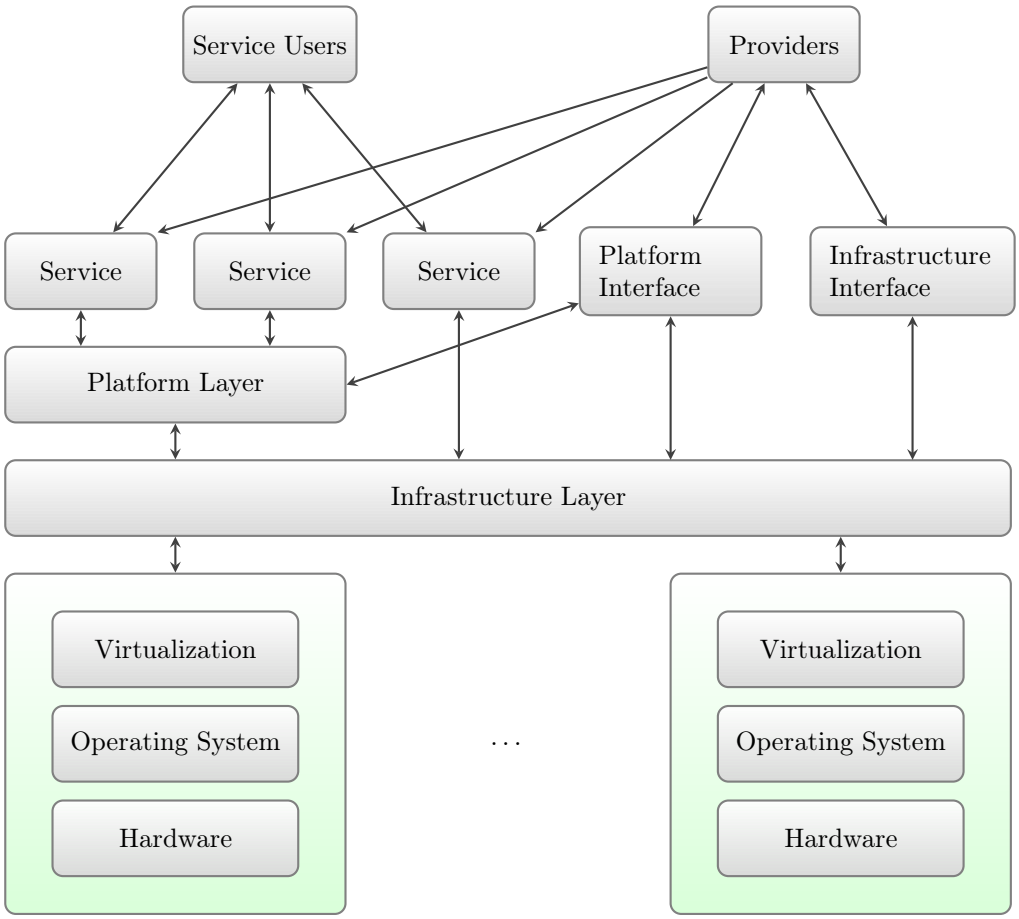


Figure 2.2.: Overview of actors and layers in Cloud Computing

software platform, the service implementation, and the related software and hardware management are invisible to the service user. Google web-based applications [56] are an example of SaaS that is offered on Internet.

Besides these three commonly named scenarios, other Cloud Computing descriptions [87] include a more data-driven context:

Database as a service (DaaS). Database as a service (DaaS) is a special form of Cloud-based offering where storage with a defined set of operations that customers can perform (such as querying the data) is offered as a service. Examples for DaaS products are Amazon SimpleDB [17] and RDS [15], Microsoft SQL Azure Database [79], and Google Cloud SQL [54].

Figure 2.2 gives an overview of actors and layers found in Cloud Computing scenarios.
In [108], Vaquero et al. propose a definition of Cloud Computing that followed a study of more than 20 other definitions and the extraction of a consensus:

> *"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."*
>
> — Vaquero, Rodero-Merino, Caceres, and Lindner [108]

Furthermore, Vaquero et al. name the three concepts that most definitions include: scalability, pay-per-use utility model, and virtualization. They also mention that there is no minimum common denominator among the definitions.

2.3. Comparison to other utility computing paradigms

A legitimate question is in how far Cloud Computing differs from other utility computing paradigms. Exemplarily, we compare Cloud Computing to Grid Computing, for which Ian Foster proposed a definition in 2002:

> *"[The Grid is] a system that coordinates resources which are not subject to centralized control, using standard, open, general-purpose protocols and interfaces to deliver nontrivial qualities of service."*
>
> — Ian Foster [47]

In light of the two definitions which show many similarities in their definition of the goal and approach, we highlight three key enabling concepts that might not be unique to Cloud Computing but are fundamental concepts of the paradigm:

1. **Virtualization** enables elasticity and the illusion of infinite capacity [19]. Furthermore, it makes features such as on-demand sharing of resources and security by isolation possible.

2. **Multitenancy** means that multiple costumers are consuming the same software service. In Cloud Computing user-specific data is isolated and service-specific functionality is shared among users. This results in cheaper development and management costs.

3. **Service-level agreements** (SLAs) are contracts between service users and providers. Thereby providers commit themselves to a concrete level of quality of service. While, for example, Grid Computing environments often only offer best-effort SLAs, Cloud Computing providers often give guarantees for availability and uptime. This clear assignment of responsibility is crucial for businesses who think about migrating to outsourced computing services.

Chapter 3

Preliminaries: Rewriting Logic and the Maude System

In order to formally specify and analyze Cloud Computing management, an appropriate semantic framework is needed. We chose rewriting logic [75] as the semantic framework and Maude [35], a language and system based on rewriting logic that offers the possibility of executing and formally analyzing specifications, as the foundation for the work in this thesis. In the following, we give brief introductions to rewriting logic and the Maude system.

3.1. A brief introduction to Rewriting Logic

Rewriting logic [75] is a simple, yet powerful, computational logic and is a general formalism that is a natural model of computation and an expressive semantic framework for concurrency, parallelism, communication, interaction, and object-orientation. It is capable of logical and distributed object reflection and, through its probabilistic and real-time extensions, it is capable of modeling real-time, stochastic, and hybrid systems. The article "Twenty Years of Rewriting Logic" [77] gives a more in-depth introduction to the topic and is a comprehensive survey of the work that has been done in that area.

In rewriting logic, concurrent and object-oriented systems are specified as *rewrite theories*, that is, as triples $(\Sigma, E \cup A, R)$, where

- Σ is a signature that defines the syntax and type structure of the system including kinds, sorts, and operators,
- $(\Sigma,\ E \cup A)$ is an order-sorted *membership equational logic* [76] theory with a set of (possibly conditional) Σ-sentences E which have equations $t = t'$ and memberships $t : s$

as atoms, and a set of equational attribute sets A (e.g., associativity, commutativity, identity, ...) for operators defined in Σ; and

- R is a set of (possibly conditional) *rewrite rules* of the form

$$t \to t' \text{ if } cond,\ cond := \bigwedge_l u_l = u'_l \wedge \bigwedge_m v_m : s_m \wedge \bigwedge_n w_n \to w'_n$$

with t, t' Σ-terms, and *cond* the rule's condition. Additionally, a rewrite theory can contain so-called frozen arguments where rewrites are forbidden.

A concurrent system is modelled by $(\Sigma, E \cup A, R)$ 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 $(\Sigma, E \cup A)$.
2. The *local atomic transitions* of the concurrent system are parametrically modeled by the rewrite rules in R; that is, a rewrite rule $t \to t'$ *if 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 [75].

Deduction in rewriting logic consists of the concurrent application of the rewriting rules in R modulo the equations in $E \cup A$.

3.2. The Maude system

The Maude system [35] is a high-performance implementation of rewriting and its underlying membership equational logic. Other examples for languages based on rewriting logic are OBJ3 [52] — the historical precursor to Maude — ELAN [28], and CafeOBJ [39]. Maude is capable of executing rewrite theories, which are specified as modules with a self-explanatory type-writer syntax that is almost isomorphic to the mathematical syntax. Modules are the key concept of Maude and can be *functional modules*, representing equational theories, and *system modules*, representing rewrite theories. Computation with these modules corresponds to deduction by rewriting. Maude and its tool environment can be used in three, mutually reinforcing ways [35]:

- as a declarative programming language,
- as an executable formal specification language, and
- as a formal verification system.

The language design aims to maximize simplicity, expressiveness, and performance. The book "All About Maude — A High-Performance Logical Framework" [35] provides a comprehensive description and documentation of Maude's features. In the following we shortly present some of the concepts that we use in this thesis.

3.2.1. Specification of real-time and probabilistic systems

Rewriting logic and Maude in particular can naturally model systems that can be both *real-time* and *probabilistic*. Real-Time systems are supported by rewrite theories $(\Sigma, E \cup A, R)$ whose underlying equational theory $(\Sigma, E \cup A)$ includes an algebraic data type to represent time instants (which may be either discrete or continuous) among its types, and whose global states are pairs of the form (t, r), with t a term representing a "discrete" state, and r a time value representing the global clock. The rewrite rules in R can then be either *instantaneous* rules, that do not change the global clock, or *tick* rules, that advance the global time (see [89]). Probabilistic systems, which may also be real-time systems, are modeled by *probabilistic rewrite rules* of the form

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

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. The distribution is, in general, not fixed but parametric in the righthand side variables $\vec{x}$ (see [35]). We refer to [35] for the systematic transformation of the declarative definition of a probabilistic rewrite theory into a corresponding Maude specification which simulates it. The Maude notation is used for probabilistic rewrite rules in this thesis.

3.2.2. Specification of object-oriented systems

The Cloud Computing systems that we consider in this thesis are object-oriented distributed systems where objects communicate via asynchronous message passing. We briefly explain how such systems are formally specified in rewriting logic and Maude. For an object o, an object-oriented specification defines a class C and a unique name o that identifies it. 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 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 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

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

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. We use

the concept of objects at several points in this thesis but sometimes introduce different syntax that better reflects the underlying specification and domain-specific syntax.

3.2.3. Parameterized modules

Both, functional and system modules, can be *parameterized* by a parameter theory P. A *parameterized module* $M[X :: P]$ has a formal parameter X satisfying P; M can be instantiated by another module Q via a theory interpretation $V : P \longrightarrow Q$, called a *view*, with the usual pushout semantics [35]. The resulting module is denoted by $M[V]$ or shorter $M[Q]$ if V is clear from the context.

3.2.4. Formal meta-object patterns

Adaptation is a challenge when designing, building, or verifying distributed systems, because these systems need to function in highly unpredictable and potentially hostile environments. To meet these adaptation challenges and the associated requirements, a modular approach based on *meta-objects* can be extremely useful. A meta-object is an object which dynamically *mediates/adapts/controls* the communication behavior of one or several objects under it. Meta-object models include work on meta-actors such as the onion-skin model [3] and the TLAM model [109]. These models have been formalized in rewriting logic and extended in various ways in, e.g., [78, 102].

In rewriting logic, a meta-object can be specified as an object of the form $\langle o : K \mid conf : c, a_1 : v_1, \ldots, a_n : v_n \rangle$, where c is a term of sort *Configuration*, and all other $v_1 \ldots, v_n$ are not configuration terms. The configuration c contains the object or objects that the meta-object o controls. If c contains a single object, the meta-object o is sometimes called an *onion-skin* meta-object [3], because o itself could be wrapped inside another meta-object, and so on, like the skin layers in an onion. More generally, c may not only contain several objects $o_1 \ldots, o_m$ inside: it may also be the case that some of these o_i are themselves meta-objects that contain other objects, which may again be meta-objects, and so on. That is, the more general reflective meta-object architectures are so-called "Russian dolls" architectures [78], because each meta-object can be viewed as a Russian doll which contains other dolls inside, which again may contain other dolls, and so on. Both the onion-skin and the TLAM [109] models are special cases of this general Russian dolls model. In this work we will present meta-object patterns that illustrate both the onion-skin case, and the general Russian dolls case.

Chapter 4

Formal Languages 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 4.1),
2. introduce the KLAIM language specification (Section 4.2),
3. develop M-KLAIM, a Maude-based formal executable specification of the KLAIM coordination language (Section 4.3),
4. extend M-KLAIM to OO-KLAIM for object-oriented specifications (Section 4.5),
5. extend OO-KLAIM to D-KLAIM for distributed object-oriented specifications (Section 4.6),
6. and lastly show how specifications based on the aforementioned languages can be formally analyzed (Section 4.7).

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.

4.1. Introduction to Coordination Languages

According to the article "Coordination Languages and their Significance" [51], 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.

4.1.1. Linda

Linda is a coordination language developed by Gelernter et al. at Yale University [51, 50] 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 $\mathbf{rdp}(t)$

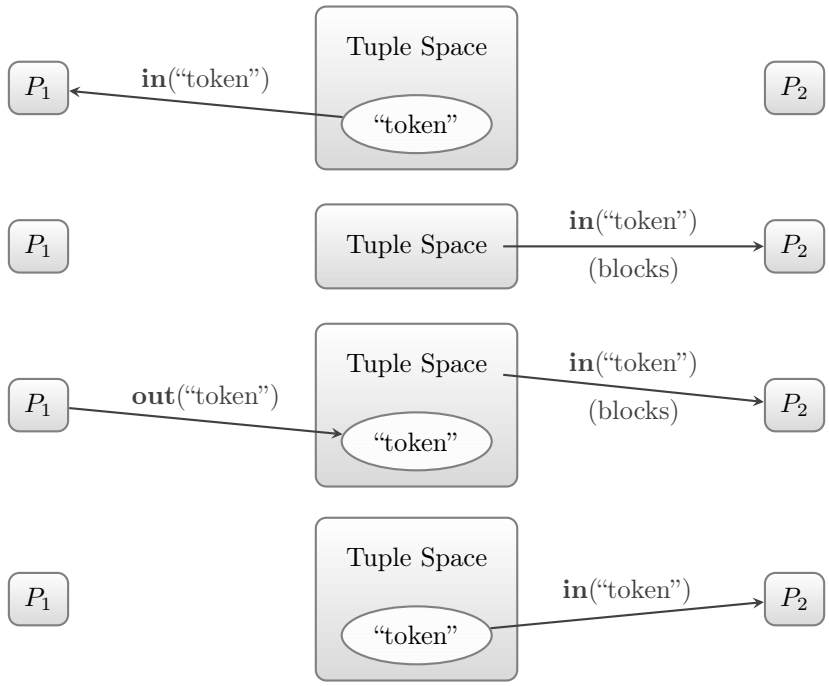


Figure 4.1.: Example of process synchronization using the Linda model

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 [119] and C++ [41].

Example 4.1: Synchronization using the Linda model
Figure 4.1 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) [40], where **in**("token") corresponds to the P-operation and **out**("token") corresponds to the V-operation on general semaphores.

4.1.2. Advantages of Coordination Languages

According to [51], 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" [118]. 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.

4.2. KLAIM

In [38], 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 [99], a calculus of explicit substitutions, and apply it to the M-KLAIM specification.

4.2.1. 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 [63], CSS [84] and the π-calculus [85]. 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, ρ) where s is a *site*, P is a *process*, and ρ 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 ρ_1 and ρ_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 ρ_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 4.2: A legal KLAIM net
In the following example let s_1, s_2 be sites, l_1, l_2 be logical localities, ρ_1, ρ_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 ρ_1 is never considered for the evaluation on l_1 as $[s_1/self] \bullet [s_1/l_1](l_1)$ is already defined.

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

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

Figure 4.2.: KLAIM process syntax

Process syntax

KLAIM processes are built using operators borrowed from Milner's CCS [84]. 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 $\mathit{out}(t)@l$, $\mathit{eval}(P)@l$, $\mathit{read}(t)@l$, and $\mathit{in}(t)@l$ 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 $\mathit{newloc}(u)$ action is used to create fresh sites. The locality variable u refers to that fresh site in the prefixed process. Figure 4.2 gives an overview of the process syntax.

In KLAIM, the $\mathit{newloc}(u).P$, $\mathit{read}(t)@l.P$ and $\mathit{in}(t)@l.P$ actions act as binders for variables that are used in the process P. For example, in $\mathit{newloc}(u).P \mid \mathit{in}(!x, !X).Q$, 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 4.3: 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:

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

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 μ gives an abstract description of what activity is performed and the label ρ stands for the allocation environment that records the local bindings that must be taken into account to evaluate μ. 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 4.3.

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

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

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

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

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

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

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

Figure 4.3.: Structural rules of KLAIM's symbolic semantics

locality names. The rules for $\mathcal{T}[\![.]\!]\rho$ are depicted in Figure 4.4, where $\mathcal{E}[\![.]\!]$ evaluates closed 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 ρ.

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

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

Figure 4.4.: Inductive definition of KLAIM's tuple evaluation function

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

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

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

Figure 4.5.: KLAIM's matching rules

is used in conjunction with the local allocation environment ρ_1 to evaluate the tuple. If the 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 4.5, 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.

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 4.4: 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 ρ_1 and ρ_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:

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

4.3. M-KLAIM — a Maude-based specification of KLAIM

Rewriting logic [75] supports the executable specification of KLAIM's syntax and structural operational semantics. This can be done in several definitional styles [101], 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 4.2 *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.

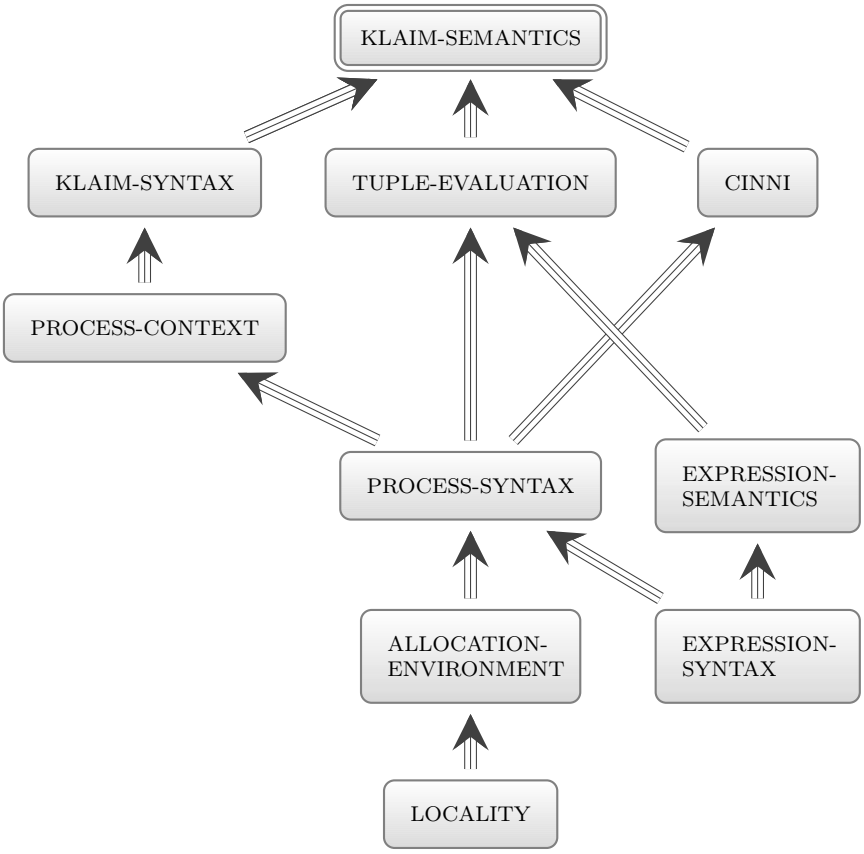


Figure 4.6.: Overview of Maude modules of the M-KLAIM specification

4.3.1. Overview

Figure 4.6 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 4.4 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.

4.3.2. Description of modules

The following descriptions of modules show how KLAIM's SOS specification style of Section 4.2 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 ρ_1 `*` ρ_2 of two allocation environments ρ_1 and ρ_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 4.4. 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.

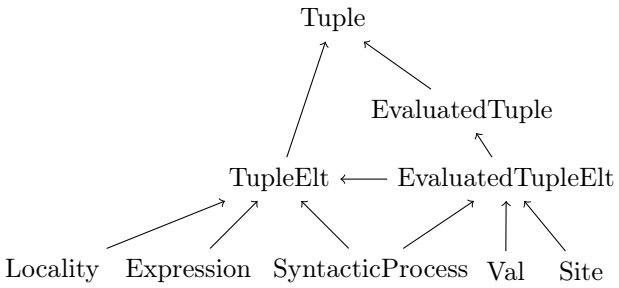


Figure 4.7.: Subsort hierarchy in the *PROCESS-SYNTAX* module

```
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 4.7 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`.

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 4.3.2).

- 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 [110], 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 4.7, 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 ρ.

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

Localities are evaluated using the evaluation function of the allocation environment (Section 4.3.2).

```
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 [35]), 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 4.6.5 and 4.7.

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 [30], Verdejo et al. [110], and Serbanuta et al. [101] 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.

4.4. 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 [99]. For a given language L and its defining syntax, the instantiation of CINNI for L is denoted by 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 [37] as the special case, in which a single name is used. CINNI uses the Berklin notation [23, 24] 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 ith binder to the left of the variable in the term.

Example 4.5: 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.

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

CINNI extends a given language with explicit substitutions as shown in equations B.1, B.2 and B.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 B.4, B.5, and B.6. It decreases the index of variables with the name X, performs the substitution S, and finally lifts the variable.

$$[X := M] \qquad \text{(simple substitution)} \tag{4.1}$$

$$\uparrow_X \qquad \text{(shift substitution)} \tag{4.2}$$

$$\Uparrow_X (S) \qquad \text{(lifted subsitution)} \tag{4.3}$$

$$\Uparrow_X (S)X_0 = X_0 \tag{4.4}$$

$$\Uparrow_X (S)X_{n+1} = \uparrow_X (SX_n) \tag{4.5}$$

$$\Uparrow_X (S)Y_n = \uparrow_X (SY_n) \text{ if } X \neq Y \tag{4.6}$$

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 4.6: 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.

$$S(n) = n,\ n \in \mathbb{N} \tag{4.7}$$
$$S(nil) = nil \tag{4.8}$$
$$S(E_1 + E_2) = S(E_1) + S(E_2) \tag{4.9}$$
$$S(P_1 \mid P_2) = S(P_1) \mid S(P_2) \tag{4.10}$$
$$S(out(E).P') = out(SE).(SP) \tag{4.11}$$
$$S(in(X).P') = in(X)(\Uparrow_X (S)P') \tag{4.12}$$

Rules 4.7 and 4.8 eliminate the substitution if it is applied to a natural number or the *nil*-process. The rules 4.9, 4.10, and 4.11 pass substitutions down to subterms. If a substitution is applied to a process P' with the *in* action prefix $in(X).P'$, rule 4.12 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:

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

$$\rightarrow [X := 3]in(X).out(X_0 + X_1).nil \mid out(4).nil \tag{4.13}$$
$$\rightarrow in(X).(\Uparrow_X ([X := 3])out(X_0 + X_1).nil) \mid out(4).nil \tag{4.14}$$
$$\rightarrow in(X).(out(\Uparrow_X ([X := 3])(X_0 + X_1)).(\Uparrow_X ([X := 3])nil)) \mid out(4).nil$$
$$\rightarrow in(X).(out(\Uparrow_X ([X := 3])(X_0) + \Uparrow_X ([X := 3])(X_1)).nil) \mid out(4).nil$$
$$\rightarrow in(X).(out(X_0 + \uparrow_X ([X := 3]X_0)).nil) \mid out(4).nil \tag{4.15}$$
$$\rightarrow in(X).(out(X_0 + 3).nil) \mid out(4).nil$$
$$\rightarrow [X := 4](out(X_0 + 3).nil) \mid nil$$
$$\rightarrow (out([X := 4](X_0 + 3)).([X := 4](nil))) \mid nil$$
$$\rightarrow (out(([X := 4]X_0) + ([X := 4]3)).nil) \mid nil$$
$$\rightarrow out(4 + 3).nil \mid nil$$

Step 4.13 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 4.14. Finally, step 4.15 applies the lifted substitution which replaces the correct variable, i.e., X_1, with the value 3.

4.4.1. 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 4.7: 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 4.8: 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`.

4.5. 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.

4.5.1. 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 4.3.

4.5.2. 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] .
```

4.5.3. 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 `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))) .
```

4.6. 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.

4.6.1. 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.

4.6.2. D-KLAIM specification overview

Figure 4.8 gives an overview of the D-KLAIM specification. The modules *D-KLAIM-METATOOLS* 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.

4.6.3. D-KLAIM modules

D-KLAIM extends the OO-KLAIM specification with two main modules, *D-KLAIM-METATOOLS* 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.

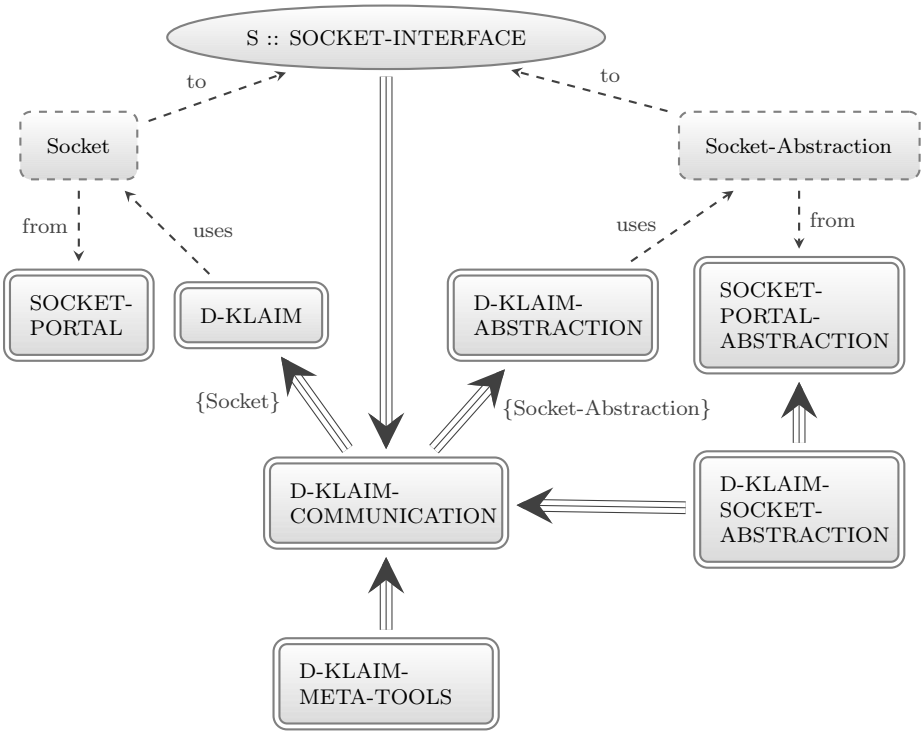


Figure 4.8.: D-KLAIM specification overview

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 4.6.4.

```
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) .
```

4.6.4. 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.

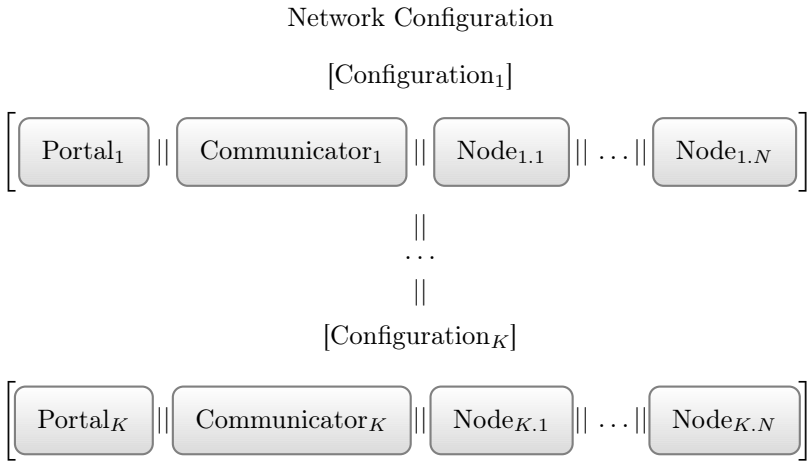


Figure 4.9.: Overview of a network configuration in the D-KLAIM socket abstraction

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

Figure 4.9 gives an 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 are 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]) .
```

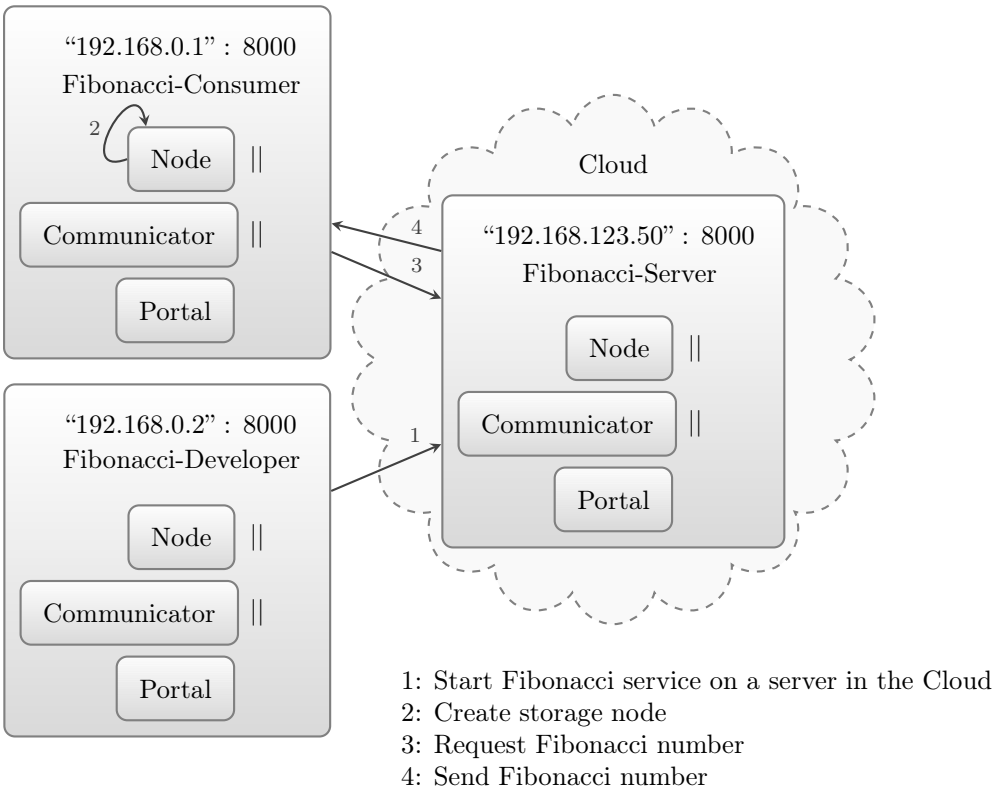


Figure 4.10.: Overview of the Fibonacci Cloud service architecture

4.6.5. 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 4.10 provides an 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.

4.7. 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.

4.7.1. Maude LTL model checking

Maude supports on-the-fly explicit state linear temporal logic (LTL) model checking of concurrent systems [100, 35]. 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*.

4.7.2. 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 4.1 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 4.6.4. 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.

4.7.3. 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 [35].

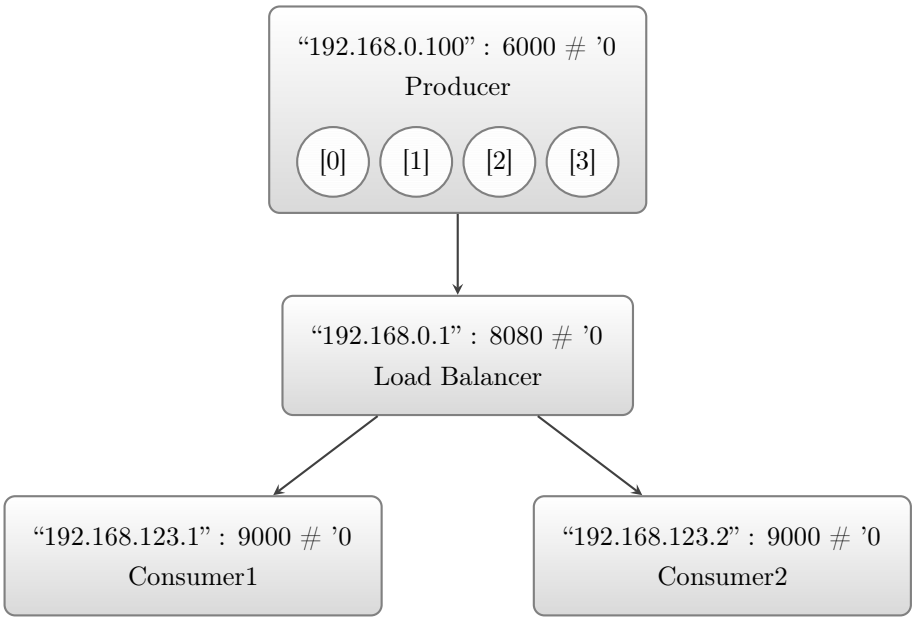


Figure 4.11.: Schematic overview of the load balancer example

4.7.4. 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 4.11 provides a 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.

4.8. Related Work

Here we discuss related work on process calculi, the formal design and analysis of distributed and Cloud Computing service-oriented architectures, and KLAIM.

Besides KLAIM [38] and Linda [50], several other calculi and formal languages that can express and analyze mobile and distributed computing have been proposed. Two examples that also influenced the work in this thesis are Mobile ambients [31] and Mobile Maude [42]: Mobile ambients is a calculus that describes the movement of processes and devices; Mobile Maude is a mobile agent language that extends Maude with the support for mobile computation.

Many of the aforementioned calculi and formal languages are influenced by the leading examples of classic process calculi including CSP [63], CCS [83], LOTOS [44], and the π-calculus [85]. PEPA (Performance Evaluation Process Algebra) [62] is a stochastic process algebra which is based classical process algebras and introduces probabilistic branching and timing of transitions.

With the rise of the Internet and Cloud Computing came a great demand to automate business processes and workflows among organizations and individuals. Service-oriented solutions require an orchestration among different services which may be distributed and run concurrently. [86] describes Orc, a theory for the orchestration of such services and provides a timed and concurrent programming language; in [6], AlTurki formally specified and analyzed the Orc language using the Maude system. Works in the Eu-project SENSORIA also face the challenges of distributed service-oriented computing with formal design and analysis: In [22], formal methods are used to model and verify a variant of SOAP that supports asynchronous communication; [120] shows how a performance analysis of a service-oriented system specified as an UML model can be achieved. Thereby, the UML model is transformed into a PEPA process which is then analyzed using a stochastic model checker or a transient analysis evaluator.

The KLAIM formalism itself has been extended and used in multiple ways. X-KLAIM [26] is a programming language, based on KLAIM, for programming distributed applications with object-oriented mobile code. There is also a compiler to translate X-KLAIM code into

Java code that uses KLAVA [25], a Java-based run-time system for KLAIM. Other work focuses on a temporal logic for the specification of properties of Klaim programs [88].

4.9. Conclusion

In this chapter, we have shown the Maude-based specification of a formal language based on the KLAIM language specification (M-KLAIM). We further extended this specification with object-orientation (OO-KLAIM) and showed how sockets can be used to execute such specifications in a distributed environment (D-KLAIM). Finally, we demonstrated how specifications based on *-KLAIM can be formally analyzed using model checking.

The various examples in this chapter have shown that the design and analysis of distributed systems such as *Cloud Computing* systems are possible using the *-KLAIM language specifications as a foundation. It was further shown that the languages provided in this chapter might prove themselves helpful for the rapid prototyping of such systems.

Chapter 5

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 5.1),
2. provide an introduction to statistical model checking (Section 5.2),
3. introduce the *Reflective Russian Dolls* model (Section 5.3),
4. extend the standard actor model to support Russian dolls models (Section 5.4),
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 5.5).

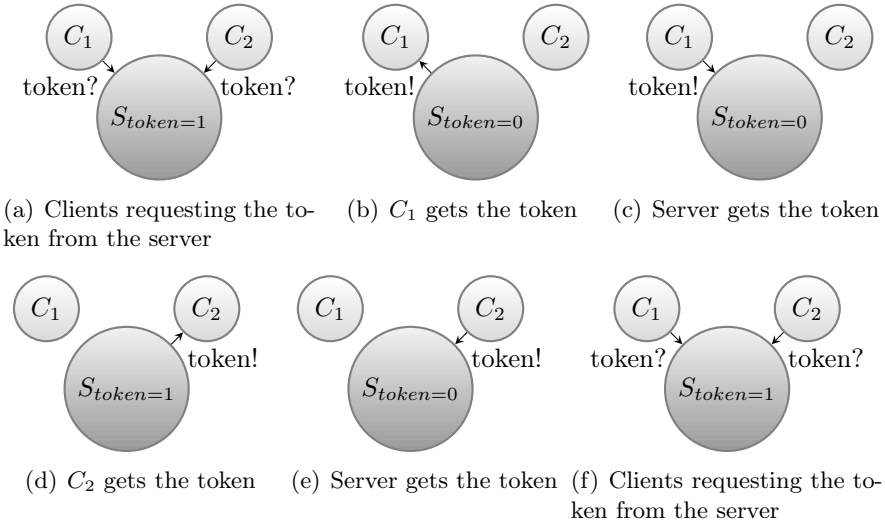
Modular, distributed, and concurrent systems can naturally be modeled and statistically model checked using the modularized actor model. The modular actor model is later used as a foundation of the specifications in the Chapters 6 and 7.

5.1. Introduction to the *Actor Model of Computation*

The *actor model of computation* [61, 60, 2] 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 5.1: 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 5.1 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 5.1.: 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 [9].

5.1.1. 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 .
```

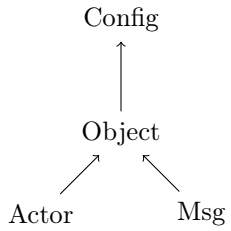


Figure 5.2.: Subsort Hierarchy of the Actor Model

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 5.2. 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 to and message 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.

5.2. 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 [8], a version of VeStA [98] which heavily exploits parallelism, can be used to statistically analyze Maude models that are based on the aforementioned actor models.

5.2.1. 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* [68, 67, 5]. 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:

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

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 ρ for the variables $\vec{y}$. ρ is chosen according to the family of probability functions $\pi(\theta)$: one function for each matching substitution θ of the variables $\vec{x}$.

In [5], 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 5.2 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 5.2: 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 [5], 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 [97, 96].

5.2.2. 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 [9], 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 [9], 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 [5], 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 [5], 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 5.3 illustrates the difference between the two approaches using the client server setting from Example 5.2.

Example 5.3: 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 5.2, 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 5.2.2.

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.

5.2.3. Statistical Analysis using the PVeStA model checker

In [97], Sen et. al. describe an algorithm for statistical model checking based on simple hypothesis testing. Formulas in *Probabilistic CTL* (PCTL) [59] and *Continuous Stochastic Logic* (CSL) [20, 21] 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 5.3.: 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 [5], 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 5.3. 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 [5].

Example 5.4: 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 $(1 - \alpha)$ [64] for the expected value is bounded by δ.

In [5], 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 α and δ) 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 [8], 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 5.6, 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.

5.3. 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.

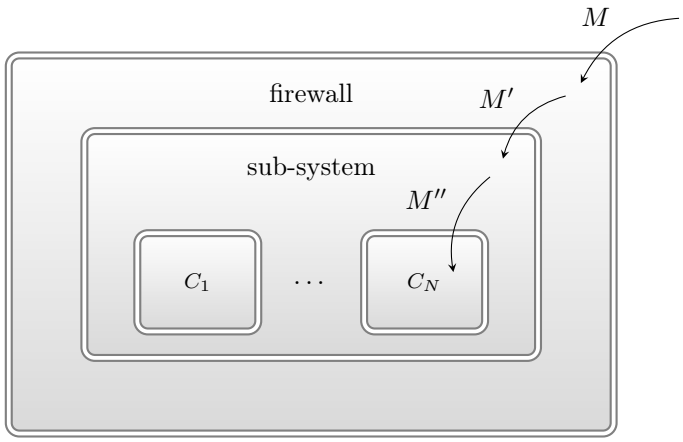


Figure 5.4.: Example of a Russian doll model of a system with boundaries

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 [78], 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 5.4 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 [78].

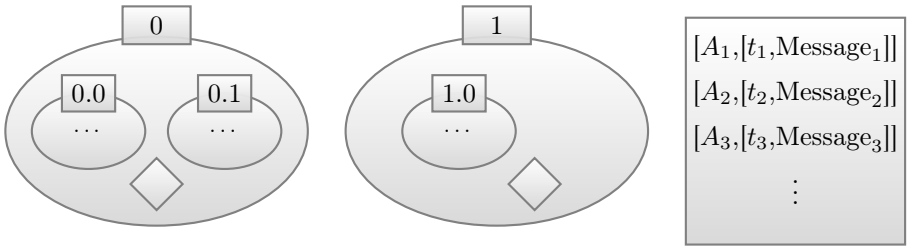


Figure 5.5.: Basic overview of the modularized actor model

5.4. 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 5.2 can handle models based on the RRD. In the following, an extension to the actor model of Section 5.1 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 6 and 7 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 5.5 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, Message_1, Message_2, and 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 $t_1 < t_2 < t_3$.

5.4.1. The Hierarchical Addressing Scheme

Figure 5.6 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 .
```

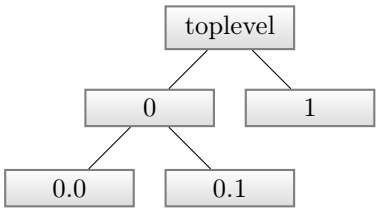


Figure 5.6.: Exemplary usage of the hierarchical naming scheme

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.

5.4.2. 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 5.7. 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] .
```

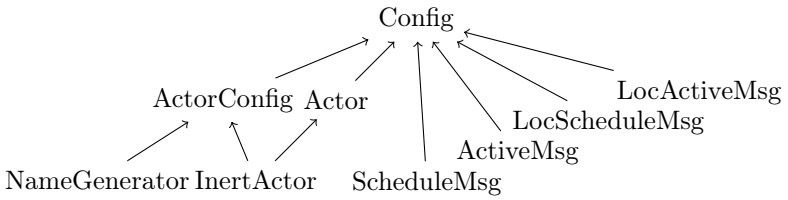


Figure 5.7.: Subsort Hierarchy of the Extended Actor Model

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`.

5.5. Multi-level scheduling for the Modularized Actor Model

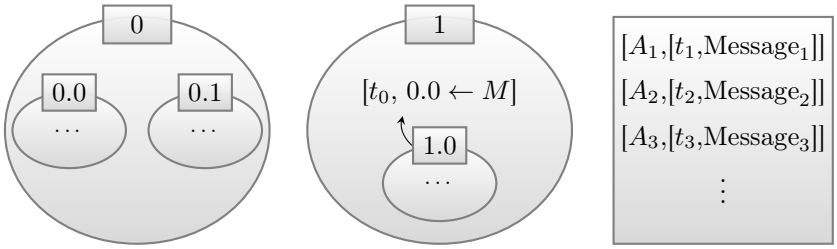
Figure 5.8 shows a schematic overview of the scheduling approach. The scheduling approach for a *scheduled messages* consists of two phases:

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.

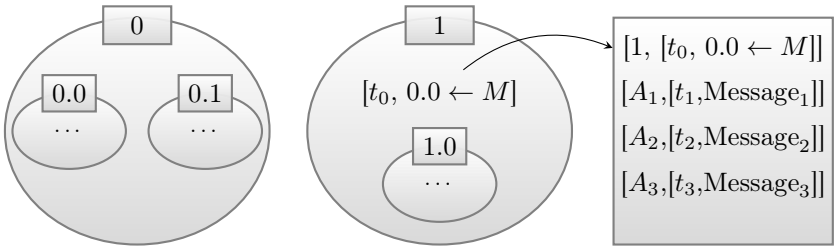
2. When a *scheduled message* is scheduled to be active, it is pushed down to the configuration where it was emitted and is inserted there as an *active message*. Internally, the *scheduled message* is thereby removed from the scheduler and inserted in the top-most configuration as a term of the sort `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.

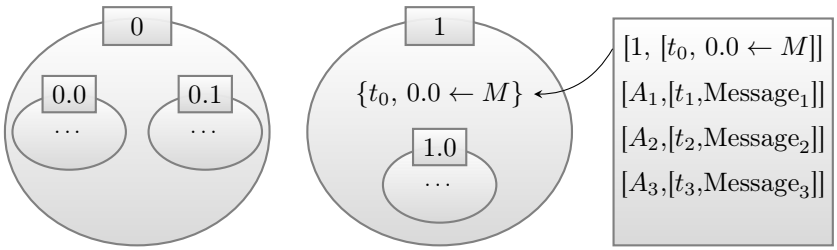
[2] There is a solution which allows for any naming scheme that prevents the collision of names to be used: While pulling the message up, the names of the actors on the path up to the top-level are collected in an ordered list that represents the path. In order to put a message back into the configuration in which it was emitted, this path is traversed in the reverse order.



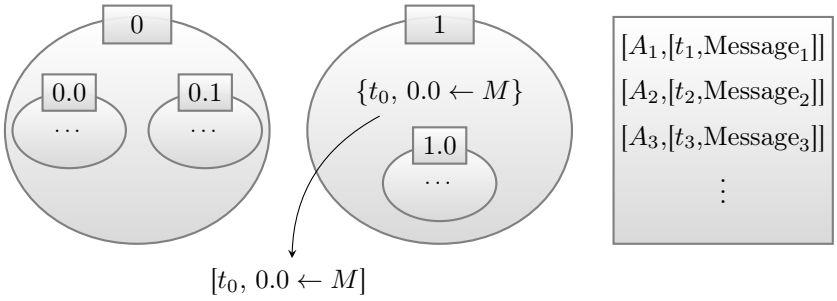
(a) The actor with address 1.0 is sending a message to the address 0.0



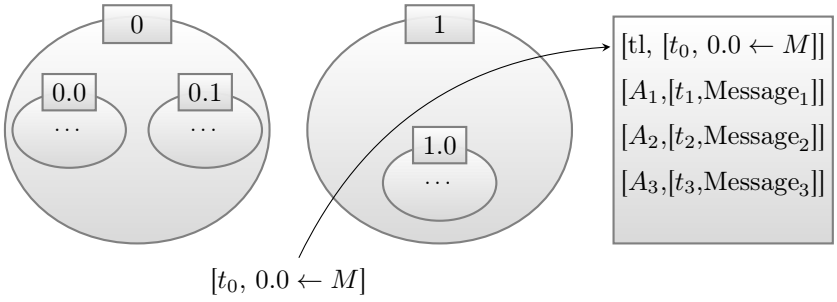
(b) The message is inserted in the scheduler with a reference to the containing actor's address



(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 5.8.: 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 [5], 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 5.5, the system reaches a final state as no more rewrites are possible.

5.5.1. 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 5.2.2. 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.

5.6. Using PVeStA to Statistically Analyze Specifications based on the Modularized Actor Model

As mentioned in Section 5.2.3, 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 C.1). The results of these analyses are shown in Sections 6.3.2, 6.4.3, 7.4, and 7.7.

5.6.1. 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`.

5.6.2. 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 recom-

mendable 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 [98, 8, 97] and Section 5.2.3. If not otherwise mentioned, the results in this thesis rely on a 99% confidence interval of size at most 0.01.

Chapter 6

Guaranteeing Stable 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 6.1),
2. describe the Adaptive Selective Verification (ASV) protocol (Section 6.2),
3. specify an ASV Wrapper meta-object in Maude and formally analyze a client-server system under attack using the ASV Wrapper (Section 6.3),
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 6.4).

The Maude specifications shown in this chapter are based on the modularized actor model described in Chapter 5 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 [6], and the result analyses in Section 6.3, although they consider a wider range of attack volumes than in [9], are in agreement with the model checking results in [9].

6.1. 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

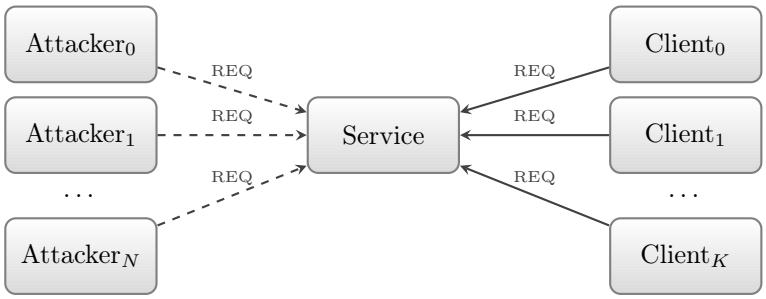


Figure 6.1.: Illustration of a service under a DDoS attack

requests which causes the computational resources to be overwhelmed. As a consequence, the system which runs the service experiences a slowdown or even crashes.

Denial **o**f **S**ervice (DoS) is one of the six categories defined in Micrsoft's STRIDE threat model [81] 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 6.1 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" [94]. 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." [72]. 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." [73].

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.

6.2. The ASV Protocol

In the *shared channel* attacker model [57], 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 [66] 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 [111] 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 REQs per second) is denoted by S, while the clients' arrival rate is denoted by ρ, 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 ith timeout window. In any given time window W_i, the system is under a DDoS attack which is flooding the server with REQs 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 REQs per second (the attacker sends $\alpha(W) * S$ fake REQs 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 REQs 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 ρ and send REQs to the server. When no ACK is received within the timeout window of T seconds, the client adaptively increases the number of REQs 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 REQs 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 REQs and, after T seconds, answers the REQs stored in the buffer. If a server receives less REQs than the buffer size, the server answers all incoming REQs. Otherwise, the server probabilistically drops or replaces REQs. The server-side protocol proceeds as follows.

S1. Initialize the window count $k \leftarrow 1$.

S2. In a window W_k, store the first $\lfloor S*T \rfloor$ REQs in the buffer. If the timeout window ends and less than $\lfloor S * T \rfloor$ REQs have been received, go to S4. Otherwise, set the packet count to $j \leftarrow \lfloor S * T \rfloor + 1$.

S3. The jth 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 jth REQ. Then, increase the packet count ($j \leftarrow j + 1$) and repeat the step until the timeout window ends.

S4. Send ACKs for each of the REQs in the buffer. Then, empty the buffer, set $k \leftarrow k + 1$ and continue at step S2.

A manual analysis of the protocol described above is a demanding task [66]. In this chapter, we specify the ASV protocol as a wrapper meta-object in Maude adapting a similar wrapper specification in [6], 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.

6.3. 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 5. We follow a modular 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

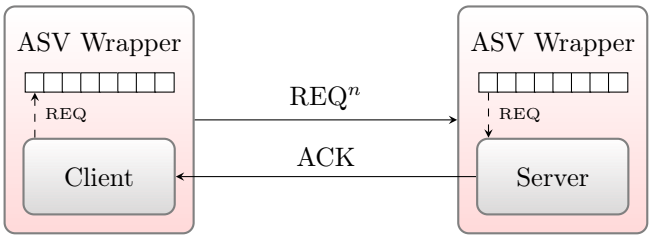


Figure 6.2.: Overview of a Cloud-based service setup using the ASV Wrapper

meta-objects for the client and the server (see Figure 6.2). 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 6.2). We then formally analyze the formal model using statistical model checking.

6.3.1. Description of the ASV specification in Maude

Figure 6.3 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.

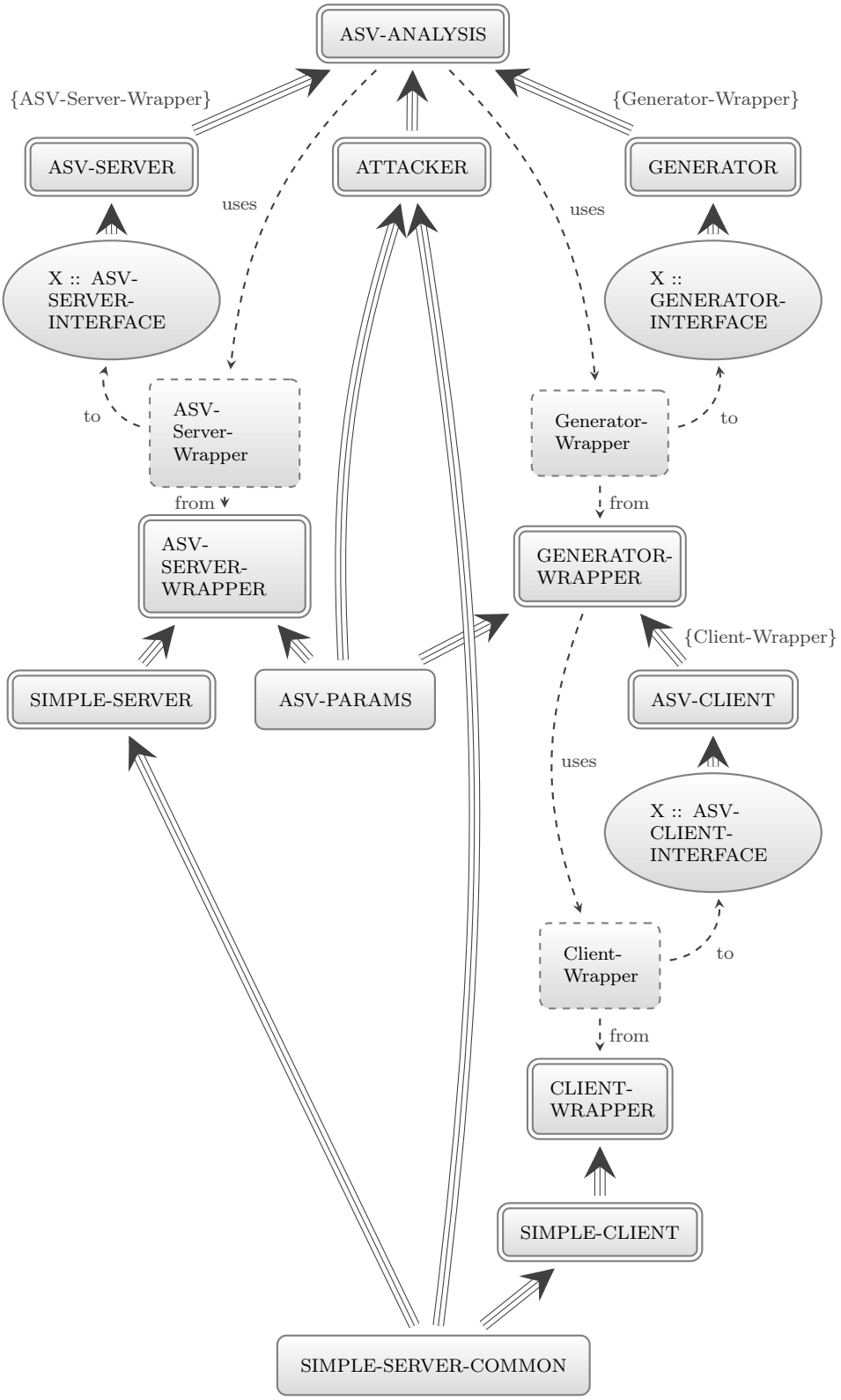


Figure 6.3.: Overview of the ASV analysis specification

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

```
sort Status .
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 6.2, 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) .
```

6.3.2. 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.

$$successRatio(t) = \textbf{if } time() > t \textbf{ then } \frac{countSuccessful()}{countClients()} \textbf{ else } \bigcirc\,(successRatio(t))$$

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.

$$avgTTS(t) = \textbf{if } time() > t \textbf{ then } \frac{sumTTS()}{countSuccessful()} \textbf{ else } \bigcirc\,(avgTTS(t))$$

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 ρ 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 [66, 9]. The properties are checked for various attack conditions represented by the constant α 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 [66, 9], we check the aforementioned properties for two setups, using

a) the ASV protocol on the server- and client-side.

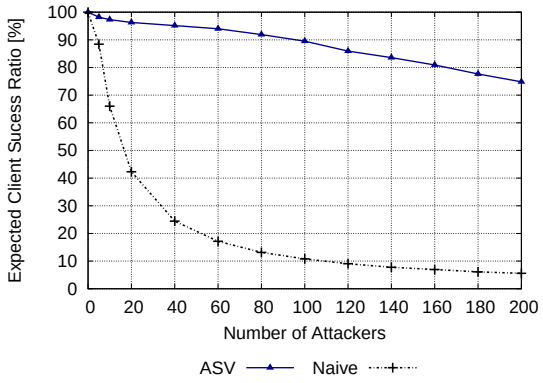
b) a non-adaptive client strategy, namely the *Naive* protocol, in which a client does not exponentially increase the number of *REQ*s after each time window but keeps sending a single *REQ*.

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*.

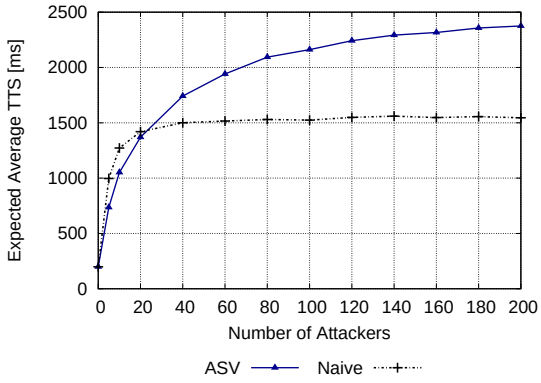
Figure 6.4 shows the results of the statistical model checking. The results clearly demonstrate the effectiveness of the ASV protocol compared to the non-adaptive client strategy. They also confirm the results in [66, 9]. Figure 6.4(a) shows that the ASV protocol can guarantee a high availability ($> 70\%$) 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 6.4(b) and 6.4(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.

6.4. ASV$^+$SR — a 2-Dimensional Protection Mechanism against DDoS Attacks

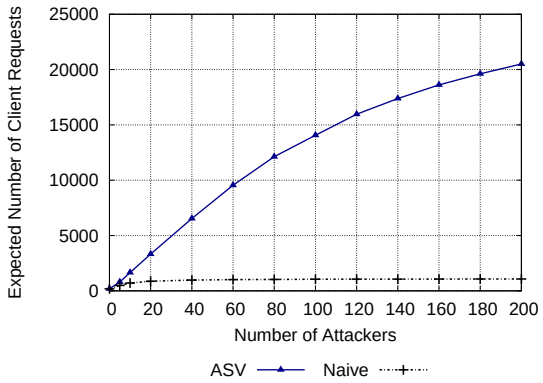
Cloud-based systems offer the possibility of provisioning resources on demand. For many applications, provisioning additional servers and replicating the service-providing application



(a) Client success ratio



(b) Average TTS



(c) Number of client requests

Figure 6.4.: Performance of the ASV protocol compared to a non-adaptive client strategy

can alleviate a DDoS attack, because more requests per second can be handled. In the following, we describe ASV$^+$SR, 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.

6.4.1. The Server Replicator meta-object and the ASV$^+$SR protocol

The Server Replicator (SR) is a meta-object that wraps around servers in a Cloud-based setup (see Figure 6.5). 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

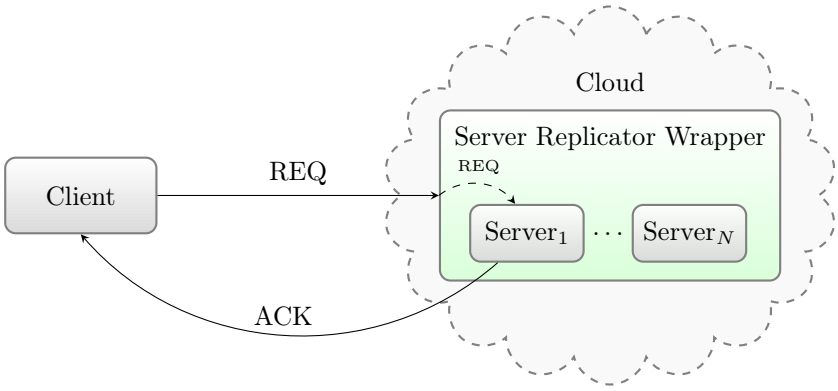


Figure 6.5.: Overview of a Cloud-based service setup using the Server Replicator Wrapper

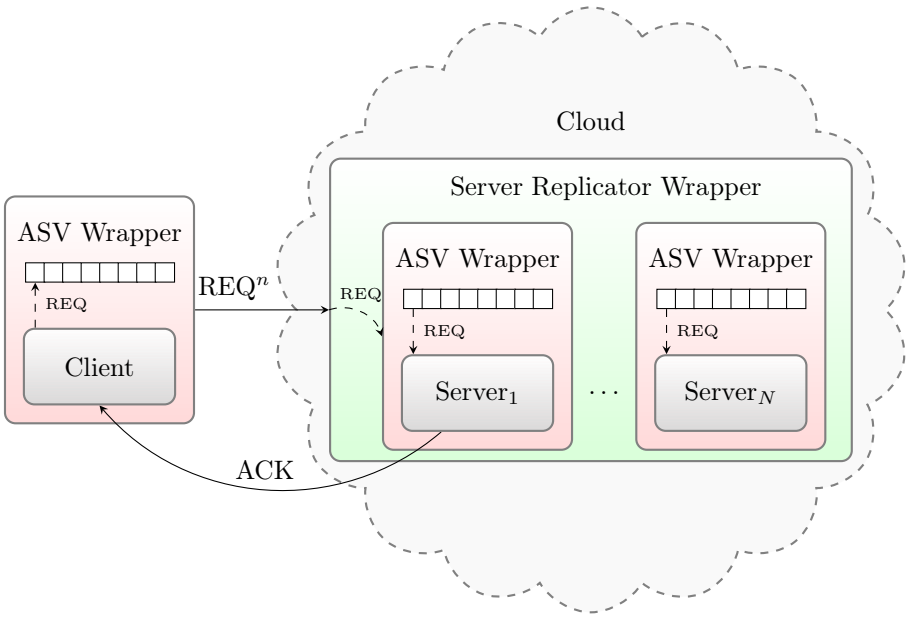


Figure 6.6.: Overview of a Cloud-based service setup using the ASV+SR protection

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 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+SR 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+SR protocol is shown in Figure 6.6. 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$^+$SR 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$^+$SR 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$^+$SR protocol.

6.4.2. Description of the ASV$^+$SR specification in Maude

The modularity of the specification of the ASV protocol (see Section 6.3) 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 6.3 gives an overview of the specification which is used for the analysis of the ASV$^+$SR 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 .
```

[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.

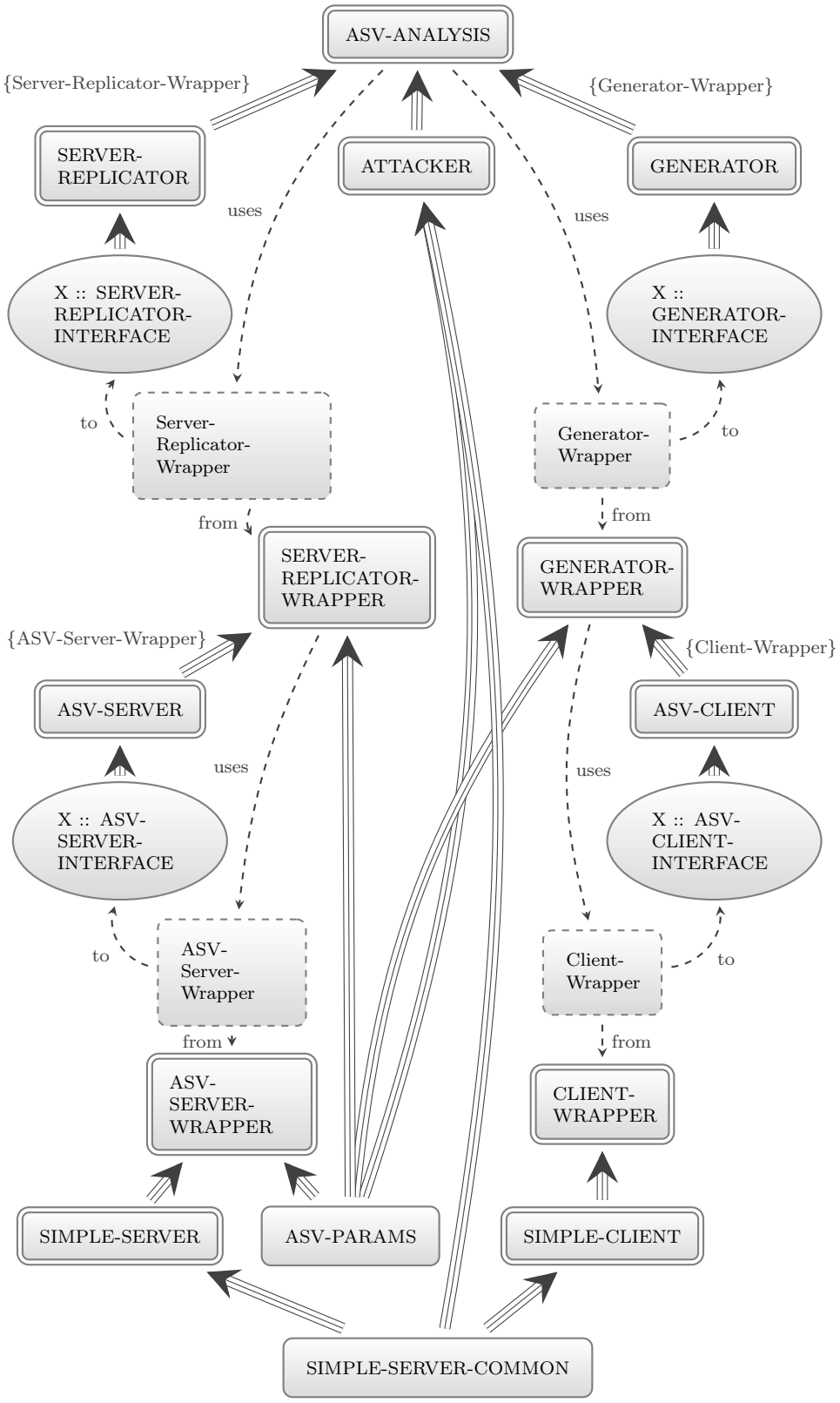


Figure 6.7.: Overview of the ASV+SR analysis specification

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.

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].

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

[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 address.

```
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 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.

6.4.3. Statistical Model Checking Results

As in Section 6.3.2, we use the specification of the ASV+SR protocol together with the client-server setting to perform statistical model checking of QuaTEx formulas analysing the behavior of ASV+SR under DoS attack using PVeStA.

We want to analyze the QuaTEx formulas defined in Section 6.3.2, 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.

$$\begin{aligned} \mathit{servers}(t) = \textbf{if } \mathit{time}() > t \textbf{ then } \mathit{countServers}() \\ \textbf{else } \bigcirc(\mathit{servers}(t)) \end{aligned}$$

with *countServers*() being equal to the size of the server list attribute of the Server Replicator meta-object.

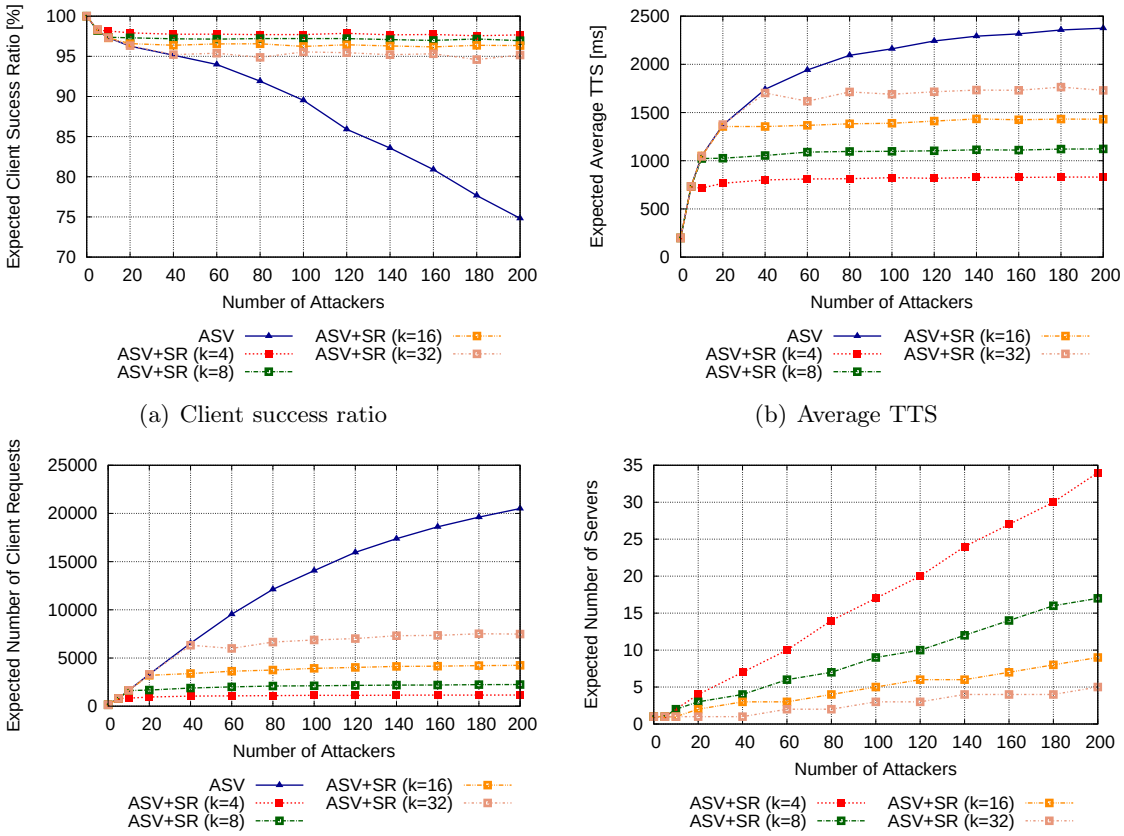
As in Section 6.3.2, for the statistical model checking of the aforementioned properties, we again fix the mean server processing rate S to 600 packets per second, the timeout window T to 0.4 seconds, the retrial span J to 7, and the client arrival rate ρ to 0.08. Additionally, an initial generation 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 α 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*.



(a) Client success ratio

(b) Average TTS

(c) Number of client requests

(d) Number of servers

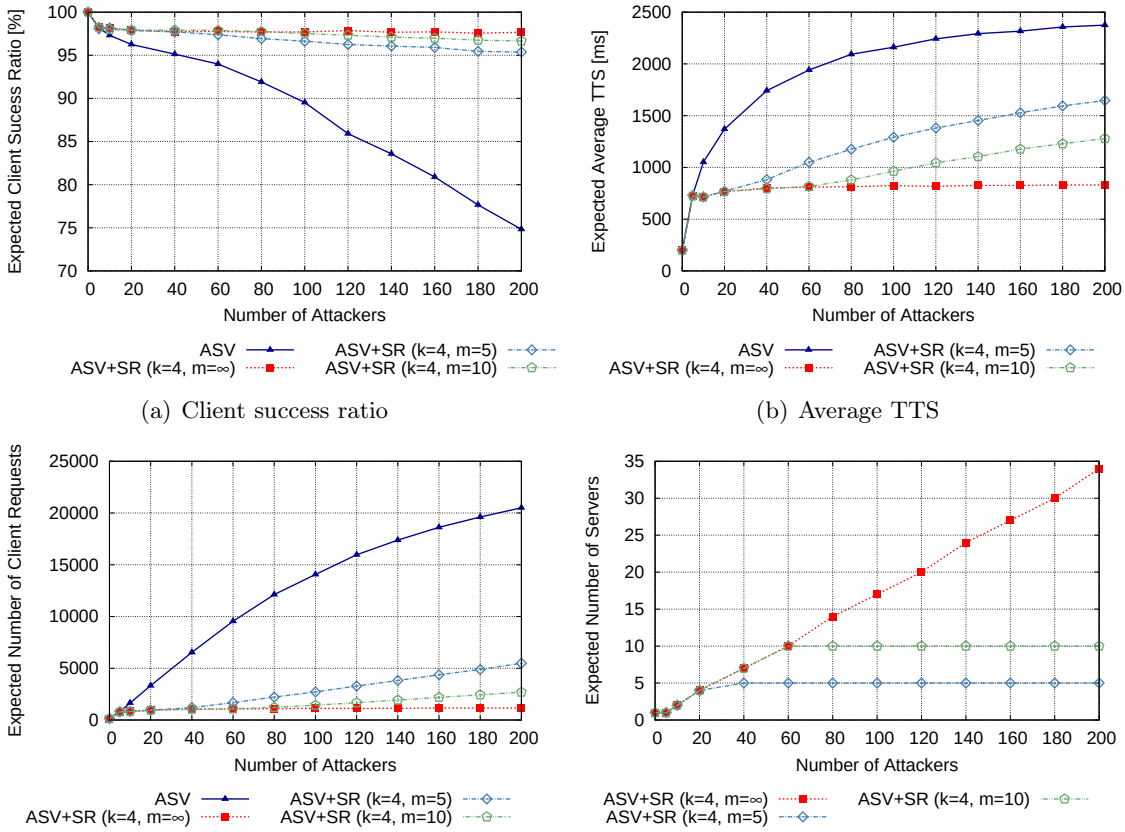
Figure 6.8.: Performance of the ASV+SR protocol with a varying load factor k and no resource bounds

Unlimited Resources

Figure 6.8 shows the model checking results for a varying load factor k and no resource limits. As indicated by Figure 6.8(a), the ASV+SR 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 6.8(b) and 6.8(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+SR outperforms the ASV protocol in all of the performance indicators. However, this is achieved at the cost of provisioning new servers. Figure 6.8(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+SR with an overloading factor of $k = 32$ requires significantly fewer resources than with an overloading factor of $k = 4$.

Limited Resources

Figure 6.9 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



(a) Client success ratio

(b) Average TTS

(c) Number of client requests

(d) Number of servers

Figure 6.9.: Performance of the ASV$^+$SR protocol with a load factor of $k = 4$ and limited resources

6.9(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 6.9(b) and 6.9(c) show that the average TTS and the number of client requests behave in a way similar to that of the success ratio.

6.5. Related Work

Various DoS defense mechanisms exist with an important class of such defenses being *currency-based mechanisms*. These mechanisms are characterized by servers that, under attack conditions, demand payments from clients in some appropriate "currency" such as actual *money* [71], *CPU cycles* [1] (e.g., by solving a puzzle [115]), or, as in the case of ASV [66], *bandwidth*. The earliest bandwidth-based defense mechanism that has been proposed

was Selective Verification (SV) [57] which includes no mechanism for adaption (and as such imposes a high cost on the system). Besides ASV, the class of adaptive bandwidth-based defense mechanisms also includes the auction-based approach in [113].

The formal analysis of DoS defense mechanisms using probabilistic rewrite theories and statistical model checking has first been performed for the SV mechanism and TCP SYN floods-based DoS attacks [4]. A general framework for the formal analysis of DoS defense mechanisms was proposed in [74]; an information flow-based framework using the Security Protocols Process Algebra for the evaluation of DoS resistance is given in [69]. Other works on formal analysis of availability properties use branching-time logics [122, 70]. A formal analysis of a meta-object-based defense mechanism has been performed for cookie-based DoS-protection wrappers [33].

6.6. 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 5, 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 analyse important qualitative properties. In this chapter, we have formally analyzed quantitative properties of systems under a DDoS attack, demonstrating that the ASV and ASV$^+$SR 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 7

QoS Analysis of Cloud-based Publish/Subscribe Systems

Publish/Subscribe is an asynchronous message exchange pattern, in which producers and consumers of messages are loosely coupled and often communicate via brokers. Brokers are intermediaries that can select and forward the published information relevant to each consumer and allow for greater flexibility and scalability in such systems. Quality of service (QoS) properties such as an on-time delivery of published messages can be crucial for these systems. Additionally, it is the task of the service management to determine how many resources should be provisioned for such a system to guarantee certain quality of service goals. However, several parameters and the unreliability of best-effort networks, especially when deployed in a worldwide setting, make it difficult to analyze systems based on a Publish/Subscribe middleware. One further source of uncertainty, and challenge in order to ensure QoS system requirements, is the variability in the number of users of the service. An interesting research question is how to enrich a Publish/Subscribe architecture with Cloud-based dynamic resource provisioning mechanisms to better meet QoS requirements. In the following, we:

1. give an introduction to Publish/Subscribe systems (Section 7.1),
2. define the model of a stock exchange information system which is built on the foundation of a Publish/Subscribe middleware (Section 7.2),
3. specify a model of the stock exchange information system in Maude (Section 7.3),
4. formally analyze the stock exchange information system model using statistical model checking (Section 7.4),

5. extend the model with a Cloud-based mechanism to provision new brokers on demand (Section 7.5),
6. specify the Cloud-based stock exchange information system in Maude (Section 7.6), and finally
7. formally analyze the Cloud-based stock exchange information system using statistical model checking (Section 7.7).

It is the primary goal of this Chapter to show that Publish/Subscribe systems, and enhancements of such systems that take advantage of a Cloud computing infrastructure, can be specified based on the modularized actor model of Chapter 5 and that predictions about quality of service properties of the specified systems can be made using statistical analysis. This approach is helpful for the management of Cloud-based systems that rely on a Publish/Subscribe middleware in order to predict the satisfaction of contracts and to allocate resources ahead of time.

7.1. Introduction to Publish/Subscribe Systems

Publish/Subscribe is a message exchange pattern and a paradigm for scalable information dissemination where publishers send new events and subscribers get notified about the new events they care about asynchronously. This paradigm is often realized in distributed architectures such as Web-based architectures, which are asynchronous and loosely coupled by nature. The W3C [112] describes the need for such a paradigm in the WS-Eventing specification which defines a protocol for event-based Publish/Subscribe interaction of web services:

> *"Web services often want to receive messages when events occur in other services and applications. A mechanism for registering interest is needed because the set of Web services interested in receiving such messages is often unknown in advance or will change over time. This specification defines a protocol for one Web service (called a "subscriber") to register interest (called a "subscription") with another Web service (called an "event source") in receiving messages about events (called "notifications"). The subscriber can manage the subscription by interacting with a Web service (called the "subscription manager") designated by the event source."*
>
> — WS-Eventing, W3C Candidate Recommendation 28 April 2011

The basic Publish/Subscribe pattern is defined by three actions:

subscribe. A subscriber subscribes to a class of events that is defined by specific event characteristics.

publish. A publisher asynchronously publishes an event.

forward. Subscribers asynchronously receive published events that they are interest in, i.e., the event satisfies the characteristics the subscriber subscribed to.

In addition to subscribers and publishers, the basic system model defines a third intermediary entity, namely the event service, which manages subscriptions and forwards events. This service can, but is not limited to, be part of the publishers (denoted as *event sources* in the WS-Eventing specification):

> *"In some scenarios the event source itself manages the subscriptions it has created. In other scenarios, for example a geographically distributed publish-and-subscribe system, it might be useful to delegate the management of a subscription [to a different entity]."*
>
> — WS-Eventing, W3C Candidate Recommendation 28 April 2011

The Publish/Subscribe communication pattern is closely related to other interaction patterns. The observer design pattern [48] allows objects to observe a subject. The subject thereby manages a list of observers and automatically notifies them about a state change of the subject. Hence, the observer pattern is a special case of the Publish/Subscribe pattern where subscribers subscribe to any event and the event service is part of the publisher. The Publish/Subscribe pattern is also related to shared data spaces, such as the one defined by the Linda coordination language (see Section 4.1.1). Using this paradigm, entities in a distributed environment communicate and synchronize using shared data that is distributed across multiple localities. A shared data space often allows entities to watch for certain data in the shared space, which resembles a subscription in the Publish/Subscribe pattern.

Besides the WS-Eventing specification by the W3C, other examples for models of Publish/Subscribe architectures include the CORBA Notification Service [90] and the Java Message Service [91].

7.1.1. Three-dimensional decoupling

Several variations of the Publish/Subscribe pattern exist. The underlying foundation, which is also the strength, of all of these variants, is a three-dimensional decoupling in space, time, and synchronization [45].

Decoupling in space. Communicating parties do not need to know about each other. The only entity known to the subscribers and the publishers is the event service.

Decoupling in time. Communicating parties do not need to be participating in an interaction simultaneously. Events can be stored in the event service and be forwarded to subscribers at a later point in time.

Decoupling in synchronization. Communicating parties do not need to communicate synchronously. Publishers are not blocked while sending an event to the event service, and subscribers can be informed about an event asynchronously.

The loose coupling between producers and consumers of information leads to an increased scalability and flexibility of information dissemination systems. Additionally, a reduction in coordination and synchronization overhead makes the Publish/Subscribe pattern well suited for distributed architectures. As we shall argue in this work, these systems can greatly benefit from Cloud-based architectures to boost their QoS properties.

7.1.2. Types of event filtering

Subscribers specify the characteristics of the events that they are interested in by defining a *filter*. Two common ways of defining those characteristics are to use topic-based or content-based event filtering.

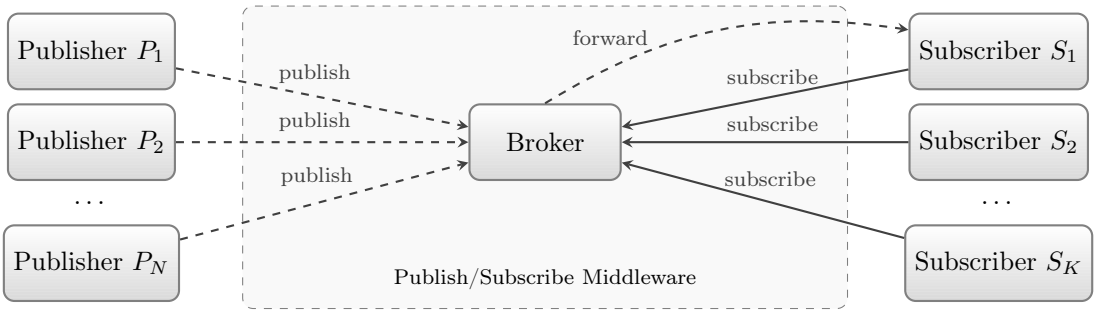


Figure 7.1.: Overview of a broker-based Publish/Subscribe middleware

Topic-based event filtering. In Publish/Subscribe systems that use topic-based event filtering, subscribers subscribe to a specific topic and receive events that are published using the same topic value. Hence, a topic defines a group of receivers which publishers broadcast to.

Content-based event filtering. In Publish/Subscribe systems that use content-based event filtering, subscribers subscribe using so-called predicate filters that define constraints on the contents or meta-data of events that they are interested in. For example, a simple way to define constraints for event contents that are made up of numerical attributes is to compare a specific content attribute (defined by an identifier) with a specific value. Predicate filters contain single constraints or logical combinations of constraints.

7.1.3. Broker-based publish/subscribe middleware solutions

Brokers are intermediaries between publishers and subscribers and take the role of the event service in broker-based Publish/Subscribe systems. A system that encapsulates a broker-based messaging service can be a middleware for complex applications in distributed systems. Figure 7.1 illustrates a broker-based Publish/Subscribe middleware. Such systems are not limited to a single broker but can be based on an overlay network of distributed brokers to provide fault tolerance, geographical distribution for reduced latencies, and scalability. Possibilities to coordinate and organize these brokers include tree-based [27, 32, 36] and peer-to-peer-based [103, 107] approaches.

7.1.4. QoS requirements and resource planning

Several Quality of Service (QoS) goals for Publish/Subscribe systems can be defined. Common QoS goals include high availability, message delivery guarantees, low time-to-service (TTS), confidentiality, etc. QoS goals are also strongly correlated with resource planning. Service providers and consumers often sign service level agreements (SLA) that give QoS guarantees. Consequently, it is vital for service providers to be able to predict the QoS properties of their system and to plan and allocate resources to guarantee certain QoS goals.

Some approaches have been proposed to manage QoS contracts for Publish/Subscribe-based systems [116] or to provide QoS-enabled information dissemination for time-critical

defense systems [114]. Other work focuses on the development of metrics to predict reliability and timeliness in Publish/Subscribe systems [93].

An area where QoS contracts are most crucial, and where maintaining QoS properties is not just a matter of costumer satisfaction, are systems like utility grids, such as the energy grid, or aviation systems. The Washington State University research on the GridStat project [117], is a Publish/Subscribe middleware framework that provides flexible, robust, timely, and secure delivery of operational status information for the electric power grid.

7.2. A Stock Exchange Information System

In this section we define a concrete model of a stock exchange information system that provides current trading information similar to Google finance [55] or Yahoo! Finance [121]. The system is built upon a content-based Publish/Subscribe middleware. Subscribers can subscribe to trading events that show specific characteristics, such as being related to a certain listing or being a high-volume trade. The middleware consists of several brokers that are located across the world. Each trading event has a specific lifetime, and the event information is only useful to subscribers if they receive it within its lifetime. In our model, we assume a high frequency[1] (one event per second) and low lifetime duration (as low as one second) event dissemination. This represents a challenging setting, as communication latencies in a best-effort worldwide network already consume a big fraction, if not all, of the lifetime. Additionally, the system is flooded with events which may lead to high workloads at the brokers. We ask three questions regarding this kind of stock exchange information system:

1. Can a QoS guarantee be made regarding the timely delivery of events?
2. Does the system scale, i.e., how many subscribers can the system handle without violating the aforementioned QoS guarantee?
3. How many resources should the service provider provision, and what are the expected operating costs?

In the following, we define the system model in more detail.

7.2.1. Events

Events published in the stock exchange information system contain information about current trading activity. An event $E \in \mathbb{E}$ is defined by the tuple (a_E, l_E, p_E) and encapsulates content attributes a_E, a lifetime duration l_E, and the publication time p_E. The content attributes a_E contain the values:

Listing $L_E \in \mathbb{N},\ 1 \leq L_E \leq 100$, is distributed according to a discrete uniform distribution and specifies which listing (symbol) the trading information is about,

[1] This system should not be confused with the high frequency trading systems that today operate in milli- or micro-seconds [58]. Instead, we assume an information system where (potentially collapsed) trading information is issued in a matter of seconds. We will see that this, together with the fact that the system operates worldwide, already causes constraints that a Publish/Subscribe system struggles to fulfil.

Price $P_E \in \mathbb{R}$, is distributed according to a normal distribution with mean 1000.0 and standard deviation 100.0 ($\mathcal{N}(1000.0, 10000.0)$) and represents the price of the trading activity,

Volume $V_E \in \mathbb{N}$, is distributed according to a discrete Pareto distribution and represents the volume of the trading activity, and

Type $T_E \in \{BUY, SELL\}$, is distributed according to a Bernoulli distribution with success probability 0.5 and specifies whether stock has been bought or sold.

We assume that the event topic is always equal to the listing value, and that the price, volume, and type values are the per-topic attributes.

Furthermore, we assume that all entities in the system have access to a globally synchronized clock that provides a global time value gt. The lifetime value l_E is either 1.0 s, 30.0 s, or 60.0 s. A subscriber receives an event on time if the time that it takes to deliver the event is less than its lifetime duration, i.e., $gt < p_E + l_E$.

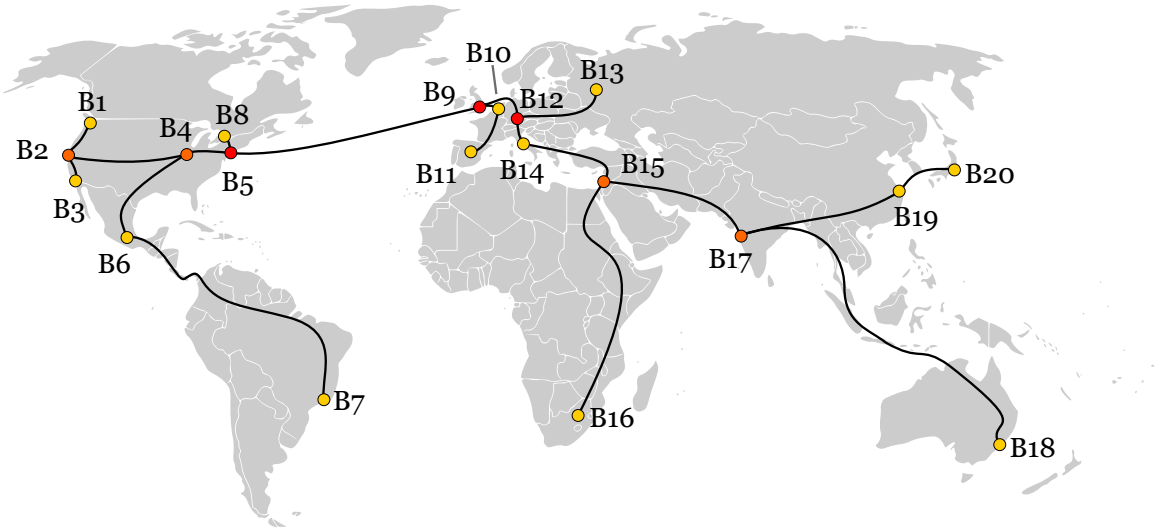
A subscriber $S \in \mathbb{S}$ is not interested in all events that are published in the stock exchange information system and therefore defines a predicate filter PF_S. Predicate filters are distributed according to the probability distribution

$$
\begin{aligned}
\mathbf{20\%}\ & PF^1 := \text{Listing} = L \\
\mathbf{5\%}\ & PF^2 := \text{Listing} = L \ \wedge\ \text{Volume} \geq V \\
\mathbf{2.5\%}\ & PF^3 := \text{Listing} = L \ \wedge\ \text{Volume} \geq V \ \wedge\ \text{Type} = \text{BUY} \\
\mathbf{2.5\%}\ & PF^4 := \text{Listing} = L \ \wedge\ \text{Volume} \geq V \ \wedge\ \text{Type} = \text{SELL} \\
\mathbf{10\%}\ & PF^5 := \text{Volume} \geq V \\
\mathbf{10\%}\ & PF^6 := \text{Listing} = L \ \wedge\ \text{Price} \geq P \\
\mathbf{10\%}\ & PF^7 := \text{Listing} = L \ \wedge\ \text{Price} \geq P \ \wedge\ \text{Type} = \text{BUY} \\
\mathbf{10\%}\ & PF^8 := \text{Listing} = L \ \wedge\ \text{Price} \geq P \ \wedge\ \text{Type} = \text{SELL} \\
\mathbf{10\%}\ & PF^9 := \text{Listing} = L \ \wedge\ \text{Price} \leq P \\
\mathbf{10\%}\ & PF^{10} := \text{Listing} = L \ \wedge\ \text{Price} \leq P \ \wedge\ \text{Type} = \text{BUY} \\
\mathbf{10\%}\ & PF^{11} := \text{Listing} = L \ \wedge\ \text{Price} \leq P \ \wedge\ \text{Type} = \text{SELL}
\end{aligned}
$$

where the listing, the volume, the price, and the type are distributed according to their respective probability distributions, which have already been specified in the definition of events.

7.2.2. Network

The network model of the stock exchange information system is an acyclic broker tree with 20 individual brokers ($B \in \{B_1, B_2, \ldots, B_{20}\}$). Each of the brokers is connected to at least one neighbor. Each connected pair of brokers is linked via an asynchronous communication link which has a static latency. Figure 7.2 shows the distribution of the brokers across the world and the respective static link latencies. The setup of the brokers is expected to lead



Link	Latency
$B_1 \leftrightarrow B_2$	20 ms
$B_2 \leftrightarrow B_3$	20 ms
$B_2 \leftrightarrow B_4$	60 ms
$B_4 \leftrightarrow B_5$	20 ms
$B_4 \leftrightarrow B_6$	60 ms
$B_5 \leftrightarrow B_8$	20 ms
$B_5 \leftrightarrow B_9$	140 ms
$B_6 \leftrightarrow B_7$	160 ms
$B_9 \leftrightarrow B_{10}$	10 ms
$B_9 \leftrightarrow B_{12}$	20 ms

Link	Latency
$B_{10} \leftrightarrow B_{11}$	30 ms
$B_{12} \leftrightarrow B_{13}$	80 ms
$B_{12} \leftrightarrow B_{14}$	20 ms
$B_{14} \leftrightarrow B_{15}$	80 ms
$B_{15} \leftrightarrow B_{16}$	160 ms
$B_{15} \leftrightarrow B_{17}$	80 ms
$B_{17} \leftrightarrow B_{18}$	180 ms
$B_{17} \leftrightarrow B_{19}$	120 ms
$B_{19} \leftrightarrow B_{20}$	30 ms

Mean latency between brokers: $\sim$ 68.95 ms

Figure 7.2.: Worldwide network setup of the stock exchange Publish/Subscribe system (red brokers indicate very high expected load; orange brokers indicate high expected load; yellow brokers indicate normal expected load)

to an unequal distribution of event processing load among the brokers, e.g., we expect that brokers that have three incoming links and are close to the center of the tree will have a much higher load than a broker that is a leaf of the tree.

The setup further consists of 100 event publishers $P \in \mathbb{P}$ and a varying number of subscribers $S \in \mathbb{S}$ (1–1000). A publisher or subscriber is connected to exactly one broker in the system. This broker is called its *home broker*. Publishers and subscribers are connected to their respective home broker via an asynchronous communication link. This link has a dynamic latency of around 30.0 ms (in our model, we assume that the latency is distributed

according to $\mathcal{N}(30.0, 64.0)$).

We further assume that links in our model have no bandwidth limit and that the amount of messages that are currently transferred over a link do not affect its latency. The chosen latency values for the communication links between brokers, subscribers and brokers, and publishers and brokers follow the results of real-world measurements published in [65, 105] and our own measurements with PlanetLab [92].

7.2.3. Behavior of subscribers, publishers, and brokers

In the following we describe the behavior of subscribers, publishers, and brokers in our stock exchange information system model.

Subscribers:

1 to 1000 subscribers $S \in \mathbb{S}$ can join the network by randomly choosing one of the 20 brokers as their home broker according to a uniform distribution. After joining the network, the subscribers send a subscription to their respective home broker. A subscription is a message that is defined by the tuple (S, PF_S) and contains an address (initially the address of the subscriber) and a predicate filter (the predicate filter of the subscriber).

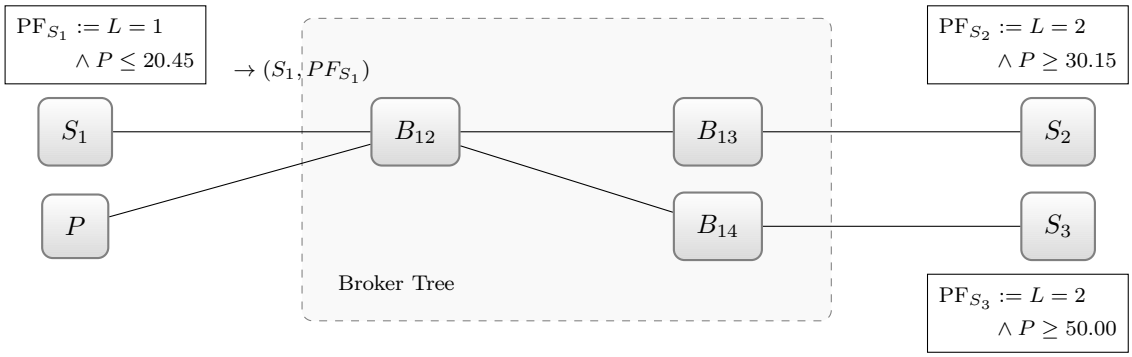
Publishers:

100 publishers $P \in \mathbb{P}$ join the network and randomly pick one of the 20 brokers as their home broker according to a uniform distribution. Then, they start publishing randomly generated events to their respective home broker. A publication is defined by the tuple (P, E) and encapsulates an address (initially the address of the publisher) and an event. In our model, publishers publish at a rate of one event per second.

Subscription handling:

When a broker B receives a subscription (S, PF_S), it stores the predicate filter PF_S and the address S in a routing table which maps from predicate filters to addresses. The broker then propagates the subscription to its neighboring brokers as a message (B, PF_S). These brokers, in turn, store the mapping between the predicate filter PF_S and the broker B in their respective routing tables. Each neighboring broker B' then propagates a message (B', PF_S) to their neighbors, except for broker B. This process is repeated until, hop by hop, the whole broker tree has been informed about the new subscription.

If a mapping between a predicate filter PF and an address A is already present in the routing table of a broker and the broker receives a subscription that contains the same predicate filter PF but a different destination A', the mapping in the routing table is updated to map from the predicate filter PF to both, A and A'. Indeed, a predicate filter can map to an arbitrary list of addresses. If a mapping between a predicate filter PF and an address A is already present in the routing table of a broker and the broker receives the subscription (A, PF) that suggests the same mapping, the broker does not alter its routing table and only forwards the subscription, i.e., the routing table is free of duplicates. Again, as the predicate filter can map to a list of addresses, it is sufficient that the list of addresses contains the address A.



Routing tables prior to the subscription of S_1

Broker	Routing Table
B_{12}	$L =$ "B" $\wedge P \geq 30.15 \mapsto B_{13}$ $L =$ "B" $\wedge P \geq 50.00 \mapsto B_{14}$
B_{13}	$L =$ "B" $\wedge P \geq 30.15 \mapsto S_2$
B_{14}	$L =$ "B" $\wedge P \geq 50.00 \mapsto S_3$

Routing tables after the subscription of S_1

Broker	Routing Table
B_{12}	$L =$ "B" $\wedge P \geq 30.15 \mapsto B_{13}$ $L =$ "B" $\wedge P \geq 50.00 \mapsto B_{14}$ $L =$ "A" $\wedge P \leq 20.45 \mapsto S_1$
B_{13}	$L =$ "B" $\wedge P \geq 30.15 \mapsto S_2$ $L =$ "A" $\wedge P \leq 20.45 \mapsto B_{12}$
B_{14}	$L =$ "B" $\wedge P \geq 50.00 \mapsto S_3$ $L =$ "A" $\wedge P \leq 20.45 \mapsto B_{12}$

Figure 7.3.: S_1 subscribes to the broker tree by sending a subscription message to its home broker B_{12}

In our model we assume that a subscription and the following subscription propagation are processed instantaneously.

Example 7.1: Subscription propagation in a subset of the stock exchange information system
To illustrate how subscription propagation works in our model, we consider the subset of the stock exchange information system consisting of the brokers B_{12}, B_{13}, and B_{14}. The subscribers S_2 and S_3 are already subscribed to the system. S_1 subscribes to the system by sending the subscription message (S, PF_{S_1}) to its home broker B_{12}. Figure 7.3 shows the broker tree and the routing tables prior to and after the subscription of S_1.

Publication handling:

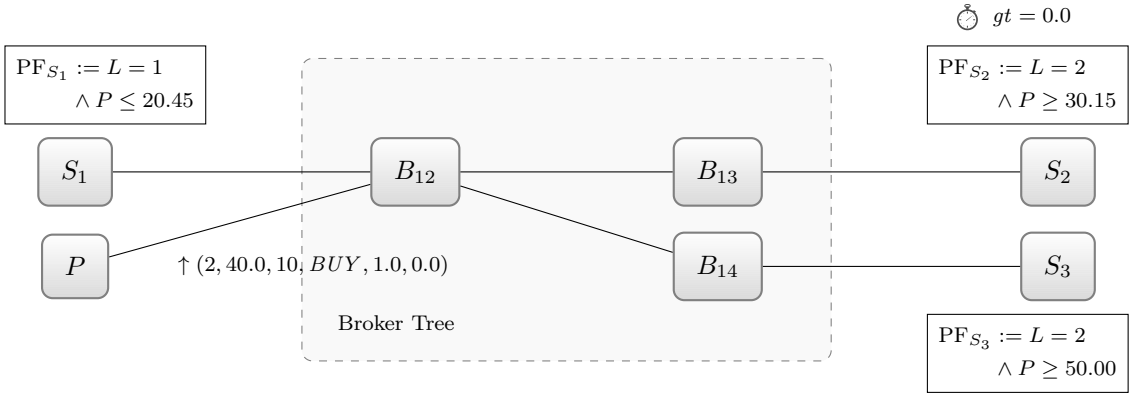
Upon receiving a publication message of the event $E = (a_E, l_E, p_E)$, a broker first checks if the event has expired, i.e., if $gt < l_E + p_E$. If it has, the broker drops the message, because the content is already outdated and useless to subscribers. Otherwise, the event and its sender are enqueued in an event queue. In our model the queue is theoretically unbounded in size. The broker processes the events in the event queue in a FIFO order. An event is processed by evaluating it against the predicate filters in the routing table. In our model we assume that a broker needs 1 ms of processing time to evaluate an event against a single predicate filter. Hence, processing of an event at a broker whose routing table contains k predicate filter entries takes k ms. When the event passes a predicate filter of the routing table, the event is forwarded to the addresses the filter maps to. Events are only propagated forward in the broker tree. Thus, if one of the mapped addresses is the address of the sender

of the event, the event is not forwarded to that address. Additionally, an event is only forwarded once per outgoing link. The forwarded publication message contains the event and the broker's address as the sender.

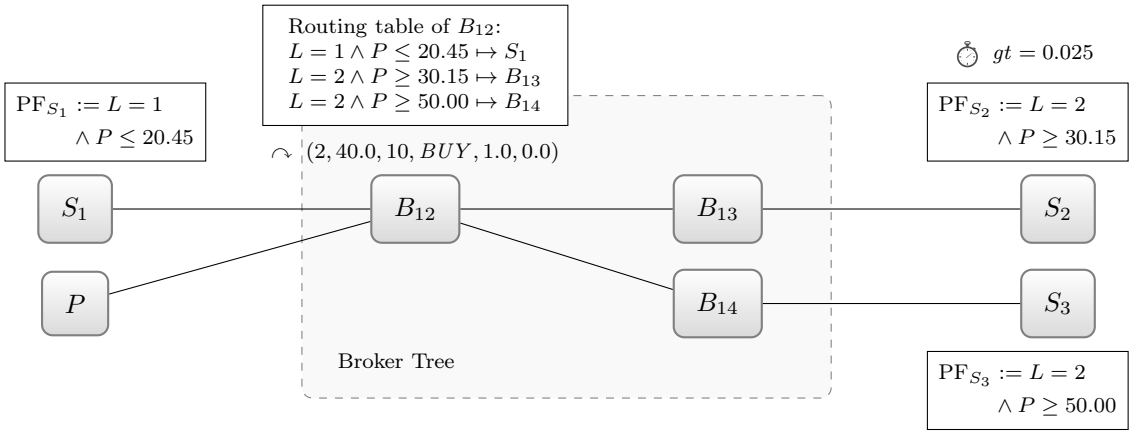
Example 7.2: Event publishing in a subset of the stock exchange information system

In the following we show an example of how a published event is forwarded by brokers in a broker tree. In this example, a subset of the broker, namely, the brokers B_{12}, B_{13}, and B_{14} form the broker tree. The three subscribers S_1, S_2, and S_3 are subscribed with their predicate filters $PF_{S_1} := L = 1 \wedge P \leq 20.45$, $PF_{S_2} := L = 2 \wedge P \geq 30.15$, and $PF_{S_3} := L = 2 \wedge P \geq 50.00$. Subscriber S_1 is subscribed to broker B_{12}, subscriber S_2 to broker B_{13}, and subscriber S_3 to broker B_{14}. Additionally, the publisher P publishes the new event $(2, 40.0, 10, BUY, 1.0, 0.0)$ to its home broker B_{12}. We assume that the global time gt starts at 0.0.

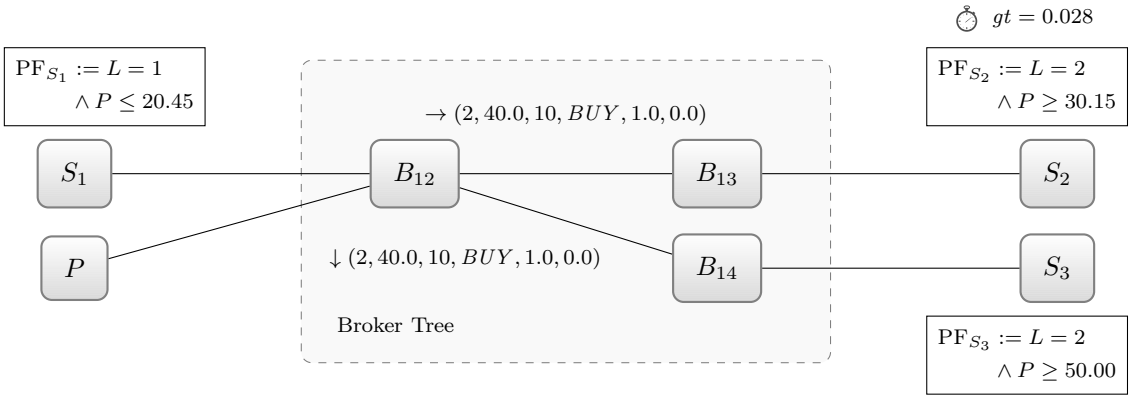
First, P sends the event $(2, 40.0, 10, BUY, 1.0, 0.0)$ to its home broker B_{12}. The current latency of the link $P \leftrightarrow B_{12}$ is 25 ms.



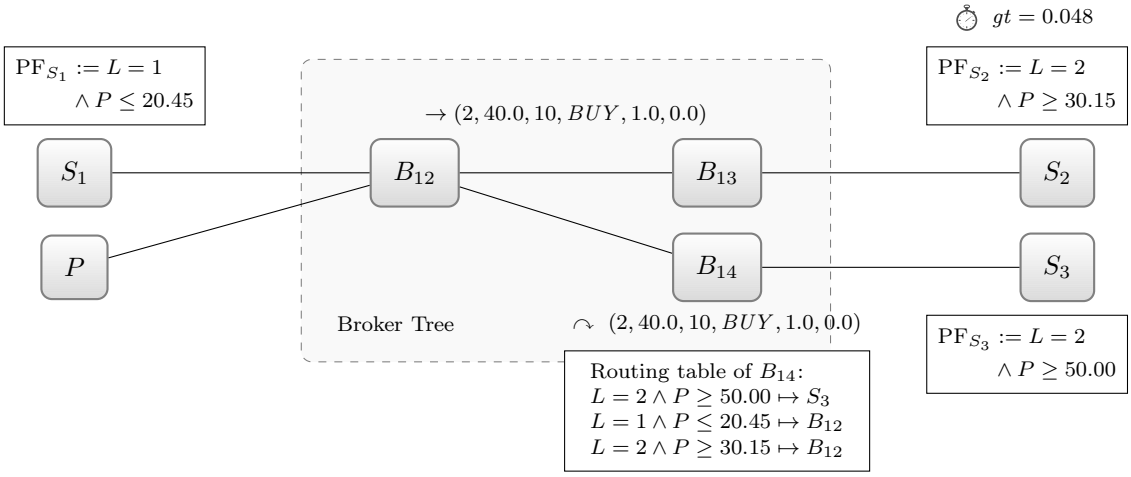
The global time advances to 0.025. Then, B_{12} receives the event and checks if the received event has expired (0.0+1.0 > 0.025). As the event has not expired, B_{12} processes the received event using its routing table. The processing time is 3 ms, because three entries are in the routing table of broker B_{12}.



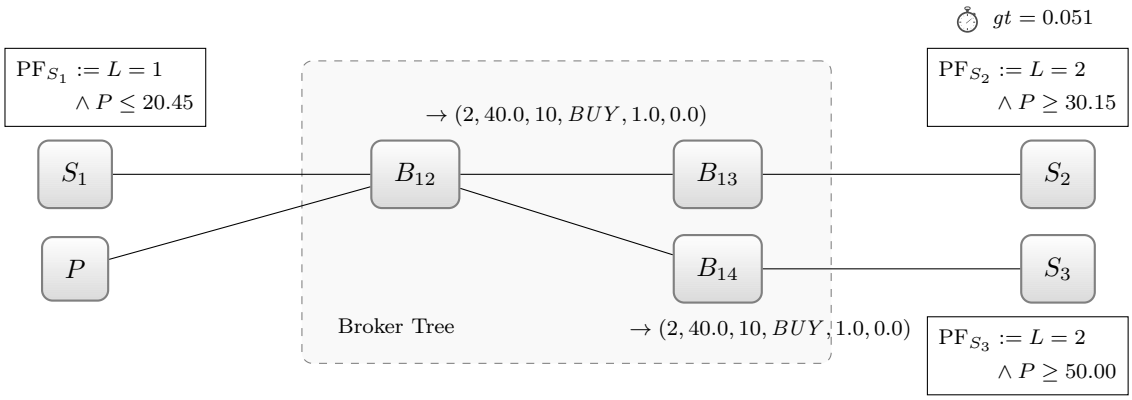
The global time advances to 0.028. B_{12} finishes the processing of the event and forwards it to B_{13} and B_{14} because the predicate filters that map to these addresses passed the event. The event is not forwarded to S_1, because the listing that is defined in the predicate filter already differs from the event's listing. The latency of the link $B_{12} \leftrightarrow B_{13}$ is 80 ms and the latency of the link $B_{12} \leftrightarrow B_{14}$ is 20 ms.



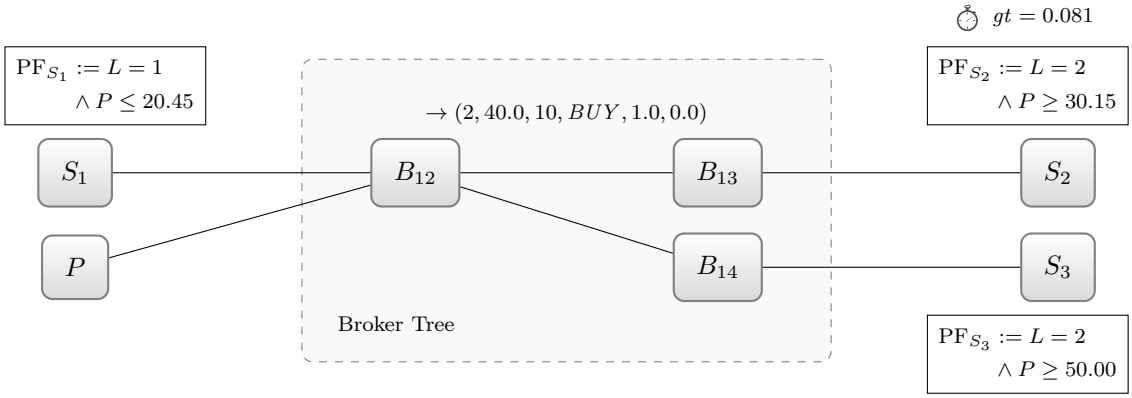
The global time advances to 0.048. Then, B_{14} receives the event and checks if the event has expired ($0.0 + 1.0 > 0.048$). As the event has not expired, B_{14} processes the received event using its routing table. The processing time is 3 ms because three entries are in the routing table of broker B_{14}. The message that forwards the event from broker B_{12} to broker B_{13} is still on its way.



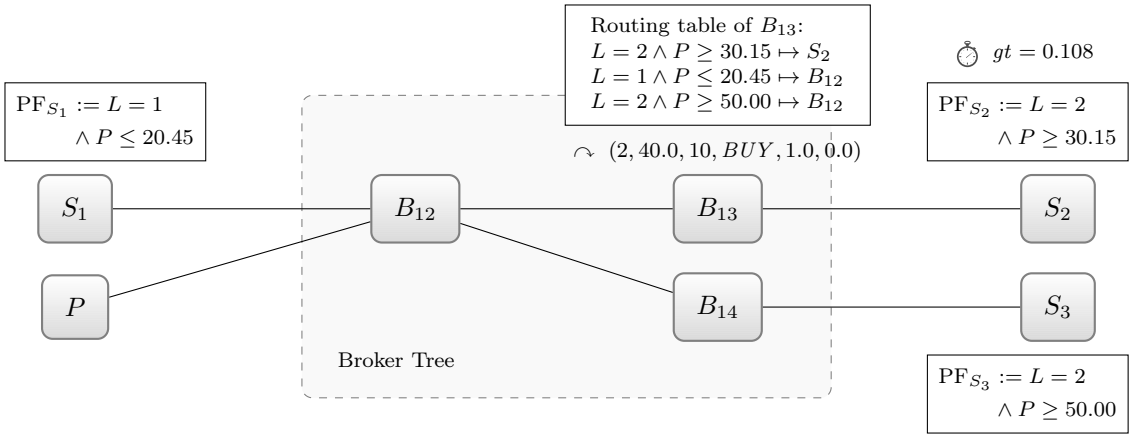
The global time advances to 0.051. B_{14} finishes the processing of the event and forwards it to S_3. The current latency of the link $B_{14} \leftrightarrow S_3$ is 30 ms. In the meantime, the message that forwards the event from broker B_{12} to broker B_{13} is still on its way.



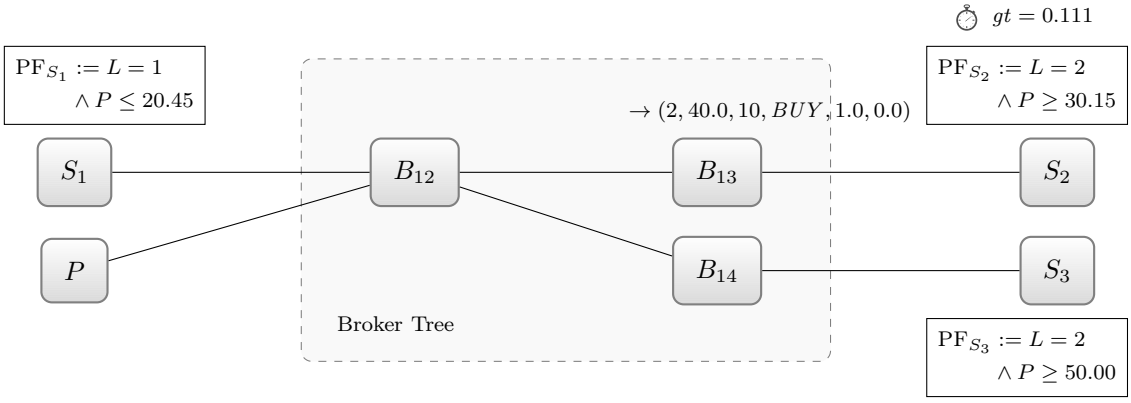
The global time advances to 0.081 and S_3 receives the event.



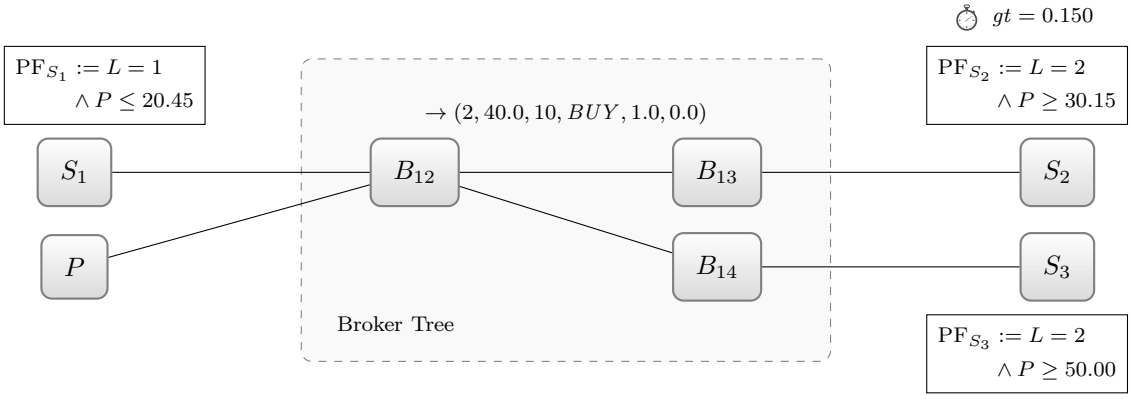
The global time advances to 0.108. B_{13} receives the event and checks if it has expired ($0.0 + 1.0 > 0.108$). As the event has not expired, B_{13} starts processing the event. The processing time is 3 ms since three entries are in the routing table of broker B_{13}.



The global time advances to 0.111. B_{13} finishes processing of the event and forwards it to S_2. The current latency of the link $B_{13} \leftrightarrow S_2$ is 39 ms.



The global time advances to 0.150. Finally, S_2 receives the event.



7.3. Specification of the Stock Exchange Information System in Maude

In this section we describe the Maude-based specification of the stock exchange information system that was described at a high level in the previous section. We base the specification on the modularized actor model that is described in Chapter 5. Additionally, we use the generic module *SAMPLER* (see Appendix C.1).

7.3.1. Overview of the Maude specification

Figure 7.4 gives an overview of the Maude modules comprising the specification and their structural dependencies. The functional modules *EVENT*, *ROUTING*, *NETWORK* define the basic syntax for events, routing tables, and the network graph, and provide operators to interact with the defined concepts. The functional module *PARAMS* defines the model parameters. Finally, the functional module *CONFIGURATION* defines the actor types, attributes, and messages used by the stock exchange information system model. The modules *PUBLISHER*, *SUBSCRIBER*, *BROKER*, and *GENERATOR* specify the behavioral rules of the actors in the model. The module *PUBLISH-SUBSCRIBE* defines the initial state of the system. The modules *STOCK-EXCHANGE-PARAMS* and *STOCK-EXCHANGE* provide

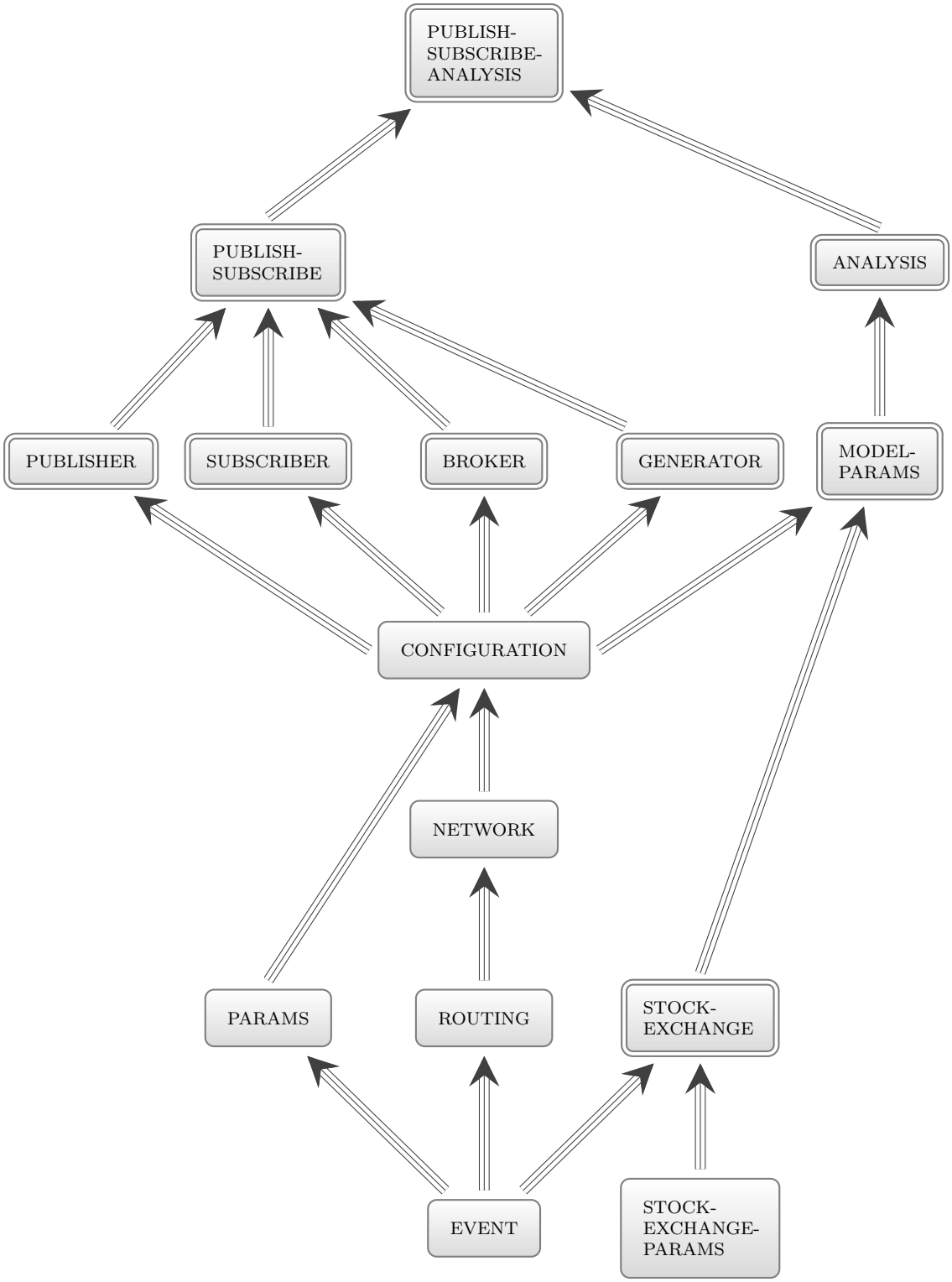


Figure 7.4.: Overview of the Maude specification of the stock exchange information system model

generators to randomly create trading events and respective predicate filters. Finally, the modules *MODEL-PARAMS*, *ANALYSIS*, *PUBLISH-SUBSCRIBE-ANAYLSIS* instantiate the model parameters and are used for the formal analysis of the system model.

7.3.2. Description of the modules

In the following, we give an in-depth description of each of the modules.

The module *EVENT*

An *event* is a triple whose first component is a pair, of sort `ContentAttributes`, of a topic and a list of attributes for the event, and whose second and third components are floating point numbers respectively describing the event's lifetime duration and its publishing time. Events are a message content (of sort `Content`)

```
sort Event ContentsAttributes .
subsort Event < Content .
```

and are constructed by the operator

```
op event : ContentsAttributes Float Float -> Event [ctor] .
```

An event is said to be on time if the current global time is less or equal than the sum of the publishing time and the lifetime duration.

The set of attributes of an event is a set of floating point values of sort `Floats`. A topic (of sort `Topic`) is thereby a special kind of value, i.e., a subsort of sort `Value`. Both topics and individual floating point numbers are values of sort `Value`.

```
sort Floats Topic Value .
subsort Float < Floats .
subsort Float Topic < Value .
```

The set of floating point per-topic attribute values is constructed by the operators

```
op nilFloats : -> Floats .
op _,_ : Floats Floats -> Floats [ctor assoc id: nilFloats] .
```

Topics are constructed by the operator

```
op topic : Nat -> Topic [ctor] .
```

which takes a natural number as argument. The constructor for contents attributes

```
op [_,_] : Topic Values -> ContentsAttributes [ctor gather(e E)] .
```

concatenates and encapsulates a topic and the per-topic attributes in a single term.

Event queues (of sort `QueueWithSize`) are used by brokers to store events for future processing. An event queue essentially is a contents queue (of sort `ContentsQueue`) which itself consists of message contents concatenated with the queue size.

```
sort QueueWithSize ContentsQueue .
subsort Content < ContentsQueue .
```

The operators

```
op nilQueue : -> ContentsQueue [ctor] .
op _;_ : ContentsQueue ContentsQueue -> ContentsQueue [ctor assoc id: nilQueue] .
op [_,_] : Nat ContentsQueue -> QueueWithSize [ctor] .
```

construct event queues.

Predicate filters (of sort `PredicateFilter`) define what information a subscriber is interested in. Predicate filters consist of filter rules (of sort `FilterRule`) that specify a filter for a specific attribute (defined by a term of sort `ValuePosition`) of an event's contents attributes. A filter rule compares the value in the contents attributes with the value in the filter rule using a binary operator (of sort `Operator`).

```
sort PredicateFilter FilterRule ValuePosition Operator .
subsort FilterRule < PredicateFilter .
```

In our model, we use the operators `<=`, `>=`, and `==`, which are already predefined for floating point numbers in Maude.

```
ops <= >= == : -> Operator [ctor] .
```

A value position, i.e., the position in the list of floats in the contents attributes of an event will always correspond, for a given event topic, to a predetermined attribute of the event, so that looking up that position in the list of floats will yield the attribute's value. Positions are constructed by the operator

```
op pos : Nat -> ValuePosition [ctor] .
```

A filter rule is constructed by the operators

```
op nilFilter : -> FilterRule [ctor] .
op ___ : ValuePosition Operator Value -> FilterRule [ctor] .
```

Finally, predicate filters are constructed by the operator

```
op _;_ : PredicateFilter PredicateFilter -> PredicateFilter
  [ctor assoc comm id: nilFilter] .
```

and are queried by the operator

```
op evalFilter : ContentAttributes PredicateFilter -> Bool .
```

which takes contents attributes and a predicate filter as arguments and returns a Boolean value indicating whether the predicate filter accepts (`true`) or rejects (`false`) the contents attributes. For the definition of the behavior of the evaluation operator, the variables

```
vars EID POS POS' TID : Nat .
var T : Topic .
vars VAL VAL' : Value .
var VALS : Values .
var D : Float .
var PF : PredicateFilter .
var FR : FilterRule .
var OP : Operator .
```

are used. The auxiliary operators

```
op evalRule : Values FilterRule Nat -> Bool .
op compare : Value Operator Value -> Bool .
```

evaluate a single rule and a single pair of values. The operator `compare` is defined by the equations

```
ceq compare(VAL, OP, VAL') = VAL <= VAL' if OP := <= .
ceq compare(VAL, OP, VAL') = VAL >= VAL' if OP := >= .
ceq compare(VAL, OP, VAL') = VAL == VAL' if OP := == .
```

which simply define the comparison to behave like the comparison of floating point values. The operator to evaluate a single rule `evalRule` is defined by the equation

```
eq evalRule((VAL, VALS), pos(POS) OP VAL', POS') =
  if POS == POS' then
    compare(VAL, OP, VAL')
  else
    evalRule(VALS, pos(POS) OP VAL', s(POS')) fi .
```

which applies the filter rule to the appropriate value of the contents attributes. Finally, the operator `evalFilter` is defined by the equations

```
eq evalFilter([T, VALS], nilFilter) = true .
eq evalFilter([T, VALS], (pos(0) == VAL) ; PF) =
  T == VAL and evalFilter([T, VALS], PF) .
eq evalFilter([T, VALS], FR ; PF) =
  evalRule(VALS, FR, 1) and evalFilter([T, VALS], PF) .
```

which recursively apply the single filter rules of the predicate filter to the specified content attributes.

The module *ROUTING*

The module *ROUTING* defines the routing table (of sort `RoutingTable`) of a broker. Routing tables consist of single routings (of sort `Routing`), which map a predicate filter to an address list. A term of sort `TableWithSize` concatenates a routing table with its size.

```
sort TableWithSize RoutingTable Routing .
subsort Routing < RoutingTable .
```

Single routings are constructed by the operators

```
op nilRouting : -> RoutingTable [ctor] .
op _->_ : PredicateFilter AddressList -> Routing [ctor] .
```

A routing table is constructed by the operator

```
op _;_ : RoutingTable RoutingTable -> RoutingTable
  [ctor assoc comm id: nilRouting] .
```

Finally, terms of sort `TableWithSize` are constructed by the operator

```
op [_,_] : Nat RoutingTable -> TableWithSize [ctor] .
```

The module *NETWORK*

The module *NETWORK* defines communication links and the structure of the network. Links are constructed by the operator

```
sort Link Latency .
subsort Float < Latency .
op [_,_] : Address Latency -> Link [ctor] .
```

which takes an address and a latency (of sort `Latency`) as arguments. In our model, latencies are defined by floating point numbers and represent the latency value in seconds.

The network is defined as an adjacency list that represents all edges of the network graph. A link list (of sort `LinkList`) is defined to concatenate links.

```
sort LinkList .
subsort Link < LinkList .
op _;_ : LinkList LinkList -> LinkList [ctor assoc comm] .
```

Mappings (of sort `Mapping`) between an address and a link list are the component of the adjacency list. Mappings are constructed by the operator

```
sort Mapping .
op _->_ : Address LinkList -> Mapping [ctor] .
```

Finally, an adjacency list (of sort `AdjacencyList`) is constructed by a mapping or the concatenation of mappings.

```
sort AdjacencyList .
subsort Mapping < AdjacencyList .
op _;_ : AdjacencyList AdjacencyList -> AdjacencyList [ctor assoc comm] .
```

The constant

```
op network : -> AdjacencyList .
```

defines the network configuration of the model and is considered global knowledge. The network can be queried using the operator

```
op latency : Address Address -> Latency [memo] .
```

which determines the latency between two given addresses. For the description of the behavior of the operator, the variables

```
var FROM A TO : Address .
var LL : LinkList .
var AL : AdjacencyList .
var L : Latency .
```

are used. The auxiliary operators

```
op getList : AdjacencyList Address Address -> Latency .
op getLink : LinkList Address -> Latency .
```

are called by the `latency` operator. The equations

```
eq getList(FROM -> LL, FROM, TO) = getLink(LL, TO) .
eq getList((A -> LL) ; AL, FROM, TO) =
  if A == FROM then
    getLink(LL, TO)
  else
    getList(AL, FROM, TO) fi .
```

define the operator `getList`, which, for a given source address, retrieves the mapped link list from the adjacency list and calls the operator `getLink`. The operator `getLink` which takes the list and the destination address as arguments proceeds as described by the equations

```
eq getLink([TO, L], TO) = L .
eq getLink([A, L] ; LL, TO) =
  if A == TO then
    L
  else
    getLink(LL, TO) fi .
```

and retrieves the appropriate latency of the link between the two addresses. Finally, the operator `latency` is just an abbreviation that abstracts from the adjacency list that is used.

```
eq latency(FROM, TO) = getList(network, FROM, TO) .
```

In the Maude specification, the network described in Section 7.2 and shown in Figure 7.2 is defined by the equation (the adjacency list) that defines the `network` constant.

```
eq network =
  (1 -> [2, 0.02]) ;
  (2 -> [1, 0.02] ; [3, 0.02] ; [4, 0.06]) ;
  (3 -> [2, 0.02]) ;
  (4 -> [2, 0.06] ; [5, 0.02] ; [6, 0.06]) ;
  (5 -> [4, 0.02] ; [8, 0.02] ; [9, 0.14]) ;
  (6 -> [4, 0.06] ; [7, 0.16]) ;
  (7 -> [6, 0.16]) ;
  (8 -> [5, 0.02]) ;
  (9 -> [5, 0.14] ; [10, 0.01] ; [12, 0.02]) ;
  (10 -> [9, 0.01] ; [11, 0.03]) ;
  (11 -> [10, 0.03]) ;
  (12 -> [9, 0.02] ; [13, 0.08] ; [14, 0.02]) ;
  (13 -> [12, 0.08]) ;
  (14 -> [12, 0.02] ; [15, 0.08]) ;
  (15 -> [14, 0.08] ; [16, 0.16] ; [17, 0.08]) ;
  (16 -> [15, 0.16]) ;
  (17 -> [15, 0.08] ; [18, 0.18] ; [19, 0.12]) ;
  (18 -> [17, 0.18]) ;
  (19 -> [17, 0.12] ; [20, 0.03]) ;
  (20 -> [19, 0.03]) .
```

The module *PARAMS*

In the module *PARAMS*, the constant operators

```
op LIMIT : -> [Float] .
op initDelay : -> [Float] .

op numBrokers : -> [Nat] .
op numSubscribers : -> [Nat] .
op numPublishers : -> [Nat] .

op pubRate : -> [Float] .
op lifetime : -> [Float] .
op procRate : -> [Float] .

op latency : -> [Float] .
op latencyMean : -> [Float] .
op latencyStdDev : -> [Float] .

op predicateFilterGenerator : -> [PredicateFilter] .
op contentsAttributesGenerator : -> [ContentsAttributes] .
```

are defined, which represent the parameters of the Publish/Subscribe system model. The model, even though it is specifically made for the stock exchange system, is specified in a generic way. Therefore, the foundations of the model can be reused for other specifications of broker trees and content-based Publish/Subscribe systems.

The module *CONFIGURATION*

The module *CONFIGURATION* defines the actor types

```
op Subscriber : -> ActorType [ctor] .
op Publisher : -> ActorType [ctor] .
op Broker : -> ActorType [ctor] .
```

Additionally, the two actor types

```
op SubscriberGenerator : -> ActorType [ctor] .
op PublisherGenerator : -> ActorType [ctor] .
```

are defined, which represent the actor types of the actors that respectively generate publishers and subscribers.

The home broker attribute is constructed by the operator

```
op homeBroker:_ : Address -> Attribute [gather(&)] .
```

which takes the address of the home broker as an argument. This attribute is used by both, the subscribers and the publishers.

The attributes

```
op predicateFilter:_ : PredicateFilter -> Attribute [gather(&)] .
op recCnt:_ : Nat -> Attribute[gather(&)] .
op avgTTS:_ : Float -> Attribute[gather(&)] .
```

are specific attributes of the subscribers. The attribute `predicateFilter` contains a subscriber's predicate filter. The attributes `recCnt` and `avgTTS` keep track of the number of incoming events and the average time it takes for these events to be delivered by the system.

The attributes

```
op routingTable:_ : TableWithSize -> Attribute [gather(&)] .
op neighbors:_ : AddressList -> Attribute [gather(&)] .
op subscribers:_ : AddressList -> Attribute [gather(&)] .
op eventQueue:_ : QueueWithSize -> Attribute [gather(&)] .
op drops:_ : Nat -> Attribute [gather(&)] .
op forwards:_ : Nat -> Attribute [gather(&)] .
op sent:_ : Nat -> Attribute [gather(&)] .
```

are specific attributes of the brokers. The attribute `routingTable` holds the routing table of the broker, the attribute `neighbors` holds an address list of neighboring brokers, the attribute `subscribers` holds an address list of subscribers that are subscribed to the broker, and the attribute `eventQueue` holds the broker's event queue. Finally, the attributes `drops`, `forwards`, and `sent` keep track of how many events have been dropped and forwarded and how many packets have been sent by the broker.

When a subscriber subscribes to its home broker, it sends a message with the message content

```
op subscribe : Address PredicateFilter -> Content .
```

Internally, this subscription is propagated by messages with the message content

```
op propagate : Address PredicateFilter -> Content .
```

Publishers publish events and brokers propagate events by sending messages with the message content

```
op publish : Address Event -> Content .
```

Additionally, the auxiliary message contents

```
op start : -> Content .
op pubTick : -> Content .
op procTick : -> Content .
op generate : -> Content .
```

are defined and are specific to the Maude specification of the stock exchange information system's behavior.

The module *PUBLISHER*

The module *PUBLISHER* defines the behavior of the publishers in the system. The rule

```
var gt : Float .
var AS : AttributeSet .
var A HB : Address .

rl [Publish] :
  < A : Publisher | homeBroker: HB, AS >
  {gt, (A <- pubTick)}
=>
  < A : Publisher | homeBroker: HB, AS >
  [gt + latency,
    (HB <- publish(A, event(contentsAttributesGenerator, lifetime, gt)))]
  [gt + pubRate, (A <- pubTick)] .
```

specifies that when a publisher receives a message with the message content `pubTick`, it generates a new event using the contents attribute generator and sends the newly generated event as a publication to its home broker. Furthermore, it schedules its next publication tick.

The module *SUBSCRIBER*

The module *SUBSCRIBER* defines the behavior of the subscribers in the system. The variables

```
vars gt d s AVGTTS : Float .
var CNT : Nat .
var AS : AttributeSet .
var PF : PredicateFilter .
var CA : ContentAttributes .
vars A HB : Address .
```

are used for the description of the rules.

The rule

```
rl [Subscribe] :
  < A : Subscriber | homeBroker: HB, predicateFilter: PF, AS >
  {gt, (A <- start)}
=>
  < A : Subscriber | homeBroker: HB, predicateFilter: PF, AS >
  [gt, (HB <- subscribe(A, PF))] .
```

states that upon receiving a message with the message content `start`, a subscriber subscribes to its home broker by sending a message with the message contents `subscribe`, which contains its address and its predicate filter.

The rule

```
rl [Receive-Publish] :
  < A : Subscriber | recCnt: CNT, avgTTS: AVGTTS, AS >
  {gt, (A <- publish(HB, event(CA, d, s)))}
=>
  < A : Subscriber | recCnt: s(CNT),
    avgTTS: AVGTTS + (((gt - s) - AVGTTS) / float(s(CNT))), AS > .
```

specifies that when an event publication is received by a subscriber, the subscriber consumes the messages and updates its statistical attributes `recCnt` and `avgTTS`. The average time to subscriber attribute is thereby updated as a cumulative ongoing average.

The module *GENERATOR*

The module *GENERATOR* describes the behavior of the generator actors which generate subscribers and publishers. For the description of the behavior of the generators, the variables

```
vars gt r : Float .
var SL : ScheduleList .
var N : Nat .
var A : Address .
var NG : NameGenerator .
```

are used.

The operator

```
op generateSubscribers : Float NameGenerator -> [Config] .
```

takes a global time and a name generator as arguments and returns a configuration that contains the generated subscribers. This operator calls the auxiliary operator

```
op generateSubscribersRec : Float NameGenerator Nat -> [Config] .
```

which takes the number of subscribers it should generate as an additional argument. The behavior of the operator `generateSubscribersRec` is defined by the equations

```
eq generateSubscribersRec(gt, NG, 0) = NG .
eq generateSubscribersRec(gt, NG, s(N)) =
  < NG .new : Subscriber |
    homeBroker: s(sampleUniWithInt(numBrokers)),
    predicateFilter: predicateFilterGenerator,
    recCnt: 0,
    avgTTS: 0.0 >
  [gt, (NG .new <- start)]
  generateSubscribersRec(gt, NG .next, N) .
```

which generate the specified number of subscribers. The newly generated subscribers have a predicate filter that is generated by the predicate filter generator and a home broker that is uniformly chosen from the available brokers[2]. In addition to the subscribers, the returned

[2] This way of randomly choosing one of the brokers as a home broker is possible due to a fixed initial configuration of the stock exchange information system model that statically defines the broker's addresses to be 1–20.

configuration also contains scheduled messages to start the subscribers and the updated name generator. Finally, the behavior of the operator `generateSubscribers` is defined by the equation

```
eq generateSubscribers(gt, NG) = generateSubscribersRec(gt, NG, numSubscribers) .
```

which states that the operator calls the operator `generateSubscribersRec` with the number of subscribers that is specified in the module *PARAMS* as an additional argument.

The rule

```
rl [Generate-Subscribers] :
  NG
  < A : SubscriberGenerator | mt >
  {gt, (A <- generate)}
=>
  generateSubscribers(gt, NG) .
```

specifies that, upon receiving a message with message contents `generate`, a subscriber generator generates the subscribers and removes itself from the configuration.

Similar to the generation of subscribers, publishers are generated by the operator

```
op generatePublishers : Float NameGenerator -> [Config] .
```

and the auxiliary operator

```
op generatePublishersRec : Float NameGenerator Nat -> [Config] .
```

that are defined by the equations

```
eq generatePublishers(gt, NG) =
  generatePublishersRec(gt, NG, numPublishers) .
eq generatePublishersRec(gt, NG, 0) = NG .
eq generatePublishersRec(gt, NG, s(N)) =
  < NG .new : Publisher |
    homeBroker: s(sampleUniWithInt(numBrokers)) >
  [gt + initDelay, (NG .new <- pubTick)]
  generatePublishersRec(gt, NG .next, N) .
```

A difference lies in the messages that are generated. The messages for the publishers contain the message contents `pubTick` and are scheduled to become active after a specified time period — an initial delay (`initDelay`).

The rule

```
rl [Generate-Publishers] :
  NG
  < A : PublisherGenerator | mt >
  {gt, (A <- generate)}
=>
  generatePublishers(gt, NG) .
```

specifies that, upon receiving a message with message contents `generate`, a publisher generator generates the publishers and removes itself from the configuration.

The module *BROKER*

The module *BROKER* defines the behavior of brokers in the stock exchange information system model. The tasks of brokers can be divided into two categories: the propagation of

subscriptions, and the handling of event publications. For the description of the behavior of the brokers, the variables

```
vars gt d s : Float .
var L : Latency .
vars F D SENT SZ SZ1 SZ2 SZ1' DROPPED : Nat .
vars PF PF1 PF2 : PredicateFilter .
var AS : AttributeSet .
var R : Routing .
vars RT RT1 RT2 : RoutingTable .
var E : Event .
var CA : ContentsAttributes .
vars CQ CQ' : ContentsQueue .
vars A B NE FROM BROKER TO : Address .
vars AL AL' N S : AddressList .
var Q : QueueWithSize .
```

are used.

Propagation of subscriptions: The operator

```
op insertSubscription : Routing TableWithSize -> TableWithSize .
```

inserts a routing into a broker's routing table. The operator calls the auxiliary operator

```
op insertSubscriptionRec : Routing RoutingTable TableWithSize
  -> TableWithSize .
eq insertSubscription(R, [SZ, RT]) =
  insertSubscriptionRec(R, RT, [SZ, nilRouting]) .
```

which recursively traverses the existing routing table and checks if either the whole routing or the predicate filter are already present in the table. If the routing is already present in the table, the routing is discarded. Otherwise, if the predicate filter of the routing is already in the table, the routing's destination is added to the mapping of that predicate filter in the routing table. If none of the aforementioned cases occur, the routing as a whole is added to the routing table. This behavior is defined by the equations

```
eq insertSubscriptionRec((PF -> A), nilRouting, [SZ, RT]) =
  [s(SZ), (PF -> A) ; RT] .
eq insertSubscriptionRec((PF1 -> A), (PF2 -> AL) ; RT1, [SZ, RT2]) =
  if PF1 == PF2 then
    if not(A in AL) then
      [SZ, (PF2 -> A ; AL) ; RT1 ; RT2]
    else
      [SZ, (PF2 -> AL) ; RT1 ; RT2] fi
  else
    insertSubscriptionRec((PF1 -> A), RT1, [SZ, (PF2 -> AL) ; RT2]) fi .
```

The operator

```
op generatePropagate : Float AddressList Address Address PredicateFilter
  -> Config .
```

generates messages to propagate subscriptions in the broker tree and is defined by the equations

```
eq generatePropagate(gt, mtAddressList, FROM, BROKER, PF) = null .
eq generatePropagate(gt, TO ; AL, FROM, BROKER, PF) =
```

```
  if TO =/= FROM then
    [gt, (TO <- propagate(BROKER, PF))]
    generatePropagate(gt, AL, FROM, BROKER, PF)
  else
    generatePropagate(gt, AL, FROM, BROKER, PF) fi .
```

The rule

```
rl [Receive-Subscription] :
  < B : Broker |
    routingTable: [SZ, RT],
    neighbors: N,
    subscribers: S, AS >
  {gt, (B <- subscribe(A, PF))}
=>
  < B : Broker |
    routingTable: insertSubscription((PF -> A), [SZ, RT]),
    neighbors: N,
    subscribers: (A ; S), AS >
  generatePropagate(gt, N, A, B, PF) .
```

states, that upon receiving a subscription from a subscriber, a broker inserts the subscriber's address to its list of subscribers, inserts the subscription to its routing table, and propagates the subscription to its neighbors in the form of a message with message contents `propagate`.

When a broker receives a message with message contents `propagate`, the rule

```
rl [Receive-Propagate] :
  < B : Broker |
    routingTable: [SZ, RT],
    neighbors: N, AS >
  {gt, (B <- propagate(A, PF))}
=>
  < B : Broker |
    routingTable: insertSubscription((PF -> A), [SZ, RT]),
    neighbors: N, AS >
  generatePropagate(gt, N, A, B, PF) .
```

states, that the broker inserts the subscription to its routing table and propagates the subscription to its neighbors.

Handling of event publications: When a broker receives an expired event, i.e., when the current time is greater or equal to the sum of the publication time and the lifetime duration of the event, it drops the received event. This behavior is specified by the rule

```
crl [Broker-Receive-Drop] :
  < B : Broker |
    drops: D, AS >
  {gt, (B <- publish(FROM, event(CA, d, s)))}
=>
  < B : Broker |
    drops: s(D), AS >
if gt >= s + d .
```

If a broker receives an event that has not expired, the rule

```
crl [Broker-Receive-Process] :
  < B : Broker |
```

```
    eventQueue: [SZ1, CQ],
    routingTable: [SZ2, RT], AS >
  {gt, (B <- publish(FROM, event(CA, d, s)))}
=>
  < B : Broker |
    eventQueue: [s(SZ1), CQ ; publish(FROM, event(CA, d, s))],
    routingTable: [SZ2, RT], AS >
  if SZ1 == 0 then
    [gt + float(SZ2) * procRate, (B <- procTick)]
  else
    null
  fi
if gt < d .
```

states that the received message contents (`publish`) is inserted into the local event queue. Additionally, if the queue was empty prior to the insertion, a self-addressed processing tick message (with the message contents `procTick`) is scheduled. The processing time (defined by the scheduled arrival time of the message) is thereby proportional to the size of the broker's routing table.

When a broker receives a processing tick message, it is forwarded to the neighboring actors that are interested in the processed event. Additionally, the event queue is cleaned at the end of the processing tick. This means that, as long as an expired event is at the beginning of the queue, the event will be dropped until either there is an event at the beginning of the queue that has not expired or the event queue is empty. The sort `CleanQueue` concatenates a cleaned event queue with the number of events that have been dropped from the queue.

```
sort CleanQueue .
op {_,_} : QueueWithSize Nat -> CleanQueue [ctor] .
```

The operator

```
op clean : Float QueueWithSize Nat -> CleanQueue .
```

takes the current time, an event queue, and a natural number as arguments and returns a term of sort `CleanQueue`. The last argument is initially 0 and indicates how many list items have been dropped. The behavior of the operator is defined by the equations

```
eq clean(gt, [0, nilQueue], DROPPED) = {[0, nilQueue], DROPPED} .
ceq clean(gt, [s(SZ), publish(FROM, event(CA, d, s)) ; CQ], DROPPED) =
  clean(gt, [SZ, CQ], s(DROPPED))
if gt >= d + s .
ceq clean(gt, [SZ, publish(FROM, event(CA, d, s)) ; CQ], DROPPED) =
  {[SZ, publish(FROM, event(CA, d, s)) ; CQ], DROPPED}
if gt < d + s .
```

The operator

```
op generateInterested : RoutingTable Address AddressList Event
  -> AddressList .
```

takes a broker's routing table, the address an event has been received from, an address list, and an event as arguments and returns an address list of neighboring actors that are interested in the event. The operator is defined by the equations

```
eq generateInterested(nilRouting, FROM, AL, E) = AL .
eq generateInterested(((PF -> mtAddressList) ; RT), FROM, AL, E) =
  generateInterested(RT, FROM, AL, E) .
```

```
eq generateInterested(((PF -> A ; AL) ; RT), FROM, AL', event(CA, d, s)) =
  if A =/= FROM and not(A in AL') and evalFilter(CA, PF) then
    generateInterested(((PF -> AL) ; RT), FROM, (A ; AL'), event(CA, d, s))
  else
    generateInterested(((PF -> AL) ; RT), FROM, AL', event(CA, d, s)) fi .
```

The operator

```
op generateForward : Float Event Address AddressList AddressList -> Config .
```

and its defining equations

```
eq generateForward(gt, E, BROKER, mtAddressList, S) = null .
eq generateForward(gt, E, BROKER, (A ; AL), S) =
  if A in S then
    [gt + latency, (A <- publish(BROKER, E))]
  else
    [gt + latency(BROKER, A), (A <- publish(BROKER, E))] fi
  generateForward(gt, E, BROKER, AL, S) .
```

generate messages that forward the event to interested actors. The scheduled arrival time of these messages reflects the delay that is introduced by the latency of the communication links from the broker to the receivers of the message.

Finally, the rule

```
crl [Broker-Process] :
  < B : Broker |
    eventQueue: [s(SZ1), publish(FROM, event(CA, d, s)) ; CQ],
    routingTable: [SZ2, RT],
    subscribers: S,
    drops: D,
    forwards: F,
    sent: SENT, AS >
  {gt, (B <- procTick)}
=>
  < B : Broker |
    eventQueue: [SZ1', CQ'],
    routingTable: [SZ2, RT],
    subscribers: S,
    drops: D + DROPPED,
    forwards: s(F),
    sent: SENT + AL .size, AS >
  generateForward(gt, event(CA, d, s), B, AL, S)
  if SZ1' =/= 0 then
    [gt + float(SZ2) * procRate, (B <- procTick)]
  else
    null fi
if AL := generateInterested(RT, FROM, mtAddressList, event(CA, d, s))
  /\ {[SZ1', CQ'], DROPPED} := clean(gt, [SZ1, CQ], 0) .
```

defines the behavior of a broker upon receiving a processing tick. The processed event is forwarded to interested actors (the result of an evaluation of the event against the predicate filters in the routing table) and the event queue is cleaned. Additionally, if the cleaned event queue is not empty, the next processing tick is scheduled.

The module *STOCKEXCHANGE-PARAMS*

The module *STOCKEXCHANGE-PARAMS* defines the three constant operators

```
op listings : -> Nat .
op stockValueMean : -> Float .
op stockValueStdDeviation : -> Float .
```

which define the parameters for the predicate filter and event content attribute generators of the stock exchange information system.

The module *STOCKEXCHANGE*

The module *STOCKEXCHANGE* defines the behavior of the predicate filter and event contents attributes generators.

The constant operator

```
op SEpfGen : -> [PredicateFilter] .
```

generates random predicate filters according to the distribution that is described in Section 7.2. The operator calls the auxiliary operator

```
op SEpfGenSwitch : [Float] -> [PredicateFilter] .
eq SEpfGen = SEpfGenSwitch(genRandom(0.0, 1.0)) .
```

with a random value between 0.0 and 1.0 that is provided as an argument. The semantics of the auxiliary operator are defined by equations such as the following

```
var SWITCH : Float .

ceq SEpfGenSwitch(SWITCH) =
  (pos(0) == topic(s(sampleUniWithInt(listings))))
if SWITCH < 0.20 .
```

which generates the predicate filter PF^1 or

```
ceq SEpfGenSwitch(SWITCH) =
  (pos(0) == topic(s(sampleUniWithInt(listings)))) ;
  (pos(2) >= floor(paretoValue)) ;
  (pos(3) == 0.0)
if SWITCH >= 0.275
  /\ SWITCH < 0.3 .
```

which generates the predicate filter PF^4. The rest of the defining equations are not shown here for reasons of brevity but can be found in Appendix E.1.

The operator

```
op SEContentsAttributesGen : -> [ContentsAttributes] .
```

generates random event contents attributes for the model of the stock exchange information system. Its behavior is specified by the equation

```
eq SEContentsAttributesGen =
  [topic(s(sampleUniWithInt(listings))),
   (floor(boxMullerValue(stockValueMean, stockValueStdDeviation)),
   floor(paretoValue),
   if sampleBerWithP(0.5) then
     0.0
   else
     1.0 fi)] .
```

7.4. Statistical Analysis of the Stock Exchange Information System

We perform statistical model checking of QuaTEx formulas, which express relevant quantitative properties, on the specification of the stock exchange information system model using PVeStA. Specifically, we ask two questions about the system:

1. Events can be forwarded and can be dropped by the brokers. What percentage of events is expected to be forwarded by the brokers? This indicates not only whether events are forwarded to subscribers on time but also how much load the brokers are facing.

2. What are the predicted operating costs for a provider of the stock exchange information system?

The properties are statistically model checked under a varying number of subscribers in the system. The expected processing time of an event increases proportionally to the subscribers that are subscribed to brokers in the system. As predicate filters of subscribers can be equal, in which case the processing time is not increased, the expected processing time is an upper bound for the real processing time.

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.

Broker processing ratio. The broker processing ratio defines the ratio of events that are forwarded by the brokers.

$$\mathit{processingRatio}(t) = \textbf{if } \mathit{time}() > t \textbf{ then } \frac{\mathit{countForwarded}()}{\mathit{countForwarded}() + \mathit{countDropped}()} \textbf{ else } \bigcirc\,(\mathit{processRatio}(t))$$

with *countForwarded*() being the result of equationally counting the number of events that are forwarded by the brokers (`forwards` attribute) and *countDropped*() being the result of equationally counting the number of events that are dropped by the brokers (`drops` attribute).

Number of outgoing packets. The formula *packets*(t) counts the number of packets that are sent by the brokers.

$$\mathit{packets}(t) = \textbf{if } \mathit{time}() > t \textbf{ then } \mathit{countPackets}() \textbf{ else } \bigcirc\,(\mathit{packets}(t))$$

with *countPackets*() being the result of equationally counting the `sent` attribute of the brokers.

We fix most of the parameters that are defined in the module *PARAMS* to the respective (probabilistic) values as defined in Section 7.2. The number of listings is set to 100. The price

values are distributed according to a normal distribution with mean 1000.0 and standard deviation 100.0 ($\mathcal{N}(1000.0, 10000.0)$). The number of brokers and the number of publishers is set to 100. Publishers publish 1 event per second, brokers can process an event with a processing time that is equal to the size of their routing table times 1 ms. The latencies for the inter-broker communication are predefined in the network setup (`network`). The latencies of links between publishers and brokers and between brokers and subscribers are random values chosen according to a normal distribution with mean 30.0 ms and standard deviation 8.0 ms ($\mathcal{N}(30.0, 64.0)$). Additionally, we define an initial delay for event publishing of 50 ms. The simulation duration of a run of the system is set to 5 minutes. The number of subscribers and the lifetime of an event are varying parameters. The parameters are defined in the module *MODEL-PARAMS*, which is presented in Appendix E.1.1.

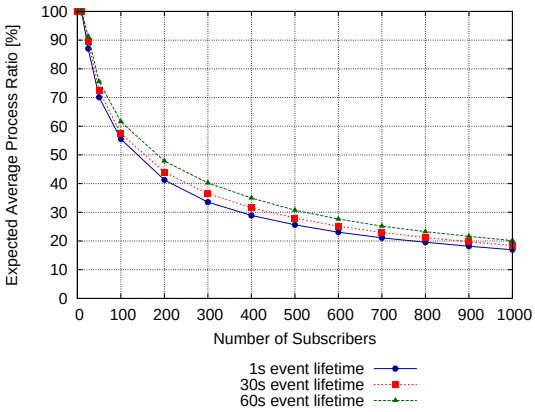
We statistically model check the above properties for the various combinations of the varying parameters. The event lifetime parameter is set to 1 s, 30 s, and 60 s. We model checked the broker process ratio for 1, 10, 25, 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, and 1000 subscribers. The number of outgoing packets were analyzed only up to 600 subscribers for time reasons. The results for the broker process ratio rely on a 99% confidence interval bounded by 0.01 for the expected value of the formula $\mathit{processRatio}(t)$. The results for the number of outgoing packets rely on a 60% confidence interval bounded by 0.05 for the expected value of the formula $\mathit{packets}(t)$.

The statistical model checking results are shown in Figure 7.5. Figure 7.5(a) shows the expected process ratio of the brokers for the varying amount of subscribers and the different lifetime durations for the events. The results reveal that the stock exchange information system is already overwhelmed by 25 subscribers. For 1000 subscribers, the expected process ratio is below 20%. The numbers barely vary for the different event lifetime durations with the process ratio not surprisingly being best for the 60 s lifetime duration. Figure 7.5(b) shows the expected number of outgoing packets of the brokers. A peak is reached for approximately 25 subscribers. After that point, as suggested by the results of 7.5(a), the system can no longer keep up with the processing of events. Consequently, the number of outgoing packets also decreases. Again, the numbers barely differ for the different event lifetime durations. Figure 7.5(c) plots the expected number of outgoing packets on a daily costs-based scale. We take the costs for regional data transfer of the Amazon EC2 platform [11] as a reference. Amazon charges \$0.120 per GB (up to 10TB/month[3]) for all data transferred between instances in different regions.

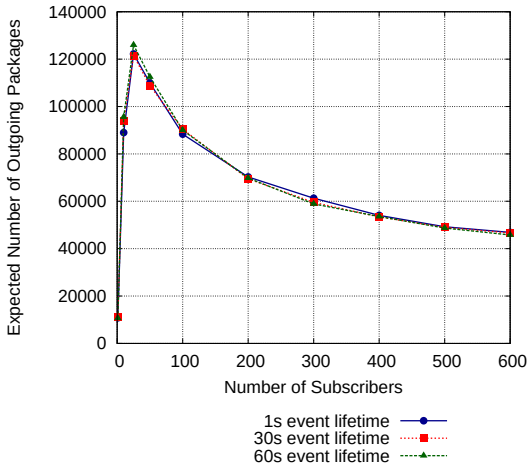
7.5. Adding Cloud-based Broker Replication

As suggested by the statistical model checking results in Section 7.4, the process ratio drops significantly for increased numbers of subscribers. A service provider who signed a service level agreement that guarantees a timely delivery of events (e.g., 80% of events arrive on time) will fail to keep the agreement in the aforementioned case. Cloud-based systems offer the possibility of provisioning new resources on demand. In this section we leverage this possibility for Cloud-based Publish/Subscribe systems. Therefore, where we previously assumed geographically distributed brokers in the stock exchange information system model

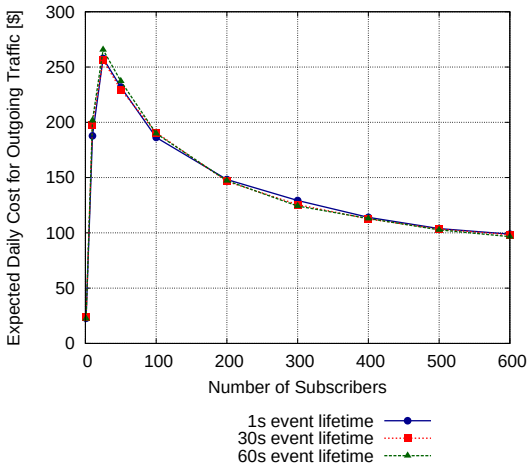
[3]Indeed, if more than 10TB/month are bought, Amazon charges lower prices for additional the data.



(a) Expected broker process ratio



(b) Expected number of outgoing packets sent by the brokers



(c) Expected operating costs for outgoing traffic of the brokers

Figure 7.5.: Statistical model checking results of the stock exchange information system model

(see Section 7.2), we now assume geographically distributed data centers that can provision extra resources on demand. However, two questions remain:

1. According to what metric should new brokers be provisioned?
2. How is the state of the broker handled?

In the following, we give answers to these questions by defining a broker replicator meta-object and a data storage and access layer.

7.5.1. Broker Data — a data storage and access interface for Cloud-based systems

The use of multiple replicated brokers raises the question of how brokers obtain the data that they need to process events. In our example, this data is the routing table. Amazon

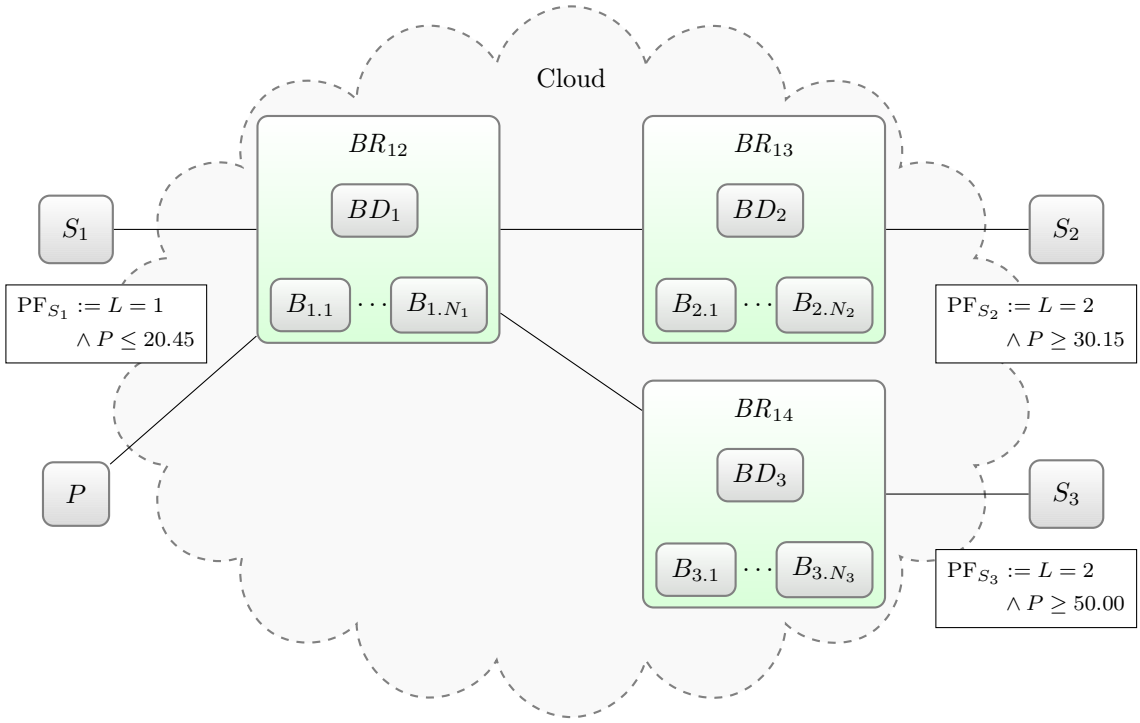


Figure 7.6.: Overview of broker replicators in a subset of the Cloud-based stock exchange information system

and other Cloud computing providers propose using shared volumes [13] or a database [10] in case multiple computing nodes need to share common data.

For the model of the Cloud-based stock exchange information system, we propose something similar: a broker data object that resembles a data storage and access interface. In each location where brokers can be replicated (where we assumed brokers before, we now assume data centers), one broker data object exists. This does not necessarily mean that the servers that provide the data are not replicated. Instead, the broker data object abstracts away how data is handled in the background. In our case, the object encapsulates the routing table of the current location and offers primitives to add a new subscription to the table and to request the current table. We assume, that the insertion algorithms for the routing table are executed in the data layer (e.g., by using a SQL query to properly insert a new subscription in a relational database).

7.5.2. Broker replication in the Cloud

To replicate brokers at the geographically distributed locations in our network, we replace each of the brokers at these locations with a server replicator. The server replicator is a meta-object that wraps broker instances according to the Russian Dolls model. Figure 7.6 illustrates how the broker replicator and the broker data objects are applied to a subset of the stock exchange information system. The broker replicator object receives incoming events, distributes them among wrapped brokers, and provisions new broker instances according to a specific metric. We propose a metric that only relies on information that can be obtained

locally. This has several advantages:

- The acquisition of global, dynamically changing knowledge requires a traversal of the system which introduces a coordination overhead.

- It should be easy and computationally inexpensive to evaluate a replication metric, because we want the server replicator to introduce a close to zero delay when forwarding events. Additionally, for a complex metric, it could be necessary to scale up the entry point itself.

In our model, the server replicator keeps track of the incoming event flow (events per second) and has knowledge about how long it takes a broker to process an event, which depends on the size of the routing table. The size information can be obtained locally from the broker data object. The server replicator provisions a new broker if at time t

$$\mathcal{F}_t > |B|_t \cdot \frac{1}{|RT|_t \cdot 0.001} \cdot \frac{E}{s}$$

where $\mathcal{F}_t$ is the event flow rate at time t in events per second ($\frac{E}{s}$), $|B|_t$ is the number of brokers at time t, and $|RT|_t$ is the size of the routing table at time t. In our model, we do not remove servers from the inner configuration of the broker replicator[4]. The replication strategy defined above is executed every time an event is received by the broker replicator.

In the following, we define how subscriptions are handled, how events are forwarded, and how they are processed by the broker instances that are wrapped by the broker replicator.

Subscription handling:

When a broker replicator receives a subscriptions, it forwards the subscription to the broker data object, which in turn stores the subscription in its routing table and forwards it to neighboring brokers. Just as in the original subscription mechanism in the model without the server replicator, the broker data object prevents duplicates and collapses equal predicate filters in the routing table. The delays that are introduced by forwarding the subscriptions to the broker data object are omitted in our model.

Event processing:

When a broker replicator receives an event, it randomly (according to a uniform distribution) picks one of the wrapped broker instances and forwards the event to this broker. The forwarding happens instantaneously, i.e., without any delay. The broker instance enqueues the received event. To process an event, a broker instance requests the current routing table from the broker data object. Upon receiving the response with the routing table, the broker instance locally processes the event and forwards it to subscribers and brokers whose filters apply. The whole process is shown in Figure 7.7.

[4]This could be done by removing a broker instance, if the flow rate drops below a certain threshold that is defined in terms of the current processing capability.

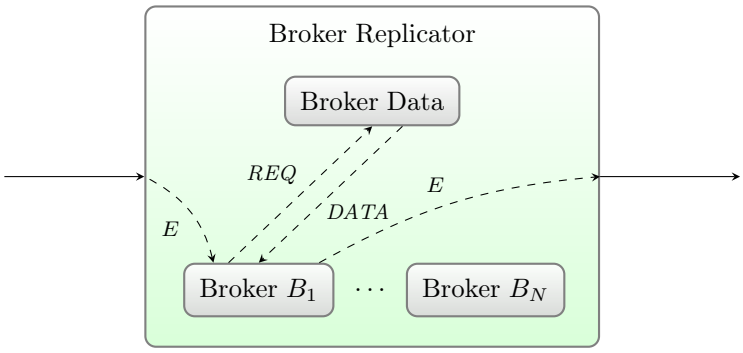


Figure 7.7.: Event processing of event E in a broker replicator that chooses broker B_1 as the processing broker

7.6. Specification of the Cloud-based Stock Exchange Information System in Maude

In this section we describe the Maude-based specification of the Cloud-based stock exchange information system that incorporates the broker replicator meta-object.

7.6.1. Overview of the Maude specification

The specification uses the specification of Section 7.3 as a foundation and replaces the modules *BROKER* with modules that describe

- the broker replicator meta-object,
- an actor that resembles a data storage and access layer in a Cloud-based architecture,
- and a broker that can be wrapped by the broker replicator meta-object and that uses the data access layer.

Additionally, the modules *BR-CONFIGURATION* and the modules *BR-ANALYSIS*, *BR-PUBLISH-SUBSCRIBE* and *BR-PUBLISH-SUBSCRIBE-ANALYSIS* replace their counterparts of the specification that does not use the broker replicator. An overview of the Cloud-based specification is shown in Figure 7.8.

7.6.2. Description of the modules of the Maude specification

The module *BR-CONFIGURATION*

The module *BR-CONFIGURATION* defines the actor types, attributes, and message contents that are added by the Cloud-based specification of the stock exchange information system. The operators

```
op BrokerReplicator : -> ActorType [ctor] .
op BrokerData : -> ActorType [ctor] .
```

define the actor types for the broker replicator and the actor that stores the common broker data of wrapped brokers in the Cloud-based model.

The broker replicator has the new attributes

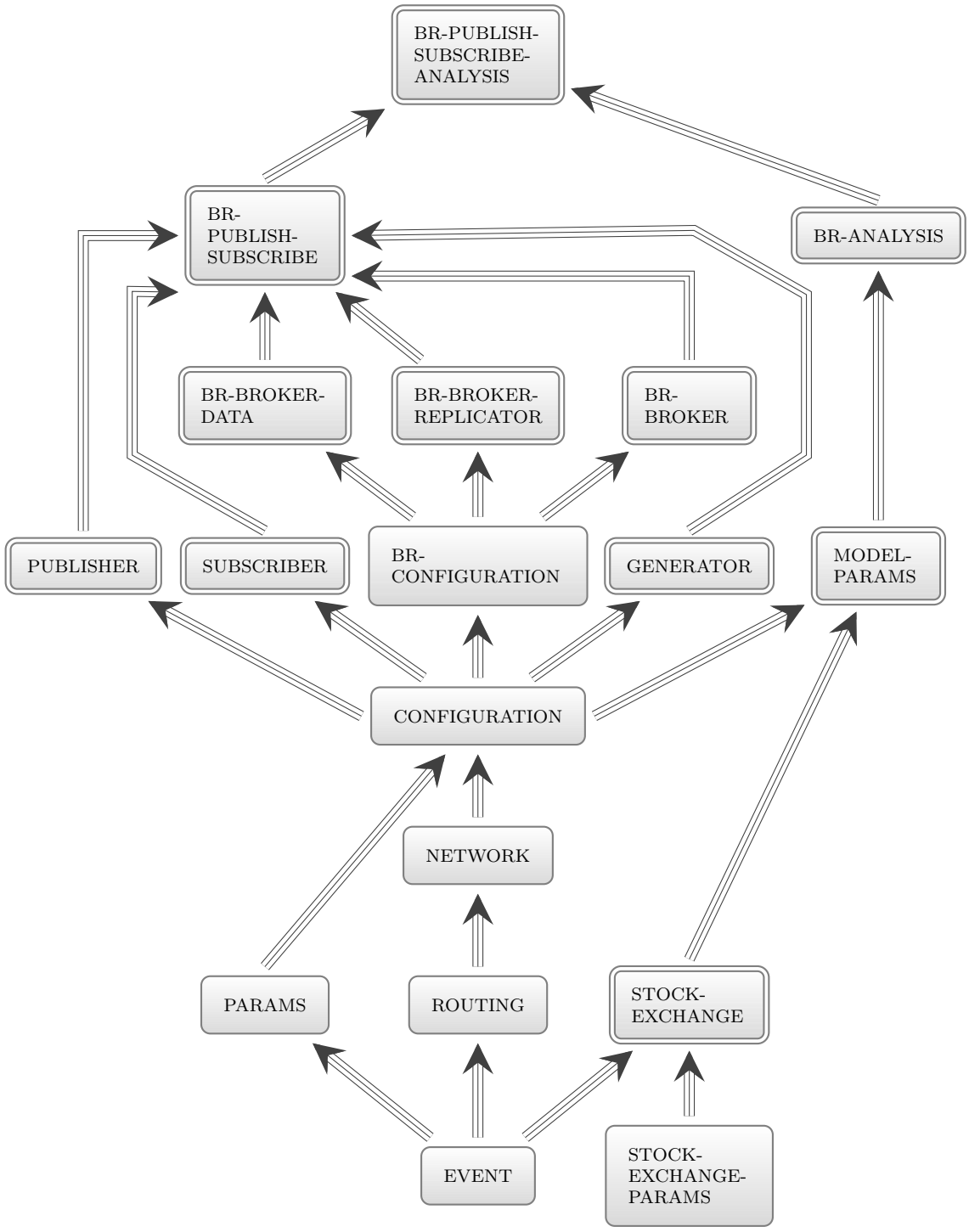


Figure 7.8.: Overview of the Maude specification of the Cloud-based stock exchange information system model

```
op brokerList:_ : AddressList -> Attribute [gather(&)] .
op eventCnt:_ : Nat -> Attribute [gather(&)] .
op eventFlowRate:_ : Float -> Attribute [gather(&)] .
```

which respectively hold a list of addresses of wrapped brokers, a counter of events, and an indicator of the current event flow rate.

Additionally, the new messages

```
op check : -> Content .
op spawnBroker : -> Content .
op startBroker : -> Content .
op dataReq : Address -> Content .
op brProcTick : TableWithSize AddressList -> Content .
```

are defined and are used by the broker replicator to handle the provisioning of new brokers and by the brokers to interact with the actor that stores the common broker data.

The module *BR-BROKER-DATA*

The module *BR-BROKER-DATA* defines the behavior of an actor that stores the common data of a group of wrapped brokers and can be queried like a data access layer. In the Maude-based specification, we use this actor to model a shared memory-like infrastructure which all brokers that are wrapped by the same server replicator can communicate with via message passing. The actor furthermore handles incoming subscriptions of the group of replicated brokers that it serves. In the following, the variables

```
vars B BR A : Address .
var SZ : Nat .
var RT : RoutingTable .
var AS : AttributeSet .
var gt : Float .
var PF : PredicateFilter .
vars N S : AddressList .
```

are used.

The rule

```
rl [State-Subscription] :
  < BR . 0 : BrokerData |
    routingTable: [SZ, RT],
    neighbors: N,
    subscribers: S, AS >
  {gt, BR . 0 <- subscribe(A, PF)}
=>
  < BR . 0 : BrokerData |
    routingTable: insertSubscription((PF -> A), [SZ, RT]),
    neighbors: N,
    subscribers: A ; S, AS >
  generatePropagate(gt, N, A, BR, PF) .
```

defines the behavior of the broker data actor when it receives a subscription. The actor inserts the subscription in its routing table and propagates it to its neighboring brokers. For this, it uses the operators `insertSubscription` and `generatePropagate`. The definitions of these operators were given in Section 7.3.2.

Similar to the rule that handles subscriptions, the rule

```
rl [State-Propagate] :
  < BR . 0 : BrokerData |
    routingTable: [SZ, RT],
    neighbors: N, AS >
  {gt, BR . 0 <- propagate(A, PF)}
=>
  < BR . 0 : BrokerData |
    routingTable: insertSubscription((PF -> A), [SZ, RT]),
    neighbors: N, AS >
  generatePropagate(gt, N, A, BR, PF) .
```

specifies the behavior of the actor when it receives a message that propagates a subscription.

Finally, the rule

```
rl [Data-Request] :
  < BR . 0 : BrokerData |
    routingTable: [SZ, RT],
    subscribers: S, AS >
  {gt, BR . 0 <- dataReq(B)}
=>
  < BR . 0 : BrokerData |
    routingTable: [SZ, RT],
    subscribers: S, AS >
  [gt + float(SZ) * procRate, B <- brProcTick([SZ, RT], S)] .
```

specifies that if a broker data actor receives a data request message from a broker, it answers the request with a process tick message that contains the current routing table and the list of subscribers.

The module *BR-BROKER-REPLICATOR*

The module *BR-BROKER-REPLICATOR* defines the behavior of the broker replicator meta-object. The broker replicator wraps around a group of (replicated) brokers and keeps track of the incoming event flow rate. It uses this metric to provision new brokers on demand. In the following description of the broker replicator behavior, the variables

```
var BL : AddressList .
vars gt FR FR' : Float .
var C : Config .
vars NAT EC SZ SZ' : Nat .
vars AS AS' : AttributeSet .
var CO : Content .
var E : Event .
var NG : NameGenerator .
vars A B NB BR BS : Address .
var PF : PredicateFilter .
var RT : RoutingTable .
var TWZ : TableWithSize .
```

are used.

The replication strategy of the broker replicator is defined by the operator

```
op strategy : Float Nat Nat -> Bool .
```

which takes the current event flow rate, the current number of brokers, and the size of the routing table as arguments. It returns a Boolean value indicating whether or not a new broker should be provisioned. The strategy is defined by the equation

```
eq strategy(FR, SZ, SZ') = FR > (float(SZ) / (float(SZ') * procRate)) .
```

It states that the strategy operator returns `true` iff the incoming flow of events per second is greater than the current processing rate of the wrapped brokers.

The operator

```
op pickRandom : AddressList -> [Address] .
eq pickRandom(BL) = BL[sampleUniWithInt(BL .size)] .
```

randomly picks (according to a uniform distribution) one of the broker addresses from the broker address list.

The rule

```
rl [Broker-Start] :
  < BR : BrokerReplicator | AS >
  {gt, BR <- startBroker}
=>
  < BR : BrokerReplicator | AS >
  [gt, BR <- spawnBroker] .
```

describes the broker replicator instantiation upon receiving a start message. The broker replicator thereby sends a self-addressed message to spawn a broker (the initial broker).

Upon receiving an event, the rule

```
crl [Broker-Replicator-Receive-Event] :
  < BR : BrokerReplicator |
    config: < BR . 0 : BrokerData |
              routingTable: [SZ, RT], AS' > C,
    brokerList: BL,
    eventCnt: EC,
    eventFlowRate: FR, AS >
  {gt, BR <- publish(A, E)}
=>
  < BR : BrokerReplicator |
    config: < BR . 0 : BrokerData |
              routingTable: [SZ, RT], AS' > C,
    brokerList: BL,
    eventCnt: s(EC),
    eventFlowRate: FR', AS >
  [gt, pickRandom(BL) <- publish(A, E)]
  if strategy(FR', BL .size, SZ) then
    [gt, BR <- spawnBroker]
  else
    null
  fi
if FR' := (float(s(EC)) / (gt + 1.0)) .
```

states that the broker replicator picks a random broker from its broker list to forward the event to. Additionally, the broker replicator checks its replication strategy and, in case of an evaluation to `true`, spawns a new broker by sending a self-addressed message with the message contents `spawnBroker`.

The rules

```
rl [Broker-Replicator-Receive-Subscription] :
  < BR : BrokerReplicator | AS >
  {gt, BR <- subscribe(A, PF)}
=>
```

```
  < BR : BrokerReplicator | AS >
  [gt, BR . 0 <- subscribe(A, PF)] .
```

and

```
rl [Broker-Replicator-Receive-Propagate] :
  < BR : BrokerReplicator | AS >
  {gt, BR <- propagate(A, PF)}
=>
  < BR : BrokerReplicator | AS >
  [gt, BR . 0 <- propagate(A, PF)] .
```

forward subscriptions and forwarded subscriptions from other brokers (message content `publish`) to the broker data actor.

Finally, the rule

```
crl [Broker-Replicator-Spawn-Broker] :
  < BR : BrokerReplicator |
    config: NG C,
    brokerList: BL, AS >
  {gt, BR <- spawnBroker}
=>
  < BR : BrokerReplicator |
    config: (NG .next) C
           < NB : Broker |
             eventQueue: [0, nilQueue],
             drops: 0, forwards: 0, sent: 0 >,
    brokerList: (NB ; BL), AS >
if NB := NG .new .
```

specifies how, when a message with message contents `spawnBroker` is received, a new broker is spawned in the inner configuration of the broker replicator.

The module *BR-BROKER*

The module *BR-BROKER* specifies the behavior of a broker that is wrapped by the broker replicator. In the following, the variables

```
vars gt d s : Float .
vars F D SZ SZ1 SZ1' SZ2 NO DROPPED SENT : Nat .
vars PF : PredicateFilter .
var AS : AttributeSet .
var R : Routing .
var RT : RoutingTable .
var E : Event .
var CA : ContentsAttributes .
var CQ CQ' : ContentsQueue .
vars A B FROM BROKER : Address .
vars AL AL' N S : AddressList .
vars TWZ TWZ' : TableWithSize .
```

are used.

Just as a normal broker, a wrapped broker drops an event upon receiving it, if it has expired.

```
crl [Broker-Receive-Drop] :
  < B : Broker |
```

```
    drops: D, AS >
  {gt, (B <- publish(FROM, event(CA, d, s)))}
=>
  < B : Broker |
    drops: s(D), AS >
if gt >= d .
```

Upon receiving an event that has not expired, the rule

```
crl [Broker-Receive-Process] :
  < B . NO : Broker |
    eventQueue: [SZ1, CQ], AS >
  {gt, (B . NO <- publish(FROM, event(CA, d, s)))}
=>
  < B . NO : Broker |
    eventQueue: [s(SZ1), CQ ; publish(FROM, event(CA, d, s))], AS >
  if SZ1 == 0 then
    [gt, B . 0 <- dataReq(B . NO)]
  else
    null fi
if gt < d .
```

states that a wrapped broker enqueues the event in its event queue and, if the queue was initially empty, starts processing the event by sending a data request to the broker data actor in order to receive the data (routing table, subscribers) that is necessary to process the event.

Finally, a wrapped broker processes an event when it receives the data from the broker data actor (message content `brProcTick`). The rule that defines this behavior

```
crl [Broker-Process] :
  < B . NO : Broker |
    eventQueue: [s(SZ1), publish(FROM, event(CA, d, s)) ; CQ],
    drops: D,
    forwards: F,
    sent: SENT, AS >
  {gt, (B . NO <- brProcTick([SZ2, RT], S))}
=>
  < B . NO : Broker |
    eventQueue: [SZ1', CQ'],
    drops: D + DROPPED,
    forwards: s(F),
    sent: SENT + AL .size, AS >
  generateForward(gt, event(CA, d, s), B, AL, S)
  if SZ1' =/= 0 then
    [gt, B . 0 <- dataReq(B . NO)]
  else
    null fi
if AL := generateInterested(RT, FROM, mtAddressList, event(CA, d, s))
  /\ {[SZ1', CQ'], DROPPED} := clean(gt, [SZ1, CQ], 0) .
```

uses the operators `clean`, `generateInterested`, and `generateForward`. The definitions of these operators were given in Section 7.3.2.

7.7. Statistical Analysis of the Cloud-based Stock Exchange Information System

Just as for the original specification (see Section 7.4), we perform statistical model checking of QuaTEx formulas, which express quantitative properties of interest, on the specification of the Cloud-based stock exchange information system model using PVeStA. Again, the properties are statistically model checked under a varying number of subscribers in the system and for three different event lifetime values (1 s, 30 s, 60 s).

Broker processing ratio (using the broker replicator). The broker processing ratio defines the ratio of events that are forwarded by the wrapped brokers.

$$\mathit{processRatio}(t) = \textbf{if } \mathit{time}() > t \textbf{ then } \frac{\mathit{countForwarded}()}{\mathit{countForwarded}() + \mathit{countDropped}()} \textbf{ else } \bigcirc(\mathit{processRatio}(t))$$

with *countForwarded*() being the result of equationally counting the number of events that are forwarded by the brokers (`forwards` attribute) and *countDropped*() being the result of equationally counting the number of events that are dropped by the brokers (`drops` attribute). It is of note that the *countForwarded*() and *countDropped*() functions are evaluated over all replicated brokers.

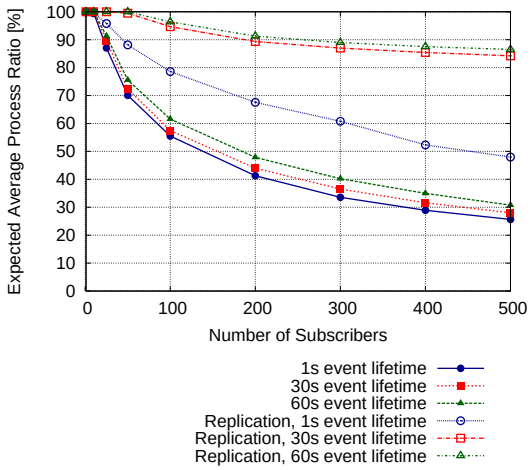
Number of brokers. The formula $\mathit{brokers}(t)$ counts the number of brokers that are provisioned by the broker replicators.

$$\mathit{brokers}(t) = \textbf{if } \mathit{time}() > t \textbf{ then } \mathit{countBrokers}() \textbf{ else } \bigcirc(\mathit{brokers}(t))$$

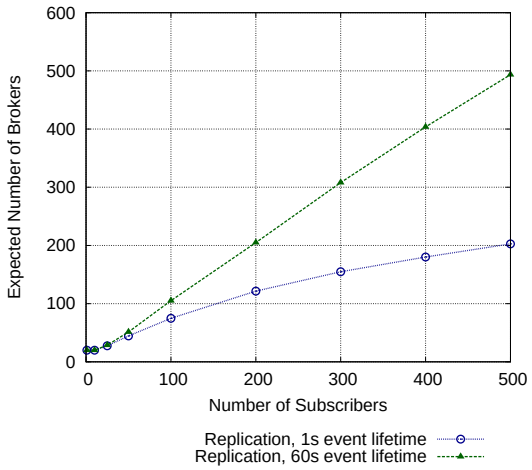
with *countBrokers*() being the result of equationally counting the size of the broker list (`brokerList` attribute) of each broker replicator.

The parameters are set to the same values as in Section 7.4. For the analysis we assume that the broker replicators are not bounded in the number of brokers that they can provision. We statistically model checked the aforementioned properties for the event lifetime durations of 1 s, 30 s, and 60 s and 1, 10, 25, 50, 100, 200, 300, 400, and 500 subscribers. The results for the broker process ratio rely on a 95% confidence interval bounded by 0.05 for the expected value of the formula $\mathit{processRatio}(t)$. The results for the number of brokers rely on a 80% confidence interval bounded by 0.05 for the expected value of the formula *countBrokers*().

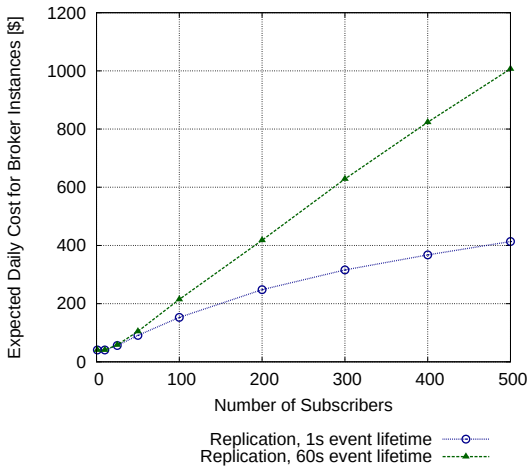
The statistical model checking results are shown in Figure 7.9. Figure 7.9(a) shows the expected process ratio of the brokers with and without broker replication for the varying amount of subscribers and the different lifetime durations for the events. With the broker replicator and the local replication strategy, a significant increase in the process ratio can be achieved. The other observation is that for a longer lifetime duration of events, the replication strategy performs better. As no global knowledge is used and brokers are just replicated so that they can keep up the local processing of events, the process ratio does not reach



(a) Expected broker process ratio (Cloud-based)



(b) Expected number of brokers



(c) Expected operating costs for brokers

Figure 7.9.: Statistical model checking results of the Cloud-based stock exchange information system model

100%. To achieve higher process ratios, global knowledge could be used to overprovision at broker replicators that have more crucial time bounds. Figure 7.9(b) shows the expected number of brokers for the broker replication. In the case of a 1 s event lifetime duration, events expire earlier in the forwarding process and less brokers are provisioned than in the case of a 60 s event lifetime duration. For a 60 s event lifetime duration, the number of brokers increases proportionally to the number of subscribers using the local replication strategy.

Figure 7.9(c) plots the expected number of brokers on a daily costs-based scale. We take the costs for an on-demand instance of the Amazon EC2 platform [11] as a reference. Amazon charges $0.085 per hour for a small standard on-demand instance.

7.8. Conclusion & Future Work

In this chapter we have shown that a system that is based on a Publish/Subscribe middleware can be naturally modeled using the modularized actor model. Furthermore, the statistical model checking analysis of the model suggested that with event lifetime durations of 1 s, 30 s, and 60 s, it is hard to guarantee a timely delivery of events in a system that is distributed across the globe. One possibility to improve to reliability of such a system is to reduce the processing delays of intermediaries in the message forwarding mechanism. We proposed a broker replicator meta-object that uses the Cloud's ability to provision resources on demand together with a local replication strategy to spawn additional broker instances and thus reduce the processing time of an event. Statistical analysis of the Cloud-based model has shown that the reliability of the system could be increased using the replication mechanism.

The following proposals to further improve the reliability of a Cloud-based Publish/Subscribe system are seen as future work:

- Introduce bandwidth limitations in the model and see if and how links can be replicated using a similar strategy as for the brokers.
- See how the second factor for delivery delays, the inter-broker latency, could be reduced by flexible routing mechanisms.
- Introduce a global management meta-object, which wraps a Publish/Subscribe system and provides global knowledge about where to provision new resources to fulfil the timeliness guarantees.

Chapter 8

Outlook and Conclusion

In this thesis, we focused on three Cloud Computing management-related obstacles: bugs in large distributed systems, service availability, and performance unpredictability. To tackle these challenges and the general complexity of Cloud Computing and distributed systems, we proposed solutions based on executable formal specifications and formal analysis, using rewriting logic as the semantic framework and Maude, a language and system based on rewriting logic that offers the possibility of executing and formally analyzing specifications, as the foundation for our work.

In Chapter 4, we presented specifications of formal languages for the design and analysis of Cloud-based architectures. In particular we provided the Maude-based specification of a formal language based on the KLAIM language specification (M-KLAIM). We further extended this specification with object-orientation (OO-KLAIM) and have shown how sockets can be used to execute such specifications in a distributed environment (D-KLAIM). Finally, we demonstrated how specifications based on *-KLAIM can be formally analyzed using model checking of qualitative properties. The various examples in this chapter suggested that the design and analysis of distributed systems such as Cloud Computing systems are possible using the *-KLAIM language specifications as a foundation. Furthermore, we outlined that the languages provided in this chapter might prove themselves helpful for the rapid prototyping of such systems in future work.

In Chapter 5, we extended the standard actor model of computation to incorporate the Russian Dolls model and fulfill the requirements for statistical model checking. To fulfill these requirements, we introduced a multi-level scheduling approach that assures the absence of un-quantified non-determinism for the extended actor model. We further demonstrated that this new extended actor model — the modularized actor model — allows the specification of hierarchically structured distributed systems and their quantitative and qualitative formal analysis.

Chapter 6 dealt with the obstacle of service availability. We outlined that we can formally describe reflective Cloud-based architectures that are protected against denial of service (DoS) attacks using meta-objects formally specified in rewriting logic. Namely, we specified the ASV Wrapper and the Server Replicator SR meta-objects. We discussed that ASV cannot provide stable availability, which means that with very high probability service quality remains very close to a threshold, regardless of how bad the DoS attack can get; and that SR cannot provide stable availability at a reasonable cost. Hence, we introduced ASV$^+$SR, the combination of the ASV and SR meta-objects, for which we have shown, by statistical model checking, that the meta-object composition can achieve stable availability at a reasonable cost under DoS attacks.

Finally, in Chapter 7, we demonstrated that a system that is based on a Publish/Subscribe middleware can be naturally modeled using the modularized actor model. Furthermore, we answered the question of how a Publish/Subscribe architecture can be enriched with Cloud-based dynamic resource provisioning mechanisms to better meet quality of service (QoS) requirements; and demonstrated how predictions about QoS properties of the specified systems can be made using statistical analysis.

It has been out of scope for this thesis to find solutions for all the challenges of Cloud Computing management. For future work we propose the formal specification and analysis of more Cloud-based systems and scenarios. The results in this thesis suggest that this methodology can help deal with the complexity of designing, building, testing, and verifying Cloud-based systems. Furthermore, it is an ambitious goal to develop a formal approach and framework for the design of correct-, secure-, and safe-by-construction distributed systems, aided by a rich tool environment.

Appendix

Appendix A
Formal Languages for the Design and Analysis of Cloud-based Architectures

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

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

$$(3)\ \frac{P \xrightarrow[\rho']{e(Q)@l} P' \qquad s = \rho' \bullet \rho(l)}{s ::_\rho P \rightarrowtail s ::_\rho Q \mid P'}$$

$$(4)\ \frac{P_1 \xrightarrow[\rho]{e(Q)@l} P_1' \qquad s_2 = \rho \bullet \rho_1(l)}{s_1 ::_{\rho_1} P_1 \parallel s_2 ::_{\rho_2} P_2 \rightarrowtail s_1 ::_{\rho_1} P_1' \parallel s_2 ::_{\rho_2} Q \mid P_2}$$

$$(5)\ \frac{P_1 \xrightarrow[\rho']{i(t)@l} P_1' \qquad s = \rho' \bullet \rho(l) \qquad P_2 \xrightarrow[\phi]{o(et)@self} P_2' \qquad match(\mathcal{T}[\![t]\!]_{\rho' \bullet \rho}, et)}{s ::_\rho P_1 \mid P_2 \rightarrowtail s ::_\rho P_1'[et/\mathcal{T}[\![t]\!]_{\rho \bullet \rho_1}] \mid P_2'}$$

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

$$(7)\ \frac{P_1 \xrightarrow[\rho']{r(t)@l} P_1' \qquad s = \rho' \bullet \rho(l) \qquad P_2 \xrightarrow[\phi]{o(et)@self} P_2' \qquad match(\mathcal{T}[\![t]\!]_{\rho' \bullet \rho}, et)}{s ::_\rho P_1 \mid P_2 \rightarrowtail s ::_\rho P_1'[et/\mathcal{T}[\![t]\!]_{\rho \bullet \rho_1}] \mid P_2}$$

$$(8)\ \frac{P_1 \xrightarrow[\rho]{r(t)@l} P_1' \qquad s_2 = \rho \bullet \rho_1(l) \qquad P_2 \xrightarrow[\phi]{o(et)@self} P_2' \qquad match(\mathcal{T}[\![t]\!]_{\rho \bullet \rho_1}, et)}{s_1 ::_{\rho_1} P_1 \parallel s_2 ::_{\rho_2} P_2 \rightarrowtail s_1 ::_{\rho_1} P_1'[et/\mathcal{T}[\![t]\!]_{\rho \bullet \rho_1}] \parallel s_2 ::_{\rho_2} P_2}$$

$$(9)\ \frac{s ::_\rho P_1 \rightarrowtail s ::_\rho P_1'}{s ::_\rho P_1 \mid P_2 \rightarrowtail s ::_\rho P_1' \mid P_2}$$

$$(10)\ \frac{P \xrightarrow[\rho']{n(u))@self} P' \qquad s' \neq s}{s ::_\rho P \rightarrowtail s ::_\rho P'[s'/u] \parallel s' ::_{[s'/self] \bullet \rho} nil}$$

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

$$(12)\ \frac{N \equiv N_1 \qquad N_1 \rightarrowtail N_2' \qquad N_2 \equiv N'}{N \rightarrowtail N'}$$

Figure A.1.: KLAIM's reduction relation

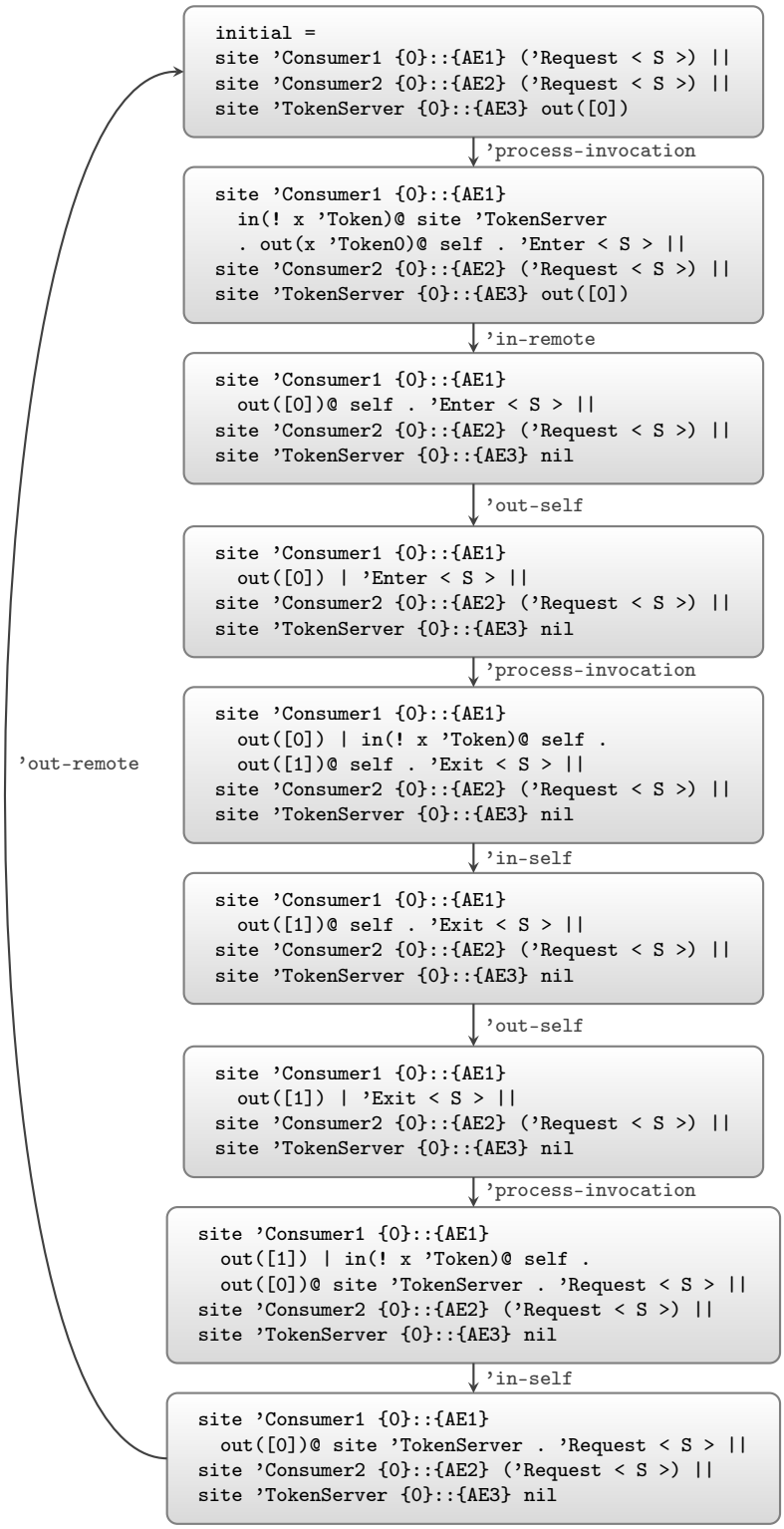


Figure A.2.: Counterexample for the liveness of `consumer2`
(`AE1 := [site 'Consumer1 / self] * [site 'TokenServer / 'TokenServer]`,
`AE2 := [site 'Consumer2 / self] * [site 'TokenServer / 'TokenServer]`,
`AE3 := [site 'TokenServer / self]`,
`S := nilProcessSeq,nilLocalitySeq,nilExpressionSeq`)

Appendix B
Automatic Generation of CINNI Instances for the Maude System

Many formal languages use the concept of names to range over essential entities of the language and are usually equipped with special binding constructs for names. For example, the λ-calculus uses variables as names and λ-abstractions as a name binders; Milner's π-calculus uses the action prefixes *in* and *new* to bind variables in a subsequent term; and first-order logic uses variables as names, which can be bound by the $\forall$ and $\exists$ quantifiers.

CINNI is a calculus of explicit substitutions that contributes a first-order representation of terms which takes variable bindings into account and captures free substitutions. The CINNI calculus is parametric in the syntax of the object language, which allows it to be applied to many different object languages.

The createCINNI tool makes the parametric nature of CINNI available to the Maude system by means of an automatic module transformation which — given a Maude module specifying the syntax of an object language L — generates a Maude module containing the instantiation CINNI_L.

B.1. Introduction

Many formal languages use the concept of *names* to range over essential entities of the language and are usually equipped with special *binding constructs* for names. For example, the λ-calculus uses variables as names and λ-abstractions as name binders; Milner's π-calculus [84] uses the action prefixes *in* and *new* to bind variables in a subsequent term; and first-order logic uses variables as names, which can be bound by the $\forall$ and $\exists$ quantifiers.

Stehr [99] proposes CINNI, a calculus of explicit substitutions that contributes a first-order representation of terms which takes variable bindings into account and captures free substitutions. The CINNI calculus is parametric in the syntax of an object language, which allows it to be applied to many different object languages. Stehr shows applications of CINNI to the λ-, ς-, and π-calculi in Maude. In other work [7, 104], the CINNI calculus is applied to manage names and bindings in various host languages.

In this work, we make the parametric nature of CINNI available to Maude users by means of an automatic module transformation which — given a Maude module specifying the syntax of an object language L — generates a Maude module containing the instantiation CINNI_L.

We first provide an overview of the CINNI calculus. Then, as a running example, we apply the CINNI calculus to Millner's π-calculus, and give, based on that example, an idea how the transformation can be generated automatically using reflection. Finally, we describe how the transformation works in more detail and present the Maude tool *createCINNI*, which performs the module transformation in Full Maude.

B.2. CINNI

Stehr [99] proposes CINNI, a calculus of explicit substitutions that contributes a first-order representation of terms which takes variable bindings into account and captures free substitutions. For a given language L and its defining syntax, the instantiation of CINNI for L is denoted by 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 [37] as a special case, in which a single name is used. CINNI uses the Berklin notation [23, 24] 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 ith binder to the left of the variable in the term.

Example B.1: 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 L with explicit substitutions as shown in equations B.1, B.2 and B.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 $\uparrow_X$ for variable X increases the index of variables with the same name X. The lifted substitution $\Uparrow_X (S)$ is defined in equations B.4, B.5, and B.6. It decreases the index of variables with the name X, performs the substitution S, and finally lifts the variable.

$$[X := M] \qquad \text{(simple substitution)} \tag{B.1}$$
$$\uparrow_X \qquad \text{(shift substitution)} \tag{B.2}$$
$$\Uparrow_X (S) \qquad \text{(lifted subsitution)} \tag{B.3}$$

$$\Uparrow_X (S)X_0 = X_0 \tag{B.4}$$
$$\Uparrow_X (S)X_{n+1} = \uparrow_X ((S)X_n) \tag{B.5}$$
$$\Uparrow_X (S)Y_n = \uparrow_X ((S)Y_n) \text{ if } X \neq Y \tag{B.6}$$

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:

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

B.3. Running Example: CINNI$_\pi$

In the following, we will describe the application of CINNI to Millner's π-calculus. We will later use this running example to illustrate out module transformation. Process terms are represented by the sort `Trm` and channels by the sort `Chan`. Process terms can be concatenated by the associative and commutative parallel composition operator `P|Q` for which the null process `nil` acts as identity. For a process term `P`, the term `out CX <CY>. P` represents a process that sends the channel `CY` over the channel `CX` and then continues with `P`. The term `in CX [Y]. P` represents a process term that receives a channel name over the channel `CX` and then continues with `P`. Finally, the term `new [Y] P` represents a process term that creates a new local name that can be used in `P` and then continues with `P`. The functional Maude module

```
fmod PI-SYNTAX is
  protecting QID .
  sorts Chan Trm .

  op _{_} : Qid Nat -> Chan .
  op nil : -> Trm [ctor] .
  op _|_ : Trm Trm -> Trm [ctor assoc comm id: nil] .
  op new[_]_ : Qid Trm -> Trm [ctor] .
  op out_<_>._ : Chan Chan Trm -> Trm [ctor] .
  op in_[_]._ : Chan Qid Trm -> Trm [ctor] .
endfm
```

describes the syntax of the π calculus.

The two terms `in CX [Y] P` and `new [Y] P` bind the channel name `Y` in the subsequent process `P`. The variables

```
vars X Y : Qid .
vars CX CZ : Chan .
vars P Q : Trm .
var S : Subst .
vars n : Nat .
```

are used in the following equations.

The application of CINNI to the syntax of the π-calculus adds a sort that represents substitutions (**sort** `Subst`), new operators and equations for the three kinds of substitution: simple, shift and lifted.

```
op [_:=_] : Qid Chan -> Subst .
op [shift_] : Qid -> Subst .
op [lift__] : Qid Subst -> Subst .
op __ : Subst Chan -> Chan .
op __ : Subst Trm -> Trm .
```

```
eq [X := CZ] X{0} = CZ .
eq [X := CZ] X{suc(n)} = X{n} .
ceq [X := CZ] Y{n} = Y{n} if X =/= Y .

eq [shift X] X{n} = X{suc(n)} .
ceq [shift X] Y{n} = Y{n} if X =/= Y .

eq [lift X S] X{0} = X{0} .
eq [lift X S] X{s(n)} = [shift X] (S (X{n})) .
ceq [lift X S] Y{n} = [shift X] (S (Y{n}))
    if X =/= Y .
```

Additionally, *syntax-specific equations* are added to the module. A substitution that is applied to the `nil` process is discarded. If a substitution is applied to the parallel composition of two process terms, it is applied to each process term individually. As `nil` acts as the identity for the associative and commutative parallel composition operator, it is important not to apply the substitution to a composition where one of the subterms is the process term `nil`. Similarly to the parallel composition, if a substitution is applied to the process term `out CX<CZ>.M`, the substitution is simply applied to all subterms. The two process terms `in CX[Y].M` and `new[Y]M` bind the name `Y` in the subsequent process term `M`, so the lifted substitution `[lift Y S]` has to be applied to `M`.

```
eq S nil=nil.
ceq S (P | Q) = (S P) | (S Q)
    if P =/= nil and Q =/= nil .
eq S (out CX<CZ>.P) = out (S CX)<S CZ>.(S P).
eq S (in CX[Y].P) = in(S CX)[Y].([lift Y S] P).
eq S (new[Y]P) = new[Y]([lift Y S]P).
```

Discussion

Using Berklin's representation, a requirement for an automatic generation of CINNI specifications is that the language differentiates between *names*, *indexed names*, and *values*. *Names* are used in binding expressions, *indexed names* are names quantified by an index that are bound to a *name* by binding operators, and *values* are terms of the language that can be substituted for *indexed names*. In our example of Millner's π calculus, these entities were mapped to entities of the target language as follows:

$$\begin{aligned} \textit{name} &\mapsto \textbf{sort } \texttt{Qid} \\ \textit{indexed name} &\mapsto \textbf{sort } \texttt{Chan} \\ \textit{value} &\mapsto \textbf{sort } \texttt{Chan} \end{aligned}$$

If this mapping is given, the CINNI transformation can be fully automated. The transformation consists of two main steps:

1. A sort that represents substitutions and operators for the simple, shift, and lifted substitutions are added. Additionally, for each constructed sort of the language, e.g., `Chan` and `Trm`, an operator to prefix substitutions is added. Then, equations defining the semantics of the simple, shift, and lifted substitution are added. These equations need to be aware of the actual sorts for *names*, *indexed names*, and *values*.

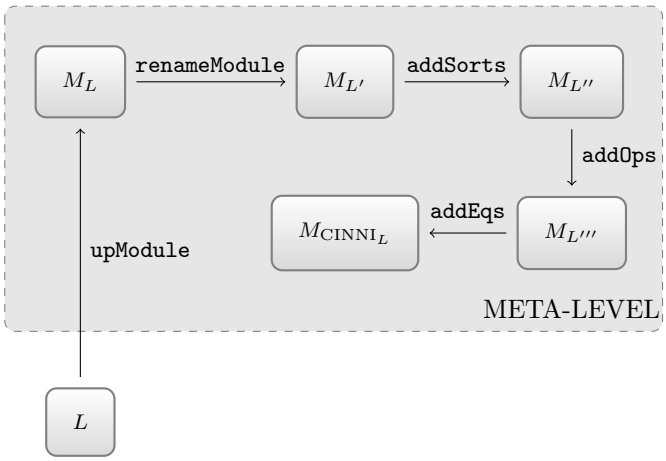


Figure B.1.: Top level view on the module transformation

2. A *syntax-specific* equation is added for each syntactic constructor of the language. Basically, there are two types of syntactic constructors: those that bind *names* in one or more of the arguments and those that don't. For example, the syntactic constructor `new[_]_` binds the first argument in the second whereas the parallel operator `_|_` does not bind any names. The *syntax-specific* equations describe the effect of applying a substitution a term built with the constructor by passing down the substitution to each of the arguments. If the constructor binds a *name* in an argument, the *name* is lifted in the substitution of that argument.

The information about which *names* are bound in which argument cannot be derived directly from the first-order declaration of a syntactic constructor. Our transformation uses Maude's `metadata` attribute so that the user can add this binding information. The two constructors `new[_]_` and `in_[_]` bind the first in the second argument. Thus they are be annotated accordingly.

```
op new[_]_ : Qid Trm -> Trm [ctor metadata "1->2"] .
op in_[_]._ : Chan Qid Trm -> Trm [ctor metadata "2->3"] .
```

Having the mapping between *names*, *indexed names*, and *values* and the corresponding sorts in the language together with the information about which *name* is bound in which argument by the syntactic constructors enables us to create a fully automated module transformation. The next section describes the transformation in more detail.

B.4. The Transformation

Figure B.1 shows a top level view of the transformation. The transformation lifts the source module to the meta-level, renames the module, and adds sorts, operators, and equations to the meta-representation of the module.

The two overloaded operators

```
op cinni : Qid Type Type Type Type -> Module .
op cinni : Module Type Type Type Type -> Module .
```

only differ in their first argument: One can either specify the module's name or give directly the meta representation of the source module. Additionally, one has to specify the sorts that will be used for: (i) substitutions, (ii) *names*, (iii) *indexed names*, and (iv) *values*, where sorsts (i)–(iv) are sorts of the source module. To apply the transformation on our running example, the transformation is executed using the command

```
red cinni('PI-SYNTAX, 'Subst, 'Qid, 'Chan, 'Chan) .
```

The equation for the `cinni` operator, which takes the name of the source module as an argument, uses the `upModule` operator to create the meta representation of the source module. The invocation of the `cinni` operator with the resulting meta-representation of the module is then evaluated by the second equation, which performs the transformation.

```
ceq cinni(MOD, SUBSTT, NAMET, INAMET, VALT) =
  cinni(M, SUBSTT, NAMET, INAMET, VALT)
  if M := upModule(MOD, false) .

ceq cinni(M, SUBSTT, NAMET, INAMET, VALT) = MWE
  if RM := renameModule(M, qid("CINNI-" + string(getModuleName(M)))
     {getParameterDeclList(M)} )
    /\ CTL := removeDoubles(getConstructedTypes(getOps(M)))
    /\ MWS := addSorts(RM, SUBSTT)
    /\ MWO := addOps(MWS, simpleSubst(NAMET, VALT, SUBSTT)
                      shiftSubst(NAMET, SUBSTT)
                      liftSubst(NAMET, SUBSTT)
                      substOps(removeDoubles(CTL INAMET),SUBSTT))
    /\ MWE := addEqs(MWO, substBase(NAMET, VALT)
                      shiftEqs(NAMET)
                      liftEqs(NAMET, SUBSTT)
                      createSpecificEqs(getOps(M), SUBSTT,
                        NAMET, INAMET, CTL)
                      createIdentityEq(CTL, SUBSTT)) .
```

The effect of applying the second equation is as follows. First, the name of the source module is prefixed with the string `CINNI` using the `renameModule` operator. Then, the following sorts, operators, and equations are added using the `addSorts`, `addOps`, and `addEqs` operators, respectively:

- The specified sort for substitutions.
- Operators for the three types of substitutions.
- For each constructed sort in the source module, one operator for prefixing substitutions.
- Equations defining the semantics of the simple, shift, and lifted substitution.
- One *syntax-specific equation* for each syntactic constructor of L.
- Identity equations are added to avoid unnecessary substitutions.

The operators to rename a module, and to add sorts, operators, or equations are not described here. A detailed description of these operators is given in the subsection "A Deadlock-Freedom Transformation" of [35, p. 480]. The automatic creation of the CINNI

operators and equations is described in more detail in the next subsections. Subsection B.4.1 describes how the CINNI operators are created, and Subsection B.4.2 shows how the CINNI equations, including the *syntax-specific equations*, are created.

B.4.1. Creation of CINNI operators

The meta-representation of a Maude operator is a term of sort `OpDecl` and is defined in the Maude module `META-MODULE`. Terms of sort `OpDeclSet` represent sets of operators and can be concatenated using the associative and commutative operator `__` for which the term `none` acts as identity. We construct the operators for the simple, shift, and lifted substitution by using the auxiliary operators `simpleSubst`, `shiftSubst`, and `liftSubst`, respectively. The equation

```
eq simpleSubst(NAMET, VALT, SUBSTT)
  = (op '`[_:=_`] : NAMET VALT -> SUBSTT [none] . ) .
```

takes the type of *names, values*, and *substitutions*, as argument, and yields the meta-representation of the simple substitution operator. In our running example, the term

```
simpleSubst('Qid, 'Chan, 'Subst)
```

would be reduced to the following term, which is the meta-representation of the simple substitution operator of CINNI_π.

```
op '`[_:=_`] : 'Qid 'Chan -> 'Subst [none] .
```

The equations

```
eq shiftSubst(NAMET, SUBSTT)
  = (op '`[shift_`] : NAMET -> SUBSTT [none] . ) .
eq liftSubst(NAMET, SUBSTT)
  = (op '`[lift__`] : NAMET SUBSTT -> SUBSTT [none] . ) .
```

take the type of *names* and *substitutions*, and create the meta-representation of the shift and lifted substitution.

Additionally, for each sort that is constructed in the source module, the corresponding operators to prepend substitutions are added. The operator `substOps` is defined recursively on the structure of the first argument. The base case takes a term of sort `Type`, and the sort of substitutions and creates the meta representation of the prefixing operator for that type.

```
eq substOps(T, ST) = (op '__ : ST T -> T [none] . ) .
```

The two recursive cases

```
ceq substOps(T TL, ST) = substOps(T, ST) substOps(TL, ST) if T =/= nil .
ceq substOps(T TL, ST) = substOps(TL, ST) if T == nil .
```

decompose the first argument — of sort `TypeList` — structurally.

B.4.2. Creation of CINNI equations

As for the meta-representation of operators, the meta-representation of equations can be found in the Maude module `META-MODULE`. Equations and sets of equations are represented at the meta-level by terms of the sorts `Equation` and `EquationSet`.

We first describe the creation of the equations that define the semantics of the CINNI substitution operators. Then, we describe the creation of the *syntax-specific equations*.

Creation of the equations for the CINNI operators

The auxiliary operators `substBase`, `shiftEqs`, and `liftEqs` create the meta representation of the equations defining the semantics of substitutions. For example, the equations

```
eq shiftEqs(NAMET) =
  shiftEq1(NAMET) shiftEq2(NAMET) .

eq shiftEq1(NAMET) =
  (eq '__['`[shift_`][qid("X:" + string(NAMET))],
    '_`{_`}[qid("X:" + string(NAMET)), 'M:Nat]]
  = '_`{_`}[qid("X:" + string(NAMET)), 's_['M:Nat]] [none] .) .

eq shiftEq2(NAMET) =
  (ceq '__['`[shift_`][qid("X:" + string(NAMET))],
    '_`{_`}[qid("Y:" + string(NAMET)), 'M:Nat]]
  = '_`{_`}[qid("Y:" + string(NAMET)), 'M:Nat]
  if '_=/=_[(qid("Y:" + string(NAMET)),
    qid("X:" + string(NAMET)))] = 'true.Bool [none] .) .
```

take the type of *names* as an argument and return the meta-representation of the following two equations of the CINNI calculus:

$$\uparrow_X X_m = X_{m+1}$$
$$\uparrow_X Y_n = Y_n \text{ if } X \neq Y$$

The equations defining the operators `substBase` and `shiftEqs` are omitted for the sake of brevity.

Creation of the *syntax specific equations*

In a last step, the *syntax-specific equations* are created. A schematic overview of the auxiliary functions, that are used, is shown in Figure B.2. *Syntax-specific equations* have to be created for each operator of the source module. The operator `getOperators` returns the meta-representation of the operators defined in the given module. For each of these operators, say `f`, the `createSpecific` operator is executed. Using the two auxiliary functions `createLeft` and `createRight`, which create the left-hand and right-hand side of the *syntax-specific equation*, the equations are created.

Let us assume that the declaration for the operator `f` contains as a `metadata` attribute,

```
op f : T1 ... TN -> T [ctor metadata i1->j1,...,iM->jM]
```

with $\mathtt{i}_1,\ldots,\mathtt{i}_M,\mathtt{j}_1,\ldots,\mathtt{j}_M \in \{1,...,N\}$. Thus, the operator `f` binds the argument $\mathtt{i}_k$ in the argument $\mathtt{j}_k$ for $k \in \{1,\ldots,M\}$. Furthermore, we assume that no two *names* are bound in the same argument. This is expressed by the requirement that $\mathtt{j}_k \neq \mathtt{j}_l$ for all $l \neq k$.

If we are applying a substitution `S` to $\mathtt{f}(t_1,\ldots,t_n)$, then the substitutions $\mathtt{S}_j$ for the jth argument t_j is of the form `[shift` $\mathtt{X}_i$`:`$\mathtt{T}_i$ `S]` if `i->j` appeared in the metadata declaration of `f`, i.e., if there is an argument i that is bound in the argument j. Otherwise, if no argument is bound in the argument j, the substitution $\mathtt{S}_j$ is equal to `S`. To better illustrate how the algorithm works, we create the *syntax specific equation* based on the operator declaration of the

```
op new[_]_ : Qid Trm -> Trm [ctor metadata "1->2"] .
```

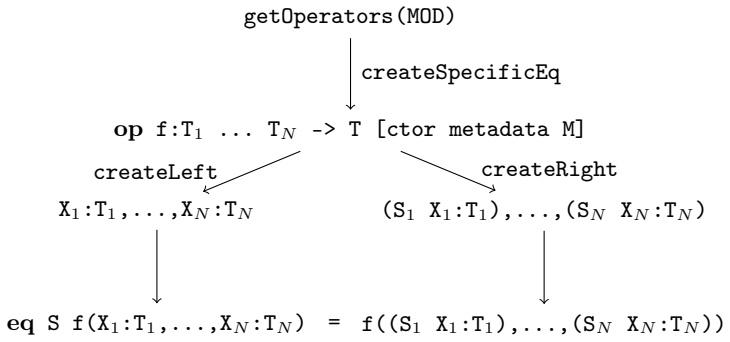


Figure B.2.: Creation of a *syntac-specific* equation

operator of our running example. The meta-representation of the *syntax specific equation* is thereby defined by the equation

```
eq __[S:'Subst, 'new'[_']_[VAR0:'Qid, VAR1:'Trm]] =
  'new'[_']_[VAR0:'Qid,'lift[VAR0:'Qid, S:'Subst]] .
```

Creating the left-hand side of a *syntax-specific equation*. Given a list of types and the index of the first variable as argument, the `createLeft` operator creates the meta-representation of the argument list of an equation.

```
op createLeft : TypeList Nat -> NeTermList .

eq createLeft(T TL, INDEX) =
  createLeft(T, INDEX) , createLeft(TL, INDEX + 1) .
eq createLeft(T, INDEX) =
  qid("VAR" + string(INDEX, 10) + ":" + string(T)) .
```

The first equation recursively decomposes the given term of sort `TypeList`, and counts the current argument position in the second argument. If the first argument is a term of sort `Type`, the second equation creates the meta-representation $\texttt{VAR}_i$ of a variable at argument position i. The term

```
createLeft('Qid 'Trm, 0)
```

in our running example is reduced to

```
VAR0:'Qid, VAR1:'Trm
```

which is the left-hand side of the *syntax specific equation* of the operator

```
op new[_]_ : Qid Trm -> Trm [ctor metadata "1->2"] .
```

Creating the right-hand side of a *syntax-specific equation*. The right-hand side of a *syntax-specific equation* also depends on the `metadata` attribute. Thus, the operator `createRight` takes the list of types from the operator declaration, the current index of the variable, the variable for the substitution, a list of types that should be lifted, the mapping between arguments and the bound arguments, and the type of *names* as arguments, and creates the meta-representation of the list of variables for the right-hand side of the equation.

```
op createRight : TypeList Nat Variable TypeList Map{Nat, Nat} Type
  -> NeTermList .
```

The first equation recursively decomposes the first argument of `TypeList`, and increases the index of the current variable. The second equation creates the meta-representation, if the first argument is a term of sort `Type`. If the type is not included in the list of relevant types for substitution, the substitution is omitted. Otherwise, as discussed above, if the argument at the current index binds a *name*, i.e., `$hasMapping(MAP, INDEX)= true`, then the lifted substitution is created. Otherwise, the substitution is simply passed down to the argument.

```
eq createRight(T TL, INDEX, SVAR, TTL, MAP, VNT) =
  createRight(T, INDEX, SVAR, TTL, MAP, VNT) ,
  createRight(TL, INDEX + 1, SVAR, TTL, MAP, VNT) .

eq createRight(T, INDEX, SVAR, TTL, MAP, VNT) =
  if T in TTL then
    if ($hasMapping(MAP, INDEX)) then
      '__['`[lift__`][qid("VAR" + string(MAP[INDEX], 10) + ":" +
      string(VNT)) ,SVAR], qid("VAR" + string(INDEX, 10) + ":" +
      string(T))]
    else
      '__[SVAR, qid("VAR" + string(INDEX, 10) + ":" + string(T))]
    fi
  else
    qid("VAR" + string(INDEX, 10) + ":" + string(T))
  fi .
```

In our running example, the right-hand side of the equation is created by the term

```
createRight('Qid 'Trm, 0, 'S:Subst, 'Qid 'Trm, 1->2, 'Qid)
```

This results in the meta-representation

```
VAR0:'Qid,'lift[VAR0:'Qid, S:'Subst]
```

Creating the *syntax-specific equation.* To bring it all together, `createSpecificEqs` and `createSpecificEq` create the *syntax specific equations*.

```
op createSpecificEqs : OpDeclSet Type Type Type TypeList
  -> EquationSet .
op createSpecificEq : OpDecl Type Type Type TypeList -> EquationSet .
```

Except for the first parameter — either a set of operator declarations, or a single operator declaration — both operators take the same parameters: the type of substitutions, the type of *names*, the type of `indexed names`, and a list of types that can be substituted. The operator `createSpecificEqs` recursively creates the *syntax-specific* equations using the `createSpecificEq` operator.

```
eq createSpecificEqs(OPD, SUBSTT, VARNAMET, VART, TERMTL)
  = createSpecificEq(OPD, SUBSTT, VARNAMET, VART, TERMTL) .
ceq createSpecificEqs(OPD OPDS, SUBSTT, VARNAMET, VART, TERMTL)
  = createSpecificEqs(OPD, SUBSTT, VARNAMET, VART, TERMTL)
    createSpecificEqs(OPDS, SUBSTT, VARNAMET, VART, TERMTL)
  if OPDS =/= none .
```

The behavior of the `createSpecificEq` operator is defined in three equations. First, if the operator's definition contains the `ctor` and `metadata S` attributes, `S` is parsed, and the `createLeft` and `createRight` operators are used to construct the resulting meta-representation of the *syntax-specific* equation.

```
ceq createSpecificEq((op N : TL -> TERMT [ctor metadata(S) AS].),
  SUBSTT, VARNAMET, VART, TERMTL) =
  (eq '__[SVAR, N[createLeft(TL, 0)]]
    = N[createRight(TL, 0, SVAR, (TERMTL VART), getPairs(S),
      VARNAMET)] [none] .)
if SVAR := qid("S:" + string(SUBSTT))
  /\ TL =/= nil /\ TL in (TERMTL VART) .
```

Second, if the operator's definition contains the `ctor` and `id(ID)` attribute (but no `metadata` attribute), then the substitution is simply passed down to the arguments. A conditional equation is created, since all arguments are required to be unequal to the identity element of the operator to prevent infinite loops.

```
ceq createSpecificEq((op N : TL -> TERMT [ctor id(ID) AS].),
  SUBSTT, VARNAMET, VART, TERMTL) =
  (ceq '__[SVAR, N[createLeft(TL, 0)]]
    = N[createRight(TL, 0, SVAR, (TERMTL VART), empty, VARNAMET)]
   if createUnequalToId(TL, ID, 0) [none] . )
if SVAR := qid("S:" + string(SUBSTT))
  /\ TL =/= nil /\ TL in (TERMTL VART) .
```

Finally, if the `ctor` attribute is contained in the operator's definition (and no `metadata` or `id` attribute), the *syntax-specific* equation is created using the `createLeft` and `createRight` operators.

```
eq createSpecificEq((op N : TL -> TERMT [AS].),
  SUBSTT, VARNAMET, VART, TERMTL) =
  if ctor in AS and TL =/= nil and TL in (TERMTL VART) then
    (eq '__[qid("S:" + string(SUBSTT)), N[createLeft(TL, 0)]]
    = N[createRight(TL, 0, qid("S:" + string(SUBSTT)),
      (TERMTL VART), empty, VARNAMET)] [none] .)
  else
    none
  fi [owise] .
```

B.5. The createCINNI Tool

The Full Maude `show module` command is used to retrieve the Maude representation of the meta-module that is created using the `cinni` command. Thus, the created meta-module has to be loaded in the Full Maude database, then printed using the `show module`, and finally the result has to be filtered. The createCINNI tool is defined by the shell script

```
#!/bin/bash
if [ $# -ne 6 ]
then
echo "Usage: ./createCINNI.sh {module name} {substitution sort}
  {name sort} {indexed name sort} {value sort} {module file}"
exit 65
fi
```

```
echo "
(select META-LEVEL .)
(fmod CREATE-CINNI is
  ex META-LEVEL .
  ex CINNI-META .
  op module : -> Module .
  eq module = cinni('$1, '$2, '$3, '$4, '$5) .
endfm)
(load module .)
(show module CINNI-$1 .)
q" | maude -no-prelude -no-banner -no-advise -no-wrap -no-ansi-color prelude.maude
     CINNIMETA.maude $6 full-maude26.maude | awk '/fmod/,/endfm/' > CINNI-$6
```

which takes six parameters: The name of the source module, the required sort of substitutions, the sort of *names*, the sort of *indexed names*, the sort of *values*, and a the name of the file containing the source module.

Basically, a new Full Maude module with name `CREATE-CINNI` is created, which extends the `META-LEVEL` and the `CINNI-META` module. Additionally, a constant operator `module` is created that is reduced to the meta-representation of the new module. Then, the new module is loaded into the Full Maude database using the `load` command. Finally, the module is printed using the `show module` command. The result is then filtered with an regular expression and the module is written in a file.

In our example of Milner's π calculus, the createCINNI tool is executed using the command

```
./createCINNI PI-SYNTAX Subst Qid Chan Chan pi-syntax-file
```

The resulting Full Maude module

```
(fmod CREATE-CINNI is
  ex META-LEVEL .
  ex CINNI-META .
  op module : -> Module .
  eq module = cinni('PI-SYNTAX, 'Subst, 'Qid, 'Chan, 'Chan) .
endfm)
```

is loaded in the Full Maude database. The resulting module is then printed and written to the file `CINNI-pi-syntax-file`.

The create CINNI tool can be found on the Maude homepage:
`http://maude.cs.uiuc.edu/tools/createcinni`.

Appendix C
A Modularized Actor Model for Statistical Model Checking

C.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) .
```

[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 $[0, 2^{32} - 1]$.

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

```
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 [29]. 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 C.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 C.1.: Operators to generate random values

[2]Paramters are indirectly set by the equations that define the constant operators `maxZipf` and `skew`.

Appendix D
Guaranteeing Stable Availability under Distributed Denial of Service Attacks

D.1. 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.

Appendix E
QoS Analysis of a Cloud-based Publish/Subscribe Middleware

E.1. Predicate filter generator for the stock exchange information system model

```
var SWITCH : Float .

op SEpfGen : -> [PredicateFilter] .
op SEpfGenSwitch : [Float] -> [PredicateFilter] .
eq SEpfGen = SEpfGenSwitch(genRandom(0.0, 1.0)) .

--- Predicate Filter 1: Listen to orders of a specific Listing.
ceq SEpfGenSwitch(SWITCH) =
  (pos(0) == topic(s(sampleUniWithInt(listings))))
if SWITCH < 0.20 .

--- Predicate Filter 2: Listen to orders of a specific Listing where the volume of
--- the orders is greater or equal to a specific value.
ceq SEpfGenSwitch(SWITCH) =
  (pos(0) == topic(s(sampleUniWithInt(listings)))) ;
  (pos(2) >= floor(paretoValue))
if SWITCH >= 0.20
  /\ SWITCH < 0.25 .

--- Predicate Filter 3: Listen to orders of a specific Listing where the volume of
--- the orders is greater or equal to a specific value and the order-type is BUY.
ceq SEpfGenSwitch(SWITCH) =
  (pos(0) == topic(s(sampleUniWithInt(listings)))) ;
  (pos(2) >= floor(paretoValue)) ;
  (pos(3) == 1.0)
if SWITCH >= 0.25
  /\ SWITCH < 0.275 .

--- Predicate Filter 4: Listen to orders of a specific Listing where the volume of
--- the orders is greater or equal to a specific value and the order-type is SELL.
```

```
ceq SEpfGenSwitch(SWITCH) =
  (pos(0) == topic(s(sampleUniWithInt(listings)))) ;
  (pos(2) >= floor(paretoValue)) ;
  (pos(3) == 0.0)
if SWITCH >= 0.275
  /\ SWITCH < 0.3 .

--- Predicate Filter 5: Listen to orders where the volume of the orders is greater
--- or equal to a specific value.
ceq SEpfGenSwitch(SWITCH) =
  (pos(2) >= floor(paretoValue))
if SWITCH >= 0.3
  /\ SWITCH < 0.4 .

--- Predicate Filter 6: Listen to orders of a specific Listing where the price is
--- greater or equal to a specific value.
ceq SEpfGenSwitch(SWITCH) =
  (pos(0) == topic(s(sampleUniWithInt(listings)))) ;
  (pos(1) >= floor(boxMullerValue(stockValueMean, stockValueStdDeviation / 2.0)))
if SWITCH >= 0.4
  /\ SWITCH < 0.5 .

--- Predicate Filter 7: Listen to orders of a specific Listing where the price is
--- greater or equal to a specific value and the order-type is BUY.
ceq SEpfGenSwitch(SWITCH) =
  (pos(0) == topic(s(sampleUniWithInt(listings)))) ;
  (pos(1) >= floor(boxMullerValue(stockValueMean, stockValueStdDeviation / 2.0)))
  ; (pos(3) == 1.0)
if SWITCH >= 0.5
  /\ SWITCH < 0.6 .

--- Predicate Filter 8: Listen to orders of a specific Listing where the price is
--- greater or equal to a specific value and the order-type is SELL.
ceq SEpfGenSwitch(SWITCH) =
  (pos(0) == topic(s(sampleUniWithInt(listings)))) ;
  (pos(1) >= floor(boxMullerValue(stockValueMean, stockValueStdDeviation / 2.0)))
  ; (pos(3) == 0.0)
if SWITCH >= 0.6
  /\ SWITCH < 0.7 .

--- Predicate Filter 9: Listen to orders of a specific Listing where the price is
--- less or equal to a specific value.
ceq SEpfGenSwitch(SWITCH) =
  (pos(0) == topic(s(sampleUniWithInt(listings)))) ;
  (pos(1) <= floor(boxMullerValue(stockValueMean, stockValueStdDeviation / 2.0)))
if SWITCH >= 0.7
  /\ SWITCH < 0.8 .

--- Predicate Filter 10: Listen to orders of a specific Listing where the price is
--- less or equal to a specific value and the order-type is BUY.
ceq SEpfGenSwitch(SWITCH) =
  (pos(0) == topic(s(sampleUniWithInt(listings)))) ;
  (pos(1) >= floor(boxMullerValue(stockValueMean, stockValueStdDeviation / 2.0)))
  ; (pos(3) == 1.0)
if SWITCH >= 0.8
  /\ SWITCH < 0.9 .
```

```
--- Predicate Filter 11: Listen to orders of a specific Listing where the price is
--- less or equal to a specific value and the order-type is SELL.
ceq SEpfGenSwitch(SWITCH) =
  (pos(0) == topic(s(sampleUniWithInt(listings)))) ;
  (pos(1) >= floor(boxMullerValue(stockValueMean, stockValueStdDeviation / 2.0)))
  ; (pos(3) == 0.0)
if SWITCH >= 0.9 .
```

E.1.1. The module *MODEL-PARAMS*

```
eq listings = 100 .

eq stockValueMean = 1000.0 .
eq stockValueStdDeviation = 100.0 .

eq numBrokers = 20 .
eq numPublishers = 100 .

eq pubRate = 1.000 .
eq procRate = 0.001 .

eq latencyMean = 30.0 .
eq latencyStdDev = 8.0 .
eq latency = boxMullerValue(latencyMean, latencyStdDev) / 1000.0 .

eq predicateFilterGenerator = SEpfGen .
eq contentAttributesGenerator = SEContentAttributesGen .

eq LIMIT = 300.0 .
eq initDelay = 0.05 .

--- Varying parameters
eq numSubscribers = 100 .
eq lifetime = 1.000 .
```

E.2. Initial configuration of the stock exchange information system model

```
eq initState =
  --- Broker in Seattle
  < 1 : Broker |
    routingTable: [0, nilRouting],
    eventQueue: [0, nilQueue],
    subscribers: mtAddressList,
    neighbors: 2,
    drops: 0, forwards: 0, sent: 0 >
  --- Broker in San Francisco
  < 2 : Broker |
    routingTable: [0, nilRouting],
    eventQueue: [0, nilQueue],
    subscribers: mtAddressList,
    neighbors: 1 ; 3 ; 4,
    drops: 0, forwards: 0, sent: 0 >
  --- Broker in San Diego
```

```
< 3 : Broker |
  routingTable: [0, nilRouting],
  eventQueue: [0, nilQueue],
  subscribers: mtAddressList,
  neighbors: 2,
  drops: 0, forwards: 0, sent: 0 >
--- Broker in Champaign
< 4 : Broker |
  routingTable: [0, nilRouting],
  eventQueue: [0, nilQueue],
  subscribers: mtAddressList,
  neighbors: 2 ; 5 ; 6,
  drops: 0, forwards: 0, sent: 0 >
--- Broker in New York
< 5 : Broker |
  routingTable: [0, nilRouting],
  eventQueue: [0, nilQueue],
  subscribers: mtAddressList,
  neighbors: 4 ; 8 ; 9,
  drops: 0, forwards: 0, sent: 0 >
--- Broker in Mexico City
< 6 : Broker |
  routingTable: [0, nilRouting],
  eventQueue: [0, nilQueue],
  subscribers: mtAddressList,
  neighbors: 4 ; 7,
  drops: 0, forwards: 0, sent: 0 >
--- Broker in Rio de Janero
< 7 : Broker |
  routingTable: [0, nilRouting],
  eventQueue: [0, nilQueue],
  subscribers: mtAddressList,
  neighbors: 6,
  drops: 0, forwards: 0, sent: 0 >
--- Broker in Toronto
< 8 : Broker |
  routingTable: [0, nilRouting],
  eventQueue: [0, nilQueue],
  subscribers: mtAddressList,
  neighbors: 5,
  drops: 0, forwards: 0, sent: 0 >
--- Broker in London
< 9 : Broker |
  routingTable: [0, nilRouting],
  eventQueue: [0, nilQueue],
  subscribers: mtAddressList,
  neighbors: 5 ; 10 ; 12,
  drops: 0, forwards: 0, sent: 0 >
--- Broker in Amsterdam
< 10 : Broker |
  routingTable: [0, nilRouting],
  eventQueue: [0, nilQueue],
  subscribers: mtAddressList,
  neighbors: 9 ; 11,
  drops: 0, forwards: 0, sent: 0 >
--- Broker in Madrid
```

```
< 11 : Broker |
  routingTable: [0, nilRouting],
  eventQueue: [0, nilQueue],
  subscribers: mtAddressList,
  neighbors: 10,
  drops: 0, forwards: 0, sent: 0 >
--- Broker in Munich
< 12 : Broker |
  routingTable: [0, nilRouting],
  eventQueue: [0, nilQueue],
  subscribers: mtAddressList,
  neighbors: 9 ; 13 ; 14,
  drops: 0, forwards: 0, sent: 0 >
--- Broker in Moskow
< 13 : Broker |
  routingTable: [0, nilRouting],
  eventQueue: [0, nilQueue],
  subscribers: mtAddressList,
  neighbors: 12,
  drops: 0, forwards: 0, sent: 0 >
--- Broker in Rome
< 14 : Broker |
  routingTable: [0, nilRouting],
  eventQueue: [0, nilQueue],
  subscribers: mtAddressList,
  neighbors: 12 ; 15,
  drops: 0, forwards: 0, sent: 0 >
--- Broker in Haifa
< 15 : Broker |
  routingTable: [0, nilRouting],
  eventQueue: [0, nilQueue],
  subscribers: mtAddressList,
  neighbors: 14 ; 16 ; 17,
  drops: 0, forwards: 0, sent: 0 >
--- Broker in Johannisburg
< 16 : Broker |
  routingTable: [0, nilRouting],
  eventQueue: [0, nilQueue],
  subscribers: mtAddressList,
  neighbors: 15,
  drops: 0, forwards: 0, sent: 0 >
--- Broker in Mumbai
< 17 : Broker |
  routingTable: [0, nilRouting],
  eventQueue: [0, nilQueue],
  subscribers: mtAddressList,
  neighbors: 15 ; 18 ; 19,
  drops: 0, forwards: 0, sent: 0 >
--- Broker in Sydney
< 18 : Broker |
  routingTable: [0, nilRouting],
  eventQueue: [0, nilQueue],
  subscribers: mtAddressList,
  neighbors: 17,
  drops: 0, forwards: 0, sent: 0 >
--- Broker in Shanghai
```

```
  < 19 : Broker |
    routingTable: [0, nilRouting],
    eventQueue: [0, nilQueue],
    subscribers: mtAddressList,
    neighbors: 17 ; 20,
    drops: 0, forwards: 0, sent: 0 >
  --- Broker in Tokyo
  < 20 : Broker |
    routingTable: [0, nilRouting],
    eventQueue: [0, nilQueue],
    subscribers: mtAddressList,
    neighbors: 19,
    drops: 0, forwards: 0, sent: 0 >
  < 21 : PublisherGenerator | mt >
  [0.0, 21 <- generate]
  < 22 : SubscriberGenerator | mt >
  [0.0, 22 <- generate]
  < 23 >
  { 0.0 | nil } .
```

E.3. Initial configuration of the Cloud-based stock exchange information system model

```
eq initState =
  --- Broker in Seattle
  < 1 : BrokerReplicator |
    brokerList: mtAddressList,
    config: < 1 . 1 >
            < 1 . 0 : BrokerData |
              routingTable: [0, nilRouting],
              neighbors: 2,
              subscribers: mtAddressList >,
    eventCnt: 0,
    eventFlowRate: 0.0 >
  [0.0, 1 <- startBroker]
  --- Broker in San Francisco
  < 2 : BrokerReplicator |
    brokerList: mtAddressList,
    config: < 2 . 1 >
            < 2 . 0 : BrokerData |
              routingTable: [0, nilRouting],
              neighbors: 1 ; 3 ; 4,
              subscribers: mtAddressList >,
    eventCnt: 0,
    eventFlowRate: 0.0 >
  [0.0, 2 <- startBroker]
  --- Broker in San Diego
  < 3 : BrokerReplicator |
    brokerList: mtAddressList,
    config: < 3 . 1 >
            < 3 . 0 : BrokerData |
              routingTable: [0, nilRouting],
              neighbors: 2,
              subscribers: mtAddressList >,
    eventCnt: 0,
```

```
  eventFlowRate: 0.0 >
[0.0, 3 <- startBroker]
--- Broker in Champaign
< 4 : BrokerReplicator |
  brokerList: mtAddressList,
  config: < 4 . 1 >
          < 4 . 0 : BrokerData |
            routingTable: [0, nilRouting],
            neighbors: 2 ; 5 ; 6,
            subscribers: mtAddressList >,
  eventCnt: 0,
  eventFlowRate: 0.0 >
[0.0, 4 <- startBroker]
--- Broker in New York
< 5 : BrokerReplicator |
  brokerList: mtAddressList,
  config: < 5 . 1 >
          < 5 . 0 : BrokerData |
            routingTable: [0, nilRouting],
            neighbors: 4 ; 8 ; 9,
            subscribers: mtAddressList >,
  eventCnt: 0,
  eventFlowRate: 0.0 >
[0.0, 5 <- startBroker]
--- Broker in Mexico City
< 6 : BrokerReplicator |
  brokerList: mtAddressList,
  config: < 6 . 1 >
          < 6 . 0 : BrokerData |
            routingTable: [0, nilRouting],
            neighbors: 4 ; 7,
            subscribers: mtAddressList >,
  eventCnt: 0,
  eventFlowRate: 0.0 >
[0.0, 6 <- startBroker]
--- Broker in Rio de Janero
< 7 : BrokerReplicator |
  brokerList: mtAddressList,
  config: < 7 . 1 >
          < 7 . 0 : BrokerData |
            routingTable: [0, nilRouting],
            neighbors: 6,
            subscribers: mtAddressList >,
  eventCnt: 0,
  eventFlowRate: 0.0 >
[0.0, 7 <- startBroker]

--- Broker in Toronto
< 8 : BrokerReplicator |
  brokerList: mtAddressList,
  config: < 8 . 1 >
          < 8 . 0 : BrokerData |
            routingTable: [0, nilRouting],
            neighbors: 5,
            subscribers: mtAddressList >,
  eventCnt: 0,
```

```
  eventFlowRate: 0.0 >
[0.0, 8 <- startBroker]
--- Broker in London
< 9 : BrokerReplicator |
  brokerList: mtAddressList,
  config: < 9 . 1 >
          < 9 . 0 : BrokerData |
            routingTable: [0, nilRouting],
            neighbors: 5 ; 10 ; 12,
            subscribers: mtAddressList >,
  eventCnt: 0,
  eventFlowRate: 0.0 >
[0.0, 9 <- startBroker]
--- Broker in Amsterdam
< 10 : BrokerReplicator |
  brokerList: mtAddressList,
  config: < 10 . 1 >
          < 10 . 0 : BrokerData |
            routingTable: [0, nilRouting],
            neighbors: 9 ; 11,
            subscribers: mtAddressList >,
  eventCnt: 0,
  eventFlowRate: 0.0 >
[0.0, 10 <- startBroker]
--- Broker in Madrid
< 11 : BrokerReplicator |
  brokerList: mtAddressList,
  config: < 11 . 1 >
          < 11 . 0 : BrokerData |
            routingTable: [0, nilRouting],
            neighbors: 10,
            subscribers: mtAddressList >,
  eventCnt: 0,
  eventFlowRate: 0.0 >
[0.0, 11 <- startBroker]
--- Broker in Munich
< 12 : BrokerReplicator |
  brokerList: mtAddressList,
  config: < 12 . 1 >
          < 12 . 0 : BrokerData |
            routingTable: [0, nilRouting],
            neighbors: 9 ; 13 ; 14,
            subscribers: mtAddressList >,
  eventCnt: 0,
  eventFlowRate: 0.0 >
[0.0, 12 <- startBroker]

--- Broker in Moskow
< 13 : BrokerReplicator |
  brokerList: mtAddressList,
  config: < 13 . 1 >
          < 13 . 0 : BrokerData |
            routingTable: [0, nilRouting],
            neighbors: 12,
            subscribers: mtAddressList >,
  eventCnt: 0,
```

```
  eventFlowRate: 0.0 >
[0.0, 13 <- startBroker]
--- Broker in Rome
< 14 : BrokerReplicator |
  brokerList: mtAddressList,
  config: < 14 . 1 >
          < 14 . 0 : BrokerData |
            routingTable: [0, nilRouting],
            neighbors: 12 ; 15,
            subscribers: mtAddressList >,
  eventCnt: 0,
  eventFlowRate: 0.0 >
[0.0, 14 <- startBroker]
--- Broker in Haifa
< 15 : BrokerReplicator |
  brokerList: mtAddressList,
  config: < 15 . 1 >
          < 15 . 0 : BrokerData |
            routingTable: [0, nilRouting],
            neighbors: 14 ; 16 ; 17,
            subscribers: mtAddressList >,
  eventCnt: 0,
  eventFlowRate: 0.0 >
[0.0, 15 <- startBroker]
--- Broker in Johannisburg
< 16 : BrokerReplicator |
  brokerList: mtAddressList,
  config: < 16 . 1 >
          < 16 . 0 : BrokerData |
            routingTable: [0, nilRouting],
            neighbors: 15,
            subscribers: mtAddressList >,
  eventCnt: 0,
  eventFlowRate: 0.0 >
[0.0, 16 <- startBroker]
--- Broker in Mumbai
< 17 : BrokerReplicator |
  brokerList: mtAddressList,
  config: < 17 . 1 >
          < 17 . 0 : BrokerData |
            routingTable: [0, nilRouting],
            neighbors: 15 ; 18 ; 19,
            subscribers: mtAddressList >,
  eventCnt: 0,
  eventFlowRate: 0.0 >
[0.0, 17 <- startBroker]

--- Broker in Sydney
< 18 : BrokerReplicator |
  brokerList: mtAddressList,
  config: < 18 . 1 >
          < 18 . 0 : BrokerData |
            routingTable: [0, nilRouting],
            neighbors: 17,
            subscribers: mtAddressList >,
  eventCnt: 0,
```

```
  eventFlowRate: 0.0 >
[0.0, 18 <- startBroker]
--- Broker in Shanghai
< 19 : BrokerReplicator |
  brokerList: mtAddressList,
  config: < 19 . 1 >
          < 19 . 0 : BrokerData |
            routingTable: [0, nilRouting],
            neighbors: 17 ; 20,
            subscribers: mtAddressList >,
  eventCnt: 0,
  eventFlowRate: 0.0 >
[0.0, 19 <- startBroker]
--- Broker in Tokyo
< 20 : BrokerReplicator |
  brokerList: mtAddressList,
  config: < 20 . 1 >
          < 20 . 0 : BrokerData |
            routingTable: [0, nilRouting],
            neighbors: 19,
            subscribers: mtAddressList >,
  eventCnt: 0,
  eventFlowRate: 0.0 >
[0.0, 20 <- startBroker]
< 21 : PublisherGenerator | mt >
[0.0, 21 <- generate]
< 22 : SubscriberGenerator | mt >
[0.0, 22 <- generate]
< 23 >
{ 0.0 | nil } .
```

Bibliography

[1] M. Abadi, M. Burrows, M. Manasse, and T. Wobber. Moderately hard, memory-bound functions. *ACM Transactions on Internet Technology*, 5(2):299–327, 2005.

[2] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986.

[3] G. Agha, S. Frolund, R. Panwar, and D. Sturman. A linguistic framework for dynamic composition of dependability protocols. In *IFIP Transactions*, pages 345–363, 1993.

[4] G. Agha, C. Gunter, M. Greenwald, S. Khanna, J. Meseguer, K. Sen, and P. Thati. Formal modeling and analysis of DoS using probabilistic rewrite theories. In *FCS*, 2005.

[5] G. Agha, J. Meseguer, and K. Sen. Pmaude: Rewrite-based specification language for probabilistic object systems. *Electronic Notes in Theoretical Computer Science*, 153:213–239, 2006.

[6] M. AlTurki. *Rewriting-based formal modeling, analysis and implementation of real-time distributed services*. PhD thesis, University of Illinois at Urbana-Champaign, 2011. `http://hdl.handle.net/2142/26231`.

[7] M. AlTurki and J. Meseguer. Dist-Orc: A Rewriting-based Distributed Implementation of Orc with Formal Analysis. In *Theoretical Computer Science*, pages 26–45, 2010.

[8] 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. 2011.

[9] 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.

[10] Amazon.
See: `http://aws.amazon.com/simpledb/` (Visited: September, 2011).

[11] Amazon. Amazon EC2 Pricing.
See: `http://aws.amazon.com/ec2/pricing/` (Visited: September, 2011).

[12] Amazon. Amazon Elastic Block Store (EBS).
See: `http://aws.amazon.com/ebs/` (Visited: November, 2011).

[13] Amazon. Amazon Elastic Block Store (EBS).
See: `http://aws.amazon.com/ebs/` (Visited: September, 2011).

[14] Amazon. Amazon Elastic Compute Cloud (Amazon EC2).
See: `http://aws.amazon.com/ec2/` (Visited: November, 2011).

[15] Amazon. Amazon Relational Database Service (Amazon RDS).
See: `http://aws.amazon.com/rds/` (Visited: November, 2011).

[16] Amazon. Amazon Simple Storage Service (Amazon S3).
See: `http://aws.amazon.com/s3/` (Visited: November, 2011).

[17] Amazon. Amazon SimpleDBAmazon SimpleDB.
See: `http://aws.amazon.com/simpledb/` (Visited: November, 2011).

[18] Arash Ferdowsi. Yesterday's Authentication Bug.
See: `http://blog.dropbox.com/?p=821` (Visited: December, 2011).

[19] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical report, University of California at Berkeley, 2009.

[20] A. Aziz, V. Singhal, F. Balarin, R. Brayton, and A. Sangiovanni-Vincentelli. It usually works: The temporal logic of stochastic systems. In *Computer Aided Verification*, volume 939 of *Lecture Notes in Computer Science*, pages 155–165. 1995.

[21] C. Baier, J.-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time markov chains. In *International Conference on Concurrency Theory*, pages 146–161, 1999.

[22] M. H. t. Beek, S. Gnesi, F. Mazzanti, and C. Moiso. Formal modelling and verification of an asynchronous extension of soap. In *European Conference on Web Services*, pages 287–296, 2006.

[23] K. Berkling. *A Symmetric Complement to the Lambda Calculus*, volume 76–77 of *Bonn Interner Bericht ISF*. Gesellschaft für Mathematik und Datenverarbeitung mbH, 1976.

[24] K. Berkling and E. Fehr. A Consistent Extension of the Lambda Calculus as a Base for Functional Programming Languages. *Information and Control*, 55(1):89–101, 1982.

[25] L. Bettini, R. De Nicola, and R. Pugliese. Klava: a java package for distributed and mobile applications. *Software - Practice and Experience*, 32:1365–1394, 2002.

[26] L. Bettini, R. D. Nicola, R. Publiese, and G. Ferrari. Interactive mobile agents in x-klaim. In *IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 110–115, 1998.

[27] Bianchi, S. and Felber, P. and Gradinariu, M. Content-Based Publish/Subscribe Using Distributed R-Trees. In *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 537–548. Springer, 2007.

[28] P. Borovanský, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285:155–185, 2002.

[29] 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.

[30] C. Braga and J. Meseguer. Modular Rewriting Semantics in Practice. *Electronic Notes in Theoretical Computer Science*, 117:393–416, 2005.

[31] L. Cardelli and A. D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155, 1998.

[32] Carzaniga, A. and Rosenblum, D. S. and Wolf, A. L. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19:332–383, 2001.

[33] R. Chadha, C. A. Gunter, J. Meseguer, R. Shankesi, and M. Viswanathan. Modular Preservation of Safety Properties by Cookie-Based DoS-Protection Wrappers. In *Formal Methods for Open Object-Based Distributed Systems*, volume 5051 of *Lecture Notes in Computer Science*, pages 39–58, 2008.

[34] T. Chou. *Cloud: Seven Clear Business Models.* Active Book Press, 2010.

[35] 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, 2007.

[36] Cugola, G. and Jacobsen, H.-A. Using publish/subscribe middleware for mobile systems. *SIGMOBILE Mobile Computing and Communications Review*, 6:25–33, 2002.

[37] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 75(5):381–392, 1972.

[38] R. De Nicola, G. L. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24:315–330, 1998.

[39] R. Diaconescu and K. Futatsugi. *CafeOBJ report: the language, proof techniques, and methodologies for object-oriented algebraic specification.* AMAST Series. World Scientific, 1998.

[40] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

[41] R. Drucker and A. Frank. A C++/Linda Model for Distributed Objects. In *Israeli Conference on Computer-Based Systems and Software Engineering*, pages 30–37, 1996.

[42] F. Durán, S. Eker, P. Lincoln, and J. Meseguer. Principles of Mobile Maude. In *International Symposium on Agent Systems and Applications*, volume 1882 of *Lecture Notes in Computer Science*, pages 73–85, 2000.

[43] J. Eckhardt. A formal analysis of security properties in cloud computing. Master's thesis, Ludwig Maximilian University of Munich, 2011.

[44] P. V. Eijk and M. Diaz, editors. *Formal Description Technique Lotos: Results of the Esprit Sedos Project*. Elsevier, 1989.

[45] Eugster, P.T. and Felber, P.A. and Guerraoui, R. and Kermarrec, A.-M. The many faces of publish/subscribe. *ACM Computing Surveys*, 35:114–131, 2003.

[46] H. Fayol. *Administration industrielle et générale: prévoyance, organisation, commandement, coordination, contrôle*. Dunod, 1947.

[47] I. Foster. What is the Grid? - a three point checklist. *GRIDtoday*, 1(6), July 2002.

[48] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[49] S. L. Garfinkel. *Architects of the Information Society: Thirty-Five Years of the Laboratory for Computer Science at MIT*. MIT Press, 1999.

[50] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7:80–112, 1985.

[51] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35:97–107, 1992.

[52] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*, pages 3–167. Kluwer Academic Publishers, 2000.

[53] Google. Google App Engine.
See: `http://code.google.com/appengine/` (Visited: November, 2011).

[54] Google. Google Cloud SQL.
See: `http://code.google.com/intl/en/apis/sql/` (Visited: November, 2011).

[55] Google. Google finance.
See: `http://finance.google.com/` (Visited: September, 2011).

[56] Google. Google Products.
See: `http://www.google.com/intl/en/about/products/index.html`
(Visited: November, 2011).

[57] C. Gunter, S. Khanna, K. Tan, and S. Venkatesh. DoS Protection for Reliably Authenticated Broadcast. In *Network and Distributed System Security Symposium*, 2004.

[58] Haldane, A.G. Patience and Finance.
See: `http://www.bankofengland.co.uk/publications/speeches/2010/speech445.pdf`
(Visited: September, 2011).

[59] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6:512–535, 1994.

[60] C. Hewitt and H. G. Baker. Laws for communicating parallel processes. In *IFIP Congress*, pages 987–992, 1977.

[61] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *International Joint Conference on Artificial Intelligence*, pages 235–245, 1973.

[62] J. Hillston. *A compositional approach to performance modelling*. Cambridge University Press, 1996.

[63] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978.

[64] R. V. Hogg, A. Craig, and J. W. Mckean. *Introduction to Mathematical Statistics*. Prentice Hall, 2004.

[65] Z. Huang, W. Wu, K. Nahrstedt, R. Rivas, and A. Arefin. SyncCast: synchronized dissemination in multi-site interactive 3D tele-immersion. In *Annual ACM conference on Multimedia systems*, pages 69–80, 2011.

[66] S. Khanna, S. Venkatesh, O. Fatemieh, F. Khan, and C. Gunter. Adaptive Selective Verification. In *IEEE Conference on Computer Communications*, pages 529–537, 2008.

[67] N. Kumar, K. Sen, J. Meseguer, and G. Agha. A Rewriting Based Model for Probabilistic Distributed Object Systems. In *Lecture Notes in Computer Science*, volume 2884, pages 32–46, 2003.

[68] N. Kumar, K. Sen, J. Meseguer, and G. Agha. Probabilistic rewrite theories: Unifying models, logics and tools. Technical report, University of Illinois at Urbana-Champaign, 2003.

[69] S. Lafrance and J. Mullins. An Information Flow Method to Detect Denial of Service Vulnerabilities. *Journal of Universal Computer Science*, 9(11):1350–1369, 2003.

[70] A. Mahimkar and V. Shmatikov. Game-based Analysis of Denial-of-Service Prevention Protocols. In *IEEE Computer Security Foundations Workshop*, pages 287–301, 2005.

[71] D. Mankins, R. Krishnan, C. Boyd, J. Zao, and M. Frentz. Mitigating Distributed Denial of Service Attacks with Dynamic Resource Pricing. In *Annual Computer Security Applications Conference*, page 411, 2001.

[72] MasterCard. MasterCard Statement.
See: `http://www.businesswire.com/news/home/20101208005866/en/MasterCard-Statement`
(Visited: September, 2011).

[73] MasterCard. MasterCard Statement.
See: `http://www.businesswire.com/news/home/20101208006660/en/MasterCard-Statement`
(Visited: September, 2011).

[74] C. Meadows. A Formal Framework and Evaluation Method for Network Denial of Service. In *IEEE Computer Security Foundations Workshop*, 1999.

[75] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[76] J. Meseguer. Membership algebra as a logical framework for equational specification. In *Recent Trends in Algebraic Development Techniques*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.

[77] J. Meseguer. Twenty years of rewriting logic. *Journal of Logic and Algebraic Programming*, 2011.

[78] J. Meseguer and C. L. Talcott. Semantic models for distributed object reflection. In *European Conference on Object-Oriented Programming*, pages 1–36, 2002.

[79] Microsoft. SQL Azure Database.
See: `http://www.microsoft.com/windowsazure/features/database/`
(Visited: November, 2011).

[80] Microsoft. Steve Ballmer: Cloud Computing.
See: `http://www.microsoft.com/presspass/exec/steve/2010/03-04cloud.mspx`
(Visited: November, 2011).

[81] Microsoft. The STRIDE Threat Model.
See: `http://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx`
(Visited: September, 2011).

[82] Microsoft. Windows Azure Platform.
See: `http://www.microsoft.com/windowsazure/` (Visited: November, 2011).

[83] R. Milner. *A Calculus of Communicating Systems*. Springer, 1982.

[84] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[85] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[86] J. Misra. Computation orchestration: A basis for wide-area computing. In *Journal of Software and Systems Modeling*, pages 10–1007, 2006.

[87] H. R. Motahari-Nezhad, B. Stephenson, and S. Singhal. Outsourcing Business to Cloud Computing Services: Opportunities and Challenges. *Development*, 10(4):1–17, 2009.

[88] R. D. Nicola and M. Loreti. A modal logic for klaim. In *International Conference on Algebraic Methodology and Software Technology*, AMAST, pages 339–354, 2000.

[89] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1–2):161–196, 2007.

[90] OMG. Notification Service.
See: `http://www.omg.org/spec/NOT/` (Visited: September, 2011).

[91] Oracle. Java Message Service (JMS).
See: `http://www.oracle.com/technetwork/java/jms/index.html`
(Visited: September, 2011).

[92] PlanetLab. PlanetLab.
See: `http://www.planet-lab.org/node/1` (Visited: September, 2011).

[93] T. Pongthawornkamol, K. Nahrstedt, and G. Wang. Probabilistic QoS modeling for reliability/timeliness prediction in distributed content-based publish/subscribe systems over best-effort networks. In *International Conference on Autonomic Computing*, pages 185–194, 2010.

[94] 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).

[95] salesforce.com. force.com.
See: `http://www.force.com/` (Visited: November, 2011).

[96] K. Sen, M. Viswanathan, and G. Agha. Statistical model checking of black-box probabilistic systems. In *Computer Aided Verification*, pages 202–215, 2004.

[97] K. Sen, M. Viswanathan, and G. Agha. On statistical model checking of stochastic systems. In *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 251–255. 2005.

[98] K. Sen, M. Viswanathan, and G. Agha. VESTA: A Statistical Model-checker and Analyzer for Probabilistic Systems. In *International Conference on the Quantitative Evaluation of Systems*, pages 251–252, 2005.

[99] M. Stehr. CINNI - A Generic Calculus of Explicit Substitutions and its Application to λ- ς- and π-Calculi. *Electronic Notes in Theoretical Computer Science*, 36:70–92, 2000.

[100] E. Steven, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In *Workshop on Rewriting Logic and its Applications*, volume 71 of *Electronic Notes in Theoretical Computer Science*, 2002.

[101] S. T.-F., G. Rosu, and J. Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 207:305–340, 2009.

[102] C. L. Talcott. Coordination Models Based on a Formal Model of Distributed Object Reflection. *Theoretical Computer Science*, 150:143–157, 2006.

[103] Terpstra, W.W. and Behnel, S. and Fiege, L. and Zeidler, A. and Buchmann, A.P. A peer-to-peer approach to content-based publish/subscribe. In *International workshop on Distributed Event-based Systems*, pages 1–8, 2003.

[104] P. Thati. *A theory of testing for asynchronous concurrent systems*. PhD thesis, University of Illinois at Urbana-Champaign, 2003.

[105] The Systems Research at Harvard (SYRAH) Group. Network Coordinate Research at Harvard.
See: `http://www.eecs.harvard.edu/~syrah/nc/` (Visited: September, 2011).

[106] The Wall Street Journal. Larry ellison's brilliant anti cloud computing rant.
See: `http://blogs.wsj.com/biztech/2008/09/25/larry-ellisons-brilliant-anti-cloud-computing-rant/` (Visited: April, 2011).

[107] Triantafillou, P. and Aekaterinidis, I. Content-based Publish/Subscribe over Structured P2P Networks. In *International workshop on Distributed Event-based Systems*, pages 24–25, 2004.

[108] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: towards a cloud definition. *Computer Communication Review*, 39:50–55, 2008.

[109] N. Venkatasubramanian, C. L. Talcott, and G. Agha. A formal model for reasoning about adaptive qos-enabled middleware. *ACM Transactions on Software Engineering and Methodology*, 13(1):86–147, 2004.

[110] A. Verdejo and N. Martí-Oliet. Executable structural operational semantics in Maude. *Journal of Logic and Algebraic Programming*, 67(1–2):226–293, 2006.

[111] W3C. Request-Response Message Exchange Pattern.
See: `http://www.w3.org/TR/2003/PR-soap12-part2-20030507/#singlereqrespmep`
(Visited: September, 2011).

[112] W3C. Web Services Eventing (WS-Eventing).
See: `http://www.w3.org/TR/2011/CR-ws-eventing-20110428/`
(Visited: September, 2011).

[113] M. Walfish, M. Vutukuru, H. Balakrishnan, D. R. Karger, and S. Shenker. DDoS defense by offense. In *ACM SIGCOMM*, pages 303–314, 2006.

[114] C. Wang, G. Wang, H. Wang, A. Chen, and R. Santiago. Quality of Service (QoS) Contract Specification, Establishment, and Monitoring for Service Level Management. In *IEEE International Enterprise Distributed Object Computing Conference Workshops*, volume 6, pages 49–49, 2006.

[115] X. Wang and M. K. Reiter. Defending Against Denial-of-Service Attacks with Puzzle Auctions. In *IEEE Symposium on Security and Privacy*, page 78, 2003.

[116] Wang, G. and Chen, A. and Wang, C. and Fung, C. and Uczekaj. S. Integrated Quality of Service (QoS) Management in Service-Oriented Enterprise Architectures. *IEEE International Enterprise Distributed Object Computing Conference*, 0:21–32, 2004.

[117] Washington State University. GridStat.
See: `http://www.gridstat.net/` (Visited: September, 2011).

[118] G. Wells. Coordination Languages: Back to the Future with Linda. In *International Workshop on Coordination and Adaptation Techniques for Software*, pages 87–98, 2005.

[119] G. C. Wells, A. G. Chalmers, and P. G. Clayton. Linda implementations in Java for concurrent systems. *Concurrency and Computation: Practice and Experience*, 16:1005–1022, 2003.

[120] M. Wirsing, A. Clark, S. Gilmore, M. Hölzl, N. Koch, and A. Schroeder. Semantic-based development of service-oriented systems. In *International Conference on Formal Methods for Networked and Distributed Systems*, volume 4229 of *Lecture Notes in Computer Science*, pages 24–45, 2006.

[121] Yahoo! Yahoo! Finance.
See: `http://finance.yahoo.com/` (Visited: September, 2011).

[122] C.-F. Yu and V. Gligor. A Specification and Verification Method for Preventing Denial of Service. *IEEE Transactions on Software Engineering*, 16(6):581–592, 1990.