REWRITING-BASED FORMAL MODELING, ANALYSIS AND IMPLEMENTATION OF REAL-TIME DISTRIBUTED SERVICES

BY

MUSAB AHMAD AL-TURKI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Doctoral Committee:

Professor José Meseguer, Chair and Director of Research
Professor Gul Agha
Professor Carl A. Gunter
Professor Jayadev Misra, University of Texas at Austin
Associate Professor Grigore Rosu

ABSTRACT

The last decade has seen an explosive growth of both: (1) enterprise service-oriented software systems, for managing enterprise resources and automating business processes, and (2) user-centric, cloud-based web applications, which provide richer experiences and more intelligent services to end-users than traditional, monolithic applications. The adoption of systems that are based on Internet-accessible software components, a class of distributed software systems to which we simply refer as *Internet software*, is expected to grow tremendously in the future. Nevertheless, designing and developing dependable Internet software poses a unique set of challenges, making the already difficult issue of whether a deployed system meets its specification requirements even harder to address than for traditional software systems.

In this dissertation, we develop formal specification, simulation, prototyping, and formal analysis techniques and tools for distributed software services, based on rewriting logic, the Maude system, and the theory of Orc, with the overall goal of improving the reliability of Internet software. The dissertation focuses on the formal specification and analysis of two fundamentally important aspects of Internet software systems: (1) the correctness of service compositions, and (2) the availability of services.

For service composition specification and analysis, we systematically use and extend methods from the rewriting logic semantics project and apply them to service orchestrations in Orc, providing a simple, elegant and efficient formal model for timed orchestration design and analysis. The rewriting specifications of the semantics of Orc is presented in three main semantics-preserving refinements in order to achieve maximum efficiency and expressiveness: (1) an SOS-based rewriting semantics, (2) a reduction rewriting semantics, and (3) an object-based rewriting semantics. A specification of the the latter in Real-Time Maude

is used as a back-end for a high-level, web-based tool, MOrc, enabling exhaustive formal verification, including model checking, of service orchestrations in Orc. Moreover, the dissertation develops a natural transformation path from formal models of Orc programs to actual, provably-correct, distributed implementations with physical timing, which enable observing actual possible behaviors of service orchestrations in realistic environments.

For the service availability problem, the dissertation extends current methods based on rewriting logic for the specification and analysis of availability properties to improve their efficiency and scalability. In particular, the dissertation first presents parallel versions of the statistical model checking algorithm of Sen, Viswanathan and Agha [1] and the statistical quantitative analysis algorithm of Agha, Meseguer and Sen [2]. The parallel algorithms we propose, which are implemented in a parallel, client/server extension of VeStA, called PVeStA, exploit an inherent parallelization opportunity within these statistical analysis algorithms, where multiple, independent Monte-Carlo simulations are performed. Performance gains as a result of parallelization can in practice be remarkable, as demonstrated using several experiments. Furthermore, using Maude and PVeStA, we apply the rewriting logic approach to availability analysis to the Adaptive Selective Verification (ASV) protocol and verify, in the presence of denial-of-service (DoS) attacks, several of its availability properties, which were previously shown either analytically or statistically by low-level network simulations.

In addition, the dissertation proposes an expressive and modular method for the formal specification and analysis of service availability against DoS in service compositions using generic ASV object wrappers. This is achieved essentially by combining techniques developed for Orc service orchestrations and service availability analysis. The method is illustrated by specifying and analyzing an ASV-endowed service orchestration pattern in Orc.

In memory of my beloved father, Ahmad B. AlTurki

Dedicated to my wife, Hanadi, and my sons, Faisal and Abdurrahman, for their love and support

ACKNOWLEDGMENTS

I, first and foremost, submit my thankful praises to *Allah* (God), the Almighty, the Most Gracious and the Ever Merciful, for granting me strength, guidance and perseverance to undertake and successfully complete this task. This accomplishment would not have been possible without His sufficient grace and mercy.

I would like to express my sincere gratitude and appreciation to my thesis adviser, Prof. José Meseguer, who has always been a source of inspiration for me. I thank him for all the expert help, guidance, and continuous support and encouragement he has been patiently providing throughout the years of my studies and research. Working with him has been an exceptional privilege, and I hope I will be able to continue to enjoy this privilege for years to come.

I would like to extend my sincere thanks and appreciation to my committee members: Prof. Gul Agha, Prof. Carl A. Gunter, Prof. Jayadev Misra (University of Texas at Austin) and Prof. Grigore Rosu, for their invaluable comments and suggestions to improve the thesis. The time and effort they had spent to thoroughly review my thesis and provide feedback is very much appreciated. I thank them for all the fruitful discussions and for their generous words of encouragement.

I am also especially grateful to Prof. Peter Csaba Ölveczky for his invaluable advice and help with several parts of the thesis. My gratitude also extends to Prof. Santiago Escobar, Prof. Francisco J. Durán and Prof. Alberto Verdejo, for their extraordinary help and support, especially with Maude-based formal analysis tools and implementations. Thanks are also due to Dr. Steven Eker for his assistance with Maude, and to Prof. Mahesh Viswanthan for his comments.

I would like to express my sincere gratitude to the Orc research team at the University of

Texas at Austin, led by Prof. Jayadev Misra and Prof. William R. Cook, for their comments and helpful discussions. Special thanks to David Kitchin, for his fruitful thoughts and ideas on Orc's formal semantics, and also to Adrian Quark and Andrew Matsuoka.

I am thankful to my colleagues for their friendship and assistance, especially Traian Florin Şerbănuţă, Mark Hills, Joe Hendrix, Michael Katelman, Ralf Sasse, Camilo Rocha, Kyungmin Bae, Ravinder Shankesi, Fariba Khan, Omid Fatemieh and Raúl Gutiérrez. I thank the Technical Support Group at the department of Computer Science and the IT Help desk at the Coordinated Sciences Laboratory for providing prompt technical assistance whenever I asked for it.

I am also thankful to King Fahd University of Petroleum and Minerals, Saudi Arabia, for sponsoring my doctoral studies and research, and to King Abdullah University of Science and Technology, Saudi Arabia, for their support through the prestigious King Abdullah Scholar Award. The research reported in the thesis has also been partially supported by NSF grants CNS 07-16638, and CCF 09-05584, and by AFOSR grant FA8750-11-22-0084.

I would like to express my unlimited thanks and sincere appreciation for the invaluable advice, tremendous support, and unceasing guidance and encouragement of my late father, my mother, my father-in-law and my mother-in-law. I would also like to express my special thanks to my darling wife, who has patiently provided all peace of mind, comfort and care for easy accomplishment of this task. My deep appreciation is also extended to my brothers, my sister and my brothers-in-law for their prayers, continuous support and enthusiasm.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The last decade has seen an explosive growth of service-oriented software systems across various application domains, such as health care, e-commerce, finance and government. Such enterprise software systems proved very effective in managing internal resources and automating business processes of the enterprise in the open, and distributed environment of the Internet. Furthermore, a new breed of user-centric, personalized web applications that are based on aggregations of cloud services provided over the Internet has recently emerged to provide richer experiences and more intelligent services to end-users on various stationary and mobile platforms. The adoption of systems that are based on Internet-accessible software components, a class of distributed software systems to which we simply refer as *Internet software*, is expected to grow tremendously in the future.

Nevertheless, designing and developing dependable Internet software poses a unique set of challenges. Internet software systems are geographically distributed, highly concurrent systems, in which service requests and responses are handled asynchronously over an unreliable communication medium. Furthermore, communication and processing overhead may affect the time it takes the system to complete a given task, and hence its overall performance and usability. In addition, increased distribution and accessibility expose Internet software to malicious uses and attacks. Consequently, such software systems admit fairly complex interaction behaviors that are hard to predict or analyze. For these reasons, the already difficult issue of whether a deployed system meets its specification requirements is even harder to address for Internet software.

In general, applying formal techniques and tools to the software development process has gained noticeably increased interest within both the research community and industry over the past few years. The mathematical rigor of formal methods, especially at the early stages

of specification and design, facilitate developing higher quality software systems. Moreover, these methods are (at least partially) mechanizable in the form of supporting tools, which can substantially improve their scalability and accessibility and reduce their adoption costs.

Research in formal methods has culminated in a wide variety of formalisms for specifying systems and their behaviors, including automata-theoretic approaches [3, 4, 5, 6, 7, 8, 9], logics and process calculi [10, 11, 12, 13, 14, 15], and (often semi-formal) visual modeling languages [16, 17, 18, 19]. These formalisms target different classes of systems and may differ widely in their underlying abstractions, expressive power, decidability properties, and tool support. One such formalism that strikes a good balance between expressiveness and verifiability is rewriting logic [10]. Rewriting logic provides a general and executable specification and verification framework through the use of an expressive equational logic for data representations and (conditional) rewrite rules for dynamic system transitions. This framework, on which this thesis builds, can naturally model deterministic and non-deterministic systems, sequential and concurrent languages, and probabilistic and real-time processes. A feature-rich implementation of rewriting logic is available in the Maude system [20], along with an ever expanding set of associated formal analysis tools.

In contrast to system specifications, which, as in the rewriting logic case, are often executable, *properties* of systems are normally specified in typically non-executable formalisms, such as modal logics [21], including most prominently temporal logic on computation trees [22], and axiomatic assertions in some form of predicate logic, such as Hoare logic [23] and JML [24]. Depending on the formalisms used, different formal techniques for property verification may be used, including deductive reasoning with a theorem prover, reachability analysis and model-checking. A significant amount of research in the area focuses on improving the efficiency and scalability of such verification techniques, and increasing automation in supporting tools.

Industrial adoption of formal methods in software development has primarily focused on embedded software systems mainly because of their safety critical concerns. A recent survey by Woodcock et. al. [25] covered a wide range of fairly large industrial formal methods projects that have been carried out over the last twenty years, including Rockwell Collins' AAMP7 microcode design verification against security policies using ACL2 [26] and

Airbus's flight control design using the SCADE toolkit [27], based on the real-time language Esterel [28]. The survey reports that the vast majority (over 90%) of these projects had seen improvements in product quality, and over one-third had reduced project durations and costs. Although most respondents to the survey were pleased with the final products, some (about 15%) believed that the formal analysis and verification tools were lacking more features and could be improved. The survey also shows an especially growing interest in applying formal methods to real-time and distributed applications with emphasis on the specification and design phases of development.

Despite these successes in the realm of embedded software systems, applications of formal methods to Internet software in industry have not been as successful or as widely accepted, sparking an increasingly growing research interest in this problem (see, for example, the papers on automated formal specification and verification of web services and systems in WWV'09 [29], WS-FM'10 [30], FACS'2010 [31], and earlier editions and references there). The unique challenges associated with developing Internet software require extending current formal methods and developing new techniques and tools to effectively and efficiently capture their essential characteristics.

1.1 Problem Description

The thesis focuses on the formal specification and analysis of two fundamentally important aspects of Internet software systems: (1) the correctness of service compositions, and (2) the availability of services.

Service Composition. Service composition is a disciplined process of combining individual service invocations into a larger, more useful service. Composability of services is one of the most critical design principles of the service-oriented computing (SOC) paradigm and its various instantiations, such as Software-as-a-Service architectures and business process and workflow management. This is because composability promotes abstractions, modularity, reusability, and interoperability of services, which are some of the fundamental goals of SOC for developing and managing large, and distributed Internet software. Within the SOC paradigm, services are normally implemented as web services, and their compositions are

specified using a web services orchestration language, such as the Business Process Execution Language (BPEL), which is now an OASIS standard known as WS-BPEL [32], and is part of an expansive set of industry standards for web services [33].

The correctness problem of timed service compositions asks whether a given composition performs its intended task (functional correctness) while satisfying its timing requirements (non-functional correctness). In practice, designing compositions of services is a heavily error-prone activity. Given the inherent dependence of Internet-accessible software components and business processes on composing services, the importance of specifying correct service compositions is quite clear and cannot be overemphasized.

One of the most prominent research proposals to address this problem, which is centrally relevant to this thesis, is J. Misra's theory of orchestration, called Orc [34], which was further developed by Misra and Cook [35]. Orc defines an elegant theory and programming model for timed service compositions that, unlike other approaches, provides simple abstractions for services with powerful and expressive orchestration primitives. Services are represented by sites, which may produce values when invoked and whose names may be passed around as values. The Orc theory inspired later developments of process-calculus-based formalisms for various aspects of service compositions, such as the Calculus of Sessions and Pipelines (CaSPiS) [36], which is a calculus for describing service sessions and their interactions, and the the Signal Calculus (SC) [37], a variant of Ambient Calculus [38] for event-notification-based service coordination. Other calculus-based approaches also exist (e.g. [39] and [40], which are based, respectively, on λ-calculus and event calculus, both with emphasis on safety properties).

Other approaches that are based directly on BPEL have also been proposed (see the manuscript [41] for a fairly comprehensive survey). Given BPEL's unfortunate lack of formal semantics, these approaches essentially provide formalizations of (subsets of) BPEL in some formal model of computation, like BPEL encodings in Petri Nets [42] and the π-calculus [43], or devise new BPEL-inspired formal languages, such as B*lite* [44], a lightweight language for service orchestrations that captures some of BPEL's salient features, such as transactions and process termination. However, as a result of its complexity and expansive feature-set, a comprehensive and practical BPEL-based formal framework for the specification and

verification of service orchestrations remains elusive.

Service Availability. The second focal point of the thesis is the problem of formally specifying and analyzing availability of services in a communication network against denial-of-service (DoS) attacks, which are cyber attacks that exploit vulnerabilities in the underlying communication protocol to deny service to legitimate users. In recent years, DoS attacks have become an increasingly prevalent form of security threat as more complex distributed systems are being deployed. Moreover, distribution and interconnectivity have enabled an even more effective and powerful form of *distributed* DoS attacks (or DDoS), in which a large number of compromised systems are used collectively to carry out the attack, such as the fairly recent, highly publicized, DDoS *flooding* attack, in which an extremely large number of malicious Domain Name Service (DNS) requests overloaded a major DNS server rendering several large e-commerce web sites inaccessible, including Amazon, Wal-Mart, and Expedia[1]. Despite quick mitigation efforts by the DNS provider, the system experienced a downtime of about an hour, and continued to experience problems a few hours after recovery.

Despite having been a serious security threat for the last few years, most currently deployed DoS countermeasures rely on ad-hoc solutions based on heuristics and remedial procedures. In particular, current approaches tend to rely on a combination of three techniques: (1) basic prevention measures, such as patching known vulnerabilities and increasing user awareness of security issues; (2) detection procedures, including sniffing network traffic, router and firewall filtering, and intrusion detection; and (3) mitigation procedures, such as bandwidth throttling, server replication, and load balancing [45]. In practice, such techniques may have helped in protecting against some known attacks. However, for these countermeasures to be more effective and comprehensive, a deeper, more formal understanding of the communication protocols, the communication channels, and the characteristics of DoS attacks is required.

Although formal modeling and analysis of DoS vulnerabilities and defense measures of protocols have been studied before [46, 47, 48, 49, 50, 51], they remain a considerably less understood subject than formal analysis of other, more traditional security properties, such as secrecy, authentication and integrity. Part of the challenge is that availability proper-

[1]See the article at `http://www.cnn.com/2009/TECH/12/24/cnet.ddos.attack/index.html`.

ties are intimately related to performance and reliability, and therefore have an inescapable quantitative nature that does not have an obvious formal model or analysis technique. In addition, DoS attacks and countermeasures are probabilistic in nature due to the inherent behavioral uncertainties, such as underlying attacker behaviors, and the use of randomized algorithms. Moreover, as distributed systems increase in complexity, the intricacy of such attacks and DoS-resilient measures continues to grow accordingly. Such complications underscore the need for expressive probabilistic formal models as well as efficient and scalable automatic formal analysis techniques and tools.

1.2 Overview of the Approach

The thesis develops formal specification, simulation, prototyping, and formal analysis techniques and tools, based on rewriting logic and Maude, and the Orc theory, for distributed software services, with the overall goal of improving the reliability of Internet software. The key idea underlying the research approach of the the thesis is that rewriting logic is both a theoretical model, with clear and well-defined mathematical foundations, and an expressive computational formalism, providing a practical method for performing concurrent computations through its logical deduction system. The logic is also supported by an advanced and efficient implementation in the Maude tool with generic formal analysis tools. Additionally, extensions to real-time and probabilistic rewrite theories and their implementations in Maude provide a unified modeling and analysis framework for the systems of interest. By combining rewriting logic with suitable models and algorithms, we are able to provide novel, expressive, efficient and scalable formal specification and analysis methods for service orchestration and availability problems.

More specifically, for the orchestration problem, we systematically use and extend methods from the rewriting logic semantics project and apply them to Misra's theory of orchestration, Orc, to provide a clean and efficient formal model for timed orchestration design and analysis. The choice of Orc over other languages and models is motivated by the simplicity, elegance, and expressiveness of its programming model. Moreover, unlike other approaches, Orc addresses the problem of timed orchestrations using high-level abstractions and constructs that

neatly capture the essential features of an orchestration without any distracting lower-level abstractions or implementation details. Moreover, the thesis develops a natural transformation path from formal models of Orc programs to actual, provably-correct, distributed implementations that enable observing actual possible behaviors of service orchestrations in realistic environments. In spite of its inherent simplicity, the formal semantics and distributed implementation of Orc present some major challenges, including: (1) the real-time nature of the language, (2) the different priorities among its actions, (3) the efficiency of specification execution and analysis, and (4) formal analysis at the implementation level. While some of these challenges are addressed by adapting existing techniques and tools, such as modeling and analysis of real-time behaviors in rewrite theories, others are addressed by new, specialized techniques that we propose as part of this work, as discussed in Chapters 3, 4 and 5 of the thesis.

For the service availability problem, the thesis extends current methods based on rewriting logic for the specification and analysis of availability properties against DoS attacks with the goal of improving their efficiency and scalability. In particular, the thesis first presents parallel versions of the statistical model checking and quantitative analysis algorithms developed in [1] and [2], which have implementations in the VeStA tool. The parallel algorithms are implemented in a parallel, client/server extension of VeStA's implementation, namely PVeStA. This development takes advantage of an inherent parallelization opportunity within the statistical model-checking algorithm, where multiple, independent Monte-Carlo simulations are performed. Furthermore, since such simulations are usually very time-consuming and tend to significantly dominate the cost of other computations in the algorithm, performance gains and increased scalability are in practice remarkable, which is demonstrated using several experiments. Despite the focus of the thesis on services, the parallel statistical model checking algorithms remain fully generic.

In addition, the thesis proposes an expressive and modular method for the formal specification and analysis of service availability against DoS in service compositions using generic Adaptive Selective Verification (ASV) wrappers. This is achieved essentially by combining techniques developed for Orc service orchestrations and service availability analysis based on rewriting logic, Maude, and PVeStA. The method is illustrated by specifying and analyzing

an ASV-wrapped service orchestration pattern in Orc.

1.3 Summary of Contributions

In general, the thesis contributes to several ongoing research efforts, while creating new possibilities for further research, within the broad areas of formal methods, programming languages, and web services. The following list highlights the main research contributions of the thesis:

1. An efficiently executable formal specification in rewriting logic of the real-time, synchronous semantics of Orc capturing Orc's intended operational semantics.

2. A high-level, web-based tool, MOrc, for the formal specification and analysis of Orc orchestrations based on Orc's rewriting semantics and Real-Time Maude.

3. A new, elegant and effective formal approach to the specification and analysis of web service orchestrations using MOrc for improving Internet software reliability.

4. A verifiably correct, distributed implementation of Orc in Maude, based on its rewriting semantics, with physical timing, endowed with formal analysis using Real-Time Maude.

5. Parallelization and optimization of existing algorithms for formal statistical model checking and quantitative analysis, and their implementation in PVeStA.

6. Further developing the rewriting-based techniques for the probabilistic modeling and statistical analysis of probabilistic systems, and using them to formally model and verify quantitative properties of the ASV protocol.

7. An object-based, probabilistic, formal specification of the ASV protocol as a generic protocol wrapper enabling modular specification of DoS-resilient communication systems.

8. A novel approach to the formal statistical analysis of availability of services under DoS attacks of service orchestrations based on Orc and ASV wrappers.

1.4 Thesis Outline

This section gives a brief outline of the thesis, along with references to previous work on which the different parts of the thesis are based.

The thesis begins in Chapter 2 by reviewing preliminaries on rewrite theories and their real-time and probabilistic extensions. The chapter also describes at a high level how such units of specifications in rewriting logic can be executed and analyzed in Maude and its real-time extension in Real-Time Maude. This is followed in Section 2.3 by a quick overview of Orc, its syntax, semantics, and some of its algebraic properties as used in the thesis.

Chapter 3 discusses in detail the first main contributions of the thesis by introducing the SOS-based rewriting semantics $\mathcal{R}_{Orc}^{sos}$ and the semantically equivalent reduction rewriting semantics $\mathcal{R}_{Orc}^{red}$ of Orc. The chapter gives proofs of desirable executability properties about them and validates with experiments the efficiency advantage of $\mathcal{R}_{Orc}^{red}$ over $\mathcal{R}_{Orc}^{sos}$. The work presented in this chapter extends previous work that appeared in [52, 53] and in the extended report [54].

Chapter 4 introduces a natural semantic extension to the reduction rewriting semantics of Orc to an object-based semantics, $\mathcal{R}_{Orc}$, on which subsequent developments are based. An initial version of this extension originally appeared in [54, 53]. This is followed by a description of MOrc, a web-based tool, based on $\mathcal{R}_{Orc}$, enabling various formal analysis methods, including LTL model checking, on Orc expressions at the level of Orc, using Real-Time Maude as a back-end. The tool is illustrated with several examples.

In Chapter 5, we describe a general transformation method, based on Maude's support for communication with external objects through sockets, in which a real-time formal specification of a language such as Orc can be turned with minimal effort into a distributed implementation with physical timing. The transformation is described in detail for Orc and illustrated with a case study. This is followed by showing how such a distributed implementation, can be formally modeled and analyzed in Real-Time Maude. This chapter is based on previously published work in [55] and the technical report [56].

In Chapter 6, we describe in detail parallel algorithms for statistical model checking of probabilistic formulas in PCTL/CSL and statistical quantitative analysis of QuaTEx ex-

pressions, and their implementations in PVeStA. Substantial speedup gains as a result of parallelization are validated through several statistical analysis examples, including statistical analysis of availability properties of the ASV protocol modeled as a probabilistic rewrite theory. The parallel statistical model checking work presented in this chapter extends previous results that appeared in [57], and the formal analysis of the ASV protocol is based on previous work in [58].

Chapter 7, introduces a *modular* approach to endow formal specifications of systems and their distributed deployments with ASV-based DoS protection measures using generic ASV protocol wrappers, specified as a probabilistic rewrite theory. This chapter, which is based on previous work in [58], describes the specification of ASV wrappers in detail and discusses the assumptions under which statistical model-checking analysis can be applied. This is followed by an application of ASV Wrappers to Orc to analyze availability of services in an Orc service orchestration pattern.

Finally, a discussion of related work is given in Chapter 8, followed by a discussion of further research directions in Chapter 9.

CHAPTER 2

BACKGROUND

In this chapter, we review, at a high level, some preliminaries on rewriting logic, the Maude system, and the theory of Orc. More details can be found in the cited references below.

2.1 Rewriting Logic

Rewriting logic [59] is a general formalism that unifies in a natural way different concurrency models [10, 60]. It is also an expressive semantic framework that is well suited to give formal semantic definitions of sequential and concurrent systems and languages (see [61, 62, 63] and references there). Furthermore, with recent real-time and probabilistic extensions, the logic is capable of succinctly modeling real-time, stochastic, and hybrid systems [64, 65, 66]. Moreover, with the availability of high-performance rewriting logic implementations, such as Maude [20], specifications can both be executed and formally analyzed.

2.1.1 Rewrite Theories

The unit of specification in the logic is a *rewrite theory*, which gives a formal description of a concurrent system including its static state structure and its dynamic behavior. Assuming that t, u, v and w (and their decorated variants) are terms and s is a sort, a rewrite theory, in its most general form, is a tuple $\mathcal{R} = (\Sigma, E \cup A, R, \phi)$, consisting of: (i) a theory $(\Sigma, E \cup A)$ in membership equational logic (MEL) [67], where Σ is a MEL signature having a set of kinds, a family of sets of operators, and a family of disjoint sets of sorts, E is a set of Σ-sentences, which are universally quantified Horn clauses with atoms that are either equations $(t = t')$ or memberships $(t : s)$, and A is a set of equational axioms, such as commutativity, associativity

and/or identity axioms for some operators in Σ; (ii) a set R of universally quantified labeled *conditional rewrite rules* of the form:

$$(\forall X)\ r : t \to t' \ \textbf{if} \ \bigwedge_i u_i = u_i' \wedge \bigwedge_j v_j : s_j \wedge \bigwedge_l w_l \to w_l' \tag{2.1}$$

where r is a label; and (iii) a function $\phi : \Sigma \to \mathcal{P}(\mathbf{N})$ that assigns to each operator symbol f in Σ of arity $n > 0$ a set of positive integers $\phi(f) \subseteq \{1, \ldots, n\}$ representing *frozen* argument positions where rewrites are forbidden.

While the MEL theory $(\Sigma, E \cup A)$ specifies the user-defined syntax and equational axioms, which define the system states as elements of the initial algebra associated to $(\Sigma, E \cup A)$, a rule $r : t \to t'$ **if** C in R gives a general pattern for a possible concurrent change or transition in its state (modulo the restrictions imposed by ϕ), with the intuition that an instance $\theta(t)$ of t (with θ a substitution) may rewrite to $\theta(t')$ in the state of the system whenever the condition $\theta(C)$ is satisfied. Such rewrites are deduced according to the inference rules of rewriting logic, which are described in detail in [68]. Using these inference rules, a rewrite theory $\mathcal{R}$ proves a statement of the form $(\forall X)\ t \to t'$, meaning that, in $\mathcal{R}$, any instance of the state term t can reach the corresponsing instance of the state term t' in a finite number of steps. A detailed discussion of rewriting logic as a unified model of concurrency and its inference system can be found in [10]. [68] gives a precise account of the most general form of rewrite theories and their models.

2.1.2 Probabilistic Rewrite Theories

Probabilistic rewrite theories [65] extend regular rewrite theories with probabilistic rules, which can specify probabilistic behaviors of systems. Assuming $\overrightarrow{x}$ and $\overrightarrow{y}$ are disjoint sets of variables, a *probabilistic rewrite rule* has the following form:

$$(\forall \overrightarrow{x}, \overrightarrow{y})\ r : t(\overrightarrow{x}) \longrightarrow t'(\overrightarrow{x}, \overrightarrow{y}) \ \textbf{if}\ C(\overrightarrow{x}) \ \textbf{with probability}\ \overrightarrow{y} := \pi(\overrightarrow{x})$$

A probabilistic rule introduces on its right-hand side term new variables $\overrightarrow{y}$, the values of which depend on a probability distribution function π parametrized by $\theta(\overrightarrow{x})$, where θ is a matching substitution satisfying the condition C. An example of a probabilistic rewrite rule, borrowed from [20], is the following rule:

$$clock(t,c) \longrightarrow \text{if } B \text{ then } clock(t+1, c-(c/1000.0)) \text{ else } broken(t,c)$$
$$\textbf{with probability } B := Bernoulli(c/1000.0)$$

The rule specifies a battery-operated clock transition that is based on the outcome B of a biased coin toss, where the bias is proportional to the current battery charge c. In general, probabilistic rewrite theories can model different probabilistic systems with discrete or continuous probability distribution functions. Furthermore, they can express models involving both probabilistic and non-deterministic features. The reader is referred to [2] for a more rigorous definition of probabilistic rewrite theories.

2.1.3 Real-Time Rewrite Theories

A real-time rewrite theory [64] extends a regular rewrite theory with support for modeling real-time behaviors of systems. In particular, in a real-time rewrite theory $\mathcal{R}^\tau = (\Sigma^\tau, E^\tau \cup A^\tau, R^\tau, \phi)$: (i) the equational theory $(\Sigma^\tau, E^\tau \cup A^\tau)$ contains a sort for *Time* representing the time domain, which can be either dense or discrete, and declares a system-wide operator that encapsulates the whole system being modeled into a special sort *GlobalSystem* for managing time elapse, and (ii) the set of rewrite rules R^τ is the disjoint union of two sets R_I and R_T, where R_I consists of *instantaneous* rewrite rules having the form (2.1) above and representing instantaneous transitions in the system, and R_T consists of *tick* rewrite rules modeling system transitions that take a non-zero amount of time to complete. A tick rewrite rule has the following form

$$r : \{t\} \xrightarrow{\tau} \{t'\} \text{ if } C$$

where τ is a term of sort *Time* representing the duration of time required to complete the transition specified by the rule. The global operator $\{\_\}$ encapsulates the whole system into the sort *GlobalSystem* to ensure the correct propagation of the effects of time elapse to every part of the system. A detailed discussion of real-time rewrite theories and their semantics, including a detailed explanation of how they can be reduced to ordinary rewrite theories by explicitly introducing a global clock as part of the global state, can be found in [64, 69].

2.2 The Maude System

Maude [20] is a high-performance implementation of rewriting logic and its underlying MEL sublogic. A basic unit of specification in Maude can be either a *functional module*, corresponding to a MEL theory $\mathcal{E} = (\Sigma, E \cup A)$, or a *system module*, defining a rewrite theory $\mathcal{R} = (\Sigma, E \cup A, R, \phi)$. A functional module may contain module inclusion assertions, sort and subsort declarations, operator symbols declarations (optionally with some equational attributes, including equational axioms A such as associativity, commutativity and/or identity), and conditional equations and membership axioms. *Admissible* functional modules, which are modules that satisfy some reasonable executability requirements, including ground confluence and termination (modulo the axioms A) and sort-decreasingness of the equations, can be executed in Maude by *equational simplification modulo axioms* using the equations E as simplification rules from left to right and Maude's matching algorithms modulo A to simplify a term to its canonical form with a least sort. Equational simplification modulo axioms of an admissible functional module yields an operational semantics, defined by the algebra of canonical forms $Can_{\Sigma/E \cup A}$, for its corresponding theory that coincides with its mathematical, initial algebra semantics, given by the initial algebra $T_{\Sigma/E \cup A}$ (see Sections 4.6–4.8 in [20] and cited references there). Simplification modulo axioms A can be performed by the `reduce` command in Maude.

An admissible system module, which may additionally contain possibly conditional rewrite rules, must satisfy the executability requirements for its equational part in addition to the ground coherence of the rules R with respect to the equations in E and to admissibility conditions on the rules, which ensures that all variables in the rules can be instantiated by

(incremental) matching. Such admissible modules can be executed in Maude by rewriting with rules (abiding by the restrictions imposed by ϕ) and oriented equations modulo the axioms A, which in this case corresponds exactly to the mathematical semantics of $\mathcal{R}$, which rewrites with R modulo the equational theory $E \cup A$ (see Section 6.3 in [20] and [68, 70]). Rewriting of system modules can be performed in Maude by means of the `rewrite` command, which applies a rule-fair strategy to explore a possible behavior of the system, the `frewrite` command, which applies a position-fair and rule-fair strategy, or the `search` command, which explores the entire reachable state space of the system, to find states instantiating a given pattern and satisfying a given semantic condition, following a breadth-first strategy. Furthermore, Maude provides a linear temporal logic (LTL) model checker for verifying safety and liveness properties, under the assumption that the set of states reachable from a given initial state is finite.

2.2.1 Simulating and Analyzing Probabilistic Rewrite Theories

In general, probabilistic rewrite theories are not directly executable, since probabilistic rewrite rules are nondeterministic, as noted in Section 2.1.2 above. However, they can be simulated in Maude by sampling values for the new variables appearing in the right-hand sides of the probabilistic rewrite rules from appropriate probability distributions. Sampling is performed using Maude's built-in random number generator function `random(s)`, with `s` a seed, and a counter function `counter` that rewrites with an internal strategy to the next natural number [2].

Monte Carlo simulations of probabilistic rewrite theories obtained in this way can be analyzed statistically using statistical model checking and quantitative analysis algorithms, provided any unquantified non-determinism is avoided [2, 1], which is easily achievable for object-oriented probabilistic rewriting specifications. Statistical model checking verifies, within a desired statistical confidence level, formulas in a probabilistic temporal logic, such as the *Continuous Stochastic Logic* (CSL) [71]. Statistical quantitative analysis, on the other hand, uses an expressive temporal language, called *Quantitative Temporal Expressions* (QuaTEx) [2], that supports parametrized, real-valued recursive function declarations, an

if-then-else construct, and a *next* operator $\bigcirc$, to quantitatively specify richer properties about probabilistic models. Both techniques rely on discrete-event simulations of the model, obtained by Monte Carlo simulations of the form described above, and are implemented in a tool called VeStA 2.0 (or simply VeStA in the rest of the thesis), with support for probabilistic rewrite theories [72]. We will elaborate on these statistical analysis algorithms in Chapter 6. Further details can be found in [2, 1, 73].

2.2.2 Real-Time Maude

While real-time rewrite theories with deterministic tick rules can be specified in Maude and analyzed using its standard analysis tools, a more expressive and flexible implementation and analysis of such theories, for both discrete and continuous time domains, is provided by Real-Time Maude (RTM) [74], which is an extension to Maude written using its reflective features. RTM modules provide the data types, operators, and execution strategies that enable the specification of timed modules with built-in or user-defined time domains. For example, the RTM module `POSRAT-TIME-DOMAIN-WITH-INF` specifies a dense time domain represented by the non-negative rationals with the infinity element. Time tick rewrite rules are in general non-deterministic, since the amount of time τ by which a system may advance its clock may be non-deterministic. Therefore, tick rules are in general not directly executable, and, for this reason, RTM defines a number of *time sampling strategies*, such as the general *maximal* sampling strategy (which advances time until the next instant when some instantaneous rewrite rule becomes enabled), which can be used to execute timed modules. Furthermore, RTM comes equipped with a range of formal analysis tools for timed modules, including timed rewriting (the commands `trewrite` and `tfrewrite`), timed and untime search (`tsearch` and `utsearch`), and time-bounded and time-unbounded `LTL` model checking (the command `mc`). For timed modules with non-deterministic time tick rule(s), the analysis is carried out with respect to a chosen time sampling strategy. Ölveczky and Meseguer [75] characterized a very broad class of timed rewrite systems for which this model checking analysis is sound and complete, when using the general maximal time sampling strategy. A complete description of RTM and its formal analysis features can be found

in [74].

2.2.3 Sockets and External Objects in Maude

Maude provides a low-level implementation of sockets, which effectively enables a Maude process to exchange messages with other processes, including other Maude instances, according to the connection-oriented TCP communication protocol. More specifically, a Maude configuration `CF`, which is a Maude term of the sort `Configuration` corresponding to a multiset of objects and messages, such that `CF` contains a *socket portal*, which is a predefined constant `<>` of sort `Portal` (a subsort of `Configuration`), may communicate with objects external to the Maude process executing `CF` through a set of special messages defining an interface to Maude sockets. Assuming that `O` is an identifier for an object in `CF`, these messages are as follows:

1. `createClientTcpSocket(socketManager, O, ADDRESS, PORT)`, which asks Maude's socket manager, a factory for socket objects, for a client socket to a server located at `ADDRESS:PORT`. Maude then responds with either a `createdSocket(O, socketManager, SOCKET)` message, indicating successful creation of the client socket `SOCKET`, or a `socketError(O, socketManager, S)` message, with `S` a string briefly describing the reason for failure.

2. `send(SOCKET, O, S)`, which asks for the string `S` to be sent through `SOCKET`. This message elicits either a message `sent(O, SOCKET)`, when the string is successfully sent, or a message `closedSocket (O, SOCKET, S)`, if an error occurred.

3. `receive(SOCKET, O)`, which solicits a response through `SOCKET`. When a response is received, Maude issues the message `received(O, SOCKET, S)`, with `S` the string received. In case of an error, the socket is closed with the message `closedSocket(O, SOCKET, S)`.

4. `createServerTcpSocket(socketManager, O, PORT, BACKLOG)`, which asks Maude's socket manager to create a server socket at port `PORT`, with `BACKLOG` a positive integer specifying the maximum allowed number of queue requests. The message elicits

either a message`createdSocket(O, socketManager, SERVER-SOCKET)`, or a message `socketError(O, socketManager, S)`.

5. `acceptClient(SERVER-SOCKET, O)`, which causes Maude to listen for incoming connections at `SERVER-SOCKET`. If a client connection is accepted, Maude responds back with the message `acceptedClient (O, SERVER-SOCKET, ADDRESS, SOCKET)`, where `ADDRESS` is the client's address and `SOCKET` is a newly created socket for communicating with the client. The message `socketError(O, socketManager, S)` is issued in case of failure.

6. `closeSocket(SOCKET, O)`, which causes Maude to close the socket and issue the message `closedSocket(O, SOCKET, S)`.

Rewriting a term with external objects in Maude is performed with the `erewrite` command (also abbreviated as `erew`). A more detailed discussion of sockets and support for external objects in Maude can be found in [20].

2.3 The Orc Theory

Orc [34, 35] is a timed theory for orchestration of services. It provides an expressive and elegant programming model for describing timed, concurrent computations. A *site* in Orc represents a service, which may range in complexity from a simple function to a complex web search engine, depending on the orchestration problem. A site may also represent the interaction with a human being, most commonly within the context of business workflows [76]. A site, when called, may produce, or *publish*, at most one value. A site may not respond to a call, either by design or as a result of a communication problem. For example, if *CNN* is a site that returns the news page for a given date d, then $CNN(d)$ might not respond because of a network failure or it may choose to remain silent because of an invalid input value d. Site calls are *strict*, i.e., they assume call-by-value semantics.

Being a timed theory, different site calls in Orc may occur at different times. A site call may be purposefully delayed using the *internal* site $Rtimer(t)$, which publishes a signal after t

$$E \in \mathit{ExpressionName} \qquad x \in \mathit{Variable} \qquad w \in \mathit{Value} \cup \{\mathbf{stop}\}$$

$$\begin{array}{rcl}
\text{Orc program} & ::= & \vec{d}\,;\ f \\
d \in \text{Declaration} & ::= & E(\vec{x}) \triangleq f \\
f, g \in \text{Expression} & ::= & \mathbf{0} \mid p(\vec{p}) \mid E(\vec{p}) \\
& \mid & f \mid g \mid f >x> g \mid g <x< f \mid f\,;\,g \\
p \in \text{Parameter} & ::= & x \mid w
\end{array}$$

Figure 2.1: Syntax of Orc

time units. Furthermore, responses from calls to *external* sites may experience unpredictable delays and communication failures, which could affect whether and when other site calls are made. Unlike external sites, however, responses from internal sites, such as *Rtimer*, are assumed to have completely predictable timed behaviors; for example, $\mathit{Rtimer}(t)$ will publish a signal in exactly t time units. Orc also assumes a few more internal sites, which are needed for effective programming in Orc. They are: (1) the $\mathit{if}(b)$ site, which publishes a signal (an Orc value *signal*) if b is true and remains silent otherwise, (2) $\mathit{let}(\vec{x})$, which publishes a tuple of the list of values in $\vec{x}$, or the value of $\vec{x}$ itself if $|\vec{x}| = 1$, and (3) *Clock*, which publishes the current time value.

2.3.1 Syntax of Orc

An Orc *expression* describes how site calls (and responses) are combined in order to perform a useful computation. Orc expressions are built up from site calls using four combinators, which were previously shown in [35] to be capable of expressing a wide variety of timed, distributed computations succinctly and elegantly[1]. The abstract syntax of Orc is shown in Figure 2.1.

We assume a syntactic category *Value* that contains not only standard Orc values, such as numeric and boolean values and the *signal* value, but also site names as a distinguished subcategory *SiteName* of values that can be called (i.e., $\mathit{SiteName} \subset \mathit{Value}$). We also assume a special site response value **stop**, which may be used to indicate termination of a site call without necessarily publishing a standard Orc value.

[1]The *otherwise* combinator was fairly recently added to Orc. Its syntax and a brief discussion on its semantics can be found in [77].

An Orc *program* consists of an optional list of declarations, giving names to expressions, followed by an Orc expression to be executed. An expression can be either: (1) the silent expression (**0**), which represents a site that never responds; (2) a parameter or an expression call having an optional list of actual parameters as arguments; or (3) the composition of two expressions by one of the following four composition operators:

Symmetric parallel composition, $f \mid g$, which models concurrent execution of independent threads of computation. For example, $CNN(d) \mid BBC(d)$, where *CNN* and *BBC* are sites, calls both sites concurrently and may publish up to two values depending on the publication behavior of the individual sites.

Sequential composition, $f >x> g$, which executes f, and for every value $v \in Value$ published by f, creates a fresh instance of g, with x bound to v, and runs that instance in parallel with the current evaluation of $f >x> g$. For example, if $Email(x)$ is a site that sends an e-mail message given by x to a fixed address a, then the expression $CNN(d) >x> Email(x)$ may cause a news page to be sent to a. If $CNN(d)$ does not publish a value, $Email(x)$ is never invoked. Similarly, the expression $(CNN(d) \mid BBC(d)) >x> Email(x)$ may result in sending zero, one, or two messages to a.

Asymmetric parallel composition, $f <x< g$, which executes f and g concurrently but terminates g once it has published its first value $v \in Value$, which is then bound to x in f. For instance, the expression $Email(x) <x< (CNN(d) \mid BBC(d))$ sends at most one message, depending on which site publishes a value first. If neither site publishes a value, the variable x is not bound to a concrete value and, therefore, the call to *Email* is never made.

Otherwise composition, $f \,;\, g$, which attempts to execute f to completion. If f terminates without ever publishing a value $v \in Value$, g is then executed. Otherwise, if f publishes a value v during its execution, g is ignored. For example, suppose *CNN* publishes a **stop** value when called with invalid date values. Then, if d is a valid date value, the composition $CNN(d) \,;\, Email(\mathsf{err\_msg})$ never invokes *Email* and may publish the news page from *CNN*. Otherwise, if d is invalid, an e-mail is sent and the value published by *Email* is the value published by the composition.

As can be noted from the informal description above, the execution of an Orc expression may in general involve several concurrently running threads, and may result in publishing a

(time-ordered) stream of values.

A variable x occurs *bound* in an expression g when g is the right (resp. left) subexpression of a sequential composition $f >x> g$ (resp. an asymmetric parallel composition $g <x< f$). If a variable is not bound in either of the two above ways, it is said to be *free*. We use the syntactic sugar $f \gg g$ (resp. $g \ll f$) for sequential composition (resp. asymmetric parallel composition) when no value passing from f to g takes place, which corresponds to x being *not a free* variable in g. To minimize use of parentheses, we assume the following precedence order: $\gg \succ \mid \succ \ll \succ ;$, with the otherwise combinator having the least precedence.

2.3.2 Some Small Examples

We now list a few example Orc expressions, borrowed from [35]. Many more examples and larger programs can be found in [35, 78, 79, 80].

Consider the Orc expression below, which specifies a timeout t on the call to a site M:

$$let(x) <x< (\, M() \mid Rtimer(t) \gg let(signal)\,)$$

Upon executing the expression, both sites M and *Rtimer* are called. If M publishes a value v before t time units, then v is the value published by the expression. But if M publishes v in exactly t time unites, then either v or *signal* is published. Otherwise, *signal* is published.

Another example is the standard programming idiom of the two-branch conditional **if** b **then** f **else** g, which can be written in Orc as the expression $if(b) \gg f \mid if(\neg b) \gg g$. Given the behavior of the internal site *if*, only one of the expressions f and g is executed, depending on the truth value of b.

A third example is the following Orc expression declaration, which defines an expression that recursively publishes a signal every t time units, indefinitely.

$$Metronome(t) \triangleq let(signal) \;\mid\; Rtimer(t) \gg Metronome(t)$$

The expression named *Metronome* can be used to repeatedly initiate an instance of a task every t time units. For example, the expression $Metronome(100) \gg\ UpdateLocation()$ calls

on the task of updating the current location of a mobile user every hundred time units.

2.3.3 Operational Semantics of Orc

A structural operational semantics for the instantaneous (untimed) behaviors of Orc was originally given by Misra and Cook [35]. Figure 2.2 lists an updated set of small-step SOS rules, based on the original SOS specification, that includes rules for the semantics of the otherwise combinator and **stop** site responses. The semantics uses two forms of internal expressions to represent intermediate transitional steps in the execution of an Orc expression, namely "$!v$", which publishes the value $v \in \mathit{Value}$, and "$?h$", with h a *handle* name, which is used to uniquely identify an unfinished site call.

The SOS semantics specifies the possible behaviors of an Orc expression as a labeled transition system with four label schemes corresponding to four types of actions an Orc expression may take: (1) publishing a value, $!v$, (2) calling a site, $M\langle \vec{v}, h \rangle$, with h a fresh handle name uniquely identifying this site call instance, (3) making an unobservable transition, τ, which may represent an expression call or a substitution event, and (4) consuming a site response, $h?w$, with h the handle for the corresponding site call and $w \in \mathit{Value} \cup \{\mathbf{stop}\}$. In Figure 2.2, n ranges over labels for non-publishing events, namely labels of types (2)–(4), while l ranges over all labels.

Two important refinements to the SOS specifications that are of central relevance to this work were proposed. First, as discussed by Misra and Cook in [35], the SOS semantics is highly non-deterministic, allowing *internal* transitions within an Orc expression (value publishing, site calls, and τ transitions) and the *external* interaction with sites in the environment (through site return events) to be interleaved in any order. This high degree of non-determinism may be undesirable. For example, in the expression

$$\mathit{let}(x) <x< \mathit{Rtimer}(1) \gg N() \mid M()$$

which is supposed to give M priority over N, the call to M may actually be delayed in this semantics, thus defeating the purpose of prioritizing it over the call to N. In order to rule

$$\frac{h \text{ fresh}}{M(\vec{v}) \overset{M\langle\vec{v},h\rangle}{\hookrightarrow} ?h} \text{ (SiteCall)} \qquad \frac{f \overset{l}{\hookrightarrow} f'}{f \mid g \overset{l}{\hookrightarrow} f' \mid g} \text{ (Sym)}$$

$$?h \overset{h?v}{\hookrightarrow} !v \text{ (SiteRetV)} \qquad \frac{f \overset{!v}{\hookrightarrow} f'}{f >x> g \overset{\tau}{\hookrightarrow} (f' >x> g) \mid [v/x]g} \text{ (Seq1V)}$$

$$?h \overset{h?\mathbf{stop}}{\hookrightarrow} \mathbf{0} \text{ (SiteRetN)} \qquad \frac{f \overset{n}{\hookrightarrow} f'}{f >x> g \overset{n}{\hookrightarrow} f' >x> g} \text{ (Seq1N)}$$

$$!v \overset{!v}{\hookrightarrow} \mathbf{0} \text{ (Publish)} \qquad \frac{f \overset{!v}{\hookrightarrow} f'}{g <x< f \overset{\tau}{\hookrightarrow} [v/x]g} \text{ (Asym1V)}$$

$$\frac{E(\vec{x}) \triangleq f \in D}{E(\vec{p}) \overset{\tau}{\hookrightarrow} [\vec{p}/\vec{x}]f} \text{ (Def)} \qquad \frac{f \overset{n}{\hookrightarrow} f'}{g <x< f \overset{n}{\hookrightarrow} g <x< f'} \text{ (Asym1N)}$$

$$\frac{f \overset{!v}{\hookrightarrow} f'}{f \,;\, g \overset{!v}{\hookrightarrow} f'} \text{ (OtherV)} \qquad \frac{g \overset{l}{\hookrightarrow} g'}{g <x< f \overset{l}{\hookrightarrow} g' <x< f} \text{ (Asym2)}$$

$$\frac{f \overset{n}{\hookrightarrow} f'}{f \,;\, g \overset{n}{\hookrightarrow} f' \,;\, g} \text{ (OtherN)}$$

Figure 2.2: Instantaneous, asynchronous structural operational semantics of Orc

out such undesirable behaviors, a *synchronous semantics* was proposed in [35] by placing further constraints on the application of SOS semantic rules. The synchronous semantics was arrived at by distinguishing between *internal* and *external* events, and splitting the SOS transition relation $\hookrightarrow$ into two sub-relations $\hookrightarrow_R$, and $\hookrightarrow_A$, and characterizing set-theoretically, the complementary subsets of expressions (quiescent vs. non-quiescent) to which they are respectively applied. In previous work [52], we have presented two different approaches, namely strategy expressions and equational conditions, in which this splitting into $\hookrightarrow_R$ and $\hookrightarrow_A$ can be faithfully captured in a rewriting logic semantics of Orc by enforcing an execution strategy that gives transitions corresponding to internal actions precedence over the external site return action. In Section 3.1.2 of Chapter 3, we describe a third, typed approach, based on sorts and subsorts, that is both more elegant and, in practice, more efficiently executable than the two previous approaches just mentioned.

A second refinement of the Orc SOS, by Wehrman et. al [81], endowed the original SOS specification with timing semantics in a way similar to timed process algebras [14]. This was achieved mainly by refining the SOS transition relation into a relation on time-shifted Orc expressions and timed labels of the form (l, t), where t is the amount of time taken by a

$$(f \mid g) \mid h = f \mid (g \mid h) \quad (2.2)$$
$$f \mid g = g \mid f \quad (2.3)$$
$$f \mid \mathbf{0} = f \quad (2.4)$$
$$(f \,;\, g) \,;\, h = f \,;\, (g \,;\, h) \quad (2.5)$$
$$f \,;\, \mathbf{0} = \mathbf{0} \,;\, f = f \quad (2.6)$$
$$\mathbf{0} >x> f = \mathbf{0} \quad (2.7)$$
$$f <x< \mathbf{0} = [\mathbf{stop}/x]f \quad (2.8)$$
$$!v \,;\, f = !v \quad (2.9)$$
$$M(\vec{p}) = \mathbf{0} \text{ if } \mathbf{stop} \in \vec{p} \quad (2.10)$$
$$w(\vec{p}) = \mathbf{0} \text{ if } w \notin \mathit{SiteName} \quad (2.11)$$
$$!\,\mathbf{stop} = \mathbf{0} \quad (2.12)$$

Figure 2.3: Some algebraic properties of Orc expressions

transition. In this extended relation, a transition step of the form $f \xrightarrow{(l,t)} f'$ states that f may take an action l to evolve to f' in time t, and, if $t \neq 0$, no other transition could have taken place during the t time period. To properly reflect the effects of time elapse, parts of the expression f may also have to be time-shifted by t. However, for simplicity of presentation, the semantics described in [81], abstracted away the non-publishing events as unobservable transitions and considered only the asynchronous semantics of Orc. Sections 3.2 and 3.3 present a rewriting logic approach to capturing timed behaviors of Orc expressions, which also takes into account the *synchronous* semantics of Orc as described above.

2.3.4 Some Algebraic Properties

Orc was shown to posses several desirable structural properties, either using bisimulations based on the original and timed SOS semantics [82, 83], or, alternatively, using graph isomorphisms in a tree-based denotational semantics [84]. We focus our attention here on the subset of these algebraic properties shown in Figure 2.3. Our choice of this subset is motivated by the fact that the equations (2.7)–(2.12) are confluent and terminating modulo the axioms (2.2)–(2.6), so that equality under (2.2)–(2.12) becomes *decidable* by rewriting. Furthermore, since a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ has both rules R and equations E, so that states are equivalence classes modulo E, we can obtain a more abstract and more efficient rewriting logic semantics of Orc by adding equations (2.2)–(2.12) to the set E of equations in the rewrite theory $\mathcal{R}$ axiomatizing Orc.

Associativity, commutativity and identity axioms for symmetric parallel composition were

proved in [82, 83, 84]. Associativity and right identity axioms of the otherwise combinator can also be proved by strong bisimulation (see Appendix A), and its left identity is assumed as a structural equivalence rule that is required to achieve its intended semantics. Proofs of the identities (2.7) and (2.10)–(2.12) are trivial, since both sides of these identities have no behavioral transitions, and are, thus, strongly bisimilar. The remaining two laws, namely (2.8) and (2.9), are also easy to show, and their proofs are given in Appendix A.

Other algebraic properties of Orc expressions, which were shown in [82, 83, 84], are not suitable for algebraic simplification purposes because, when viewed as equations, they either fail to satisfy executability requirements, such as confluence and/or coherence with the Orc semantic rules, or they do not necessarily compute simpler normal forms. In particular, algebraic laws shown using weak bisimulations that ignore τ transitions, such as the law $f >x> let(x) = f$ [83], may break coherence of the semantic rules when used as equational properties, since they may cause an Orc expression to *miss* some behavioral transitions. Other identities may result in equations that are not confluent, such as the restricted left associativity law of sequential composition [82, 83], where $FV(h)$ computes the set of free variables in h:

$$f >x> (g >y> h) = (f >x> g) >y> h \text{ if } x \notin FV(h)$$

(consider for example the term $f_1 >x> (f_2 >y> (f_3 >z> f_4))$, with $x \notin FV(f_3) \cup FV(f_4)$ and $y \notin FV(f_4)$). Finally, some identities, when used as oriented equations, may compute normal forms that are not necessarily structurally simpler than the original expressions, such as, for example, the law of distributivity of parallel composition over sequential composition [82, 83]: $(f \mid g) >x> h = f >x> g \mid g >x> h$. Such equations add extraneous "simplification" steps that may adversely affect execution performance without actually arriving at simpler normal forms. We prove in Sections 3.2.4 and 3.3.4 that the identities listed in Figure 2.3 satisfy all the desirable executability requirements when used as equations in both the SOS-based and the reduction rewriting semantics specifications of Orc.

CHAPTER 3

REWRITING SEMANTICS OF ORC

Orc is a language for orchestration of services that offers simple, yet powerful and elegant, constructs to program sophisticated orchestration applications. Despite its simplicity, developing an efficiently executable formal semantics of Orc poses interesting challenges, including, most importantly, its real-time nature and the urgency of internal actions over external actions. The rewriting logic semantics of Orc described in this chapter captures the real-time, synchronous semantics of Orc, and is based on the operational semantics of Orc by Misra and Cook [35] and Orc's algebraic properties, as described in Section 2.3 of Chapter 2. We begin by describing the specification of the two variants of the rewriting semantics of Orc: the SOS-based rewriting semantics $\mathcal{R}^{sos}_{Orc}$ and the reduction rewriting semantics $\mathcal{R}^{red}_{Orc}$.

3.1 The Semantic Infrastructure

The different styles of the rewriting logic semantics of Orc share a common infrastructure, which can be specified as a sub-theory $\mathcal{R}_\Omega \subset \mathcal{R}^{sos}_{Orc}$ and $\mathcal{R}_\Omega \subset \mathcal{R}^{red}_{Orc}$ describing the structures for the semantic entities and the common behaviors that are needed for a complete specification of Orc's Semantics. We, therefore, begin by describing the most important components of $\mathcal{R}_\Omega$, on which all later developments are based.

3.1.1 Parameters and Substitution

We assume a sort Var for Orc variables. To account for substitution of variables with other parameters, we use the CINNI calculus of explicit substitution [85]. This is consistent with our choice of a first-order representation of Orc in rewriting logic and does not im-

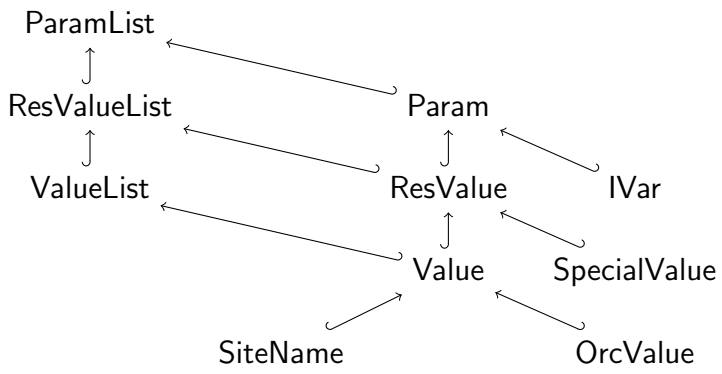


Figure 3.1: Parameter subsort structure

pair readability, since the CINNI notation is just a slight refinement of the usual textbook notation for higher-order syntax with explicit names. Therefore, in instantiating CINNI to Orc, we introduce *indexed variables*, captured by the sort IVar and declared with syntax: $\_\{\_\} : \mathsf{Var}\ \mathsf{Nat} \to \mathsf{IVar}$, with Nat the sort of natural numbers. An indexed variable represents either a free or a bound occurrence of a variable (as opposed to non-indexed names of variables in *binding* operators, which are captured by the sort Var). The index of an indexed variable name indicates the level at which the variable may be bound. For example, in the specialization of CINNI to Orc, the standard representation of the Orc expression $f(x) >x> g(x)$ in CINNI is $f(x_0) >x> g(x_0)$, where the index 0 means that the variable represented by x_0 may be bound by the closest binding instance of x, if available, i.e., since x in $f >x> g$ binds the occurrences of x in g, x is free in f but bound in g. This is in contrast to, for example, the representation $f(x_0) >x> g(x_1)$, in which both instances of x occur free in the expression.

In addition, we assume a sort Param for Orc parameters, which, according to Orc's syntax in Figure 2.1, are either (indexed) variables of the sort IVar or site response values (including the special value **stop**) of the sort ResValue. Furthermore, response values other than **stop** are identified as either standard data types, such as integers and booleans, of the sort OrcValue, or as site names of the sort SiteName, which are values representing sites that can also be called. This classification of parameters is crucial to the semantics and is neatly captured by the subsorted structure illustrated in Figure 3.1. In this figure, a separate sort SpecialValue is used to represent the **stop** value, and three list super-sorts are declared.

	$[p/x]$	$\uparrow_{\boldsymbol{x}}$	$\Uparrow_{\boldsymbol{x}}\alpha$
$x\{0\}$	p	$x\{1\}$	$x\{0\}$
$x\{1\}$	$x\{0\}$	$x\{2\}$	$\uparrow_{\boldsymbol{x}}(\alpha\ (x\{0\}))$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$x\{n\}$	$x\{n-1\}$	$x\{n+1\}$	$\uparrow_{\boldsymbol{x}}(\alpha\ (x\{n-1\}))$
$y\{n\}$	$y\{n\}$	$y\{n\}$	$\uparrow_{\boldsymbol{x}}(\alpha\ (y\{n\}))$

Table 3.1: The effects of applying CINNI's explicit substitution operators to indexed variables

The sort Subst is the sort of substitutions, which, according to the CINNI calculus of explicit substitutions [85], may have one of three forms: (1) the simple substitution $[p/x]$, which accounts for substituting a parameter for a *free* variable (assuming no free variable capture), (2) the shift-up substitution $\uparrow_{\boldsymbol{x}}$, having the effect of substituting fresh variable names for free variables, and (3) the lift substitution $\Uparrow_{\boldsymbol{x}}\alpha$, which represents a more general substitution that avoids capturing free instances of x while applying the substitution α. Table 3.1 illustrates the effects of these substitutions on indexed variables, where we assume that $x \neq y$ and α is a meta-variable ranging over terms of sort Subst. A more detailed discussion of the CINNI calculus can be found in [85].

3.1.2 Orc Expressions

The set of Orc expressions that can be constructed from the syntax of Figure 2.1, in addition to the internal publishing and handle expressions of the forms $!w$ and $?h$, is represented by a sort Expr, which is subsorted into the (singleton) zero expression subsort ZExpr, containing only $\mathbf{0}$ (which is declared as $\mathbf{0} : \rightarrow$ ZExpr), and the subsort of non-zero expressions, NZExpr. This distinction between $\mathbf{0}$ and other expression will simplify the specification and will help achieve a more efficiently executable semantics, as we will see later.

In the synchronous semantics of Orc, the contrast between internal actions (publishing of values, site calls, and τ transitions) and the external action of a site return induces a corresponding distinction between expressions that can make an internal transition and others that cannot. To capture the synchronous semantics, we make this distinction explicit in the type structure by introducing the notions of *active* and *inactive* Orc expressions.

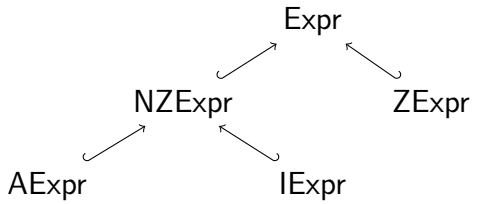


Figure 3.2: The subsort structure of Orc expressions

Intuitively, an expression is active if it contains as a sub-expression a value publishing, a site call, or an expression call sub-expression that is *enabled*, and is inactive otherwise. This notion is made more precise in the following definition.

Definition 1 (Active and inactive Orc expressions)**.** *The set of* active *expressions* $\mathcal{F}_a$ *is the smallest set generated by the following rules:*

1. $M(\vec{v})$, $E(\vec{p})$, *and* $!v$ *are in* $\mathcal{F}_a$.
2. *If* $f \in \mathcal{F}_a$, *then* $f >x> g \in \mathcal{F}_a$ *and* $f \,;\, g \in \mathcal{F}_a$.
3. *If* $f \in \mathcal{F}_a$ *or* $g \in \mathcal{F}_a$, *then* $f \mid g \in \mathcal{F}_a$ *and* $g <x< f \in \mathcal{F}_a$.

A non-zero expression f *is called* active *if* $f \in \mathcal{F}_a$*; otherwise,* f *is* inactive.

Note that this notion of active expressions corresponds exactly to that of non-quiescent expressions in [35] (see Section 3.2.3 there). This notion can be elegantly captured in the type structure of the rewriting semantics by further subsorting NZExpr into two subsorts: AExpr, for active expressions, and IExpr for inactive expressions. The subsorting structure of Orc expressions is shown in Figure 3.2.

Since any non-zero Orc expression must either be active or inactive (and cannot be both), the subsorts must partition the sort NZExpr. This is achieved by a combination of subsort-overloaded function symbol declarations for Orc's syntax, along with appropriate equational axioms and frozenness information, and a few simple membership axioms based on Definition 1, as we explain below.

Basic Orc expressions. An expression call $E(\vec{p})$, which is always active, has a corresponding declaration of the form $\_(\_) : \mathsf{ExprName} \times \mathsf{ParamList} \to \mathsf{AExpr}$, whereas a parameter

call expression $p(\vec{p})$, which has the general declaration $\_(\_) : \mathsf{Param} \times \mathsf{ParamList} \to \mathsf{Expr}$, is active if and only if p is a site name $M \in \mathsf{SiteName}$ and $\vec{p}$ is a list of values $\vec{v} \in \mathsf{ValueList}$, and hence we have the following subsort-overloaded declaration

$$\_(\_) : \mathsf{SiteName} \times \mathsf{ValueList} \to \mathsf{AExpr}$$

which precisely characterizes active parameter calls. For Inactive calls, which are calls that fail to satisfy the condition above (and are *not* semantically equivalent to $\mathbf{0}$), a third declaration $\_(\_) : \mathsf{IVar} \times \mathsf{ValueList} \to \mathsf{IExpr}$ and two membership predicates

$$M(\vec{p}) : \mathsf{IExpr} \text{ if } \vec{p} \notin \mathsf{ValueList} \wedge \mathbf{stop} \notin \vec{p}$$
$$x(\vec{p}) : \mathsf{IExpr} \text{ if } \vec{p} \notin \mathsf{ValueList} \wedge \mathbf{stop} \notin \vec{p}$$

capture precisely when a parameter call is inactive. Given the parameter subsort structure in Figure 3.1, this declaration and the two membership predicates define inactive parameter calls as those in which either: (1) the called parameter is a site name and the list of arguments contains at least one variable and no **stop** values, or (2) the called parameter is a variable and the argument list may contain variables or non-**stop** values. Note that by identities (2.10) and (2.11) in the structural equivalence properties of Figure 2.3, the other cases, in which the called parameter is a value or the argument list contains a **stop** value, are all semantically equivalent to $\mathbf{0}$, and are, therefore, of the sort ZExpr.

The other basic expressions, comprising handle expressions $?h$ and publishing expressions $!p$ are similarly specified. In particular, handle expressions $?h$ are always inactive and are simply specified by the declaration $?\_ : \mathsf{Handle} \to \mathsf{IExpr}$. Publishing expressions $!p$, which are active when $p \in \mathsf{Value}$ and inactive when $p \in \mathsf{IVar}$, are specified by the following subsort-overloaded family of declarations:

$$!\_ : \mathsf{Param} \to \mathsf{Expr} \qquad !\_ : \mathsf{Value} \to \mathsf{AExpr} \qquad !\_ : \mathsf{IVar} \to \mathsf{IExpr}$$

Note that the third case, when p is **stop**, is equivalent to $\mathbf{0}$, according to identity (2.12) in

Figure 2.3, and is, therefore, of the sort ZExpr.

Composed Orc expressions. To complete the specification of active and inactive expressions, function symbol declarations for the four Orc combinators are also subsort-overloaded according to Definition 1. Specifically, the symmetric parallel composition combinator, which has associativity, commutativity and **0** as its identity all specified as equational axioms, has the following subsort-overloaded family of declarations:

$$\begin{aligned}
&_ \mid _ : \mathsf{Expr} \times \mathsf{Expr} \to \mathsf{Expr} && [\mathsf{assoc\ \ comm\ \ id} : \mathbf{0}] \\
&_ \mid _ : \mathsf{AExpr} \times \mathsf{Expr} \to \mathsf{AExpr} && [\mathsf{ditto}] \\
&_ \mid _ : \mathsf{IExpr} \times \mathsf{IExpr} \to \mathsf{IExpr} && [\mathsf{ditto}]
\end{aligned}$$

which precisely define when a symmetric parallel composition is active or inactive, according to Definition 1. Similarly, the following declarations specify the sequential composition operator:

$$\begin{aligned}
&_ >_> _ : \mathsf{Expr} \times \mathsf{Var} \times \mathsf{Expr} \to \mathsf{Expr} && [\mathsf{frozen}(3)] \\
&_ >_> _ : \mathsf{AExpr} \times \mathsf{Var} \times \mathsf{Expr} \to \mathsf{AExpr} && [\mathsf{ditto}] \\
&_ >_> _ : \mathsf{IExpr} \times \mathsf{Var} \times \mathsf{Expr} \to \mathsf{IExpr} && [\mathsf{ditto}]
\end{aligned}$$

Since the right subexpression of a sequential composition has no behavioral transitions, the sequential combinator symbol is declared frozen on its third argument; i.e., we define $\phi(\_ >\_> \_) = \{3\}$, so that no rewriting is allowed on the third argument. The declarations state that a sequential composition is active (resp. inactive) if and only if its left subexpression is active (resp. inactive). The operator declarations for the asymmetric parallel combinator are similar to those of symmetric composition:

$$\begin{aligned}
&_ <_< _ : \mathsf{Expr} \times \mathsf{Expr} \to \mathsf{Expr} && _ <_< _ : \mathsf{Expr} \times \mathsf{AExpr} \to \mathsf{AExpr} \\
&_ <_< _ : \mathsf{IExpr} \times \mathsf{IExpr} \to \mathsf{IExpr} && _ <_< _ : \mathsf{AExpr} \times \mathsf{Expr} \to \mathsf{AExpr}
\end{aligned}$$

Function symbol declarations for the otherwise combinator are similar, except that the

$$\begin{array}{lll}
\alpha(\mathbf{0}) = \mathbf{0} & \alpha(p(\vec{p})) = (\alpha\ p)(\alpha\ \vec{p}) & \alpha(f >x> g) = (\alpha\ f) >x> (\Uparrow_{\boldsymbol{x}} \alpha\ g) \\
\alpha(?h) = ?h & \alpha(E(\vec{p})) = E(\alpha\ \vec{p}) & \alpha(f \mid g) = (\alpha\ f) \mid (\alpha\ g) \\
 & \alpha(!p) = !(\alpha\ p) & \alpha(f <x< g) = (\Uparrow_{\boldsymbol{x}} \alpha\ f) <x< (\alpha\ g) \\
 & & \alpha(f\ ;\ g) = (\alpha\ f)\ ;\ (\alpha\ g)
\end{array}$$

Figure 3.3: CINNI Substitutions on Orc expressions

symbol is declared associative with the identity $\mathbf{0}$, and frozen on its second argument:

$$\begin{array}{ll}
_\,;\,_ : \mathsf{Expr} \times \mathsf{Expr} \to \mathsf{Expr} & [\mathsf{frozen}(2)\ \mathsf{id} : \mathbf{0}] \\
_\,;\,_ : \mathsf{AExpr} \times \mathsf{Expr} \to \mathsf{AExpr} & [\mathsf{ditto}] \\
_\,;\,_ : \mathsf{IExpr} \times \mathsf{Expr} \to \mathsf{IExpr} & [\mathsf{ditto}]
\end{array}$$

Substitution. To complete the specification of Orc's instance of the CINNI calculus, substitution is extended from Orc parameters to Orc expressions. In general, a CINNI substitution α is extended to language expressions by adding, for each syntactic constructor f of arity n in the language, an equation of the form (with $\Uparrow_x \alpha$ a lift substitution):

$$\alpha\ f(P_1, \ldots, P_n) = f(\Uparrow_{P_{j_1,1}} \ldots \Uparrow_{P_{j_1,m_1}} \alpha\ P_1, \ldots, \Uparrow_{P_{j_n,1}} \ldots \Uparrow_{P_{j_n,m_n}} \alpha\ P_n) \tag{3.1}$$

where each P_i is an expression in the language, and $P_{j_i,1}, \ldots, P_{j_i,m_i}$ are the variable arguments that f binds in the ith expression argument P_i. In the case of Orc, the substitutions are defined using the equations shown in Figure 3.3, which are instances of the equation 3.1 above. Intuitively, when a substitution is applied to an expression, the substitution is propagated down the expression tree while keeping track of bound variable instances, so that the substitution can be correctly performed with no free variable capture.

Algebraic properties. To fully account for the algebraic properties of Orc expressions, the semantic infrastructure includes equations that correspond to the algebraic identities (2.7)–(2.12) in Figure 2.3. As mentioned above, identities (2.2)–(2.6) are specified as equational axioms of the respective Orc combinators, declared by the assoc, comm and id at-

tributes in their operators' declarations.

3.1.3 Orc Configurations

Following the strongly bisimilar MSOS-to-rewriting logic transformation of [86], a state in the execution of an Orc program is defined by an Orc *configuration*, which is a pair $\langle f, r \rangle$, where f is the Orc expression to be executed and r is a record structure consisting of fields of the form $a_i : u_i$, with a_i the field index and u_i its value. The fields of the record r hold state information required for the semantics of f. In the Orc semantics, five fields are used: (1) a label field $lbl : l$, (2) an environment for expression names $env : \sigma$, (3) a pool of pending messages $msg : \rho$, (4) a set of currently used handle names $hdl : \eta$, and (5) a clock $clk : t$. A more detailed description of these semantic fields follows.

Clock. Time is abstracted by the sort Time, which is specified as a totally ordered set with a least element zero. A supersort TimeInf of Time also includes ∞ as a top element, which is useful for specifying the proper timed semantics of Orc. An instantiation of the sort TimeInf can, therefore, be either discrete or dense. In our specifications we assume a dense time domain implemented by the non-negative rationals and maintained by the clock field in a configuration.

Environment. Since expressions may be referred to by their expression names, an environment σ is maintained by the configuration to resolve references to such names. An environment, a term of sort Env, is defined as a set of declarations, which are terms of sort Decl, with Decl $<$ Env, formed with an associative and commutative multiset union operator $\_,\_$, with the empty set as its identity element. Initially, an environment is created out of the declaration list $\vec{d}$ of an Orc program $\vec{d}\,;\ f$ so that the following conditions hold: (1) a later declaration in the list hides all previous declarations with the same expression name; and (2) all declarations in the resulting environment are visible to each other. This implies that an expression name has a unique defining declaration in an environment, and that (mutual) recursion is directly available.

Handles. A *handle* is a name of sort Handle that uniquely identifies a pending site call, which is a call waiting for a response from the environment. Since, by the SiteCall rule of

Figure 2.2, fresh handle names need to be generated, a configuration maintains in its handles field a set η of currently used handles against which new names may be created. Sets of handles, of sort HandleSet, are constructed by an associative, commutative comma-denoted union operator, with the empty set of handles as its identity element.

Messages. To simulate interactions of an Orc expression with its environment, we maintain in an Orc configuration a message pool (MsgPool) as a multiset of messages, constructed by the empty juxtaposition operator with the empty set as the identity element. A *message*, which is a term of the sort Msg, is either a site call message or a site return message. A site call message is of the form $[M, \vec{v}, h]$, with M the name of the called site, $\vec{v}$ the list of actual parameters of the call, and h the handle name identifying the site call. Since in the SOS semantics the environment is treated as a "black box", which is reasonable, since responses from external sites are unpredictable, we need to *simulate* environment responses to obtain an executable Orc semantics. This is achieved by equationally converting site call messages into potential site return messages of the form $[\mathsf{app}(M, \vec{v}, t), h]$, which represents a *potential* response back to the Orc expression for the call identified by h. The operator app, which is of sort PreValue, a supersort of ResValue, serves two purposes. First, it provides a uniform and abstract means by which the response of a particular site M can be modularly defined. Second, it may optionally associate a (possibly random) delay, given by t, to responses of external sites. The operational meaning is that a well-formed response is not generated until the delay reaches the value zero. Once the delay is zero (and assuming the external site was known to the environment), the term $\mathsf{app}(M, \vec{v}, 0)$ is evaluated according to the value of M to an actual response value w (a ground term of sort ResValue), and the message becomes a site return message of the form $[w, h]$. Only then, the site response is ready to be consumed by the Orc expression.

Labels. The label field keeps track of the last event generated as a result of a configuration evolving into another, which is needed in the SOS semantics for inferring one-step transitions. To represent the four labels in the SOS rules in Figure 2.2, we define a sort Label, and declare

four operators of this sort:

$$\_\langle \_, \_ \rangle : \mathsf{SiteName} \times \mathsf{ValueList} \times \mathsf{Handle} \to \mathsf{Label} \qquad \tau :\to \mathsf{Label}$$

$$\_?\_ : \mathsf{Handle} \times \mathsf{Value} \to \mathsf{Label} \qquad !\_ : \mathsf{Value} \to \mathsf{Label}$$

The label constructors above have syntax that is identical to that of the labels in the SOS specification. We also introduce a special constant $\epsilon :\to \mathsf{Label}$ to represent absence of a label.

Therefore, given an Orc program $\vec{d}\,;\ f$, its initial configuration, which can be constructed by an operator $[\_] : \mathsf{Program} \to \mathsf{Config}$, is of the form:

$$\langle f,\ lbl : \epsilon \mid env : \mathsf{init}(\vec{d}) \mid msg : \emptyset \mid hdl : \emptyset \mid clk : 0 \rangle$$

where init is a function that initializes an environment structure from a list of declarations $\vec{d}$ according to the description given above.

Additionally, as part of the semantic infrastructure, we define two notions about Orc configurations (borrowed from Real-Time Maude (RTM) [69]) that will be useful for defining the timed behaviors of Orc for both the SOS-based and the reduction rewriting semantics. The first is the notion of *eager* configurations, which are configurations that can make an instantaneous (internal or external) transition, i.e., configurations of the form $\langle f, r \rangle$ where either f is active or r has a pending site response that can be consumed. This notion is made more precise in the following definition, where $\hat{f}$ ranges over active Orc expressions and $\bar{f}$ over inactive expressions.

Definition 2 (Eager Orc configuration)**.** *An Orc configuration $\mathcal{C}$ is* eager *if $\mathcal{C}$ is of one of the following forms: (i) $\langle \hat{f}, r \rangle$; or (ii) $\langle \bar{f}, msg : \rho[w, h] \mid r \rangle$ with h a handle in $\bar{f}$.*

This notion is easily captured by a (partial) predicate $\mathsf{eager} : \mathsf{Config} \to [\mathsf{Bool}]$ that evaluates to true if and only if the given configuration is eager using two equations corresponding to cases (i) and (ii) in Definition 2 above. The second notion is that of the *maximal time elapse* (or *mte*) of an Orc configuration, which specifies the maximum time shift until the next point in time when an instantaneous event (corresponding to the evaluation of an Orc expression as opposed to just advancing time on the configuration) may be enabled.

Definition 3 (*mte* of an Orc configuration)**.** *The maximum time elapse (mte) of an Orc configuration* $\langle f,\ msg : \rho \mid r \rangle$ *is the minimum time delay across all messages in* ρ *if* ρ *is non-empty, and is* ∞ *otherwise.*

The time shift needed to advance the clock of an Orc configuration to the next point in time when an instantaneous action becomes enabled is determined by a function $\mathsf{mte} : \mathsf{Config} \rightarrow \mathsf{TimeInf}$, which is defined equationally according to the definition above.

3.2 The SOS-based Rewriting Semantics $\mathcal{R}^{sos}_{Orc}$

We now present a rewriting logic semantics of Orc that is based directly on the SOS semantics of Orc of Section 2.3.3. The rewriting logic semantics is obtained by mapping the SOS rules in Figure 2.2 into a corresponding rewrite theory $\mathcal{R}^{sos}_{Orc} = (\Sigma^s, E^s \cup A^s, R^s, \phi^s)$ according to Meseguer and Braga's semantics-preserving transformation from Modular SOS [86] to rewriting logic. An initial version of the SOS-based rewriting semantics of Orc appeared in [52], where we described two different ways of capturing Orc's synchronous semantics: (1) strategy expressions, and (2) additional equational conditions. The semantics in [52] also captured timed behaviors in Orc, although timing, as specified there, was limited to discrete time domains, such as the natural numbers.

Since the initial version in [52], the rewriting semantics of Orc given by $\mathcal{R}^{sos}_{Orc}$ has been thoroughly refined and extended to achieve a more complete, elegant, and efficiently executable specification. First, using order-sorted structures for Orc values, a concise representation of the new *otherwise combinator* and its semantics has been achieved. Moreover, order-sorted declarations for Orc expressions and action labels, and membership equations enable a simpler and more elegant specification of instantaneous actions that can be executed and analyzed more efficiently than with just the many-sorted specifications used before. Furthermore, the semantics is now capable of handling dense time domains, using ideas from real-time rewrite theories [64] and Real-Time Maude [69], with implementations in both (Core) Maude and Real-Time Maude.

Therefore, in summary, the semantics $\mathcal{R}^{sos}_{Orc}$ we present here is characterized primarily by

being:

1. *comprehensive*, as it captures all the essential features of Orc, including, most notably, urgency of internal actions and timed behaviors,

2. *provably correct* with respect to the semantics in [35], since its instantaneous part follows almost immediately from the SOS semantics and exploits the strong bisimulation meta-theorem between MSOS and rewriting logic specifications in [87],

3. *executable*, so that one may simulate Orc programs and observe their possible behaviors, and

4. *formally analyzable* using exhaustive verification methods, such as breadth-first search and model checking.

The rewriting semantics $\mathcal{R}^{sos}_{Orc}$ builds on the semantic infrastructure discussed in Section 3.1, i.e., $\mathcal{R}^{sos}_{Orc} \supset \mathcal{R}_{\Omega}$. Below, we describe how $\mathcal{R}^{sos}_{Orc}$ captures the timed, synchronous semantics of Orc expressions, and discuss some of its important properties.

3.2.1 Instantaneous Rewriting Semantics Rules

Since, by rewriting logic's **transitivity** inference rule, a rewrite computation $t \longrightarrow t'$ may involve a sequence of one-step rewrites $t \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \ldots \longrightarrow t_n \longrightarrow t'$, we need to restrict rewrites of Orc configurations to be exactly one-step rewrites, corresponding to the single-step SOS behavior, as explained in [87]. For this purpose, we employ the SOS one-step modifier technique of [86, 62], in which two operators are declared: (1) a (frozen) prefix dot operator $\cdot\_ : \mathsf{Config} \to \mathsf{Config}$, and (2) a non-frozen operator $\mathsf{smallstep} : \mathsf{Config} \to \mathsf{Config}$. By defining the rewrite rules that correspond to the SOS rules in Figure 2.2 in the following format

$$\cdot\langle f, r\rangle \to \langle f', r'\rangle \text{ if } \bigwedge_{i=1}^{n} \cdot\langle f_i, r_i\rangle \to \langle f_i', r_i'\rangle \wedge C$$

with C an equational condition, we effectively restrict rewriting to single steps using the equation:

$$\mathsf{smallstep}(\langle f,\ lbl : l \mid r\rangle) = \mathsf{smallstep}(\cdot\langle f,\ lbl : \epsilon \mid r\rangle)$$

$$\textsc{SiteCall}: \cdot\langle M(\vec{v}),\ lbl : l \mid msg : \rho \mid hdl : \eta \mid r\rangle \\ \rightarrow \langle ?h,\ lbl : M\langle \vec{v}, h\rangle \mid msg : \rho[M, \vec{v}, h] \mid hdl : \eta, h \mid r\rangle \ \text{ if } \ h := \mathsf{fresh}(\eta)$$

$$\textsc{SiteRetV}: \cdot\langle ?h,\ lbl : l \mid msg : \rho[v, h] \mid hdl : \eta, h \mid r\rangle \rightarrow \langle !v,\ lbl : h?v \mid msg : \rho \mid hdl : \eta \mid r\rangle$$

$$\textsc{SiteRetN}: \cdot\langle ?h,\ lbl : l \mid msg : \rho[\mathbf{stop}, h] \mid hdl : \eta, h \mid r\rangle \\ \rightarrow \langle \mathbf{0},\ lbl : h?\mathbf{stop} \mid msg : \rho \mid hdl : \eta \mid r\rangle$$

$$\textsc{Publish}: \cdot\langle !v,\ lbl : l \mid r\rangle \rightarrow \langle \mathbf{0},\ lbl : !v \mid r\rangle$$

$$\textsc{Def}: \cdot\langle E(\vec{p}),\ lbl : l \mid env : \sigma, E(\vec{x}) \triangleq f \mid r\rangle \ \rightarrow \langle [\vec{p}/\vec{x}]f,\ lbl : \tau \mid env : \sigma, E(\vec{x}) \triangleq f \mid r\rangle$$

Figure 3.4: Rewrite rules in $\mathcal{R}^{sos}_{Orc}$ for basic expressions

where the label field is reset in preparation for the next transition step, which is enabled by the newly introduced prefix dot.

The rewrite rules in $\mathcal{R}^{sos}_{Orc}$ that specify the semantics of the basic Orc expressions are shown in Figure 3.4. The rules precisely match the correspondingly labeled SOS rules in Figure 2.2.

When executing a site call, according to the site call rewrite rule, the call expression is replaced by the handle expression ($?h$), where h is a fresh handle name generated by a function fresh with respect to the currently used set of handle names η, using a *matching equation* in the condition (see [88], Section 4.3). Furthermore, the rule emits a message targeted to M into the message pool, adds a site call event label, and updates the handles set. When a site response that corresponds to the call with handle h appears in the message pool, one of the site return rules applies, depending on whether the response is the **stop** value or not. In both cases, the site return rules replace the handle expression with the appropriate Orc expression, add a site return event label, remove the message from the pool, and update the set of handles. The rules for publishing expressions and expression calls are very similar to their counterparts in the SOS specification.

Figure 3.5 lists the rewrite rules in $\mathcal{R}^{sos}_{Orc}$ that specify the *synchronous*, instantaneous semantics of the Orc combinators. In the figure, we let $\hat{f}, \hat{g}$ range over active expressions (of the sort AExpr), $\bar{f}, \bar{g}$ over inactive expressions (IExpr), and $\widetilde{f}, \widetilde{g}$ over non-zero expressions

(NZExpr). We also let i denote an internal action label (i.e., a non-site-return label), and n a non-publishing internal action label (i.e., a site call or a τ label). An important distinction between the rewrite rules in $\mathcal{R}^{sos}_{Orc}$ and the SOS rules in Section 2.3.3, is that the former capture the *synchronous* semantics of Orc expressions whereas the rules in Section 2.3.3 describe the unrestricted *asynchronous* semantics. This explains the larger number of rules in Figure 3.5 compared to those in Figure 2.2. Indeed, for each rule in the SOS rules for Orc's combinators in Figure 2.2, there are one or more rewrite rules in $\mathcal{R}^{sos}_{Orc}$ that correspond to it. For example, The SOS rule SYM for symmetric parallel composition has two corresponding rewrite rules in $\mathcal{R}^{sos}_{Orc}$, one capturing internal actions for active expressions (SYMI), while the other deals with inactive expressions consuming site returns (SYME). Since the symmetric combinator is commutative with identity $\mathbf{0}$, the two rewrite rules fully specify the synchronous semantics of parallel composition. Similar observations also apply to the remaining rewrite rules in Figure 3.5.

3.2.2 The Tick Rule

Following the standard approach for specifying time in rewriting semantic definitions of real-time systems [64] – using either regular or real-time rewrite theories, the theory $\mathcal{R}^{sos}_{Orc}$ includes a time *tick* rewrite rule to capture the timed semantics of Orc, in addition to the *instantaneous* rewrite rules in Figure 3.4 and Figure 3.5. The tick rule in $\mathcal{R}^{sos}_{Orc}$ is a one-step rule defined as follows (with t' of the sort Time):

$$\text{TICK} : \cdot\langle f, clk : t \mid r\rangle \to \langle f', clk : t + t' \mid \delta(r, t')\rangle \quad \text{if } \mathsf{eager}(\langle f, clk : t \mid r\rangle) \neq \mathsf{true} \ \wedge \ t' := \mathsf{mte}(r) \ \wedge \ t' \neq 0$$

Note that the variable t' only appears in the righthand side. The value of t' is determined, by the matching equation $t' := \mathsf{mte}(r)$ in the condition, to be the maximum time elapse (see Definition 3). The function δ propagates the effect of a clock tick t' down the record structure of a configuration (somewhat similar to time-shifting in [81]). It essentially updates delays of messages in the message pool, which makes response messages from site calls become

$\textsc{SymI}: \cdot\langle \hat{f} \mid \widetilde{g},\ lbl : l \mid r\rangle \rightarrow \langle f' \mid \widetilde{g},\ lbl : i \mid r'\rangle$ if $\cdot\langle \hat{f}, lbl : \epsilon \mid r\rangle \rightarrow \langle f', lbl : i \mid r'\rangle$

$\textsc{SymE}: \cdot\langle \bar{f} \mid \bar{g},\ lbl : l \mid r\rangle \rightarrow \langle f' \mid \bar{g},\ lbl : h?w \mid r'\rangle$ if $\cdot\langle \bar{f}, lbl : \epsilon \mid r\rangle \rightarrow \langle f', lbl : h?w \mid r'\rangle$

$\textsc{Seq1V}: \cdot\langle \hat{f} >x> g,\ lbl : l \mid r\rangle \rightarrow \langle (f' >x> g) \mid [v/x]g,\ lbl : \tau \mid r'\rangle$
if $\cdot\langle \hat{f}, lbl : \epsilon \mid r\rangle \rightarrow \langle f', lbl : !v \mid r'\rangle$

$\textsc{Seq1NI}: \cdot\langle \hat{f} >x> g,\ lbl : l \mid r\rangle \rightarrow \langle f' >x> g,\ lbl : n \mid r'\rangle$ if $\cdot\langle \hat{f}, lbl : \epsilon \mid r\rangle \rightarrow \langle f', lbl : n \mid r'\rangle$

$\textsc{Seq1NE}: \cdot\langle \bar{f} >x> g,\ lbl : l \mid r\rangle \rightarrow \langle f' >x> g,\ lbl : h?w \mid r'\rangle$
if $\cdot\langle \bar{f}, lbl : \epsilon \mid r\rangle \rightarrow \langle f', lbl : h?w \mid r'\rangle$

$\textsc{Asym1V}: \cdot\langle g <x< \hat{f},\ lbl : l \mid r\rangle \rightarrow \langle [v/x]g,\ lbl : \tau \mid r'\rangle$ if $\cdot\langle \hat{f}, lbl : \epsilon \mid r\rangle \rightarrow \langle f', lbl : !v \mid r'\rangle$

$\textsc{Asym1NI}: \cdot\langle g <x< \hat{f},\ lbl : l \mid r\rangle \rightarrow \langle g <x< f',\ lbl : n \mid r'\rangle$ if $\cdot\langle \hat{f}, lbl : \epsilon \mid r\rangle \rightarrow \langle f', lbl : n \mid r'\rangle$

$\textsc{Asym1NEa}: \cdot\langle \bar{g} <x< \bar{f},\ lbl : l \mid r\rangle \rightarrow \langle \bar{g} <x< f',\ lbl : h?w \mid r'\rangle$
if $\cdot\langle \bar{f}, lbl : \epsilon \mid r\rangle \rightarrow \langle f', lbl : h?w \mid r'\rangle$

$\textsc{Asym1NEb}: \cdot\langle \mathbf{0} <x< \bar{f},\ lbl : l \mid r\rangle \rightarrow \langle \mathbf{0} <x< f',\ lbl : h?w \mid r'\rangle$
if $\cdot\langle \bar{f}, lbl : \epsilon \mid r\rangle \rightarrow \langle f', lbl : h?w \mid r'\rangle$

$\textsc{Asym2I}: \cdot\langle \hat{g} <x< \widetilde{f},\ lbl : l \mid r\rangle \rightarrow \langle g' <x< \widetilde{f},\ lbl : i \mid r'\rangle$ if $\cdot\langle \hat{g}, lbl : \epsilon \mid r\rangle \rightarrow \langle g', lbl : i \mid r'\rangle$

$\textsc{Asym2E}: \cdot\langle \bar{g} <x< \bar{f},\ lbl : l \mid r\rangle \rightarrow \langle g' <x< \bar{f},\ lbl : h?w \mid r'\rangle$
if $\cdot\langle \bar{g}, lbl : \epsilon \mid r\rangle \rightarrow \langle g', lbl : h?w \mid r'\rangle$

$\textsc{OtherV}: \cdot\langle \hat{f} \mathbin{;} \widetilde{g},\ lbl : l \mid r\rangle \rightarrow \langle f',\ lbl : !v \mid r'\rangle$ if $\cdot\langle \hat{f}, lbl : \epsilon \mid r\rangle \rightarrow \langle f', lbl : !v \mid r'\rangle$

$\textsc{OtherNI}: \cdot\langle \hat{f} \mathbin{;} \widetilde{g},\ lbl : l \mid r\rangle \rightarrow \langle f' \mathbin{;} \widetilde{g},\ lbl : n \mid r'\rangle$ if $\cdot\langle \hat{f}, lbl : \epsilon \mid r\rangle \rightarrow \langle f', lbl : n \mid r'\rangle$

$\textsc{OtherNE}: \cdot\langle \bar{f} \mathbin{;} \widetilde{g},\ lbl : l \mid r\rangle \rightarrow \langle f' \mathbin{;} \widetilde{g},\ lbl : h?w \mid r'\rangle$ if $\cdot\langle \bar{f}, lbl : \epsilon \mid r\rangle \rightarrow \langle f', lbl : h?w \mid r'\rangle$

Figure 3.5: Rewrite rules in $\mathcal{R}^{sos}_{Orc}$ for the combinators

eventually available.

Ticking the clock and updating the record structure accordingly are not enough for the proper timed semantics of Orc, because if not appropriately controlled, new undesirable behaviors may be introduced, such as advancing time indefinitely or beyond a point when an instantaneous action was enabled (and, in effect, *missing* that action). This is avoided by defining a *maximal, time-synchronous* execution semantics, in which an Orc configuration with no enabled instantaneous actions is allowed to advance its clock all the way up to the next point in time when an instantaneous action will be enabled[1]. This restriction is formally specified by making the tick rule conditional on: (1) the configuration *not* being *eager*, i.e., being incapable of making an instantaneous transition as defined by the eager predicate, and (2) the time shift t' being equal to the *maximal time elapse* of the configuration, as defined by the mte function, which must be non-zero (see Section 3.1.3 for definitions of eagerness and maximal time elapse of configurations). These conditions ensure that time is advanced as much as possible in every application of the tick rule, but only enough so as to be able to capture all events of interest.

3.2.3 Correctness of $\mathcal{R}^{sos}_{Orc}$

The original SOS transition relation $\hookrightarrow$ proposed in [35] (a variant of which was shown in Figure 2.2) and it's refinement, also in [35], into two sub-relations $\hookrightarrow_R$ and $\hookrightarrow_A$ for quiescent and non-quiescent Orc expressions, respectively, defined the synchronous semantics of the instantaneous actions in Orc. Although a non-trivial timed extension of the original SOS specification was later proposed in [81], the extension did not consider the synchronous semantics, and abstracted non-publishing actions as unobservable actions for simplicity of presentation. We, therefore, show correctness of the rewriting semantics given by $\mathcal{R}^{sos}_{Orc}$ with respect to the SOS semantics in [35] by comparing the transition systems defined by the refined SOS relation $\hookrightarrow_R \cup \hookrightarrow_A$ and the instantaneous part of the rewrite theory $\mathcal{R}^{sos}_{Orc}$,

[1]For the analysis to be mechanizable, we also assume Orc programs with "non-Zeno" behaviors [75], such that only a finite number of instantaneous transitions are possible within any finite period of time.

namely, the theory $\mathcal{R}^{sos}_{Orc}$ without the TICK rule.[2]

First, a simple lemma relating the notions of non-quiescent expressions in [35] and active expressions as defined in Section 3.1.2.

Lemma 1. *f is non-quiescent (resp. quiescent) iff f is active (resp. inactive).*

Proof. Straightforward by structural induction on f. □

We denote by $\mathcal{R}^{sos}_{Orc} \vdash t \rightarrow^1_I t'$ a single rewrite step obtained by an application of an instantaneous (non-tick) rewrite rule, i.e., a rule $I \in R^s - \{\text{TICK}\}$. Correctness of $\mathcal{R}^{sos}_{Orc}$ is expressed by the following theorem.

Theorem 1 (Correctness of $\mathcal{R}^{sos}_{Orc}$)**.** *For any two Orc expressions f and f', and for $X \in \{A, R\}$,*

$$f \hookrightarrow_X f' \iff \mathcal{R}^{sos}_{\text{Orc}} \vdash \mathsf{smallstep}(\cdot\langle f,\ lbl : l \mid r\rangle) \rightarrow^1_I \mathsf{smallstep}(\langle f',\ lbl : l_X \mid r'\rangle)$$

with l_A an internal action label and l_R a site return action label.

Proof. The proof follows trivially by construction of $\mathcal{R}^{sos}_{Orc}$, based on the correctness of the MSOS-to-rewriting logic transformation methodology, given by the strong bisimulation theorem (Theorem 1) in [87], and from Lemma 1. □

3.2.4 Executability Properties of $\mathcal{R}^{sos}_{Orc}$

As we show in this section, the specification of the theory $\mathcal{R}^{sos}_{Orc}$ is not only correct with respect to Orc's synchronous semantics but also satisfies some desirable admissibility and executability properties that make it computable and amenable to sound and complete formal analysis and verification. In particular, the signature Σ^s is A^s-preregular [20] and the equations E^s and the rules R^s are deterministic. Furthermore, the equations E^s are operationally terminating, confluent, and sort-decreasing modulo the axioms A^s, and the

[2]Further refining the timed semantics in [81] to capture Orc's synchronous semantics within the SOS framework, and investigating its relationship to the rewriting semantics $\mathcal{R}^{sos}_{Orc}$ is an interesting direction for future work.

rules R^s are coherent with E^s. As a result, the specification $\mathcal{R}^{sos}_{Orc}$, through its implementation in Maude as the system module named `SOS-ORC`, can be executed and formally analyzed.

Since the current Maude tools for checking confluence of equations and coherence of rules with respect to equations [89, 90] only support order-sorted specifications (as opposed to more general membership equational specifications), we apply a simple, semantics-preserving transformation to convert the underlying conditional membership equational logic theory $(\Sigma^s, E^s \cup A^s)$ into a semantically equivalent conditional *order-sorted* equational logic theory $(\hat{\Sigma}^s, \hat{E}^s \cup A^s)$ with no kinds or membership assertions so that the original theory is executable iff the transformed one is. The resulting rewrite theory is denoted $\hat{\mathcal{R}}^{sos}_{Orc}$ (see Appendix B.1 for details on this transformation). Moreover, the corresponding Maude specification in `SOS-ORC` is also transformed so that it is order-sorted, and does not use the `owise` attribute or any built-in function [89, 90].

While the syntactic requirements of preregularity and determinism can be easily checked automatically by Maude, the other properties require a more thorough and careful investigation.

Operational termination of the conditional membership rewrite theory associated with (order-sorted) equational theory $(\Sigma^s, E^s \cup A^s)$ can be automatically proved using the Maude Termination Tool (MTT) [91] with AProVE [92] as the back-end tool. When supplied with the specification of the theory $(\Sigma^s, E^s \cup A^s)$, MTT applies a sequence of non-termination-preserving theory transformations (see [93]) to convert the conditional order-sorted equational theory into an unsorted, unconditional term rewriting system (TRS) $\mathcal{T}$, which is then fed into AProVE for an automatic termination check[3]. Therefore, we have the following result.

Lemma 2 (Operational Termination)**.** *The set of equations $\hat{E}^s$ in $\hat{\mathcal{R}}^{sos}_{Orc} = (\hat{\Sigma}^s, \hat{E}^s \cup A^s, R^s, \phi^s)$ is operationally terminating modulo the axioms A^s.*

Since $\hat{E}^s$ is terminating modulo A^s, the ground confluence and descent properties of $\hat{E}^s$ can be verified (semi-)automatically using Maude's Church-Rosser Checker (CRC) tool [89]. The

[3]The TRS $\mathcal{T}$ was obtained using the C;Uk;B transformation in MTT [93]. Its specification in TPDB notation and the generated termination proof script are available online at `http://www.cs.illinois.edu/~alturki/`.

CRC tool, when given $\hat{\mathcal{R}}^{sos}_{Orc}$ as input, computes all the (conditional) critical pairs between the equations in $\hat{E}^s$ and attempts, using various techniques, to show them either unfeasible or context joinable. For $\hat{E}^s$ in $\hat{\mathcal{R}}^{sos}_{Orc}$, the tool was able to discharge the vast majority of the critical pairs generated, leaving only a few that can be easily discharged by inductive reasoning. Some more details about such proof obligations generated by the tool can be found in Appendix B.1.

Lemma 3 (Ground Church-Rosser)**.** *The set of equations $\hat{E}^s$ in $\hat{\mathcal{R}}^{sos}_{Orc} = (\hat{\Sigma}^s, \hat{E}^s \cup A^s, R^s, \phi^s)$ is ground confluent modulo the axioms A^s and ground sort-decreasing.*

Since R^s contains conditional rewrite rules with rewrites in their conditions, the Maude Coherence Checker (ChC) tool [90], which assumes that the conditions of rules are equational, cannot be directly used to show coherence of the specification. It turns out, however, that the coherence property for $\hat{\mathcal{R}}^{sos}_{Orc}$ can be shown manually fairly easily, primarily because $\hat{\mathcal{R}}^{sos}_{Orc}$ is a top-most theory, with Config as the top sort. A fairly detailed proof showing sufficient conditions checked for ground coherence can be found in Appendix B.1.

Lemma 4 (Ground coherence)**.** *The set of rewrite rules R^s in $\hat{\mathcal{R}}^{sos}_{Orc} = (\hat{\Sigma}^s, \hat{E}^s \cup A^s, R^s, \phi^s)$ is ground coherent with respect to $\hat{E}^s$.*

Executability of $\hat{\mathcal{R}}^{sos}_{Orc}$ (and, hence, executability of $\mathcal{R}^{sos}_{Orc}$) follows immediately from its admissibility properties and Lemmas (2)–(4) above.

Theorem 2 (Executability of $\hat{\mathcal{R}}^{sos}_{Orc}$)**.** *The specification given by $\hat{\mathcal{R}}^{sos}_{Orc}$ satisfies the executability requirements of generalized rewrite theories.*

3.3 The Reduction Rewriting Semantics $\mathcal{R}^{red}_{Orc}$

Although the rewriting specification $\mathcal{R}^{sos}_{Orc}$ is readily understandable and its correctness with respect to the SOS semantics in [35] is straightforward, its execution, in practice, is quite expensive and inefficient. This is partly because $\mathcal{R}^{sos}_{Orc}$ makes extensive use of *conditional* rewrite rules (corresponding to the rules in the SOS specifications) which are particularly expensive to execute as compared to unconditional rules. Moreover, most of these rewrite

rules, besides being conditional, have rewrites (as opposed to equations) in their conditions, which is typical of the SOS specification style. Rewrite conditions, as opposed to equational conditions, can be particularly expensive to find a proof for or to disprove as they are non-deterministic in nature. In addition, the relatively large number of such rules in the specification can potentially cause nested (recursive) rewrite checks when checking a rule's conditions, which adversely affect performance of execution and analysis.

This section introduces a specification for a rewrite theory $\mathcal{R}^{red}_{Orc}$ that is not directly based on the SOS specifications but is instead more akin to a *reduction semantics*. It utilizes the inherently distributed semantics of rewriting logic, and uses both equations, for modeling *deterministic* computation steps, and rewrite rules, for modeling the *non-deterministic* transitions. This is achieved primarily by localizing the rewrite rules as much as possible, and specifying equationally any required propagation of information between the subexpression to be rewritten (the *redex*) and the enclosing Orc configuration (the *context*). In effect, this approach minimizes the number of rewrite rules needed and reduces their complexity, and thus achieves a simpler and superior semantic specification that can be executed and analyzed much more efficiently.

Despite the superior execution and analysis efficiency of $\mathcal{R}^{red}_{Orc}$ over $\mathcal{R}^{sos}_{Orc}$, the semantics defined by $\mathcal{R}^{red}_{Orc}$ is still operational, in that it describes in detail how Orc programs are evaluated, and is, in fact, equivalent to $\mathcal{R}^{sos}_{Orc}$, in the sense that, given any Orc program P, the state transition systems of the semantics of P given by $\mathcal{R}^{sos}_{Orc}$ and $\mathcal{R}^{red}_{Orc}$ are strongly bisimilar, assuming that program configurations are closed[4]. This implies that $\mathcal{R}^{red}_{Orc}$ captures precisely the intended semantics of Orc while providing an efficient means for execution and formal analysis of Orc programs.

The specification of the reduction rewriting semantics builds on the semantic infrastructure introduced in Section 3.1, with the main difference that, unlike for $\mathcal{R}^{sos}_{Orc}$, action labels are not essential to $\mathcal{R}^{red}_{Orc}$, since label information is implicitly managed by auxiliary operators in $\mathcal{R}^{red}_{Orc}$. However, to maintain equivalence with $\mathcal{R}^{sos}_{Orc}$, the label field $lbl : l$ is maintained in Orc configurations as before. Furthermore, the one-step modifier strategy used in the

[4]Roughly speaking, a configuration $\langle f, r\rangle$ is *closed* if every expression name referenced in f has a declaration in r. This is discussed in more detail when we introduce the equivalence theorem in Section 3.4.

SOS-based semantics to implement one-step rewrites is no longer needed, as the semantics is *not* a direct translation of the SOS specification in Figure 2.2, but is instead a rewriting logic semantics.

We describe next the specification of the synchronous, timed semantics given by the theory $\mathcal{R}^{red}_{Orc}$.

3.3.1 The Internal Actions

Transition steps that correspond to internal actions of Orc expressions are specified using the IAction rewrite rule (with $\mathsf{act}^{\uparrow}$ an auxiliary function symbol, which will be described shortly):

$$\textsc{IAction} : \langle \hat{f}, r \rangle \to \langle \mathsf{act}^{\uparrow}(f', i), r \rangle \;\text{ if }\; \hat{f} \to \mathsf{act}^{\uparrow}(f', i)$$

The rule simply states that an (eager) configuration with an active expression may make an internal transition if the expression is able to make that transition. Note that, this rule is global at the configuration level, which is required in order to maintain equivalence with the interleaving semantics defined by the original Orc semantics, and is also essential for executability of the specification. An active expression may make an internal transition according to one of the following rules:

$$\textsc{SiteCall} : M(\vec{v}) \to \mathsf{act}^{\uparrow}(\mathsf{tmp}, \mathsf{siteCall}(M, \vec{v}))$$
$$\textsc{ExprCall} : E(\vec{p}) \to \mathsf{act}^{\uparrow}(\mathsf{tmp}, \mathsf{exprCall}(E, \vec{p}))$$
$$\textsc{Publish} : !v \to \mathsf{act}^{\uparrow}(\mathbf{0}, \mathsf{publish}(v))$$

Therefore, an active, basic sub-expression may rewrite to a frozen, auxiliary operator symbol $\mathsf{act}^{\uparrow} : \mathsf{Expr} \times \mathsf{InternalEvent} \to [\mathsf{Expr}]$, whose purpose is to propagate the action up the expression tree all the way to the top so that: 1) its effects are reflected in the configuration (e.g. emitting a site call message into the configuration) , and 2) any necessary information in the configuration can be propagated back to the sub-expression (e.g. getting globally fresh handle names for site calls). This process of propagating information back and forth between redexes and contexts is specified equationally by induction on the structure of

Orc expressions. In particular, for site and expression calls, $\mathsf{act}^{\uparrow}$ replaces the call with a temporary placeholder expression tmp and propagates the action up to the configuration according to the following equations (where c stands for a site call event $\mathsf{siteCall}(M, \vec{v})$, or an expression call event $\mathsf{exprCall}(E, \vec{p})$):

$$\mathsf{act}^{\uparrow}(f, c) \mid \widetilde{g} = \mathsf{act}^{\uparrow}(f \mid \widetilde{g}, c) \qquad \mathsf{act}^{\uparrow}(f, c) >x> g = \mathsf{act}^{\uparrow}(f >x> g, c)$$
$$\mathsf{act}^{\uparrow}(f, c) <x< \widetilde{g} = \mathsf{act}^{\uparrow}(f <x< \widetilde{g}, c) \qquad \mathsf{act}^{\uparrow}(f, c) \,;\, \widetilde{g} = \mathsf{act}^{\uparrow}(f \,;\, \widetilde{g}, c)$$
$$g <x< \mathsf{act}^{\uparrow}(f, c) = \mathsf{act}^{\uparrow}(g <x< f, c)$$

Once the call reaches the root of the expression, the effect of the call is reflected in the containing configuration, using one of the following equations, depending on the call type:

$$\langle \mathsf{act}^{\uparrow}(f, \mathsf{siteCall}(M, \vec{v})),\ lbl : l \mid msg : \rho \mid hdl : \eta \mid r \rangle$$
$$= \langle \mathsf{act}^{\downarrow}(f, ?h),\ lbl : \epsilon \mid msg : \rho[M, \vec{v}, h] \mid hdl : \eta, h \mid r \rangle \ \text{ if } \ h := \mathsf{fresh}(\eta)$$

$$\langle \mathsf{act}^{\uparrow}(f, \mathsf{exprCall}(E, \vec{p})),\ lbl : l \mid env : \sigma, E(\vec{x}) \triangleq g \mid r \rangle$$
$$= \langle \mathsf{act}^{\downarrow}(f, [\vec{p}/\vec{x}]g),\ lbl : \epsilon \mid env : \sigma, E(\vec{x}) \triangleq g \mid r \rangle$$

which capture precisely the semantics of site and expression calls, respectively. The specifications of the effects of site calls and expression calls on the record structure of a configuration are identical to those in the SiteCall and Def rules of the SOS-based semantics of $\mathcal{R}^{sos}_{Orc}$, except that the label field is reset to ϵ. Note that since both the handle h in a site call and the instantiated body g of the expression definition in an expression call need to propagate back to the subterm where the call was made (which was temporarily substituted by the expression tmp), $\mathsf{act}^{\uparrow}$ does not rewrite immediately to f, but rather to another (frozen) operator, $\mathsf{act}^{\downarrow} : \mathsf{Expr} \times \mathsf{Expr} \to \mathsf{Expr}$, that traverses down the expression tree until it reaches

the appropriate subterm, using the following equations:

$$\mathsf{act}^{\downarrow}(\widetilde{f} \mid \widetilde{f}', g) = \mathsf{act}^{\downarrow}(\widetilde{f}, g) \mid \mathsf{act}^{\downarrow}(\widetilde{f}', g) \qquad \mathsf{act}^{\downarrow}(\widetilde{f}\,;\, \widetilde{f}', g) = \mathsf{act}^{\downarrow}(\widetilde{f}, g)\,;\, \widetilde{f}'$$

$$\mathsf{act}^{\downarrow}(\widetilde{f} >x> f', g) = \mathsf{act}^{\downarrow}(\widetilde{f}, g) >x> f' \qquad \mathsf{act}^{\downarrow}(b, g) = b$$

$$\mathsf{act}^{\downarrow}(f <x< \widetilde{f}', g) = \mathsf{act}^{\downarrow}(f, g) <x< \mathsf{act}^{\downarrow}(\widetilde{f}', g) \qquad \mathsf{act}^{\downarrow}(\mathsf{tmp}, g) = g$$

where b is a basic Orc expression, and g is either a handle expression (for a site call), or the body expression of a declaration (for an expression call).

The internal action of publishing a value is defined slightly differently, although the overall operational behavior is similar. This is primarily because published values may be bound in an expression by sequential or asymmetric parallel compositions. In particular, if the published value v is *not* bound in the expression, the value is propagated all the way to the top, using the following equations:

$$\mathsf{act}^{\uparrow}(f, \mathsf{publish}(v)) \mid \widetilde{g} = \mathsf{act}^{\uparrow}(f \mid \widetilde{g}, \mathsf{publish}(v))$$

$$\mathsf{act}^{\uparrow}(f, \mathsf{publish}(v)) <x< \widetilde{g} = \mathsf{act}^{\uparrow}(f <x< \widetilde{g}, \mathsf{publish}(v))$$

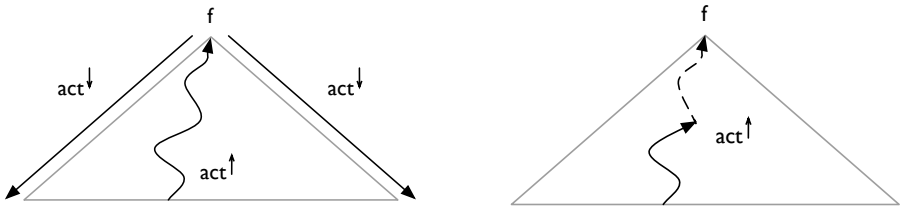
$$\mathsf{act}^{\uparrow}(f, \mathsf{publish}(v))\,;\, \widetilde{g} = \mathsf{act}^{\uparrow}(f, \mathsf{publish}(v))$$

In this case, the published value reaches the top of the expression in the enclosing configuration: $\langle \mathsf{act}^{\uparrow}(f, \mathsf{publish}(v)), lbl : l \mid r \rangle = \langle f, lbl : \epsilon \mid r \rangle$. Otherwise, if the value published is bound by a sequential composition expression or an asymmetric parallel composition expression, then one of the following equations applies:

$$\mathsf{act}^{\uparrow}(f, \mathsf{publish}(v)) >x> g = \mathsf{act}^{\uparrow}(f >x> g \mid [v/x]g, \mathsf{publish}^{\tau})$$

$$g <x< \mathsf{act}^{\uparrow}(f, \mathsf{publish}(v)) = \mathsf{act}^{\uparrow}([v/x]g, \mathsf{publish}^{\tau})$$

These equations reflect the semantics specified by the SOS rules SEQ1V and ASYM1V of Figure 2.2 (and the corresponding rewrite rules in $\mathcal{R}^{sos}_{Orc}$). They also change the value publishing event to a τ publishing event $publish^{\tau}$, which ultimately causes the label field of the configuration to reset (the equations for terms of the form $\mathsf{act}^{\uparrow}(f, \mathsf{publish}^{\tau})$ are similar).



(a) Site and expression calls (b) Publishing and τ actions

Figure 3.6: Schematic diagrams of the equational propagation of information in an expression tree

Notice that in both cases, when a publishing (or a τ) event reaches the configuration, no further information needs to be communicated back down the expression, unlike the cases of site and expression calls. Figure 3.6 gives a schematic representation of the mechanics of the internal actions. The figure shows that the structures of a site call and an expression call are similar, although the information propagated in both directions (and the side effects on the enclosing configurations) are different.[5]

3.3.2 The Site Return Action

The external action of a site return is modeled by the following rewrite rule:

$$\textsc{SiteReturn} : \langle \bar{f},\ lbl : l \mid msg : \rho[w,h] \mid hdl : \eta, h \mid r \rangle \\ \rightarrow \langle \mathsf{sret}(\bar{f}, w, h),\ lbl : \epsilon \mid msg : \rho \mid hdl : \eta \mid r \rangle \ \text{ if } \ h \in \mathsf{handles}(\bar{f})$$

which corresponds to the SiteRetV and SiteRetStop rules in the SOS rules and the SOS-based rewrite rules of $\mathcal{R}^{sos}_{Orc}$. Note that application of the site return rule above is subjected to the condition that the handle name of the message to be consumed is referenced in $\bar{f}$. This is to avoid useless transitions that could take place when a thread, having an unfinished site call, is pruned using asymmetric parallel composition, and thus, maintains a comparable behavior to site returns in the SOS specification. In addition, the rule SiteReturn captures the synchronous semantics of Orc by matching an *inactive* expression $\bar{f}$ to consume the

[5]Unwanted concurrent execution of site calls, expression calls and publishing of values is avoided by equations that will introduce an *error* constant of the kind [Expr] in such cases. For reasons of confluence of the equations E^r, an expression having *error* as a subterm immediately collapses to *error* (see [54]).

site return message. By this rule, the expression $\bar{f}$ rewrites to a frozen auxiliary operator $\mathsf{sret} : \mathsf{Expr} \times \mathsf{ResValue} \times \mathsf{Handle} \to \mathsf{Expr}$ which carries the response parameters down to the appropriate pending handle expression, using a set of equations with a similar structure to the equations defining the semantics of the operator $\mathsf{act}^{\downarrow}$ above, except for handle expressions, which are specified with the following equations:

$$\mathsf{sret}(?h', v, h) = \mathsf{if}\ (h' == h)\ \mathsf{then}\ !v\ \mathsf{else}\ ?h'\ \mathsf{fi}$$
$$\mathsf{sret}(?h', \mathbf{stop}, h) = \mathsf{if}\ (h' == h)\ \mathsf{then}\ \mathbf{0}\ \mathsf{else}\ ?h'\ \mathsf{fi}$$

3.3.3 Timed Semantics

Like the SOS-based rewriting semantics $\mathcal{R}^{sos}_{Orc}$, time and the effects of time elapse are specified in the reduction semantics $\mathcal{R}^{red}_{Orc}$ using a tick rewrite rule and the δ methodology. In fact, the tick rewrite rule is almost identical to that of $\mathcal{R}^{sos}_{Orc}$ – but without the SOS one-step modifier:

$$\textsc{Tick} : \langle f, clk : t \mid r\rangle \to \langle f', clk : t + t' \mid \delta(r, t')\rangle \quad \text{if } \mathsf{eager}(\langle f, clk : t \mid r\rangle) \neq \mathsf{true} \ \wedge\ t' := \mathsf{mte}(r) \ \wedge\ t' \neq 0$$

The rule uses in its condition the eager and mte functions to capture the maximal, time-synchronous execution semantics, described before in Section 3.1.3.

3.3.4 Executability Properties of $\mathcal{R}^{red}_{Orc}$

This section shows that, like $\mathcal{R}^{sos}_{Orc}$, the theory $\mathcal{R}^{red}_{Orc} = (\Sigma^r, E^r \cup A^r, R^r, \phi^r)$ is executable, which implies that formal analysis using its implementation in Maude, as a system module `RED-ORC`, is both sound and complete. However, the additional equations defining the auxiliary operators for propagating event information and the fact that rewrite proofs for applications of the IAction rule may contain rewrites at the expression level (using the rules SiteCall , ExprCall, and Publish), all make the task of checking these executability properties slightly more involved.

As before, the membership equational logic sub-theory of the rewrite theory $\mathcal{R}^{red}_{Orc}$ is transformed to a semantically equivalent *order-sorted* equational theory, so that we may leverage existing Maude tools for checking executability properties. The transformed theory is denoted $\hat{\mathcal{R}}^{red}_{Orc} = (\hat{\Sigma}^r, \hat{E}^r \cup A^r, R^r, \phi^r)$. The transformation is also reflected in the corresponding Maude module `ORC-RED`.

Operational termination of the conditional membership rewrite theory associated with $(\hat{\Sigma}^r, \hat{E}^r \cup A^r)$ can be shown automatically using the MTT [91] and AProVE [92], using the same sequence of transformations as for the SOS-based rewriting semantics specification in Section 3.2.4, yielding the following result.

Lemma 5 (Operational termination)**.** *The set of equations $\hat{E}^r$ in $\hat{\mathcal{R}}^{red}_{Orc}$ is operationally terminating modulo the axioms A^r.*

The ground confluence and descent properties can be shown using Maude's CRC tool and inductive reasoning. More details can be found in a proof sketch in Appendix B.2 of the following lemma.

Lemma 6 (Ground Church-Rosser)**.** *The set of equations $\hat{E}^r$ in $\mathcal{R}^{red}_{Orc}$ is ground confluent modulo the axioms A^r and ground sort-decreasing.*

For ground coherence, the specification of $\hat{\mathcal{R}}^{red}_{Orc}$ cannot be directly checked using Maude's ChC tool because R^r has a rule with a rewrite condition, namely the rule IAction. However, since there is only one such rule and since the rewrite condition is the only condition of this rule, we may consider the sub-theory $\hat{\mathcal{R}}_\circ$ of $\hat{\mathcal{R}}^{red}_{Orc}$ without the IAction rule, show that it is ground coherent using Maude's ChC tool and induction on ground terms, and then reason inductively on the case when a rewrite is made by IAction in $\hat{\mathcal{R}}^{red}_{Orc}$. The details of this proof technique are given in Appendix B.2.

Lemma 7 (Ground coherence)**.** *The set of rewrite rules R^r in $\hat{\mathcal{R}}^{red}_{Orc} = (\hat{\Sigma}^r, \hat{E}^r \cup A^r, R^r, \phi^r)$ is ground coherent with respect to $\hat{E}^r$.*

We now have the following essential theorem, which is an immediate consequence of the admissibility properties of $\mathcal{R}^{red}_{Orc}$ and Lemmas (5)–(7) above.

Theorem 3 (Executability of $\mathcal{R}_{Orc}^{red}$)**.** *The specification given by $\mathcal{R}_{Orc}^{red}$ satisfies the executability requirements of generalized rewrite theories.*

3.4 Equivalence of $\mathcal{R}_{Orc}^{sos}$ and $\mathcal{R}_{Orc}^{red}$

We shall now show that the SOS-based rewriting semantics, $\mathcal{R}_{Orc}^{sos}$, and the reduction-based rewriting semantics, $\mathcal{R}_{Orc}^{red}$, are semantically equivalent, in the sense that an Orc program behaves in exactly the same way in both semantic models. We show this by proving a more general result, stating that the semantic models given by $\mathcal{R}_{Orc}^{sos}$ and $\mathcal{R}_{Orc}^{red}$ of any closed Orc configuration are strongly bisimilar. We first define what we mean by a configuration being closed.

Definition 4 (Closed configurations)**.** *An Orc configuration $\langle f, r\rangle$ is* well-formed *if: (i) f does not contain any auxiliary function symbol, such as* $\mathsf{act}^{\uparrow}$*,* sret*, or* tmp*; and (ii) r contains at least the five fields introduced in Section 3.1, namely: (1)* $\mathrm{lbl} : l$*, with $l \in \mathsf{Label}$, (2)* $\mathrm{hdl} : \eta$*, with $\eta \in \mathsf{HandleSet}$, (3)* $\mathrm{env} : \sigma$*, with $\sigma \in \mathsf{Env}$, (4)* $\mathrm{msg} : \rho$*, with $\rho \in \mathsf{MsgPool}$, and (5)* $\mathrm{clk} : t$*, with $t \in \mathsf{Time}$. Moreover, a* closed *configuration is a well-formed configuration in which no expression name appears free in f or σ.*

We observe that a closed configuration in $\mathcal{R}_{Orc}^{sos}$ is also a closed configuration in $\mathcal{R}_{Orc}^{red}$ and vice versa. This is due to the fact that both $\mathcal{R}_{Orc}^{sos}$ and $\mathcal{R}_{Orc}^{red}$ use the same semantic infrastructure. Moreover, we have the following easy lemma.

Lemma 8 (Preservation of closed configurations)**.** *Let $\mathcal{C}$ be a closed configuration. If $\mathcal{R}_{\mathrm{Orc}}^{sos} \vdash \mathcal{C} \to \mathcal{C}'$ for some configuration $\mathcal{C}'$, then $\mathcal{C}'$ is closed. Similarly, if $\mathcal{R}_{\mathrm{Orc}}^{red} \vdash \mathcal{C} \to \mathcal{C}'$, then $\mathcal{C}'$ is closed.*

Proof. This can be proved by rule induction on the rewriting relations induced by $\mathcal{R}_{Orc}^{sos}$ and $\mathcal{R}_{Orc}^{red}$, respectively. □

Intuitively, preservation of well-formedness is trivial in both $\mathcal{R}_{Orc}^{sos}$ and $\mathcal{R}_{Orc}^{red}$ by a quick examination of the rewrite rules. It is also easy to see that closed configurations in $\mathcal{R}_{Orc}^{sos}$ rewrite to configurations that are also closed. Essentially, the only rule in $\mathcal{R}_{Orc}^{sos}$ that might

introduce expression names in an expression is the [DEF] rule. But since all expression declarations are closed (have no free occurrences), and since actual parameters cannot be expression names, the resulting expression must also be closed. A similar argument also applies to $\mathcal{R}^{red}_{Orc}$. In what follows, we assume all configurations are closed.

The following lemma states that the definitions of eager configurations in $\mathcal{R}^{sos}_{Orc}$ and $\mathcal{R}^{red}_{Orc}$ coincide. The lemma is an easy consequence of the fact that the definitions of the eager predicate, the mte function, active and inactive expressions, messages, and auxiliary functions for the time domain and handle names, are all shared in the same semantic infrastructure given by $\mathcal{R}_\Omega$.

Lemma 9 (Timing strategy equivalence)**.** *For any configuration $\mathcal{C}$, $\mathcal{R}^{sos}_{\mathrm{Orc}} \vdash \mathsf{eager}(\mathcal{C}) = \mathrm{true}$ iff $\mathcal{R}^{red}_{\mathrm{Orc}} \vdash \mathsf{eager}(\mathcal{C}) = \mathrm{true}$, and, similarly, $\mathcal{R}^{sos}_{\mathrm{Orc}} \vdash \mathsf{mte}(\mathcal{C}) = t$ iff $\mathcal{R}^{red}_{\mathrm{Orc}} \vdash \mathsf{mte}(\mathcal{C}) = t$, for any $t \in \mathsf{Time}$.*

Now we are ready to present the equivalence theorem, for which a detailed proof can be found in Appendix B.3

Theorem 4. *For any configurations $\mathcal{C}$ and $\mathcal{C}'$, the following equivalence holds,*

$$\mathcal{R}^{sos}_{\mathrm{Orc}} \vdash \mathsf{smallstep}(\cdot\mathcal{C}) \to^1 \mathsf{smallstep}(\cdot\mathcal{C}') \iff \mathcal{R}^{red}_{\mathrm{Orc}} \vdash \mathcal{C} \to^1 \mathcal{C}'$$

The main result of this section can be derived as a consequence of Theorem 4 by taking as $\mathcal{C}$ the *initial configuration* of a program P given by $[P]$ (see Section 3.1.3).

Corollary 1. *For any Orc program P and configuration $\mathcal{C}$, we have*

$$\mathcal{R}^{sos}_{\mathrm{Orc}} \vdash [P] \to^1 \mathsf{smallstep}(\cdot\mathcal{C}) \iff \mathcal{R}^{red}_{\mathrm{Orc}} \vdash [P] \to^1 \mathcal{C}$$

Proof. Immediate from Theorem 4 and the fact that $[P]$ is closed, by definition. □

Therefore, for any Orc program P, the state transition systems defined by $\mathcal{R}^{sos}_{Orc}$ and $\mathcal{R}^{red}_{Orc}$ are strongly bisimilar.

3.5 Specification in Maude

Both rewrite theories $\mathcal{R}_{Orc}^{sos}$, for the SOS-based rewriting semantics, and $\mathcal{R}_{Orc}^{red}$ for the reduction rewriting semantics, have been specified as *admissible* and *executable* system modules in Maude, namely `SOS-ORC` and `RED-ORC`, respectively. These modules can be used to execute Orc programs, explore traces of computations, and verify safety and liveness properties about them by model checking. The desirable executability properties of both theories shown in Section 3.2.4 and Section 3.3.4 guarantee that the formal analysis performed using the corresponding Maude modules is both sound and complete. To take advantage of some of the time-based versions of the analysis tools available in Real-Time Maude (RTM), such as timed search and timed model checking, we have also developed corresponding *timed modules* in RTM for $\mathcal{R}_{Orc}^{sos}$ and $\mathcal{R}_{Orc}^{red}$ [6].

Since Maude supports mixfix user-definable syntax, the syntax in the Maude specification is a readable, typewriter version of the Orc syntax shown in Figure 2.1. For example, the declarations (introduced by the keyword `op`) below specify the syntax of active and inactive *otherwise* compositions:

```
op _;_ : Expr Expr -> Expr [assoc id: zero frozen(2)] .
op _;_ : AExpr Expr -> AExpr [ditto] .
op _;_ : IExpr Expr -> IExpr [ditto] .
```

where the combinator `_;_` is declared associative with the expression `zero` as its identity, and *frozen* in its second argument. Similarly, equational properties of the combinators, corresponding to the algebraic laws in Figure 2.3, and definitions and properties of other auxiliary operators, such as substitution, operations on handle sets and environment initializers, are all specified as (possibly conditional) equations and membership predicates. For instance, laws (2.7) – (2.9) in Figure 2.3 are declared using the following equations (introduced with the `eq` keyword):

[6] The RTM specification is almost identical to that of Maude, with the main exception that configurations in the RTM specification do not have to maintain a clock field since a global clock is implicitly managed by the tool. Furthermore, our specifications of $\mathcal{R}_{Orc}^{sos}$ and $\mathcal{R}_{Orc}^{red}$ can be easily shown to model *time-robust* systems [75], for which formal analysis with respect to the maximal time-synchronous execution strategy is complete.

```
eq zero > X > F = zero .                    eq ! V ; NZF = ! V .
eq F < X < zero = [X := stop] F .
```

assuming that `F`, `X`, `V` and `NZF` are Maude meta-variables representing, respectively, terms of the sorts `Expr`, `Var`, `Value`, and `NZExpr`. The term `[X := stop] F` represents a substitution applied to `F` mapping the Orc variable given by `X` to the special value `stop`. Finally, rewrite rules in $\mathcal{R}_{Orc}^{sos}$ and $\mathcal{R}_{Orc}^{red}$ are introduced in the corresponding Maude modules using the keyword `rl` (and `crl` for conditional rules). An example is the OTHERV rule in $\mathcal{R}_{Orc}^{sos}$ (shown in Figure 3.5):

```
crl [OtherV] : . < AF ; NZF , lbl : L | R > => < F' , lbl : ! V | R' >
    if . < AF , lbl : nl | R > => < F' , lbl : ! V | R' > .
```

with `AF` and `F'` ranging, respectively, over terms of the sorts `AExpr` and `Expr`, and `R`, `R'` over records in a configuration. The variable `L` ranges over labels while the constant `nl` corresponds to ϵ and represents absence of a label.

The full specification is available for download online at `http://www.cs.illinois.edu/~alturki/`.

To illustrate the use of the Maude specifications for formal analysis of Orc programs, we use an Orc program TIMEOUT based on the timeout expression given in Section 2.3.2 to demonstrate simple use cases of rewriting-based simulation and breadth-first search analysis. A discussion of more sophisticated analysis of LTL properties using Maude's model checker is given in Section 4.2.

Suppose that the initial configuration of TIMEOUT is given by the constant `to-config`, which is defined using the following equations (assuming that `x`, `t`, `y` are Orc variables, and `Timeout` is an expression name):

```
eq to-decl = Timeout(x,t) := let(y{0}) < y < x{0}() | Rtimer(t{0})
                           >> let(signal)
eq to-config = [ to-decl ; Timeout(M, 3) ] .
```

in which the expression to be evaluated is the expression call `Timeout(M, 3)`, which times out in 3 time units a call to a site `M`. Recall that the operator `[_]` constructs an initial configuration of a given Orc program (see Section 3.1.3).

We first obtain a sample execution of the program using the `rewrite` command (assuming no message transmission delays):

```
rewrite in RED-ORC : to-config .
result Config: < zero, lbl : nl | env : ... | clk : 3 |
                                msg : [signal,h(1)] | hdl : h(1) >
```

We observe that the call to M did not timeout as the site response `signal` caused by the call to `let` in the message `[signal,h(1)]` was never consumed. To investigate the sequence of rewrites that led to this (final) state, we may use Maude's `trace` command to output a trace of equations and rules applied with different display and detail options [20]. An easier, although much less powerful, approach, which we demonstrate here for its simplicity, is to use the `print` attribute to selectively annotate equations and rules of interest with output messages that get displayed whenever the corresponding statement is applied [88]. Using such annotations, the following trace can be produced using the rewrite command above:

```
Maude> rew to-config .
rewrite in RED-ORC : to-config .
Tau (Expr Call): Timeout(M,3)
Site Call:       M(); with Handle h(0)
Site Call:       Rtimer(3); with Handle h(1)
Site Return:     value 1 for handle h(0)
Tau (bound value)
Site Call:       let(1); with Handle h(0)
Site Return:     value 1 for handle h(0)
Publishing:      value 1
Time Tick :      from 0 by 3
result Config: < zero, lbl : nl | env : ... | clk : 3 |
                               msg : [signal,h(1)] | hdl : h(1) >
```

The trace lists the types of the actions taken, each of which is followed by some instantiation details. We note that, since execution begins at time 0 and time is not advanced until the very last step, all the instantaneous actions of this trace occur at time 0. The last step time-shifts the configuration by 3 time units, which is the timeout period, but no site return action takes places after that, since the thread that had the corresponding site call was already pruned at the second `tau` event in the trace. The (fairly obvious) fact that, in the absence of transmission delays, the call to `M` will never timeout can be exhaustively verified by, for example, the following `search command`:

```
Maude> search to-config =>* < F:Expr, msg : [self, 1, H:Handle]
    MS:MsgPool | clk : T:Time  | R:Record > such that T:Time  >= 3 .
No solution.
states: 18  rewrites: 371 in 1ms cpu (10ms real) (247498 rewrites/second)
```

which searches for a configuration having a pending response from `M` after 3 time units have passed, and fails to find a solution as expected. However, when the possibility of message transmission delays of 3 time units or longer are introduced, the response from `M` may now timeout, which may be witnessed by a simulation, or – when the set of possible delays is finite – shown by breadth-first search.

3.6 Performance Comparison

Despite being bisimilar, the reduction rewriting semantics given by $\mathcal{R}_{Orc}^{red}$ enjoys a significant performance advantage over the SOS-based rewriting semantics given by $\mathcal{R}_{Orc}^{sos}$. This section validates this claim by comparing the formal simulation and analysis performance of the two rewrite theories through their specifications in Maude.

Throughout all experiments, performance is measured in terms of the CPU time – in milliseconds – taken to perform a particular task as reported by Maude. The tasks are: (1) simulating six Orc programs using Maude's `rewrite` command, (2) exploring the state space of these six programs using Maude's breadth-first `search` command, and (3) model checking deadlock-freeness in four problem instances of a deadlock-free specification of the

$$
\begin{aligned}
BCast(m, x) \triangleq\ & (if(empty(x)) \gg let(\mathsf{signal}) \\
& \mid if(\neg empty(x)) \gg head(x) >a> a(m) \gg BCast(m, tail(x))) \\
CList(x) \triangleq\ & (if(empty(x)) \gg let([]) \\
& \mid if(\neg empty(x)) \gg head(x) >a> (append(y, ys) \\
& \quad <y< (a() \mid Rtimer(5) \gg let(\mathsf{signal})) \\
& \quad <ys< CList(tail(x))))
\end{aligned}
$$

Figure 3.7: Expression declarations for BCast and CList examples

		TIMEOUT	PRIORITY	PAR-OR	TIMED-M	BCAST	CLIST
$\mathcal{R}_{Orc}^{sos}$	rewrite	1.0	1.0	8.0	14.0	3.0	10.0
	search	1.0	2.0	84.0	326.0	5,492.0	6.0×10^5
$\mathcal{R}_{Orc}^{red}$	rewrite	1.0	1.0	1.0	1.0	1.0	1.0
	search	1.0	1.0	3.0	4.0	56.0	4.7×10^4

Table 3.2: A performance comparison of the rewriting semantics of Orc using Maude's `rewrite` and `search` commands (times in milliseconds)

the dining philosophers problem using Maude's LTL model checker. The Orc programs used as benchmarks for these tasks were borrowed or inspired from examples in [35]. In particular, TIMEOUT, which was also given in Section 2.3.2, PRIORITY, which prioritizes a site call over another, PAR-OR, which specifies a parallel (lazy) disjunction function, and TIMED-M, which makes four timed calls to a site, were all borrowed from [35]. The Orc programs BCAST, which implements a *sequential* broadcast, and CLIST, which constructs *in parallel* a tuple of responses from external sites with timeout, were both inspired by examples in [35]. The expression definitions for BCAST and CLIST are shown in Figure 3.7, where *empty*, *head*, *tail* and *append* are (local) sites implementing list functions. . For the dining philosophers benchmark, we use the deadlock-free specification given in [35]. For simplicity, we assume no message delays for all the benchmarking tasks above.

The results of these experiments are summarized in Table 3.2 and Table 3.3. To guarantee fairness in comparison, the experiments were carried out on the same machine, using the same version of Maude, and under similar operating conditions[7]. The results clearly show

[7]The experiments were carried out on a 2.93GHz quad-core machine with 24GB of memory using Maude 2.6.

Problem Size	2	3	4	5
$\mathcal{R}_{Orc}^{sos}$	22.0	2,423.0	4.8×10^5	∞
$\mathcal{R}_{Orc}^{red}$	3.0	107.0	3,230.0	1.6×10^5

Table 3.3: A performance comparison of the rewriting semantics of Orc using Maude's LTL model checker applied to four instances of the dining philosophers problem (times in milliseconds)

that the reduction semantics of Orc is much more efficiently executable than the SOS-based semantics. This is mainly because the SOS-based semantics makes extensive use of rewrite rules that are mostly conditional with rewrites in their conditions, which are, by their non-deterministic nature, more expensive to compute than unconditional rules. The reduction semantics, on the other hand, minimizes both the number of rewrite rules and the number of rewrites in the conditions, while using equations to specify the deterministic features of the language. Furthermore, unlike the SOS-based semantics, where several rules may have to be attempted before successfully making a transition, attempts to apply the instantaneous action rules in $\mathcal{R}_{Orc}^{red}$ never fail (as can be verified by Maude's profiler) since transition steps corresponding to instantaneous actions are specified only by two rules that match active expressions for internal actions (the rule labeled IAction), and inactive expressions for the external action of site return (the SiteReturn) rule). This significantly reduces the need for backtracking-like behaviors when (recursively) searching for proofs of rewrite conditions.

As Table 3.2 shows, the performance difference between $\mathcal{R}_{Orc}^{red}$ and $\mathcal{R}_{Orc}^{sos}$ is only marginal for small expressions with limited parallelism, such as TIMEOUT and PRIORITY. However, as expressions get more complex, the performance gap between the SOS-based semantics and the reduction semantics increases considerably, especially with the `search` command, since searching tries to build proofs of all reachable states, exposing it to the limitations of the SOS-based semantics even more. This is justified since the more complex an expression is, the larger the number of (parallel) compositions used, which translates into a larger number of conditional rewrite checks in the SOS-based semantics[8].

[8]It is important to note that, like the original synchronous SOS specification of Orc, both the SOS-based semantics and the reduction semantics are fairly detailed, operational semantics of Orc, and are, as a result, vulnerable to the state space explosion problem, when using the search or LTL model checking commands, particularly for programs with increasing levels of non-determinism (the CList instance, for example, had over 8.5×10^5 states).

The performance advantage of $\mathcal{R}_{Orc}^{red}$ over $\mathcal{R}_{Orc}^{sos}$ is more pronounced in the model checking experiments of the notoriously non-deterministic dining philosophers specification, as shown by Table 3.3. For the SOS-based semantics, the model checker did not finish within a reasonable amount of time for the problem instance with five philosophers.

CHAPTER 4

OBJECT-BASED REWRITING SEMANTICS OF ORC AND THE MORC TOOL

The Orc semantics, as described above in the SOS and the reduction rewriting semantics specifications, focuses on the, possibly concurrent, evaluation of a single Orc expression, abstracting away its interactions with external sites in an environment as if the environment were a black box. It is however very natural to view both Orc expressions and sites as *distributed objects*, which interact with each other through message passing. Indeed, in practical applications, many orchestration problems, especially relatively large ones, can be thought of as compositions of multiple Orc subexpressions that independently orchestrate different but related tasks. For example, an online auction management expression may be composed of subexpressions managing: (1) seller inventories and product auction announcements, (2) bid collection and winner announcements, and (3) payments and shipping coordination. Such subexpressions need not be located on the same machine for the orchestration effort to be completed, but are, in fact, more often run on physically distributed nodes spread across the web. Furthermore, sites normally maintain local states to support the services they provide, such as counter sites and channel (buffer) sites. Below, we describe a simple *semantic* extension of the Orc theory to a distributed, object-based programming model that is both natural and useful in specifying and analyzing distributed computations with explicit treatment of external sites and messages.

4.1 Distributed Object-based Semantics $\mathcal{R}_{Orc}$

The distributed object-based semantics encapsulates the Orc programming model as the underlying model for Orc expressions, and in this respect, its rewriting specification, denoted $\mathcal{R}_{Orc}$, directly builds on the reduction semantics specification $\mathcal{R}_{Orc}^{red}$ using rewriting logic's

approach to distributed objects [94]. $\mathcal{R}_{Orc}$ generalizes the reduction semantics to multiple Orc expressions and models the environment explicitly by (possibly external) site objects and asynchronous message passing. Although still a formal specification, this distributed object semantics can be seamlessly transformed into a distributed Orc *implementation* using Maude, as we discuss in Chapter 5.

4.1.1 Object-based Orc Configurations

In the distributed object semantics $\mathcal{R}_{Orc}$, an Orc program configuration is defined as a *multi-set* of objects and messages, specified by associative and commutative empty juxtaposition, with *none* as the identity element. An object is a term of the form $\langle O : C \mid A \rangle$, with O a unique object identifier, C the class name of the object, and A a set of attribute-value pairs, each of the form $attr : value$, where $attr$ is the attribute's name, and $value$ is its corresponding value. There are two main classes of Orc objects: *Expression* objects (of the class Expr) and *Site* objects (of the class Site), corresponding, respectively, to Orc expressions and sites. An expression object for an Orc program $\vec{d}\,;\ f$ has three attributes: (i) env, which holds the set of expression declarations corresponding to $\vec{d}$, (ii) exp, which maintains the Orc expression f to be evaluated, and (iii) hdl, which keeps a set of handle names that are currently being used by the expression. For instance, a timeout expression object may have the following *initial* form:

$$\begin{aligned}\langle O : \mathsf{Expr} \mid\ & env : Timeout(x,t) \triangleq let(y)\ {<}y{<}\ x()\ \mid\ Rtimer(t) \gg let(signal),\\ & exp : Timeout(M, 10),\ hdl : \emptyset\rangle\end{aligned}$$

In contrast, an Orc site object has the following three attributes: (i) $name$, which holds the site's name, such as *if*, *CNN*, *M*, ...etc, (ii) $state$, which abstractly maintains the current state of the site (the concrete definition of the state is site-specific), and (iii) op, which tells whether the site object is currently blocking (performing some operation) or accepting

incoming messages. For example, a simple *CNN* site object may have the following form:

$$\langle O : \mathsf{Site} \mid name : CNN,\ op : ready,\ state : (d_0, p_0), \cdots (d_n, p_n) \rangle$$

where *ready* is a constant signifying that the site is ready to accept calls, and the state is a list of dated news pages. Note that fundamental sites, such as *if* and *Rtimer*, and other basic sites, such as arithmetic functions, are stateless and thus make no use of the *state* field.

In keeping with the philosophy of the Orc theory, expression objects are modeled as *active* objects with one or more threads of control (given as an Orc expression), and are capable of initiating (asynchronous) message exchange. Site objects, on the other hand, are *reactive* objects having internal states but only capable of responding to incoming requests. In order to capture timing behaviors, an additional simple *Clock* object is also included in the configuration.

Messages in an Orc configuration are either site call or site return messages. Within an expression object O, a site call expression $M(\vec{v})$ causes a site call message of the form $M \leftarrow sc(O, \vec{v}, t)$ to be emitted into the object configuration, with t a non-negative value representing the delay of the message; that is, the time it takes for the message to reach the site M. Once the message is received and processed by M, the site may reply back with a site return message $O \leftarrow sr(M, w, t')$, with w the value published by M, and t' the message delay.

4.1.2 Object-based Semantics of Orc

The distributed semantics of an object-based Orc configuration is essentially given by the semantics of the individual expression objects within the configuration, which is precisely the semantics specified by the reduction rewriting semantics of $\mathcal{R}_{Orc}^{red}$, with two exceptions: (i) messages are now managed by the object-based Orc configuration, and (ii) site responses, which were only simulated in the reduction semantics (and the SOS-based semantics), are now generated based on the internal states of site objects, in addition to the site call parameters.

Since Orc expressions and sites are now encapsulated within expression and site objects, respectively, in the object-based configuration, rewrite rules that capture the *instantaneous* actions are no longer global rules that match the entire state, like in $\mathcal{R}_{Orc}^{red}$, but are instead *localized* to individual objects. In effect, the distributed semantics adds a new level of concurrency between objects in a configuration, in addition to concurrency within an Orc expression using parallel composition combinators. In particular, the rewrite rule in $\mathcal{R}_{Orc}$ specifying internal action transitions is of the following form (using object-oriented notation where attributes that are not used in the rule need not be mentioned):

$$\textsc{IAction} : \langle O : \mathsf{Expr} \mid exp : \hat{f} \rangle \rightarrow \langle O : \mathsf{Expr} \mid exp : \mathsf{act}^{\uparrow}(f', i) \rangle \;\text{ if }\; \hat{f} \rightarrow \mathsf{act}^{\uparrow}(f', i)$$

The rule states that an expression object (possibly among other expression objects in the configuration) may perform an internal transition if its active expression is able to perform that transition. As in $\mathcal{R}_{Orc}^{red}$, an expression $\hat{f}$ may perform an internal action using one of the expression-level rules SiteCall, ExprCall, and Publish, given in Section 3.3.1. Equations that reflect the effects of an internal action on the state of an expression object and the configuration are also localized to the relevant expression objects. For instance, the equation that captures the effects of a site call has the following form:

$$\begin{aligned}&\langle O : \mathsf{Expr} \mid exp : \mathsf{act}^{\uparrow}(f, \mathsf{siteCall}(M, \vec{v})), hdl : \eta \rangle \\ &= \langle O : \mathsf{Expr} \mid exp : \mathsf{act}^{\downarrow}(f, ?h), hdl : h, \eta \rangle \;\; M \leftarrow sc(O, \vec{v}, h, t) \quad \text{if} \;\; h := \mathit{fresh}(\eta)\end{aligned}$$

which causes a site call message $M \leftarrow sc(O, \vec{v}, h, t)$ to be emitted into the configuration, with t a (possibly random) time delay. The two equations capturing the effects of expression calls and publishing of values are similarly specified.

When a site call message to a site M becomes ready for consumption (i.e., its transmission delay reaches zero), a site object for M may consume the message according to the following

rule:

$$\textsc{ProcessCall} : \langle O' : \mathsf{Site} \mid name : M, op : ready \rangle \;\; M \leftarrow sc(O, \vec{v}, h, 0) \\ \rightarrow \langle O' : \mathsf{Site} \mid name : M, op : app(\vec{v}, h, O) \rangle$$

where $app(\vec{v}, h, O)$ indicates that the site object is about to process the call from the expression object identified by O. In general, the behavior of a site M in response to a site call may depend on its current state, given by the *state* field, and the call parameters in $app(\vec{v}, h, O)$. Fundamental sites, such as *Rtimer* and *Clock*, and functional sites, like arithmetic *add* and logical *or*, have no internal states, and their behaviors are specified equationally in $\mathcal{R}_{Orc}$. For example, the behavior of the *Rtimer* site object is defined simply by the following equation:

$$\langle O' : \mathsf{Site} \mid name : Rtimer, op : app(t, h, O) \rangle \\ = \;\; O \leftarrow sr(signal, h, t) \;\; \langle O' : \mathsf{Site} \mid name : Rtimer, op : ready \rangle$$

where the response is purposefully delayed to achieve the semantics of *Rtimer*. This is in contrast to, say, a remote counter site *counter* that responds to a *next* message based on the current value of its counter:

$$\langle O' : \mathsf{Site} \mid name : counter, op : app(next, h, O), state : count(k) \rangle \\ = \;\; O \leftarrow sr(k, h, d) \;\; \langle O' : \mathsf{Site} \mid name : counter, op : ready, state : count(k+1) \rangle$$

with d a suitable message transmission delay.

Likewise, the external action of a site return, in which an available site return message of the form $O \leftarrow sr(w, h, 0)$ is consumed by the expression object O, is also specified at the level of expression objects using the following rule:

$$\textsc{SiteReturn} : \langle O : \mathsf{Expr} \mid exp : \bar{f}, hdl : h, \eta \rangle \;\; O \leftarrow sr(w, h, 0) \\ \rightarrow \langle O : \mathsf{Expr} \mid exp : \mathsf{sret}(\bar{f}, w, h), hdl : \eta \rangle \;\; \text{if} \;\; h \in \mathsf{handles}(\bar{f})$$

which consumes an incoming site return message only when its delay is zero. It is important to note that all the auxiliary operators, including $\mathsf{act}^{\uparrow}$, $\mathsf{act}^{\downarrow}$, sret and $\mathsf{handles}$, are defined exactly as before (see Sections 3.1 and 3.3).

The tick rewrite rule, however, must be specified globally at the configuration level to properly propagate the effects of a time tick across all objects and messages in the configuration. This is accomplished by encapsulating the entire object-based configuration using an operator $\{\_\} : \mathsf{Config} \to \mathsf{System}$, and specifying the tick rule as follows (cf. [69]):

$$\textsc{Tick} : \{\langle O : \mathsf{Clock} \mid clk : t\rangle\ \mathcal{C}\} \to \{\langle O : \mathsf{Clock} \mid clk : t + t'\rangle\ \delta(\mathcal{C}, t')\}$$
$$\text{if } \mathsf{eager}(\mathcal{C}) \neq \mathsf{true}\ \wedge\ t' := \mathsf{mte}(\mathcal{C})\ \wedge\ t' \neq 0$$

where $\mathcal{C}$ is the configuration with all the concurrent expression and site objects, and the operators eager and mte are defined similarly as before in Section 3.1.

In summary, the distributed, object-based semantics given by $\mathcal{R}_{Orc}$ generalizes the reduction semantics of $\mathcal{R}^{red}_{Orc}$ to multiple Orc expressions, and provides an explicit treatment of sites and message exchanges between expression and site objects. An implementation of $\mathcal{R}_{Orc}$ in Maude (and Real-Time Maude – RTM) as the module `OO-ORC` has also been developed, and is used as the back-end of the tool MOrc, which we describe in Section 4.2.

4.1.3 Executabilty of $\mathcal{R}_{Orc}$

Like the rewriting logic specifications for the SOS-based rewriting semantics $\mathcal{R}^{sos}_{Orc}$ and the reduction rewriting semantics $\mathcal{R}^{red}_{Orc}$, the specification of the distributed object-based semantics given by $\mathcal{R}_{Orc} = (\Sigma, E \cup A, R, \phi)$ satisfies the executability requirements of rewrite theories. Indeed, the membership equational theory $(\Sigma, E \cup A)$ can be shown operationally A-terminating using the MTT and AProVE, and ground confluent and sort-decreasing using Maude's CRC as before. Moreover, a proof of ground coherence of R with E can be given using a modular argument similar to that in the proof of Lemma 7 in Section 3.3.4.

4.1.4 Object-level Concurrency vs. Orc's Parallel Composition in $\mathcal{R}_{Orc}$

As noted before, $\mathcal{R}_{Orc}$ localizes the rewrite rules capturing the untimed semantics to objects, which implies that behavioral transitions across different objects may be taken concurrently, adding a new dimension of concurrency to Orc's semantics that is orthogonal to Orc's *internal* concurrency using its parallel combinators. A key observation is that object-level concurrency and *symmetric* parallel composition in $\mathcal{R}_{Orc}$ both model independent threads of computation running in parallel. When the Orc expression of an Orc expression object O is a symmetric parallel composition of the form $f \mid g$, the possible behavioral transitions of O are essentially identical to those of two separate objects O_f and O_g whose underlying Orc expressions are f and g, respectively. Intuitively, this is because any enabled internal transition in $f \mid g$ is enabled either in f or in g (the IAction rule), and by assuming that the enclosing configurations of O and of O_f and O_g have the same set of Orc site objects and the same set of pending messages (with appropriate renaming of object references), the object O and the pair of objects O_f and O_g will exhibit identical external and time tick transitions (using SiteReturn, ProcessCall and Tick).

Despite being semantically equivalent, having both concurrency constructs (that is, the Orc operator $\_ \mid \_$ and the configuration multiset union operator $\_\_$) incorporated in $\mathcal{R}_{Orc}$ provides a conceptual advantage for modeling *distributed* concurrent systems. In particular, an Orc symmetric parallel composition may be preferred when modeling *tightly-coupled* multi-threaded computations that typically run on the same node, while concurrency across objects may be preferred when modeling *loosely-coupled* parallel computations that are more often distributed across different, independent nodes. This distinction for modeling distributed systems is exploited when we introduce the distributed implementation of Orc in Chapter 5. Object-level concurrency in $\mathcal{R}_{Orc}$ is also used in Chapter 7 to arrive at simpler models amenable to statistical model checking analysis.

4.2 The MOrc Tool

MOrc is a web-based formal specification and analysis tool for Orc programs based on

RTM and the real-time, object-oriented, rewriting logic specification of Orc, $\mathcal{R}_{Orc}$. MOrc provides a user-friendly interface for specifying the Orc program or expression to be analyzed, any custom sites and their definitions, and the desired formal analysis task and its parameters. The tool supports three kinds of formal analyses: (1) rewriting-based simulation using the timed rewrite command `trew`; (2) untimed and timed breadth-first search using, respectively, the `utsearch` and `tsearch` commands; and (3) untimed and time-bounded model-checking analysis using the `mc` command. The tool is designed to balance both simplicity and expressiveness by supporting user inputs in standard Orc notation, by hiding interactions with RTM, and by providing generic templates for specifying parametric predicates (for both searching and model-checking), using which a wide range of properties can be specified.

4.2.1 Components of MOrc

MOrc is implemented as a dynamic web-based application with both: (1) a front-end client process (implemented using Javascript and the JQuery library) to display appropriate interactive visual elements and manage interactions with the user, and (2) a back-end server (implemented in PHP and C) that pre-processes user input, handles communication with RTM, and post-processes Maude's output. The diagram in Figure 4.1 shows the main components of the tool and illustrates their interactions. MOrc can be accessed online at `http://www.cs.illinois.edu/~alturki/`.

The front-end of MOrc provides an intuitive interface for specifying inputs and displaying analysis results. The main screen, shown in a screen-shot in Figure 4.2, is divided into three main sections: (1) an Orc program/expression and site input section, (2) a preloaded examples section, and (3) a tabbed analysis input/output section. The user may wish to load, and then perhaps edit, one of the examples by clicking on it on the right panel, or he/she may specify an entirely different Orc program/expression, essentially using the mathematical notation of the syntax of Orc defined in Section 2.3.1 (a more precise definition of MOrc-admissible Orc syntax will be discussed in Section 4.2.2 below). Furthermore, custom sites can be defined in the "Custom Sites" panel of the program input area. Currently,

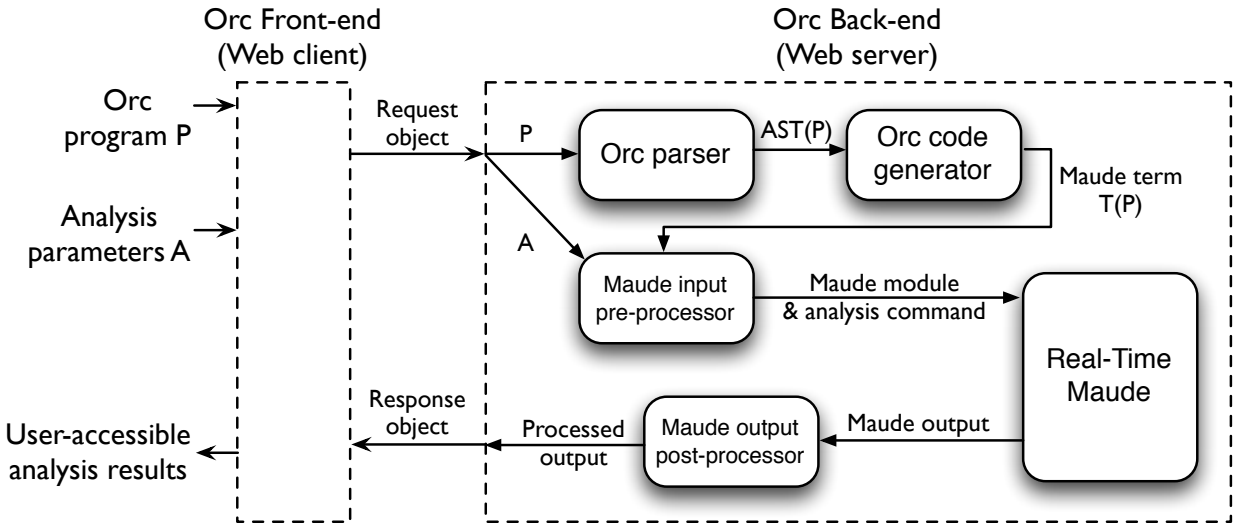


Figure 4.1: The architecture of MOrc

only a limited form of *functional* sites can be defined, which is achieved by specifying the site's name, its messaging interface, and its (possibly time-sensitive) behavior. Finally, the formal analysis section presents a tabbed interface with three tabs corresponding to the three analysis modes supported, namely, simulation, search, and model-checking. Each tab provides a customized panel for specifying analysis parameters for the corresponding analysis mode, such as an expression pattern for searching, or custom atomic predicates for model-checking.

Upon specifying these parameters and pressing an analysis submission button, the front-end constructs a request object (in Javascript Object Notation format, or JSON) encapsulating all relevant input parameters, submits it to the back-end server, and waits for a response. As illustrated in Figure 4.1, the back-end server passes the Orc program text P to a parser and a code generator (both written in C) to build the Maude term $T(P)$ corresponding to the initial state of P. The term $T(P)$, along with the user-supplied analysis parameters, is then fed into a RTM pre-processor that is responsible for generating the appropriate RTM formal analysis command and, for searching and model-checking, a user module that extends the RTM Orc module `ORC` with custom predicates capturing the analysis parameters. After that, the generated module and command are supplied to RTM for execution. The analysis output of RTM is parsed and processed before a server, JSON-encoded object is created and

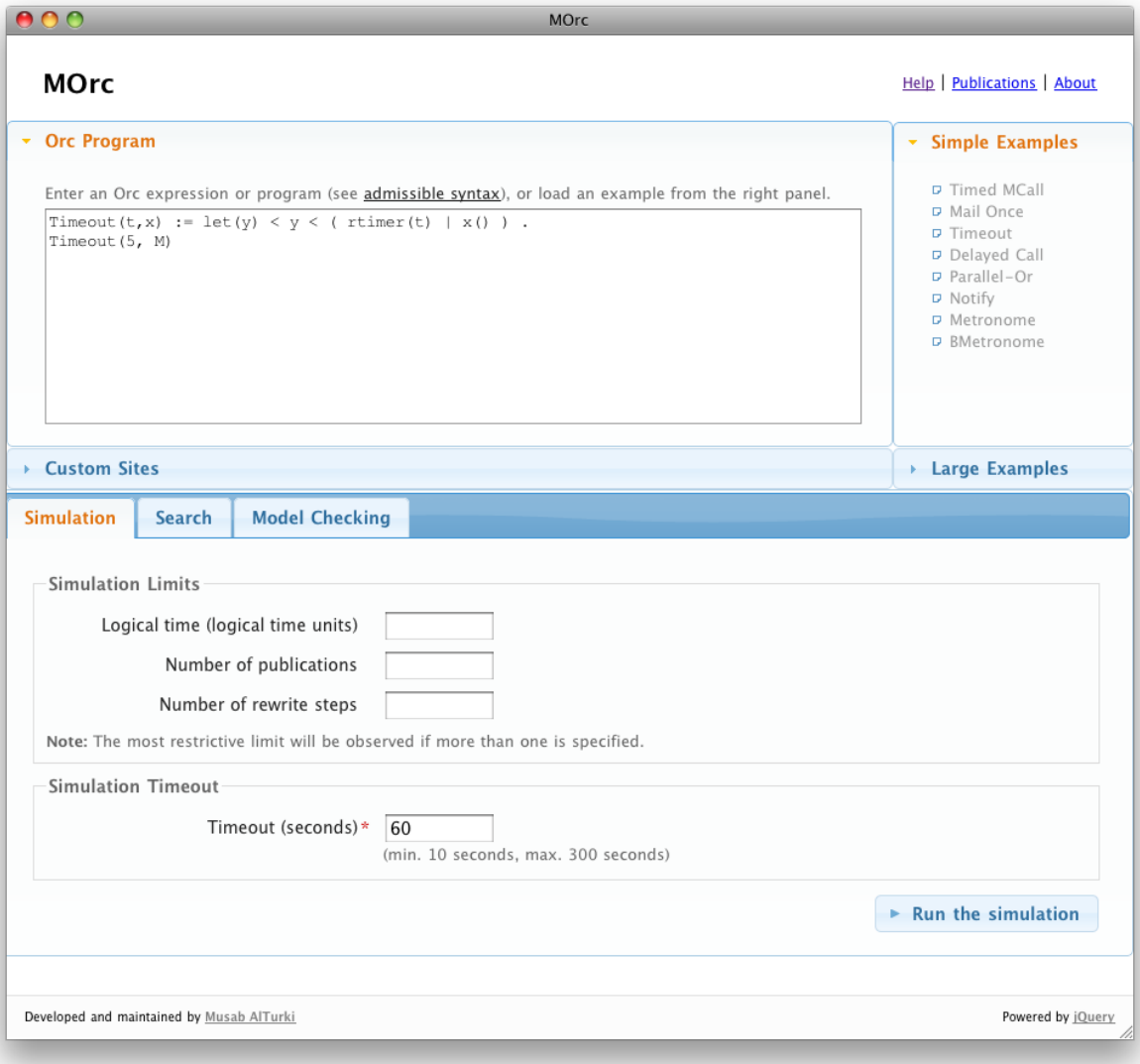


Figure 4.2: The main screen of MOrc's front-end

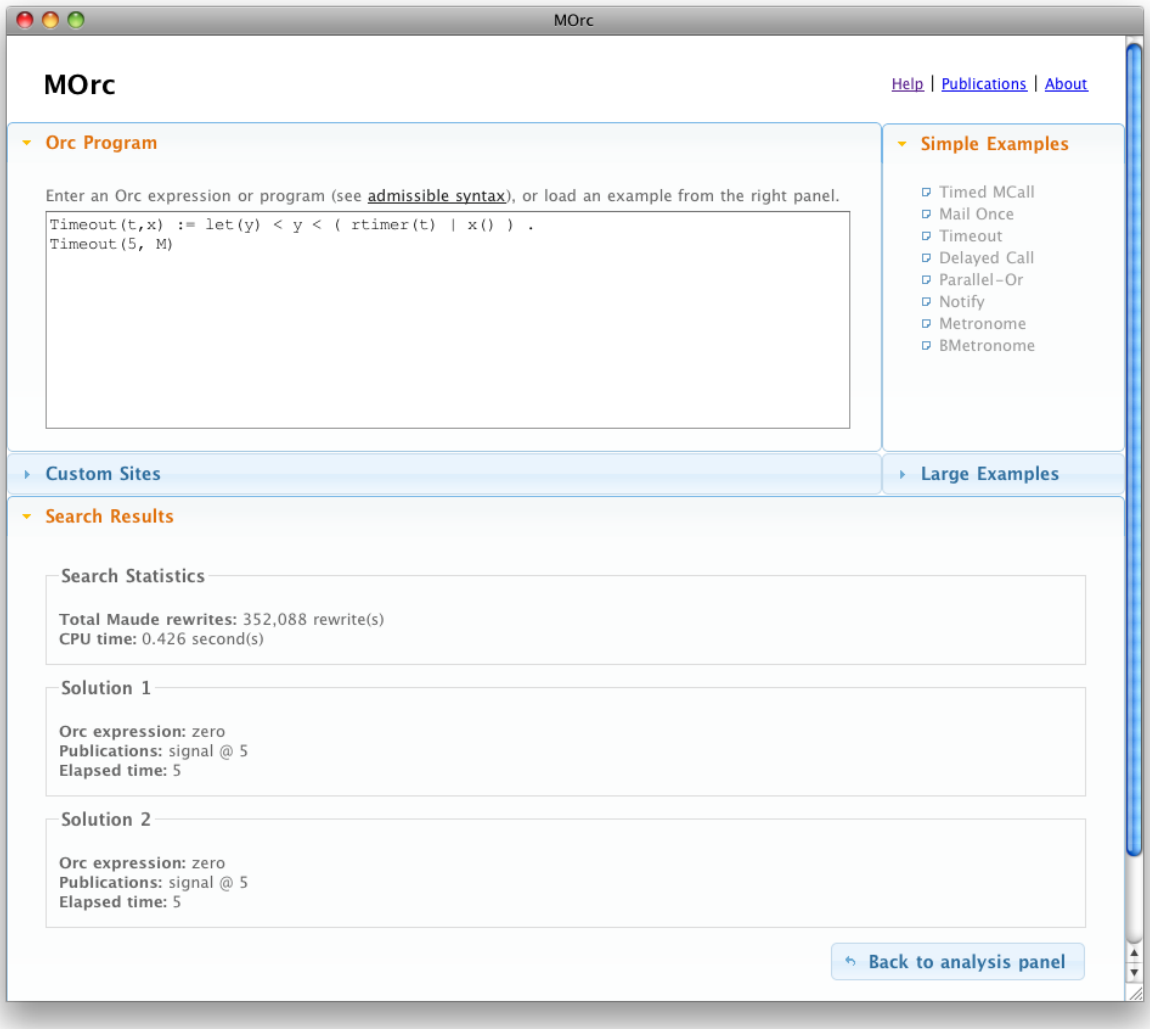


Figure 4.3: The main screen of MOrc showing the results of executing a search command

sent to the client.

When the client receives the server response object, the front-end replaces the analysis panel in the user interface with a results panel that displays in a structured way the analysis results extracted from the response object. For example, Figure 4.3 shows the results of a timed search analysis command. The user may then press the "Back to analysis panel" button to navigate back to the last used analysis panel.

```
<program>          ::=   <declarations> '.' <expr>  |  <expr>
<declarations>     ::=   <declarations> '.' <declaration>  |  <declaration>
<declaration>      ::=   ID '(' ')' ':=' <expr>
                     |   ID '(' <call_expr_list> ')' ':=' <expr>
<expr>             ::=   <expr> '>' ID '>' <expr>   |   <expr> '>>' <expr>
                     |   <expr> '<' ID '<' <expr>   |   <expr> '<<' <expr>
                     |   <expr> '|' <expr>          |   <expr> ';' <expr>
                     |   '(' <expr> ')'             |   <call_expr_list>
                     |   'zero'
<call_expr_list>   ::=   <call_expr_list> ',' <call_expr>   |   <call_expr>
<call_expr>        ::=   ID '(' <call_expr_list> ')'   |   <local_call_expr>
                     |   ID   |   NUMBER   |   STRING
                     |  'signal'   |    'true'   |   'false'
<local_call_expr>  ::=   ID [ NUMBER ]
                     |   <call_expr> <infix_bin_sn> <call_expr>
                     |   <infix_un_sn> <call_expr>
                     |   <prefix_bin_sn> ( <call_expr> , <call_expr> )
<infix_bin_sn>     ::=   '+'   |   '-'   |   '*'   |   '/'   |   '%'
                     |   '=='  |  '>'  |  '>='  |  '<'  |  '<='  |  '!='
                     |   '&&'   |  '||'
<infix_un_sn>      ::=   '!'
<prefix_bin_sn>    ::=   'min'   |   'max'
```

Table 4.1: BNF grammar of Orc syntax accepted by MOrc

4.2.2 Formal Analysis Using MOrc

This section explains and illustrates with examples how MOrc may be used to formally analyze Orc programs. We begin by first defining the syntax of Orc accepted by the tool.

Admissible Orc Syntax

An Orc program can be specified in MOrc according to the BNF grammar shown in Table 4.1, which is essentially an extension of standard Orc syntax, as defined in Section 2.3.1, with built-in site names for standard arithmetic, relational, and logical operators. The grammar also defines syntax for a local site call that returns the i-th element of a k-tuple x, with $0 \leq i < k$ and $k > 0$, using the site call $x[i]$. Identifiers, represented by the token `ID`, are sequences of alphanumeric characters beginning with an alphabetic letter. The tokens `NUMBR` and `STRING` denote, respectively, natural numbers and double-quoted string literals.

Simulating Orc Programs

The simulation analysis panel implements RTM's timed rewrite command `trew`. For simulating Orc programs, one or more limits on the simulation task may optionally be specified, which are particularly useful for simulating non-terminating Orc programs. The possible simulation limits are: (1) a *logical time limit*, which specifies an upper bound on the logical time value of the state of the program; (2) a *publications limit*, which specifies an upper bound on the number of publications allowed before stopping the simulation; and (3) a *rewrite steps limit*, which specifies an upper bound on the number of rewrite steps allowed, where a rewrite step corresponds to a transition in the semantics of Orc (i.e. an application of one of the rewrite rules in the rewriting semantics of Orc: a site call, an expression call, the publishing of a value, a site return, or a time tick transition). If more than one limit is specified, simulation proceeds until one of the limits is reached. As an example, we can load the METRONOME example, whose declaration was given in Section 2.3.2, from the "Examples" panel on MORC's interface:

```
Metronome(t) := let(signal) | rtimer(t) >> Metronome(t) .
Metronome(5)
```

Giving a logical time limit of 20 time units, and a publications limit of 2 causes the simulation to stop once the second publication of `signal` is made at logical time 5, with the resulting Orc expression being of the form: `?h >> Metronome(5)`. Finally, a timeout - in seconds - should be specified to guard against having simulations running forever. For instance, running the simulation command on METRONOME without specifying any simulation limits will cause the tool to display a timeout message.

Reachability Analysis Using Search

The search analysis panel implements the breadth-first search command of RTM, which is either timed (using `tsearch`), or untimed (using `utsearch`). Timed search takes into account explicit time-stamps of states, and allows reasoning about timed properties of the timed transition system of the subject Orc program, such as the number of publications within

`$F[0-9]*`	Expressions	`$E[0-9]*`	Expr. names	`$M[0-9]*`	Site names
`$V[0-9]*`	Values	`$X[0-9]*`	Variables	`$P[0-9]*`	Parameters
`$VL[0-9]*`	Value lists	`$XL[0-9]*`	Variable lists	`$PL[0-9]*`	Param. lists

Table 4.2: Definitions of supported Orc expression pattern meta-variables

the first t time units, the time at which a specific value is published, or whether a pattern is reachable after a given timeout. When timed search is selected, the user may specify the type and value of the search time bound, or leave it unspecified for a timed search with no time limit. Untimed search, on the other hand, ignores time-stamps on states. Therefore, to allow for a *potentially* more efficient analysis, untimed search is performed with respect to a slightly more abstract version of the rewriting semantics of Orc, `ORC-UT`, in which the clock object is not maintained and only an untimed publications trace is recorded in the state, which potentially reduces the reachable state space of a given program. This implies, however, that only untimed properties can be specified and checked, as we will see below.

There are three configurable search parameters. First, an optional upper bound on the number of solutions to be found can be specified. Otherwise, if left unspecified, the tool will try to find all possible solutions. The second parameter, which is required, is the search timeout, which specifies the timeout - in seconds - on the RTM process performing the search. Finally, the third parameter is an option that when checked causes the search to consider solutions corresponding only to terminal (deadlocked) states (i.e. states that cannot be further rewritten). By default, if this option is not specified, the search considers all states reachable by one or more rewrite steps.

To perform a search, the user may optionally provide an *Orc expression pattern* that a solution state must satisfy. If specified, the search command will look for states whose Orc expression components match at the top the given pattern. An Orc expression pattern can be either a *concrete* Orc program specified according to the Orc syntax BNF grammar above in Section 4.2.2, for example `let(x)` and `if(x == y) >> let(true)`, or a *symbolic* pattern containing pattern *meta-variables* ranging over terms of appropriate types. Syntactically, a pattern meta-variable is an identifier of the form `$[A-Z][A-Z]?[0-9]*`, where the prefix dollar sign `$` distinguishes pattern meta-variables from Orc variables, and the first one or two uppercase letters specify the type of the terms over which the pattern meta-variable ranges.

Table 4.2 lists the currently supported meta-variables and their corresponding types. For example, the pattern `$F > $X > $F` matches a sequential composition of two identical sub-expressions, while the pattern `$M($VL) | $E(1, $P)` matches a parallel composition of an enabled site call and an expression call with two parameters, the first of which is the value 1. Pattern meta-variables, which are internally translated into Maude meta-variables of appropriate sorts, enable symbolic reachability analysis without requiring users of MOrc to be familiar with underlying Maude.

In addition to Orc expression patterns, semantic constraints on publications of Orc values can also be specified when using search. First, the reachable state space of an Orc program can be constrained by an upper bound on the number of publications allowed. Furthermore, it is possible to specify timed constraints (for timed search) or untimed constraints (for untimed search) on what values are published (Type I constraints) or the number of values published (Type II constraints) in a state. MOrc provides two generic templates, corresponding to both types, for the specification of such constraints, which are internally translated into state predicates in RTM. For example, a timed Type I constraint may be that the Orc program must have published a signal within the first five time units of its execution, whereas an untimed Type II constraint may require that the program must have published at least two values. The user may specify as many such constraints as desired through MOrc's interface. For a search task, the solution states will have to satisfy all the specified publication constraints (i.e., the conjunction of the corresponding state predicates in RTM).

As an example, consider the Delayed Response example from the list of examples on the right pane, which is specified as:

```
DelayedResponse(x,t) := rtimer(t) >> let(z) < z < x() .
DelayedResponse(clock, 5) | let(signal)
```

Using timed search with a strict upper bound of 5 time units (i.e. $t < 5$), and using the default values for the other search parameters, we may verify the simple property that the site response from `clock` (which is the value 0) is never published before 5 time units have passed. This can be achieved by adding a Type I constraint that looks for states with time-

stamps strictly less than 5 in which the value 0 has been published, and making sure that no solution is found.

Model-checking LTL formulas

The model checking analysis panel implements the linear temporal logic (LTL) model checking command `mc` of RTM. As for search, model checking is either untimed, so that state time-stamps are abstracted away, or time-bounded, (normally) with a given time bound, in which a property is checked in the reachable (timed) state space up to the given time bound. For efficiency of analysis, untimed model checking is based on a version of the rewriting semantics of Orc, namely `ORC-NT`, in which publication traces are *not* kept in the state, which implies that properties that can be verified for this type of model checking analysis cannot refer to what, or how many, publications are made. This is in contrast to time-bounded model checking, for which properties may refer to publications and the times at which they were made. Other parameters include specifying: (1) an optional upper bound on the number of publications (for time-bounded model-checking), which restricts the analysis to the subset of the reachable state space satisfying this bound, and (2) a model checking timeout, which is required, on the RTM process.

The LTL property to be verified in MOrc can be either the generic absence-of-deadlock property, which is a safety property that stipulates that no reachable state is terminal, or deadlocked, or, alternatively, a custom, user-defined formula, typically based on user-defined atomic predicates. When the custom LTL property radio button is pressed, the form fields for specifying the property and any atomic predicates are displayed. MOrc provides three generic templates for specifying named, parametric atomic predicates:

1. *Expression pattern predicate*, which specifies a predicate that is true in a state whose expression component matches at the top the given pattern. The pattern can be specified as described before in Section 4.2.2. Expression pattern predicates can be used in both untimed and time-bounded model checking analyses.

2. *Publication predicate*, which specifies a predicate that is true in a state in which the

```
<formula>  ::=  'True'   |   'False'
             |  AP                              (* An atomic proposition *)
             |  'O' <formula>                   (* Next  *)
             |  <formula> 'U' <formula>         (* Until  *)
             |  '<>' <formula>                  (* Eventually  *)
             |  '[]' <formula>                  (* Always  *)
             |  <formula> 'W' <formula>         (* Weak until  *)
             |  <formula> 'R' <formula>         (* Release  *)
             |  <formula> '/\' <formula>        (* And  *)
             |  <formula> '\/' <formula>        (* Or  *)
             |  '~' <formula>                   (* Not
             |  <formula> '->' <formula>        (* Implies *)
             |  <formula> '<->' <formula>       (* Equivalent *)
             |  <formula> '|->' <formula>       (* Leads to *)
             |  <formula> '=>' <formula>        (* Always implies *)
             |  <formula> '<=>' <formula>       (* Always equivalent *)
```

Table 4.3: BNF grammar of LTL formulas

given value is published within the given time constraints. This predicate is only meaningful for time-bounded model checking.

3. *Publication length predicate*, which specifies a predicate that is true in a state in which at least the given number of publications are made within the given time constraints. This predicate is also only meaningful for time-bounded model checking.

The LTL formula field, which is a required field when specifying a custom formula, allows for specifying an LTL formula that may make use of user-defined atomic predicates according to the BNF grammar shown in Table 4.3.

In addition to the user-defined predicates above, the set of atomic predicates AP currently includes as a predefined predicate (that may also be used in the specification of the LTL formula) the predicate "deadlock", which is true in deadlocked states.

We illustrate how the model-checking interface panel may be used by means of two specifications of the Dining Philosophers problem in Orc, DF1 and DF2, given in the examples panel on the right. We first verify the mutual exclusion property for both specifications, which entails that no two philosophers (processes) are eating (the critical section) at the same time, using untimed model-checking and the custom formula:

```
[] ~ ((eat1 /\ eat2) \/ (eat2 /\ eat3) \/ (eat1 /\ eat3))
```

where `eat1`, `eat2` and `eat3` are user-defined Orc expression pattern predicates that are true in a state when the corresponding philosopher process is about to call the site `eat`, i.e. when the expression component of the state matches the pattern `eat(`i`) >> $F | $F1` for atomic predicate $\texttt{eat}_i$. We may also show that the no-deadlock property holds for DF2 but not for DF1, using untimed model-checking and the predefined property "Absence of Deadlock" in MOrc.

To illustrate time-bounded model checking in MOrc, we use a specification in Orc of Fischer's protocol with two processes, which is a timed, shared-variable-based, synchronization protocol (assuming atomic reads and writes) for controlling multi-process access to a critical section (see [95] for a detailed discussion of the protocol and variations). The specification in Orc, named FP_{NT}, uses two sites: (1) *shared*, which represents the shared (synchronization) variable, which can be set to a positive value, reset to 0, or waited on (for a possibly unbounded amount of time) to be reset again; and (2) *csection*, which represents the critical section and blocks, when called, for a fixed (finite) period of time before responding back (current specification of *csection* responds after 2/3 time units elapses). The mutual exclusion property can be verified in FP_{NT} using untimed model checking of the formula: `[] ~ (cs1 /\ cs2)`, where `cs(`i`)`, for $i \in \{1, 2\}$, are user-defined atomic predicates specified using pattern expressions of the form `csection(`i`) >> $F1 | $F2`. Alternatively, time-bounded model checking (with a reasonable time bound, say 15) may also be used to verify this property up to the given bound. Another important property that is desired in such a protocol is absence of *livelock*, so that some progress is being made by either process in the protocol. This can be verified using untimed (or time-bounded – with a reasonable time bound) model checking and the formula: `[] <> (cs1 \/ cs2)`.

Finally, we briefly describe a third example, illustrating the use of publication predicates, based on the specification of a simplified online auction management application Auction, which manages posting new items for auction, coordinates the bidding process, and announces winners (the reader is referred to [55] for a detailed description of this example). The specification of Auction, which can be loaded from the examples menu on the right,

publishes the pair of values (N, "*bidStarted*") when a bidding process is started for item N, and the pair (N, "*won*") when its auction ends. Also, for this instance of AUCTION, we assume a single seller with an item, labeled 1910, available for auction for 5 time units. A property that is typically required in an auction management system is that an item with at least one bid is eventually sold. This property can be specified using Type I publication predicates, which specify constraints on what values are published, using the values published by the expression. In particular, we first add a Type I atomic predicate, named `bid`, and specify the tuple `(1910, ‘‘bidStarted’’)` as the published value with no time constraints (i.e. time ≥ 0). Similarly, we add a second Type I predicate, "won", with value `(1910, ‘‘bidStarted’’)` and no particular time constraint. Time-bounded model checking with a time bound greater than or equal to 5 of the formula `bid |-> won` verifies that the desired property holds in AUCTION.

CHAPTER 5

DISTRIBUTED IMPLEMENTATION OF ORC

In this chapter, we propose a seamless, semantics-preserving transformation path from a language specification to a distributed language implementation, which substantially narrows the gap between the theoretical level of a distributed language like Orc and an actual implementation. In addition to its formal correctness guarantees, the transformation path enables formally reasoning about programs written in such a language in their implemented form. To demonstrate its effectiveness, the transformation method is applied to Orc to arrive at a distributed implementation of Orc with physical timing. This *semantics-based* transformation builds on the object-based rewriting logic semantics of Orc, given by $\mathcal{R}_{Orc}$ and discussed in Chapter 4, and which we have shown semantically equivalent to Orc's intended semantics.

The key idea behind this transformation methods is that concurrent rewriting is *both* a theoretical model and a practical method of distributed computation. Specifically, in Maude, asynchronous message-passing between distributed objects can be accomplished by concurrent rewriting via sockets. For Orc, the objects are either Orc expressions, which play the role of clients, or sites, which play the role of servers. But since for Orc real time is of the essence, an important issue that must be addressed is how real time is supported in the implementation. Here, the key observation is that Orc programs assume an asynchronous and possibly unreliable distributed environment such as the Internet, and therefore implicitly rely on their *local* notion of time for their computations. As a consequence, time is supported by *local ticker objects*, that interact in a tightly-coupled way with their co-located Orc objects.

Formal verification of Orc programs running in the rewriting-based distributed implementation can be performed *indirectly* by *formally specifying* both Maude sockets, supporting external communication, and the ticker objects, supporting the real-time behavior of Orc

expressions. In this way, both distributed message-passing computation between Orc expression clients and web sites, and time elapse are faithfully *simulated* in the formal specification, in which our desired program can then be model checked. As we explain in this chapter, under reasonable assumptions about the granularity of time chosen for the tickers and the Orc expressions, this simulated formal analysis gives us corresponding guarantees about the actual Orc programs running in the actual distributed Orc implementation.

5.1 Dist-Orc: A Distributed Implementation of Orc

In general, the method of transforming a real-time, object-based rewriting semantics into a real-time distributed implementation consists of three fundamental steps:

1. Defining the distributed structure of the system being specified by specifying locations and a globally unique naming mechanism for objects

2. Specifying the rewriting semantics of the underlying communication model for distributed objects in the system

3. Devising a mechanism for capturing physical, wall clock timing information and extending the rewriting semantics of time to incorporate this information

As explained below, a crucial enabling feature for steps (2) and (3) above is Maude's support for socket-based communication [20]. Through sockets, a Maude process is able to exchange messages with other processes, including other Maude instances, according to the connection-oriented TCP communication protocol.

By applying these transformation steps to the provably correct real-time, object-based rewriting semantics of Orc, $\mathcal{R}_{Orc}$, we obtain in Maude a real-time, distributed implementation of Orc, which we call Dist-Orc, which is deployable on a physically distributed communication network. The transformation entails only minor modifications to $\mathcal{R}_{Orc}$, such as the changes of data representation needed to exchange data through sockets. This considerably increases our confidence in the correctness of the implementation and greatly narrows the gap between implementation and formal analysis.

Below, we discuss in some detail how this method is applied to Orc's rewriting semantics, outline the design and implementation choices in Dist-Orc and explain how they are specified in Maude. The full specification of Dist-Orc is available online at `http://www.cs.illinois.edu/homes/alturki/dist-orc`.

5.1.1 Distributed Orc Configurations

In the distributed implementation, an Orc configuration may span multiple nodes in an interconnected network, and is thus called a *distributed* Orc configuration. Both expression and external site objects in a distributed configuration are identified partly by their *location* (a term of the sort `Loc`), which is defined as a combination of an address (such as a URI or an IP address) and a port number.

```
sort Loc .     op loc : String Nat -> Loc [ctor] .
```

To fully identify expression and external site objects, expression object identifiers, of sort `EOid`, and external site identifiers, of sort `XSOid`, also include a locally unique sequence number.

```
op s : Loc Nat -> XSOid [ctor] .     op e : Loc Nat -> EOid [ctor] .
```

Internal site objects, such as *if* and *rtimer*, are identified simply by their names, since their locations are implicit.

Within a distributed Orc configuration, a *local configuration* (a term of sort `LocalSystem`), or simply a *configuration*, which is a configuration that is located at some node, is managed by an independent instance of Maude. In addition to expression and site objects, each such configuration contains a clock object for maintaining local time and a socket portal for exchanging messages with external objects in other configurations (more on this below).

```
sort LocalSystem .   op [_] : Configuration -> LocalSystem [ctor] .
```

The operator `[_]` encapsulates a local configuration to support managing the local clock and the effects of time elapse (similar to the ideas presented in Real-Time Maude (RTM) [69]).

5.1.2 Sockets and Messaging

In agreement with the Orc theory, the communication model between Orc expressions and sites follows very closely that of the client-server architecture, where Orc expressions are client objects requesting and using services from sites as server objects. In particular, when an expression object `O` within some Orc configuration makes a site call with actual parameters `C` to an external site `N` located at `loc(SR, PT)`, with `SR` and `PT` the node's address and port, a site call message of the form `s(loc(SR, PT), N) <- sc(O, C, H)` is created within the configuration, where `S(loc(SR, PT), N)` is the site object identifier, as described above, and `H` is a temporary handle that uniquely identifies this call. This message then triggers the creation of a client socket to the called site through the following equation:

```
eq s(loc(SR, PT), N) <- sc(O, C, H)
   = < p(O, H) : Proxy | param : C, response : "" >
     createClientTcpSocket(socketManager, p(O, H), SR, PT) .
```

Beside asking Maude's socket manager for a client socket to the site, the rule creates a temporary *proxy* object identified by `p(O, H)`, which manages external communication for this particular site call on behalf of the expression object `O`. The proxy object also serves as a buffer for the site's response, since TCP sockets do not preserve message boundaries in general.

If a client TCP socket to `loc(SR, PT)` is successfully created, Maude introduces the message `createdSocket(OP, socketManager, SC)` targeted to the proxy `OP` into the configuration, which causes the proxy to send the site call message to the external site object through the socket `SC`, as specified by the following rewrite rule (the variable `AS` denotes the rest of the attributes in the object):

```
rl [SendExtCall] :
  < OP : Proxy | param: C, AS > createdSocket(OP, socketManager, SC)
  => < OP : Proxy | param: C, AS > send(SC, OP, (toString(C) + sep)) .
```

where `toString` is a function that properly serializes Orc values into strings that can be transmitted through sockets. The function uses a separator `sep` to distinguish message

boundaries. At the other end, Orc values are built back from such strings using another function `toValue`. Orc value serialization is described in detail in Section 5.1.3 below.

There is also the possibility of an unsuccessful client socket creation attempt due, for example, to an unavailable server or a network failure. In this case, Maude reports the error by issuing the message `socketError(OP, socketManager, S)`, with `S` a string describing the cause of the error. Such an error is a run-time error, which, for simplicity, is considered fatal in DIST-ORC, so that the site call and any subsequent transitions that depend on it will fail.

Once the site call message is sent, the reply `sent(OP, SC)` appears in the configuration and the proxy object waits for a response by introducing a `receive(SC, OP)` message:

```
rl [RecExtResponse] :
  sent(OP, SC) < OP : Proxy | AS > => < OP : Proxy | AS > receive(SC, OP) .
```

When some string `S` is received through the socket, the message `received(OP, OD, S)` appears, and the proxy object stores `S` in its buffer and waits for further input.

```
rl [AccumExtResponse ] :
  < OP : Proxy | param: C, response: S' > received(OP, SC, S)
  => < OP : Proxy | param: C, response: S' + S > receive(SC, OP) .
```

The proxy will keep waiting for and accumulating input through the socket until it is remotely closed by the site, when the reply `closedSocket` appears. At this point, the site response is reconstructed and handed in to its expression object:

```
rl [ProcessExtResponse] :
  < p(O,H) : Proxy | response: S' , AS > closedSocket(p(O,H), SC, S)
  => O <- sr(toValue(S'), H) .
```

On the server side, when a site is first initialized, it creates a server TCP socket, through which it keeps listening for incoming connections.

```
eq [InitializeSite] :
```

```
  < s(loc(SR, PT), N) : Site | op : free, status : idle, AS >
  = < s(loc(SR, PT), N) : Site | op : free, status : initializing, AS >
    createServerTcpSocket(socketManager, s(loc(SR, PT), N), PT, 10) .

rl [CreatedServerSocket] :
  createdSocket(xOS, socketManager, LISTENER)
  < xOS : Site | op : free, status : initializing , AS >
  => < xOS : Site | op : free, status : active , AS >
     acceptClient(LISTENER, xOS) .
```

Once a client has been connected with a socket `CLIENT`, the site becomes ready for an incoming site call through `CLIENT`, while listening for other potential clients:

```
rl [AcceptedClient] :
  acceptedClient(xOS, LISTENER, IP, CLIENT)
  < xOS : Site | op : free, status : active, AS >
  => < xOS : Site | op : free, status : active , AS > receive(CLIENT, xOS)
     acceptClient(LISTENER, xOS) .
```

The site then accumulates the request from `CLIENT` (which contains the actual parameters for the site call):

```
crl [AccumulateRequest] :
  < xOS : Site | op : free , buffer : S, AS > received(xOS, CLIENT, S')
  => < xOS : Site | op : free , buffer : S + S', AS > receive(CLIENT, xOS)
        if find(S + S', sep, 0) == notFound .
```

The rule above buffers the serialized request by checking whether the message boundary indicator, given by `sep`, has been received. Once the message is received in its entirety, the following rule fires:

```
crl [PrepareReply] :
   received(xOS, CLIENT, S') < xOS : Site | op : free , buffer : S, AS >
   => < xOS : Site | op : exec(toValue(substr(S'', 0, length(S'')
```

```
                                    + (- length(sep)))), CLIENT),
          buffer : "", AS >
        if S'' := S + S' /\ n := find(S'', sep, 0) .
```

This rule causes the site to process the call using the function `exec(...)`, whose definition is site-dependent. When appropriate, the site might publish a value as a result of this site call, which then causes a response to be sent back to the client:

```
eq CLIENT <- sr(c, xOS, 0) < xOS : Site | AS >
  = < xOS : Site | AS > send(CLIENT, xOS, toString(c)) .
```

Once the response is sent, the site closes the socket:

```
rl [ReplySent] :
  sent(xOS, CLIENT) < xOS : Site | AS >
  => < xOS : Site | AS > closeSocket(CLIENT, xOS) .


rl [ClosedClientSocket] :
  closedSocket(xOS, CLIENT, S) < xOS : Site | AS >
  => < xOS : Site | AS > .
```

Note that, when a site remains silent, no response is generated, and the client blocks waiting for a response through the other end of the open socket `CLIENT`.

It is worth noting here that, just like in any other physically distributed communication mechanism, messaging through sockets is inherently prone to various potential communication problems. In addition to the socket creation errors mentioned above, these include dropped connections, lossy channels and unpredictably long delays. In Dist-Orc, such problems are dynamic errors that might be exposed while executing a distributed Orc program, and typically cause the Orc objects in which they appear to fail.

5.1.3 Serialization of Orc Values

Local Orc configurations communicate with each other by external site calls and returns, which involve exchanging Orc values either as actual parameters to calls or as return values.

Since this communication takes place through Maude's TCP sockets, Orc values need to be serialized as transferable strings just before they are transmitted, and then reconstructed back to Orc values just after they are received. Similarly to how this issue was previously approached in Maude [96, 97], we utilize Maude's meta-level capabilities, provided by the module `META-LEVEL`, to produce proper serialization and reconstruction procedures for Orc values. More specifically, we define a module `VALUE-STRING-CONVERSION` that imports both the `PARAMETER` module, containing declarations of Orc value sorts and operators, and the `META-LEVEL` module.

```
mod VALUE-STRING-CONVERSION is
  inc PARAMETER .
  pr META-LEVEL .
  ...
endm
```

Within this module, two main operators are defined: `toString` and `toValue`, which were briefly introduced above. The operator `toString` defines a partial function that attempts to convert an Orc value or a list `C` of values (a term of sort `ValueList`) into a string `S` in Maude.

```
op toString  : ValueList ~> String .
eq toString(c(S)) = S .
eq toString(C) = qidListString(metaPrettyPrint(
                  upModule('PARAMETER, false), upTerm(C), none)) [owise] .
```

Encapsulated strings of the form `c(S)` are easily converted into strings by using their underlying string values, while conversion of any other value or list of values is performed by building up a string out of their meta-representations. The operator `qidListString` creates a string out of the list of quoted identifiers returned by the meta-level operator `metaPrettyPrint` representing the different values in `C` (see [20] for a detailed description of the meta-level operators).

The dual operator is `toValue`, which is defined in `VALUE-STRING-CONVERSION` and is partially shown below.

```
 op toValue : String -> Value .
ceq toValue(S) = downTerm(getTerm(metaParse(
     upModule('PARAMETER, false), stringQidList(S), 'Value)), error(S))
     if substr(S, 0, 6) == "signal"   ---- signal value
     or substr(S, 0, 4) == "b `("     ---- bool
     or substr(S, 0, 4) == "l `("     ---- tuple
     or substr(S, 0, 4) == "s `("     ---- site object id
     or substr(S, 0, 3) == "let"      ---- `let' site id
     ...
     or substr(S, 0, 7) == "_`,_ `(" .  ---- ValueList
 eq toValue(S) = c(S) [owise] .       ---- arbitrary string
```

If the input string `S` to the function `toValue` matches the meta-representation of a known Orc value (or list of values), which is decided by examining an appropriate prefix of `S`, the first equation applies and the conversion process is carried out by Maude's meta-level operators. Otherwise, the string is considered an encapsulated Orc string value. The operator `stringQidList` converts a string built by `qidListString` back into a list of quoted identifiers to be processed by the meta-level (see [56] for more details).

5.1.4 Timed Behavior

Orc is a timed theory. Therefore, a faithful implementation of Orc requires capturing its timed behaviors. The notion of time in a language implementation is typically captured by a clock against which events in a program in that language may take place. There are several different ways in which clocks can be used to maintain timing information. For our distributed implementation, however, a number of requirements influence the design choices we have made. First, Orc's communication model is asynchronous. This suggests the use of *distributed clocks* (as opposed to a centralized clock), where each node in the distributed configuration maintains its local clock. Indeed, the distributed clocks architecture emphasizes Orc's philosophy of having the communicating expressions and sites as loosely coupled as possible. Furthermore, for all the applications we have so far specified in Orc, distributed

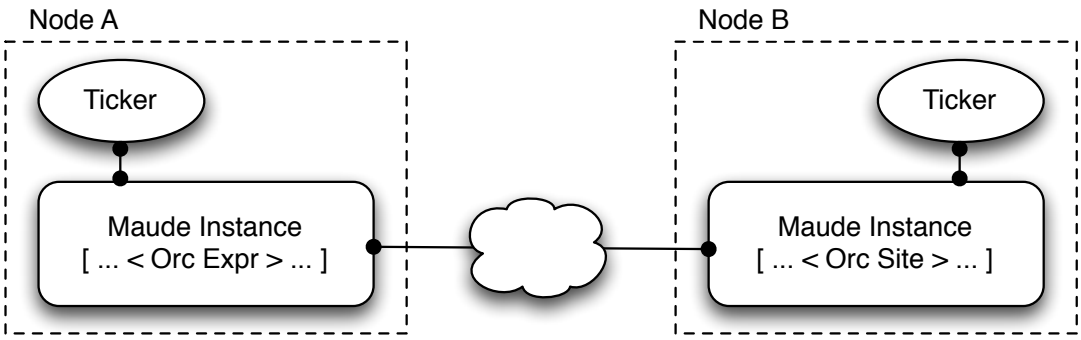
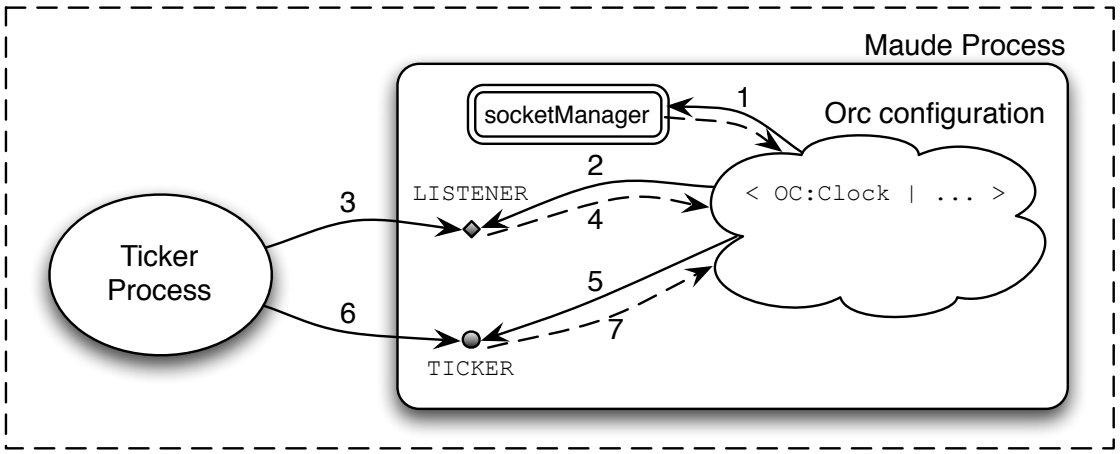


Figure 5.1: A schematic diagram illustrating the general structure of a distributed Orc configuration. Dashed rectangles represent node boundaries, solid rounded rectangles represent local configurations, and darkened circles represent endpoints of TCP sockets.

clock synchronization is not required for preserving program correctness. This is primarily due to the fact that in most of the applications clocking information is used either locally (for example with the local *rtimer* site) or to time incoming responses. This greatly simplifies the implementation, since no clock synchronization mechanism, such as Lamport counters [98] or vector clocks [99, 100], is needed. Finally, since the implementation supports communication using sockets with external objects, which is inherently unpredictable given the possible transmission delays and network failures, any design of a clocking mechanism that depends on external communication with expressions or sites would also be unpredictable and unreliable.

Therefore, in Dist-Orc, for each node in the distributed configuration, the local clock is managed by an independent and local *ticker* object with access to the node's real-time system clock. Since in Maude there is currently no direct support for accessing the system clock, we employ sockets as a means of transmitting clock time ticks to Maude. It is important to note here that, although we use sockets to implement it, the ticker object is *local* to its corresponding Orc configuration and is thus guaranteed to provide fairly accurate clocking information. Figure 5.1 illustrates schematically the deployment architecture of a distributed Orc configuration with timing.

The diagram in Figure 5.2 outlines the steps involved in initializing a connection with the co-located ticker object and receiving the first clock tick. Upon initialization, the clock object within an Orc configuration requests a server socket, by issuing the message



1: `createServerTcpSocket(socketManager, OC, PORT, ...)`
- creates the server socket `LISTENER`,
- elicits `createdSocket(OC, socketManager, LISTENER)`

2: `acceptClient(LISTENER, OC)`

3: Ticker requests a connection through `LISTENER`

4: `acceptedClient(OC, LISTENER, IP, TICKER)`
- creates the client socket `TICKER`

5: `receive(TICKER, OC)`

6: Ticker sends a tick string `S` to `TICKER`

7: `received(OC, TICKER, S)`

Figure 5.2: The steps involved in establishing a connection with the ticker object and receiving clock ticks.

`createServerTcpSocket(...)`, to be used for listening for a connection from the local ticker process, which is a Java process that is run in every node of a distributed configuration. The ticker process uses the built-in Java classes `Timer` and `Socket` to generate and send a tick message every t milliseconds to its corresponding Maude process, where t is a positive integer value. The clock object waits for a connection as soon as the server socket is created:

```
rl [InitClockSocket1] :
  < OC : Clock | AS > createdSocket(OC, socketManager, LISTENER)
  => < OC : Clock | AS > acceptClient(LISTENER, OC) .
```

where `LISTENER` is the newly created clock server socket. Once the ticker object is connected, the message `acceptedClient(OC, LISTENER, IP, TICKER)` appears, with `IP` the originating address of the ticker object, and `TICKER` the newly created client socket for communicating with the ticker object. This causes the clock object to become ready for incoming clock ticks according to the following rule:

```
rl [InitClockSocket2] :
```

```
  acceptedClient(OC, LISTENER, IP, TICKER) < OC : Clock | AS >
   => < OC : Clock | AS > receive(TICKER, OC) .
```

Upon receiving a clock tick, the clock object updates its clock and reflects the effect of time elapse on the rest of the Orc configuration using the time-updating function `delta`, which decrements the relative time delays in pending messages.

```
crl [tick] :
  [received(OC, TICKER, S) < OC : Clock | clk: c(N) > CF]
   => [< OC : Clock | clk: c(N + 1) > receive(TICKER, OC) delta(CF)]
       if find(S, "#", 0) =/= notFound .
```

The variable `CF` denotes the rest of the local configuration. Recall that the operator `[ CF ]` encapsulates the local configuration `CF`. The equational condition in the above tick rule checks whether the tick message has been fully received, as buffering is (again) required to ensure proper message transmission (which is specified by a different rule similar to the rule `[AccumulateRequest]` shown in Section 5.1.2 for sites). The process of receiving and processing time tick messages keeps repeating as long as the Ticker object is supplying those ticks through the clock socket.

An important observation is that the use of physical, wall clock time in Dist-Orc to time Orc transitions eliminates the possibility of Zeno behaviors, which are a well-known artifact of logical time. This implies that for the intended semantics to be preserved, and hence for the correctness of the analysis later in Section 5.3, the transitions internal to an Orc configuration must be completed before the next real-time clock tick arrives. In other words, a single clock tick should be long enough to accommodate the instantaneous transitions of an Orc configuration. The minimum length of a clock tick so that this property is satisfied is specific to the Orc application and the machines used to run it. For example, for the distributed auction case study below, and using a 2.0GHz dual-core node with 4GB of memory, the clock tick can be made as short as 0.2 seconds. In general, deciding on a minimum size for a clock tick given an application is hard to anticipate and is typically accomplished through experimentation.

$$Posting(seller) =_{def} seller(\text{“postNext”}) > x > Auction(\text{“post”}, x) \gg rtimer(1) \gg Posting(seller)$$

$$Bidding =_{def} Auction(\text{“getNext”}) > (id, d, m) > Bids(id, d, m, 0) > (wn, wb) > (\ if(wn = 0) \gg Bidding() \mid if(wn \neq 0) \gg Auction(\text{“won”}, wn, id, wb) \gg Bidding() \ )$$

$$Bids(id, d, wb, wn) =_{def} (\ if(d \leq 0) \gg let(wb, wn) \mid if(d > 0) \gg clock() > t_a > min(d, 1) > t > TimeoutRound(id, wb, t) > x > (\ if(x = signal) \gg Bids(id, d - t, wb, wn) \mid if(x \neq signal) \gg rtimer(1) \gg clock() > t_b > Bids(id, d - (t_b - t_a), x_0, x_1) \ ) \ )$$

$$TimeoutRound(id, bid, t) =_{def} let(x) < x < (\ rtimer(t) \ \mid \ Bidders(\text{“nextBidList”}, id, bid) > bl > MaxBid(bl) \ )$$

Figure 5.3: Orc expressions in the AUCTION program

5.2 Case Study: A Distributed Implementation of AUCTION

To illustrate DIST-ORC, we describe a distributed implementation `Dist-Auction` of the online auction management application in Orc, AUCTION, which was first introduced in [55], and whose expression declarations are also shown in Figure 5.3 for reference. The distributed configuration of the auction application contains two expression configurations: one with the *Posting* expression object, which is responsible for retrieving and posting items for sale by a given seller, and the other contains the *Bidding* expression object for managing the bidding process. For instance, the initial local configuration for the *Posting* expression object has the form:

```
[ <>
  < C : Clock | clk : c(0) >
  createServerTcpSocket(socketManager, C, 54200, 10)
  < e(loc("10.0.0.2", 44200), 0) : Expr |
    env: Posting s := s("postNext") > x > AUCTIONID("post",x) >> rtimer(1) >>
                      Posting(s),
    exp: Posting(SELLERID), ... >  ... ]
```

where `SELLERID` and `AUCTIONID` are object identifiers for the *Seller* and *Auction* sites, respectively. The *Posting* expression declaration is stored in the environment attribute `env` of

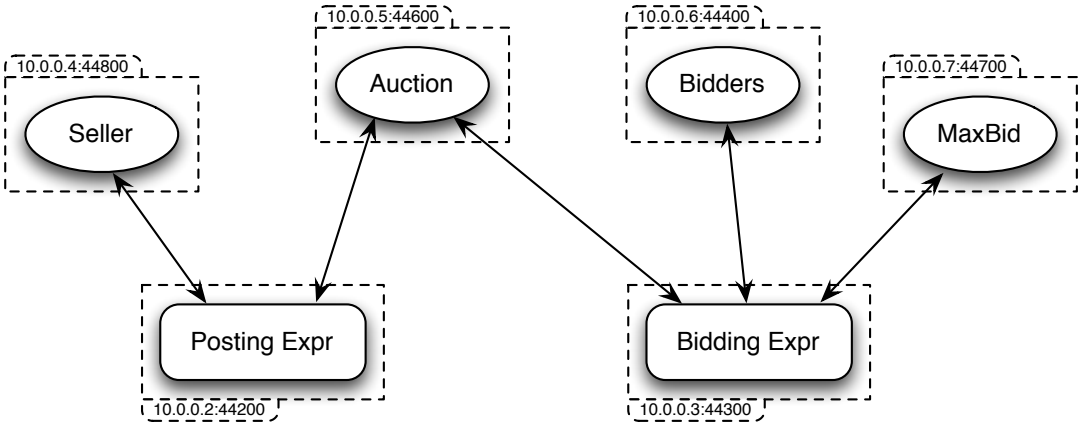


Figure 5.4: The deployment architecture of the `Dist-Auction` Orc program

the expression object, while the attribute `exp` keeps the actual expression to be evaluated. The configuration also includes objects for internal (fundamental) sites, such as *if* and *let*, which are omitted here for brevity.

In addition to the *Posting* and *Bidding* expression configurations, there are four site object configurations in the distributed configuration of `Dist-Auction`, one configuration for each of the sites assumed by AUCTION, namely *Seller*, *Bidders*, *MaxBid*, and *Auction*. For example, the initial local configuration for a *Seller* site with two items for auction (identified by numbers `1910` and `1720`) may have the form:

```
[ <>
  < C : Clock | clk : c(0) >
  createServerTcpSocket(socketManager, C, 54800, 10)
  < s(loc("10.0.0.4", 44800), 0) : Site |
      name : 'seller, state : (item(1910, 5, 500), item(1720, 7, 700)) , ... >
  createServerTcpSocket(socketManager, s(loc("10.0.0.4", 44800), 0), 44800, 10)]
```

We note that the site attempts to create two server sockets: one for listening to expression object requests and the other for listening to the local ticker object.

Each local configuration in `Dist-Auction` may run on a different node in a communication network. The diagram in Figure 5.4 depicts a physical deployment of `Dist-Auction`, with bidirectional arrows representing communication patterns. A physical deployment can be

conveniently achieved using an appropriate shell script to run Maude, load the DIST-ORC module and `Dist-Auction`, and execute the external rewrite command `erew`. For example, with `initAuction`, an operator that creates the initial state of the Auction site configuration, the following command executes the Auction site:

```
echo "erew initAuction ." | maude orc-distributed.maude auction-manager.maude
```

with the following sample output, generated by the `print` attribute of Maude statements (with auction items `1910` and `1720`):

```
00 erewrite in DIST-AUCTION : initAuction .
01 Site "10.0.0.5":44600 0 initializing... Site is ready.
02 Clock server socket created.
03 Awaiting connection from ticker ...
04 Ticker connected.
05 Received "post"
06 Item 1910 posted
07 Received "getNext"
08 Bidding to start for 1910
09 Tick! ...  (5 time ticks)
10 Received "won"
11 Item 1910 won by Bidder 3
12 Received "getNext"
13 Tick!
14 Received "post"
15 Item 1720 posted
16 Bidding to start for 1720
17 Tick! ...  (6 time ticks)
18 Received "won"
19 Item 1720 won by Bidder 3
20 Received "getNext"
21 ...
```

In this particular run, the auction site receives a `post` request from the *Posting* expression object and posts item `1910`. Meanwhile, a request for the next item to be auctioned is received from the *Bidding* expression object. The auction site then publishes the item details back to the *Bidding* expression, which takes care of orchestrating the bidding process for this item. After five time units (the duration of the auction for item `1910`), Bidder 3 is announced as the winner and a similar process is repeated for the second item `1720`.

5.3 Formal Analysis of Distributed Orc Programs

The real-time, distributed implementation of Orc described above is very useful in prototyping and deploying Orc programs on physically distributed nodes in an interconnected network. As we saw in Section 5.2, the implementation enables observing actual possible

behaviors in practical environments, in which the effects of physical limitations of communication networks are taken into account.

However, the implementation technique as outlined above does not result in a language specification that is immediately amenable to more rigorous formal analysis such as reachability analysis and model-checking. This is fundamentally due to the fact that the implementation technique makes use of facilities that are outside the scope of the Maude formal analysis tools. In particular, there are two fundamental facilities in the implementation that complicate formal analysis: TCP sockets and the ticker objects. While support for sockets is built into Maude, sockets do not have a direct and immediate logical representation that can be subjected to formal analysis. Furthermore, the ticker objects, being written in another general-purpose language with access to the system's real, wall-clock time, introduce yet another obstacle in achieving a formally analyzable specification.

Our solution to this problem, which we explain in some detail in the rest of this section, is to develop rewriting logic specifications for these facilities, so that the distributed implementation can be turned, with minimal effort, into a formally analyzable specification in Maude. In particular, both Maude sockets and externally defined configuration clocks must be formally modeled at the object-level. This is discussed next.

5.3.1 Formal Specification of TCP Sockets

Maude's TCP sockets can be formally specified by defining abstractions of Maude instances, sockets, and their behaviors. We develop a rewriting specification $\mathcal{R}_{Socket}$ of sockets, which is based on previous work on Mobile Maude [97, 20] and algorithmic skeletons in Maude [96]. The specification models sockets as a rewrite theory $\mathcal{R}_{Socket}$, in which Maude instances are abstracted with objects of the class *Process*, and server and client sockets as objects of *ServerSocket* and *Socket* classes, respectively. Abstract processes and sockets in $\mathcal{R}_{Socket}$ introduce a higher layer of abstraction in which socket objects mediate communication between processes, which encapsulate local Orc configurations.

More specifically, a process object has the form $\langle PID : Process \mid sys : S \rangle$, with PID an object identifier and S an encapsulated local configuration. A client socket object of the

form

$$\langle SID : Socket \mid endpoints : [PID_1, PID_2] \rangle$$

abstracts a bidirectional client TCP socket set up between processes PID_1 and PID_2 (where $[PID_1, PID_2]$ is an unordered pair), while a server socket has the form:

$$\langle SID : ServerSocket \mid address : A, port : N, backlog : K \rangle$$

where K is a positive integer specifying the maximum allowed number of queue requests. While client sockets are created and destroyed during the course of execution of the different configurations, server sockets are created for server objects using a *Manager object*, which abstracts Maude's socket manager (which we first encountered in Section 5.1.2). The manager object itself is a simple object maintaining a counter for creation of fresh socket object identifiers.

To maximize specification modularity and reusability, socket objects in $\mathcal{R}_{Socket}$ have interfaces (i.e. message formats) that are almost identical to those of Maude sockets. This minimizes the need for making any changes in the distributed implementation of Orc when switching between Maude sockets and their abstractions given by $\mathcal{R}_{Socket}$.

Different useful abstractions of socket behaviors can be defined. For the formal model of Dist-Orc, we choose an abstraction level that captures most interesting behaviors and yet can be efficiently analyzed. The abstraction essentially considers potential client socket creation errors and a somewhat limited form of communication delays and failures. This design choice abstracts away uninteresting behaviors, such as server socket creation problems, and approximates actual messaging problems, such as unavailable or unreachable servers, and unreliable networks. The main features of $\mathcal{R}_{Socket}$ are explained below.

Server socket creation is straightforward, and is modeled with the following rewrite rule:

```
rl [CreateServerTcpSocket] :
  < PID : Process | sys : [createServerTcpSocket(socketManager, 0, PT) CF] >
  < socketManager : Manager | counter : N >
  => < PID : Process | sys : [createdSocket(0, socketManager, server(N)) CF] >
```

```
    < socketManager : Manager | counter : (N + 1) >
    < server(N) : ServerSocket | address : "localhost", port : PT > .
```

This rule creates a server socket object, identified by `server(N)`, with an arbitrary address and a given port, and transforms the socket creation request message into an appropriate response.

When an Orc object within a process attempts to create a client socket to a server by issuing the message `createClientTcpSocket (socketManager, O', SR, PT)`, two different transitions are possible, depending on whether the client socket creation is successful or not. The success case is modeled by the following rule:

```
rl [CreateClientSocketSuccess] :
< PID : Process | sys : [acceptClient(server(N), O) CF] >
< PID' : Process | sys : [createClientTcpSocket(socketManager, O', SR, PT) CF'] >
< socketManager : Manager | counter : M >
< server(N) : ServerSocket | address : SR, port : PT >
=> < PID : Process | sys : [acceptedClient(O, server(N), SR, socket(M)) CF] >
   < PID' : Process | sys : [createdSocket(O', socketManager, socket(M)) CF'] >
   < socketManager : Manager | counter : (M + 1) >
   < server(N) : ServerSocket | address : SR, port : PT >
   < socket(M) : Socket | endpoints : [PID : PID'] > .
```

In this rule, the server is in a state accepting incoming connections from clients, specified by matching a server at address and port `SR:PT` that is accepting connections using the message `acceptClient(...)`. The rule also creates a socket object `socket(M)` that will mediate communication between the client and the server.

Client socket creation may also fail, representing situations where the server is unreachable or unavailable. This case is modeled by a similar rule, labeled `[CreateClientSocketFail]`, with the same starting state as the rule above but with a different resulting state, where now the client process gets the `socketError (O', socketManager, "")` message from the socket manager, and no new socket object is created.

Once a socket is successfully created, a connection through this socket is established, and bidirectional message exchanges may take place using `send(...)` and `receive(...)` messages. The following rule specifies message exchange between two processes:

```
crl [exchange] :
 < PID : Process | sys : [send(SOCKET, O, C) CF] >
 < PID' : Process | sys : [receive(SOCKET, O') CF'] >
 < SOCKET : Socket | endpoints : [PID : PID'] >
 < DID : Delays | ds : DS >
 => < PID : Process | sys : [sent(O, SOCKET) CF] >
    < PID' : Process | sys : [received(O', SOCKET, C, R) CF'] >
    < SOCKET : Socket | endpoints : [PID : PID'] >
    < DID : Delays | ds : DS >
 if DS' R DS'' := DS .
```

The `send(...)` and `receive(...)` messages are respectively transformed into a `sent(O, SOCKET)` message, acknowledging the send action to the sender, and a `received(O', SOCKET, C, R)` message, signaling *delayed* delivery of the sent message to the receiver *in* `R` *time units*. That is, the value(s) sent, `C`, are *delayed* by some amount of time `R` and will only be available to the receiver object after `R` time units have elapsed. The delay value for any transmitted message is non-deterministically extracted using a matching equation in the condition from a non-empty, set of delays `DS` maintained by a special object `< DID : Delays | ds : DS >` in the global configuration. To maintain feasibility of exhaustive formal analysis techniques, the set `DS` should obviously be finite. In fact, for most reasonably sized distributed Orc programs, the delay set should have a fairly small size. An appropriate delay set for a given distributed Orc application can be specified as part of its initial state using the `Delays` object. It is worth noting here that different behaviors may result by giving different delay sets. Two special cases of interest are: (1) $DS = \{0\}$, in which case messages are assumed to experience no delays, and (2) $\infty \in DS$, which represents the case of a lossy communication channel. As we will see in Section 5.3.2 below, the real-time semantics of the model will eventually make such delayed messages available to the receiver for processing.

Finally, closing a socket is straightforwardly modeled by the following equation:

```
eq [close] :
  < PID : Process | sys : [closeSocket(SOCKET, O) CF] >
  < PID' : Process | sys : [receive(SOCKET, O') CF'] >
  < SOCKET : Socket | endpoints : [PID : PID'] >
  = < PID : Process | sys : [closedSocket(O, socketManager, "") CF] >
    < PID' : Process | sys : [closedSocket(O', socketManager, "") CF'] > .
```

The equation drops the closed socket, and issues the `closedSocket(...)` message to its endpoints.

5.3.2 Global Logical Time

As before, time and its effects on the distributed Orc configuration are formally specified using the standard and general technique of capturing logical time in real-time rewrite theories [64], and facilitated by RTM [74]. Essentially, the time domain is represented by a sort TimeInf (time with infinity), and a *global tick* rewrite rule is used to synchronously advance time and propagate its effects across the *encapsulated global configuration*, a term of the sort GlobalSystem, of the form $\{\mathcal{C}\}$, where $\mathcal{C}$ is the Orc configuration consisting of all process and socket objects. configuration at a given point in time. Furthermore, the tick rule, which plays the role of the ticker objects in the distributed implementation, is defined globally as follows (with `R'` a variable ranging over the positive rational numbers):

```
crl [tick] :
  {CF} => {delta(CF, R')} in time R'
    if eager({CF}) =/= true /\ R' <= mte(CF) [nonexec] .
```

The tick rule computes on the global Orc configuration the function `delta`, which advances time for all local clock objects and updates time delays in all site calls and returns present in the configuration. For example, clocks and delayed external messages are updated, respectively, by the following two equations (`plus` and `monus` define addition and subtraction on time domains):

```
eq delta(< O : Clock | clk : c(R) > CF, R')
   = < O : Clock | clk : c(R plus R') > delta(CF, R') .
eq delta(received(O, O', C, R) CF, R')
   = received(O, O', C, R monus R') delta(CF, R') .
```

The tick rule above is not immediately executable (which is indicated by the `[nonexec]` attribute), as it introduces a new variable `R'` representing the amount of time elapse on its right hand side. A strategy for sampling time needs to be specified for the rule to be executable. As before, we assume a general *maximal* strategy that in each tick advances time by the *maximum time elapse*, which is defined by the function `mte` as the minimum delay across all site call messages and returns in the global Orc configuration. The combination of the maximal time sampling strategy and the condition `R' <= mte(Conf)` in the tick rule ensures that time is advanced as much as possible in every tick but only enough to be able to capture all events of interest.

To properly capture the synchronous semantics of Orc [52, 53], the tick rule is also made conditional on the fact that no other behavioral (instantaneous) transition is possible. This imposes a precedence of rule application, where time ticks have the lowest priority among all transitions. This is precisely captured by the `eager` predicate in the `tick` rule's condition. As was discussed before in Chapter 3, the condition is necessary to precisely capture the intended semantics of the Orc theory.

It is important to note that the abstraction of time and how it affects the global Orc configuration as specified by the tick rule is consistent with the real-time distributed implementation Dist-Orc in that, in Dist-Orc, we assumed that the granularity of a single time tick in real-time is always large enough for instantaneous transitions within a configuration to complete. Furthermore, the tick rule *synchronously* updates all clock objects in all processes. This also defines yet another abstraction over Dist-Orc, where individual clocks are not necessarily synchronized. However, since clock synchronization is not required for Dist-Orc, as was discussed in Section 5.1.4, the abstraction considers only those behaviors in Dist-Orc that make sense under these assumptions about time.

5.3.3 Further Abstractions For Performance

Unlike the formal specifications of sockets and time described above, the abstractions outlined below are not essential for formal reasoning about distributed Orc programs. They describe further optional abstractions that are useful for obtaining a more efficiently executable specification without affecting the kinds of properties that one would want to verify about Orc programs. The first optimization is to drop the meta-level operations in the definition of external communication between Orc objects across different processes, and define socket-based messaging at the level of Orc values rather than at the lower-level of strings. This results in a higher abstraction that does not have to deal with serialization and de-serialization of Orc values, as was required in the Dist-Orc implementation (see Section 5.1.3). It also entails a slight modification to the syntax of the socket specification in $\mathcal{R}_{socket}$. In particular, `send` and `received` messages each now take a list of Orc values rather than a string as a parameter:

```
op send : Oid Oid ValueList -> Msg [ctor msg] .
op received : Oid Oid ValueList Time -> Msg [ctor msg] .
```

The rules and equations defining external, socket-based message exchange are also appropriately updated.

Another optimization, which aims at reducing the reachable state space of a distributed Orc program without changing the semantics of the underlying Orc expressions and sites, is to impose a slightly more restrictive rule application strategy. More specifically, we may give internal transitions of an Orc expression (site calls, expression calls, and publishing of values) priority over socket-based transitions (creating sockets, and sending and receiving external messages). Besides being natural, this strategy does not conflict with Orc's synchronous semantics, as internal transitions still have precedence over the external transition of consuming a site return. Furthermore, it turns out that this extension can be easily specified by simply changing the relevant rewrite rules in $\mathcal{R}_{socket}$ so that the underlying Orc expression objects have expressions that are *inactive*. For instance, here is a fragment of the modified rule specifying successful client socket creation `[CreateClientSocketSuccess]`. The rule

matches an inactive expression `iF` in the Orc expression object trying to make a connection with an external site:

```
rl [CreateClientSocketSuccess] :
  < PID : Process | sys : [acceptClient(server(N), O) CF] >
  < PID' : Process |
     sys : [createClientTcpSocket(socketManager, p(OE,H ), ADDRESS, PORT)
            < p(OE,H) : Proxy | > < OE : Expr | exp : iF > CF' ] >
  < socketManager ... > < server(N) ... >
  => < PID : Process |
        sys : [acceptedClient(O, server(N), ADDRESS, socket(M)) CF] >
     < PID' : Process |
        sys : [createdSocket(p(OE,H), socketManager, socket(M))
               < p(OE,H) : Proxy | > < OE : Expr | exp : iF > CF'] >
     < socketManager ... > < server(N) ... > < socket(M) ... > .
```

5.3.4 Formal Analysis of `Dist-Auction`

The formal specification of sockets and logical time provides a formal model of Dist-Orc that can be used to verify properties about distributed applications in Orc, which take into account some of the actual problems that can take place, not only in Orc expressions, but also in socket communications. To illustrate this formal verification capability, we use RTM to formally analyze the distributed implementation `Dist-Auction` of the auction case study. In particular, we perform time- bounded linear temporal logic model checking with commands of the form (`mc` *term* `|=t` *formula* `in time <=` *timeLimit* `.`), and timed search using (`find earliest` *term* `=>*` *pattern* `such that` *condition* `.`), which finds a state reachable within the shortest possible time that matches the given pattern and satisfies the given condition. Verification is applied on a *closed* system specification that includes definitions of all required sites (servers) and expressions (clients) in the Auction application.

In our analysis, we assume a single seller site with two items for sale, labeled `1910` and

1720, and offered for auction for 5 and 7 time units, respectively. The function `initial(DS)` constructs an initial state for `Dist-Auction` in which the set of possible message transmission delays is `DS`, which, in the analysis examples below, is the singleton set $\{0.1\}$, unless otherwise indicated. The atomic predicates used are:

1. *commError*, which is true in states with communication errors:

```
op commError : -> Prop .
eq {< PID: Process | sys: [socketError(O, O', S) CF] > CF'}
   |= commError = true .
```

2. *sold*(*id*), which is true in states where the item *id* has been sold:

```
op sold : Nat -> Prop .
eq {< PID : Process |
    sys : [< O : XSite | name : 'auction,
                         state : won(winner(N, id, M), WN) OST > CF] > CF'}
    |= sold(id) = true .
```

   where the term `winner(N, id, M)` matches a winning bid `M` on item `id` by the `N`th bidder.

3. *hasBid*(*id*), which is true when the item *id* has been bid on:

```
op hasBid : Nat -> Prop .
eq {< PID : Process |
   sys : [< O : XSite | name : 'bidders,
                        state : bidders(b(N, [id, M] IBS) BS) OL > CF] > CF'}
    |= hasBid(id) = true .
```

   where the term `b(N, [id, M] IBS)` matches a bid `M` on item `id` by the `N`th bidder.

4. *conflict*(*id*), which is true when item *id* has two different winners:

```
op conflict : Nat -> Prop .
eq {< PID : Process |
        sys : [< O : XSite | name : 'auction ,
                             state : won(winner(N, id, M),
                                     winner(N', id, M'), WN)
              OST > CF] > CF'}
   |= conflict(id) = true .
```

A property that is typically required in an auction management system is that an item with at least one bid is eventually sold: $\Box \bigwedge_i (hasbid(id_i) \rightarrow \Diamond sold(id_i))$. This can be shown to be guaranteed by `Dist-Auction` in the absence of communication problems and excessively large delays. The property itself is specified in Real- Time Maude as the following formula `commitAllNoErrors` (with ~ denoting the LTL negation operator) :

```
op commit : Nat -> Formula .
eq commit(id) = hasBid(id) -> <> sold(id) .
op commitAllNoErrors : -> Formula .
eq commitAllNoErrors = ([] ~ commError) -> [] (commit(1910) /\ commit(1720)) .
```

The property is then verified with the time-bounded model checking command:

```
Maude> (mc initial(1/10) |=t commitAllNoErrors in time <= 15 .)
rewrites: 7052663 in 14413ms cpu (14420ms real) (489317 rewrites/second) ...
Result Bool : true
```

The property is satisfied with a communication delay of 0.1 time units. In fact, the property is satisfied when communication delays are bounded by 0.25 time units. This is because the timeout value for collecting bids in a single bidding round in the *TimeoutRound* expression is 1.0, while a delay of 0.25 translates into a cumulative round trip delay of 1.0 for its two sequential site calls, which may result in an uncommitted bid. This can be verified by the resulting counterexample when executing the command above but with `initial(1/4)`.

Another property an auction management system must guarantee is that every item sold has a unique winner: $\Box \bigwedge_i \neg conflict(id_i)$. This property can be shown satisfiable in

`Dist-Auction` regardless of communication errors. The property is specified in RTM as the formula `uniqueWinnerAll`:

```
op uniqueWinner : Nat -> Formula .
eq uniqueWinner(id) = ~ conflict(id) .
op uniqueWinnerAll : -> Formula .
eq uniqueWinnerAll = [] (uniqueWinner(1910) /\ uniqueWinner(1720)) .
```

The property is verified by the following command :

```
Maude> (mc initial(1/10) |=t uniqueWinnerAll in time <= 15 .)
rewrites: 8613539 in 19627ms cpu (19800ms real) (438843 rewrites/second) ...
Result Bool : true
```

Finally, given a delay of 0.1, one can verify that the first item cannot be won before 5.5 time units have passed using the following command:

```
Maude> (find earliest initial(1/10) =>* {C:Configuration}
          such that {C:Configuration} |= sold(1910) .)
rewrites: 268287407 in 1525921ms cpu (1544117ms real) (175819 rewrites/second) ...
Result: {< did : Delays | ds : 1/10 > ... } in time 11/2
```

CHAPTER 6

STATISTICAL MODEL CHECKING ANALYSIS

Statistical model checking (see, e.g., [1, 101]) is an attractive formal analysis method for probabilisitic systems. Although the properties model checked can only be ensured up to a user-specified level of statistical confidence (as opposed to the *absolute* guarantees provided by standard probabilistic model checkers), the approximate nature of the formal analysis is compensated for by its better scalability, the fact that the models to be analyzed can often be known only approximately, and the interest in analyzing quantitative properties for which an approximate result within known bounds is quite acceptable.

There are many systems for which this kind of statistical model checking analysis can be very useful. For example, distributed real-time systems, including so-called cyber-physical systems, are often probabilistic in nature, both because they often use probabilistic algorithms, and due to the uncertain, stochastic nature of the environments with which they interact. Furthermore, *quality of service properties* may be as important as traditional boolean-valued properties such as safety properties. For example, in a secure communications system, *availability* of vital information may be as important as its *secrecy*, but availability may be utterly lost due to a denial of service (DoS) attack with no loss of secrecy. Suppose that such a system is hardened against DoS attacks. How should one formally analyze the effectiveness of such a hardening? What is needed is not a Boolean-valued yes/no answer, but a *quantitative* one in terms of the expected *latency* of messages under certain assumptions about the attacker and the network. Quantitative information may include probabilities $p \in [0, 1]$, but need not be reducible to probabilities. For this reason, it is important to support statistical model checking not only of standard probabilistic temporal logics such as *PCTL*/*CSL*, but also of *quantitative temporal logics* like QuaTEx [2], where the result of evaluating a temporal formula on a path is a real number. This of course

includes the case of probabilities, as values $p \in [0, 1]$, and even of standard truth values, as values in $\{0, 1\}$, as special cases.

In this chapter, we introduce parallel algorithms that drastically increase the scalability of statistical model checking, and also make such scalability of analysis available to tools like Maude, where probabilistic systems can be specified at a high level as *probabilistic rewrite theories* [2], which are theories in rewriting logic [10] that may contain, in addition to regular rewrite rules, *probabilistic rewrite rules* modeling probabilistic transitions of such systems. We present PVeStA, an extension and parallelization of the VeStA statistical model checking tool [72]. PVeStA supports statistical model checking of probabilistic real-time systems specified as either: (i) discrete or continuous Markov Chains; or (ii) probabilistic rewrite theories in Maude. Furthermore, the properties that it can model check can be expressed in either: (i) *PCTL*/*CSL*, or (ii) QuaTEx. Since statistical model checking is based on Monte-Carlo simulations, which are naturally parallelizable, the performance gains can be very high, as our experiments show.

This chapter also further develops a rewriting-based approach to the formal specification and verification of probabilistic systems initiated in [102] and further applied to other systems in [103, 104], and applies it to a novel DoS protection mechanism, namely the Adaptive Selective Verification protocol [105]. In this approach, real-time and probabilistic protocols and DoS attackers are naturally modeled using probabilistic rewrite theories, while properties are expressed by quantitative, real-valued formulas in QuaTEx, whose formal verification is based on statistical quantitative analysis [2], using Maude and PVeStA.

6.1 Parallel Statistical Model Checking and Quantitative Analysis Algorithms

Sen et. al. [1] described an algorithm $\mathcal{A}$ based on simple hypothesis testing for statistical model checking of formulas in both: (1) *Probabilistic CTL* (*PCTL*) [106], which extends standard *CTL* by associating probability measures to computation paths of a probabilistic system and qualifying the temporal logic formulas with probability bounds, and (2) *Continuous Stochastic Logic* (*CSL*) [107, 108], which further extends *PCTL* by continuous timing

and qualifying temporal logic operators by time bounds. Given a probabilistic model $\mathcal{M}$, a *PCTL/CSL* formula $\mathcal{P}_{\bowtie p}(\varphi)$, with φ a state or path formula[1], and error bounds α and β, the algorithm $\mathcal{A}$ checks satisfiability of the formula by setting up a statistical hypothesis testing experiment such that its Type I and Type II errors are bounded, respectively, by α and β. The test is based on the sample mean of n random samples of φ computed over n Monte-Carlo simulations of the model. The algorithm uses standard statistical methods to precompute the total number n of samples needed to achieve the desired test strength (see [1] for more details).

To be able to express not just probabilities of satisfaction of temporal logic formulas but also *quantitative properties* such as, for example, the expected latency of a probabilistic communication protocol, *PCTL* and *CSL* have been generalized to a logic of *Quantitative Temporal Expressions* (QUATEX) in [2], in which state formulas and path formulas are generalized to user-definable, *real-valued state expressions* and *path expressions*. In [2], Agha et. al. proposed a statistical quantitative analysis algorithm $\mathcal{Q}$ for estimating the expectation of a temporal expression in QUATEX. Given a probabilistic model $\mathcal{M}$, an expectation QUATEX formula of the form $\mathbf{E}[Exp]$, with *Exp* a QUATEX state or path expression, and bounds α and δ, the algorithm $\mathcal{Q}$ approximates the value of $\mathbf{E}[Exp]$ within a $(1-\alpha)100\%$ confidence interval, with size at most δ, by generating a large enough number n of random sample values $x_1, x_2, \ldots, x_n$ of *Exp* computed from n independent Monte Carlo simulations of $\mathcal{M}$. The value returned by the algorithm as the estimator for $\mathbf{E}[Exp]$ is the sample mean $\bar{x} = \frac{\Sigma_{i\in[1,n]}x_i}{n}$. To guarantee the quality and size requirements of the confidence interval (given respectively by α and δ) for $\bar{x}$, the number n of sample values must be large enough. In general, the more accurate the estimator, the larger the number of samples required. To generate enough samples, the algorithm $\mathcal{Q}$ uses student's t-distribution to compute a $(1-\alpha)100\%$ confidence interval by iteratively generating them in batches of N samples each (with $N > 5$). Once the size of the computed interval falls below the threshold δ, $\mathcal{Q}$ halts and the sample mean $\bar{x}$ is returned (more details can be found in [2]).

A key observation is that the both statistical analysis algorithms $\mathcal{A}$ and $\mathcal{Q}$ expose a real op-

[1]We restrict our attention to non-nested probabilistic formulas here, although the algorithm of [1] can handle nested formulas as well.

Algorithm 1 A parallel algorithm $\mathcal{A}^p$ for checking (non-nested) probabilistic *CSL* formulas

Input: Model $\mathcal{M}$, Probabilistic formula $\mathcal{P}_{\bowtie p}(\varphi)$, α, β, Parameters $\vec{p}$, Resources R
Output: True or false, with Type I and II errors bound by α and β, respectively (see Theorem 1 in [1])

1: $n \leftarrow \mathsf{computeSampleSize}(p, \bowtie, \alpha, \vec{p})$
2: $sum \leftarrow 0$,
3: $sims \leftarrow \lfloor N \ / \ |R| \rfloor$, $r\_sims \leftarrow N \mod |R|$
4: **for** $i \leftarrow 1$ **to** $|R|$ **do**
5: $\quad m_i \leftarrow (i < r\_sims) \ ? \ sims + 1 \ : \ sims$
6: **for** $i \leftarrow 1$ **to** $|R|$ **do in parallel**
7: $\quad sum_i \leftarrow 0$
8: $\quad$ **for** $j \leftarrow 1$ **to** m_i **do**
9: $\quad\quad x_{ij} \leftarrow \mathsf{sat}(\varphi, p, \bowtie, \pi(\mathcal{M})$
10: $\quad\quad sum_i \leftarrow sum_i + x_{ij}$
11: **for** $i \leftarrow 1$ **to** $|R|$ **do**
12: $\quad sum \leftarrow sum + sum_i$
13: $b \leftarrow \mathsf{performHT}(sum, n, \alpha, \beta, \vec{p})$
14: **return** b

portunity for parallelization through their dependence on performing batches of independent Monte Carlo simulations. Moreover, the cost of performing a single discrete-event simulation from a model of a probabilistic or cyber-physical system is usually high compared with other computations in the algorithms, due to the typically complex nature of these systems. Consequently, by exploiting this opportunity, performance gains through parallelization can be significant and the scalability of this kind of statistical model checking analysis can be substantially improved.

We develop parallel, map-reduce versions $\mathcal{A}^p$ and $\mathcal{Q}^p$ of both algorithms in which the task of computing a set of n sample values for a state or path formula in *PCTL*/*CSL* or QuaTEx is done in parallel by performing n Monte Carlo simulations in parallel. Both parallel algorithms make no assumptions about the underlying parallel architecture. For *PCTL*/*CSL*, $\mathcal{A}^p$ assumes non-nested probabilistic formulas.

An outline of $\mathcal{A}^p$ is shown in Algorithm 1. In addition to the model $\mathcal{M}$, the probabilistic formula $\mathcal{P}_{\bowtie p}(\varphi)$ and the verification parameters, the algorithms take as input a list of available computing resources R on which the task of generating random samples is mapped. This task is first distributed as evenly as possible by determining the number of simulations

Algorithm 2 A parallel algorithm $\mathcal{Q}^p$ for estimating expected values of QuaTEx expressions

Input: Model $\mathcal{M}$, QuaTEx expression Exp, α, δ, load factor k, Resources R
Output: An estimator for $\mathbf{E}[Exp]$, with a $(1-\alpha)100\%$ confidence interval of size at most δ

1: $sum \leftarrow 0, \quad n \leftarrow 0, \quad d \leftarrow \infty$
2: **if** $N < |R|$ **then**
3: $\quad N \leftarrow k \cdot |R|$
4: **else**
5: $\quad N \leftarrow k \cdot \lceil N \,/\, |R| \rceil \cdot |R|$
6: $m \leftarrow N \,/\, |R|$
7: **while** $d > \delta$ **do**
8: $\quad n \leftarrow n + N$
9: $\quad$ **for** $i \leftarrow 1$ **to** $|R|$ **do in parallel**
10: $\quad\quad sum_i \leftarrow 0, \quad sumSquare_i \leftarrow 0$
11: $\quad\quad$ **for** $j \leftarrow 1$ **to** m **do**
12: $\quad\quad\quad x_{ij} \leftarrow \mathsf{eval}(Exp, \pi(\mathcal{M}))$
13: $\quad\quad\quad sum_i \leftarrow sum_i + x_{ij}$
14: $\quad\quad\quad sumSquare_i \leftarrow sumSquare_i + x_{ij}^2$
15: $\quad$ **for** $i \leftarrow 1$ **to** $|R|$ **do**
16: $\quad\quad sum \leftarrow sum + sum_i$
17: $\quad\quad sumSquare \leftarrow sumSquare + sumSquare_i$
18: $\quad d \leftarrow \mathsf{computeCISize}(sum, sumSquare, n, \alpha)$
19: **return** $sum \,/\, n$

m_i to be performed by each available computing resource R_i in R. Since the total number of samples n is precomputed, m_i is either $\lfloor n/|R| \rfloor$ or $\lfloor n/|R| \rfloor + 1$. The body of the **for** loop at line (6) is done in parallel, where each resource R_i checks satisfiability of the formula φ on m_i random executions of the model $\mathcal{M}$ obtained by discrete-event simulations of $\mathcal{M}$, generating m_i random sample results. The function $\mathsf{sat}(\varphi, p, \bowtie, \pi(\mathcal{M})$ verifies the state or path formula φ on the given computation path $\pi(\mathcal{M})$, and returns 1 if the formula is satisfied and 0 otherwise. Each resource R_i then returns the sum sum_i of its computed random samples. Once all partial sums are collected, the algorithm performs the simple hypothesis testing experiment (denoted by the function $\mathsf{performHT}$) as explained in [1], using the total sum of all random samples generated by the resources R.

The parallel QuaTEx evaluation algorithm $\mathcal{Q}^p$ is outlined in Algorithm 2. Like $\mathcal{A}^p$, the algorithm takes as input a list of available computing resources R. The number m of simulations to be performed by each computing resource R_i is computed as a positive integer

multiple of $|R|$ and the *load factor* k, which is a parameter to $\mathcal{Q}^p$ that can be used to increase the number of simulations performed by each resource in a round. Given a verification task, the load factor k can be tuned to optimize performance, especially for lightweight simulations when the desired statistical confidence is high, as we will see in Section 6.2. Once m is determined, each resource R_i computes m random samples using the function $\mathsf{eval}(\varphi, \pi(\mathcal{M}))$, which evaluates a QuaTEx state or path expression φ on a computation path π, obtained by a discrete-event simulation of $\mathcal{M}$, according to the semantics of such expressions given in [2]. Each resource R_i returns a partial sum, sum_i, and a partial sum of squares, $sumSquare_i$, of sample values, which are then used in the sequential computation of the size d of the new confidence interval (denoted by the function computeCISize). The algorithm halts and returns the overall sample mean once d becomes less than or equal to the confidence interval size bound δ.

6.2 Implementation in PVeStA

We have implemented a client-server prototype, PVeStA, of both parallel algorithms $\mathcal{A}^p$ and $\mathcal{Q}^p$, in Java, based on the Java implementation of the original algorithms in VeStA [72]. The tool, which is available for download online at `http://www.cs.illinois.edu/~alturki/pvesta`, consists of two command-line-based executable programs: (1) a client program `pvesta-client`, which implements the sequential parts of the algorithms performing simple hypothesis testing for *PCTL*/*CSL* formulas and confidence interval computations for QuaTEx expressions, and (2) a server program `pvesta-server`, which implements the role of a resource R_i that computes random samples by performing discrete-event simulations of a given model expressed as a Markov chain or as a probabilistic rewrite theory. Figure 6.1 presents a schematic diagram of the structure and interactions of the client and server parts of the tool.

The client program first reads a list of servers R that are available for performing simulations. It then creates, using Java's managed concurrency library, a thread pool of $|R|$ *callable* computation threads, which are Java threads that implement the `Callable` interface by specifying a `run` method to be called when the thread is invoked. Each thread, which

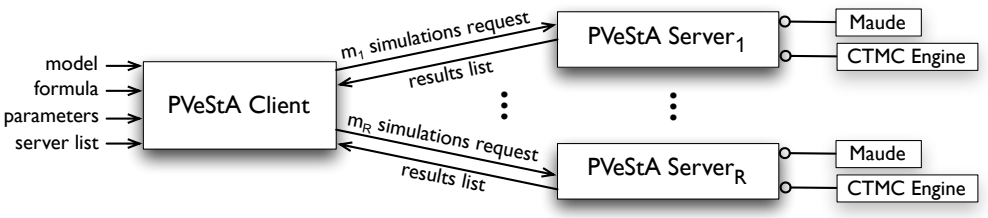


Figure 6.1: Components and interactions of PVeStA

will manage simulation requests and responses with a particular server in R, is supplied with a pseudo-random seed to be used by its corresponding server to guarantee statistical independence of the simulations. The thread pool is then submitted to an *executor* object, which invokes all the threads in the pool, commencing communication with the servers in R. Upon receiving the simulations request, each PVeStA server performs the requested number of simulations using either Maude (for models expressed as probabilistic rewrite theories) or the built-in Continuous-Time Markov Chain (CTMC) engine (for CTMC models) and produces a list of sample results. The client collects all samples in an array of $|R|$ `Future` Java class objects, from which the results are extracted and then used in performing the appropriate sequential computations. For confidence interval computations, the client may need to repeat this process until enough samples are collected.

Experimental Evaluation. We have conducted two sets of experiments with PVeStA to evaluate the performance gains of parallelization using two different parallel architectures: (1) a high-performance computing (HPC) architecture, in which simulation tasks are distributed over different nodes in a PC cluster, and (2) a multi-core architecture, in which simulations are distributed over different processing cores within a single node. The HPC benchmarks were executed on a PC cluster consisting of 256 nodes, each of which has two (single-core) AMD Opteron 2.2GHz CPUs with 2GB of RAM. The second set of experiments was performed on a server machine having two quad-core 2.66GHz Intel Xeon processors with 16GB of RAM.

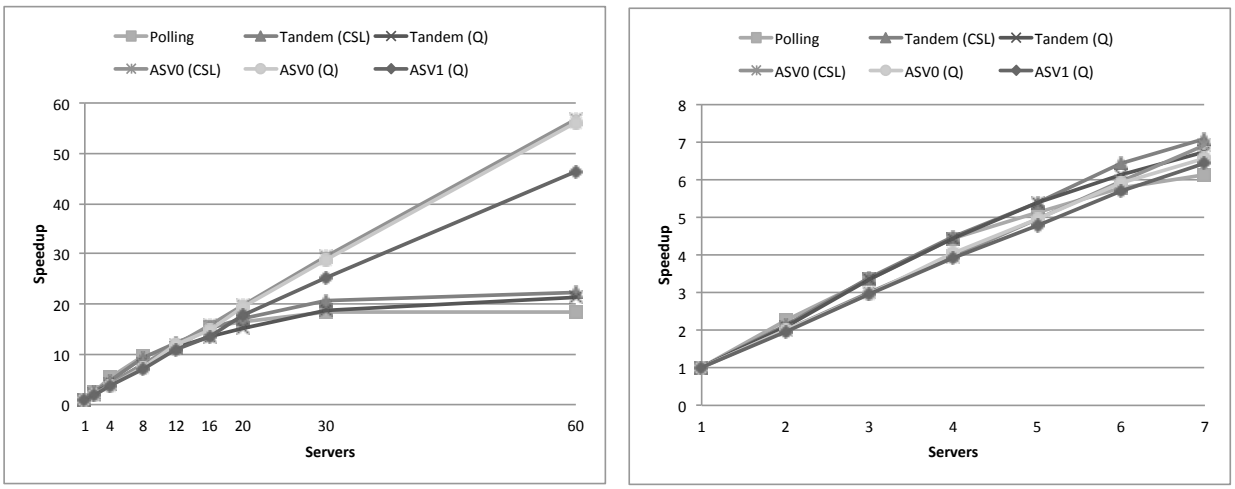
We use two examples from [72]: (1) a simplified server polling system, *Polling*, and (2) a simple tandem queuing system, *Tandem*, both expressed as continuous-time Markov chains. In addition, we use two variants of a larger case study, described in some detail in Section 6.3

	Polling (CSL)	Tandem (CSL)	Tandem (Q)	ASV_0 (CSL)	ASV_0 (Q)	ASV_1 (Q)
Simulations	16,906	16,906	46,380	1051	706	1,308
Servers	HPC Cluster					
1	6.78	9.54	17.36	494.9	770.8	1,584.3
2	2.61	4.06	8.56	248.4	385.4	798.5
4	1.24	2.01	4.26	124.2	197.1	410.5
8	0.70	1.02	2.19	62.1	103.4	221.9
12	0.59	0.77	1.53	41.4	65.3	144.3
16	0.44	0.63	1.27	31.1	52.3	116.6
20	0.42	0.56	1.14	25.1	39.4	89.9
30	0.37	0.46	0.93	16.9	26.7	63.1
60	0.38	0.43	0.82	8.7	13.7	34.2
Servers	Multi-core Computer					
1	3.83	5.53	11.26	367.7	559.7	1,167.9
2	1.70	2.60	5.43	184.5	281.1	589.5
3	1.15	1.62	3.36	122.9	189.4	396.5
4	0.86	1.24	2.53	92.3	138.7	298.3
5	0.74	1.03	2.09	74.2	113.1	243.0
6	0.66	0.86	1.84	61.8	94.5	204.5
7	0.62	0.78	1.66	53.1	85.1	181.2

Table 6.1: The (average) times in seconds taken by PVeStA to complete six verification tasks using a PC cluster and a multi-core computer

below, that specifies a probabilistic model in rewriting logic of the Adaptive Selective Verification (ASV) protocol [105] for thwarting DoS attacks. The first variant, denoted ASV_0, assumes a reliable communication channel, a fixed attack rate, and no message transmission delays, while the second variant, ASV_1, is more realistic, as it assumes a lossy channel, a variable attack rate, and random delays. Benchmarking is performed by measuring the total time required (including any additional time required for file and network I/O, thread and object management, and so on) to verify a probabilistic *CSL* formula in *Polling*, *Tandem*, and ASV_0, or a QuaTEx expectation expression in *Tandem* (load factor, $k = 100$), ASV_0 ($k = 1$), and ASV_1 ($k = 1$). The results are summarized in Table 6.1.

As the table clearly shows, performance gains as a result of parallelization can be substantial. For example, in the analysis of ASV_1, a verification task that would normally require about 27 minutes, can be completed in about 34 seconds on an HPC cluster using 60 nodes, and a 20-minute task can be done in just above 3 minutes on a multi-core machine using seven cores in parallel. In practice, several factors influence the speedups achieved by PVeStA, including the complexity of the model and the formula, and the statistical strength of the result. Figure 6.2(a) plots the speedups achieved against the number of servers used for HPC experiments in Table 6.1. We note that while performance scales almost linearly with the number of servers used for ASV_0 and ASV_1, the speedups for both *Polling* and *Tandem*



(a) HPC cluster

(b) Multi-core computer

Figure 6.2: The speedup using multiple PVeStA servers

begin to decelerate beyond 20 servers. This is primarily because the models *Polling* and *Tandem* are so simple that, as the number of servers increases, the time needed to generate random samples begins to be dominated by other computations in the tool. For the *Tandem*-Q experiment, which requires a fairly high statistical confidence, and thus a higher number of random samples, achievable speedups are greatly influenced by the chosen load factor k. In general, for such simple models, a higher value of k (and thus a higher number of simulations performed by a server in each round) translates into reduced processing and communication overhead and increased efficiency. For example, speedup tripled when using $k = 100$ compared with $k = 1$ for *Tandem*-Q with 60 servers. Of course, excessively high values of k result in an unnecessarily excessive number of simulations and degrade performance. Appropriate values of k can be determined by experimentation using the above ideas as guidelines. Figure 6.2(b), which plots speedups on a multi-core architecture, shows a similar pattern to Figure 6.2(a).

6.3 Statistical Analysis of the Adaptive Selective Verification Protocol

For the formal analysis of availability under DoS attacks, we often need to define protocol and attacker models that are real-time and probabilistic in nature. Such models are usually too complex to analyze manually or through symbolic manipulations. For such systems, it is well known that statistical methods, such as statistical quantitative analysis, provide effective and flexible means for automatically approximating properties about their behaviors with reasonable levels of statistical confidence. But when these methods are paired with executable formal models, a much stronger level of assurance can be achieved.

We apply the rewriting-based approach to the formal specification and verification of DoS resilience initiated in [102] to the Adaptive Selective Verification protocol [105] by: (1) modeling the behavior of both the protocol and the DoS attacker by means of probabilistic rewrite rules [2], (2) specifying properties using quantitative, real-valued formulas in QuaTEx [2], and (3) formally verifying these properties using statistical quantitative analysis with PVeStA and Maude.

In general, this approach provides a useful middle ground for the analysis of availability properties between manual mathematical analysis and simulation-based analysis, adding significant analytic power to these approaches. Specifically, the results obtained confirm by automatic statistical quantitative analysis techniques analytic results proved by hand in [105]. They also confirm, with a much stronger level of assurance, various protocol properties suggested by the simulation analyses reported as well in [105]. In this way, a considerably higher level of assurance can be gained for both analytical properties proved by hand, and for properties suggested by simulation analyses. Furthermore, this assurance can be gained for scenarios and realistic deployment conditions too complex to be amenable to manual mathematical analysis.

6.3.1 The Shared Channel Model and the ASV Protocol

Unlike the Dolev-Yao model [109], in which an attacker has full control over the communication channel, the *shared channel model* is a more appropriate model for the analysis of availability properties [110], since attackers can only probabilistically share a channel with legitimate clients to the server. However, an attacker may also replay modified (or faked) versions of previously seen legitimate packets at some maximum rate, specified as a parameter in the model.

The Adaptive Selective Verification (ASV) protocol [105] is a cost-based, DoS-resistant protocol in which bandwidth is the currency. ASV assumes the shared channel model as its underlying attack model, in which the goal of each legitimate client is to get a service from the server represented in the protocol by an acknowledgment message. The key idea of DoS-resilience in the protocol is for clients to spend more bandwidth to compete with attacker bandwidth usage, and for the server to selectively process incoming requests.

More precisely, we denote the server's mean processing rate by S, and the server and client timeout periods by T_s and T_c, respectively. The current client request rate is denoted by ρ, with the assumption that $\rho \in [\rho_{\min}, \rho_{\max}]$. Similarly, The current attack rate is denoted by $\alpha \in [\alpha_{\min}, \alpha_{\max}]$. The client replication threshold is specified by the protocol as 2^J, where $J = \lceil \log(\alpha_{\max}/\rho_{\min}) / \log(2) \rceil$ (called the *retrial span*). Under the ASV protocol, the server and clients behave as follows:

Client. When a client first arrives, it initializes its *retries count* $j \leftarrow 0$, and then sends a single copy of its request to the server. If the client receives and acknowledgment within T_c time units, the client succeeds and quits. Otherwise, it increments j ($j \leftarrow j + 1$) and then sends 2^j copies of its request to the server. This process is repeated until either an acknowledgment is received, or the retrial threshold is reached, i.e. $j > J$. In the latter case, the client fails and quits.

Server. The server first initializes its *window count* $k \leftarrow 1$, and its *request count* $j \leftarrow \lfloor ST_s \rfloor + 1$. During the *kth* window, the server attempts to collect the first $\lfloor ST_s \rfloor$ incoming requests. At this point, there are two cases:

1. If it times out before the reservoir is filled, the server sends an acknowledgment for each request in the reservoir, empties its reservoir, increments its window count k, and repeats the process for the next window.

2. If the reservoir is filled before a timeout occurs, the server places the jth incoming request in the reservoir with probability $p \leftarrow \lfloor ST_s \rfloor / j$ and discards it with probability $1 - p$. If the request is to be placed in the reservoir, the server replaces a request in the reservoir selected uniformly at random with the accepted request. The server increments its request count and processes the next request in the same way. This step is repeated until the server times out (signaling the end of the current window). Once a timeout occurs, the server empties its reservoir after acknowledging its requests, increments k, resets j to $\lfloor ST_s \rfloor + 1$, and the whole process is repeated for the next window.

Despite the simplicity of the protocol, analyzing it manually under simplifying assumptions turns out to be a fairly demanding task [105]. In the following, we describe a model of the protocol that enables automatic statistical verification of its properties and analyze the results in simple scenarios, and also under circumstances that would be too complex for manual analysis.

6.3.2 Formal Probabilistic Modeling of ASV in Rewriting Logic

Our model of the ASV protocol is based on a representation of actors with asynchronous message passing in rewriting logic [2], which is built using the logic's Maude object-based programming framework [20]. Within this framework, the system state is represented by a *configuration*, which is a soup of objects and messages, in which an object is a term of the form $\langle name : O \mid A \rangle$, with O a unique object identifier, and A a set of attribute-value pairs representing the state of the object, and a message is a term of the form $O \leftarrow C$, with O the target object id and C the contents of the message.

There are three main classes of objects, namely *client*, *attacker*, and *server* objects. A client object represents a legitimate client trying to get a service from the server. The object

maintains, among other attributes, the current number of retries and the current replication count, which are required to implement the adaptive protocol. The attacker object is a simple object that maintains only a reference to the server object on which the attack is to be carried out. The server object maintains at least two attributes: a buffer attribute *buffer*, which holds incoming requests between server timeouts, and a request packet count attribute *reqcnt*, which is used in determining the probability of accepting an incoming request when the buffer is full: $\langle name : O_s \mid buffer : L, reqcnt : R \rangle$. A server object, with id O_s, accepts two kinds of messages: (i) a connection request message of the form $O_s \leftarrow req(O)$, with O the object id of the client or attacker object which initiated the request, and (ii) an internal timeout message $O_s \leftarrow timeout$, which signals a new server time window. Self-addressed messages are commonly used in actor-based systems to schedule internal events [2].

In addition to the three classes above, a fourth, auxiliary class of objects is the *generator* class, of which a single object is used in the configuration to model new clients coming in at a rate ρS, with S the server's mean processing rate. The generator object maintains a counter for generating fresh client identifiers and the name of the server to which generated clients will attempt to connect: $\langle name : G \mid cnt : I, server : O_s \rangle$. The generator object G uses a self-addressed message of the form $G \leftarrow spawn$ to schedule the creation of the next client object every $1/\rho S$ time units.

Beside the objects described above, a configuration contains a few other components that support its dynamic behavior. First, in order for it to properly support statistical model checking and quantitative analysis, the specification must avoid any unquantified non-determinism. We adopt a specification style originally developed in [102] to support this requirement. In this style, the configuration uses a *scheduler* that stores a list of scheduled messages to be made active and ready for consumption by the appropriate object in the configuration. The scheduler, which is a term of the form $\{T \mid S\}$, with T the current global clock of the configuration and S a list of scheduled messages, enforces the property that only one message becomes active at any given instant of time. A *scheduled message* is a term of the form $[t, m, d]$, where t is the time at which the message is scheduled for delivery, m is the message itself, and d is a drop flag that is used when modeling lossy channels to indicate whether the message is to be dropped or kept.

The use of the scheduler object also provides a mechanism for managing the elapse of time and its effect on the configuration. This is achieved with the help of an operator *mytick*, which is used to extract the next message from the scheduler and update the current global time accordingly. The specification uses this operator repeatedly to advance time and process successive messages from the scheduler until the given time limit, specified by a parameter in the model, is reached. This flexible mechanism enables us to specify the granularity of a round in terms of the amount of time we wish to run a Monte Carlo simulation.

The behavior of the model is specified (mostly) by rewrite rules, some of which are probabilistic. As an example, the following is a simplified version of the probabilistic rule for capturing the event of handling an incoming request by the server.

[ProcessReq] :
$\{T \mid S\}\ \ O_s \leftarrow req(O)\ \ \langle name : O_s \mid buffer : L, reqcnt : P \rangle$
$\rightarrow mytick(\{T \mid S\})$
if the request buffer L is full **then**
if B **then** $\langle name : O_s \mid buffer : replace(L, req(O), U), reqcnt : P + 1 \rangle$
else $\langle name : O_s \mid buffer : L, reqcnt : P + 1 \rangle$ **fi**
else $\langle name : O_s \mid buffer : add(L, req(O)), reqcnt : P \rangle$ **fi**
with probability $B := Bernoulli(\mathcal{A}(P)) \ \wedge\ U := Uniform(|L|)$

When a server receives a connection request message $req(O)$, it first checks whether its request buffer stored in the *buffer* attribute is full. If the buffer is not yet full, the request is simply added to the list. Otherwise, if the buffer has already reached its maximum capacity, the server tosses a biased coin with success probability B, given by a function of its *reqcnt* attribute P, and uses the outcome of this experiment to decide on whether to replace an existing request selected uniformly at random with the incoming request or to drop the incoming request altogether.

A detailed discussion of the specification of the model can be found in [58].

6.3.3 Properties of ASV as QuaTEx Expressions

We have used the ASV model described above to perform statistical quantitative model checking analysis of various QuaTEx formulas using PVeStA and Maude to produce a point estimator of the quantity of interest for each of these formulas, given a desired confidence interval for the experiment and its maximum tolerable size. For this purpose, we specify the nature of the quantities to be statistically estimated. The quantities specified as QuaTEx formula declarations are listed below.

Connection ratio. This is the ratio of clients successfully connected to the total number of clients in a configuration:

$$\begin{aligned} \mathit{connRatio}(t) \;\; = \;\; & \textbf{if } \mathit{time}() > t \textbf{ then } \mathit{countConnected}()/\mathit{countClients}() \\ & \textbf{else } \bigcirc\,(\mathit{connRatio}(t)) \end{aligned}$$

with *time*() a state function that returns the global clock time in the current configuration. The number of accepted clients *countConnected*() is computed by equationally counting the number of clients whose status field is *connected*, while the total number of clients *countClients*() can be easily extracted from the client counter attribute maintained in the client generator object.

Average TTS. This is the ratio of the total time-to-service added up over all accepted clients to the number of accepted clients:

$$\begin{aligned} \mathit{avgTTS}(t) \;\; = \;\; & \textbf{if } \mathit{time}() > t \textbf{ then } \mathit{sumTTS}()/\mathit{countConnected}() \\ & \textbf{else } \bigcirc\,(\mathit{avgTTS}(t)) \end{aligned}$$

The total TTS *sumTTS*() is computed by adding up the time intervals given by the arrival time and service time attributes of every accepted client. The number of accepted clients is computed as described above.

Bandwidth usage. This is the amount of bandwidth used by legitimate clients attempting to establish a connection with the server. Bandwidth usage is measured in terms of

the number of legitimate requests and is given by the legitimate client request counter attribute of the server object.

$$bw(t) = \textbf{if } time() > t \textbf{ then } bwUsage() \textbf{ else } \bigcirc\,(bw(t))$$

Connection Confidence. This is the probability that a given client will successfully establish a connection to the server:

$$\begin{aligned} connConfidence(i,t) \;\; = \;\; & \textbf{if } time() > t \textbf{ then } hasConnected(i) \\ & \textbf{else } \bigcirc\,(connConfidence(i,t)) \end{aligned}$$

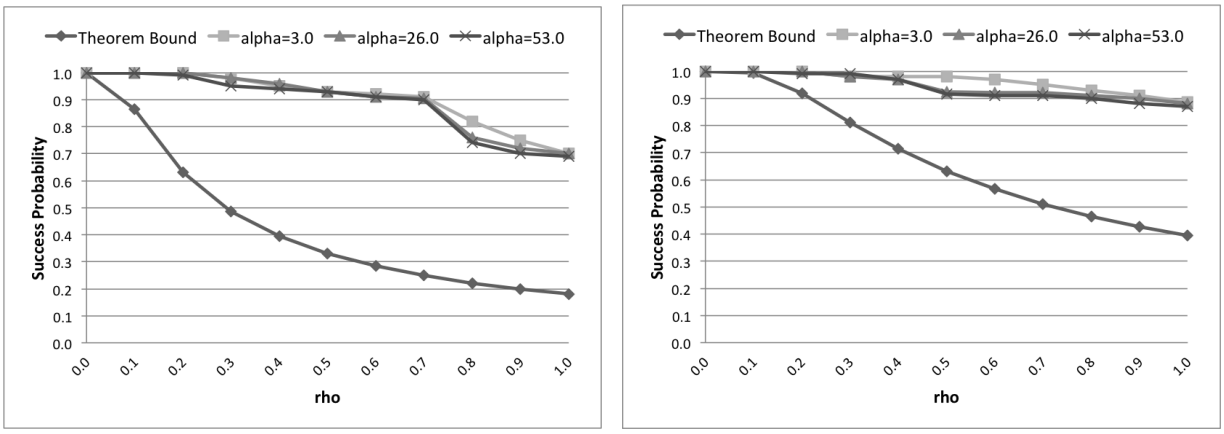
This is computed using a simple function $hasConnected(i)$ that, given a client id, returns 1.0 if the client has actually connected to the server and 0.0 otherwise.

Throughout this section, we assume a 95% confidence interval with size at most 0.05. We also fix S, the mean server processing rate, to 600 packets per time units, the server and client timeouts, T_s and T_c, to 0.4 time units, unless otherwise specified.

6.3.4 Verification of ASV Properties

In this section, we use the ASV model to formally verify two important properties of the protocol, which were given in [105]. The properties provide guarantees on the connection confidence and legitimate bandwidth consumption of ASV.

To provide a benchmark for the performance of ASV with respect to these properties, a simpler protocol, namely, the *omniscient* protocol, in which the server and clients are always aware of the current ρ and α is also described in the cited paper. We modeled the omniscient protocol to compare the bounds given by these theorems and provide formal statistical evidence of their correctness. In the omniscient model, the server accepts client requests with probability that depends only on the (now known) ρ and α, which implies a simpler server object specification as it no longer needs to maintain a buffer *reqlist* nor a client request replication count *reqcnt*. Furthermore, a client uses the value $\lceil \alpha/\rho \rceil$ as its



(a) The ASV protocol (b) The omniscient protocol

Figure 6.3: Connection confidence: Theorem bound vs. estimated values

replication count, and fails if no ACK is received after T_c units of time.

Connection confidence (Theorems 1 and 2 of [105]). The property gives a lower bound on the probability with which a given client will be able to establish a connection to the server, given a condition on the client request rate ρ. In particular, for the omniscient protocol, Theorem 1 states that if $\rho_{\max}$ is at most $1/(-2\log\delta)$, with δ a given confidence parameter, then any given client will be accepted with probability at least $1-\delta$. A similar lower bound is guaranteed for ASV under a slightly stronger condition on $\rho_{\max}$, which is that $\rho_{\max} \leq 1/(-5\log\delta)$. These properties are verified by fixing a client i (say the first client[2]) and then estimating the expectation of the $connConfidence(i,t)$ formula. Figure 6.3 plots the estimated probabilities of the first client getting connected versus the bound given by the theorem at different confidence parameter values, giving rise to different upper bounds on $\rho_{\max}$, and assuming three different levels of attack (low, medium, and high). For this analysis, we assume a worst-case analysis with $\rho = \rho_{\max}$. As both Figures 6.3(a) and 6.3(b) show, the estimated success probabilities under both protocols are always higher than the respective theorem bounds over the whole range of values of ρ, which confirms the statements of the theorems. We also note that, with respect to the connection confidence property, both protocols are able to maintain high success probabilities at higher attack levels compared to those at low attack levels.

[2]For this analysis, the choice of the client is immaterial since all clients behave identically and are introduced to statistically similar attack conditions.

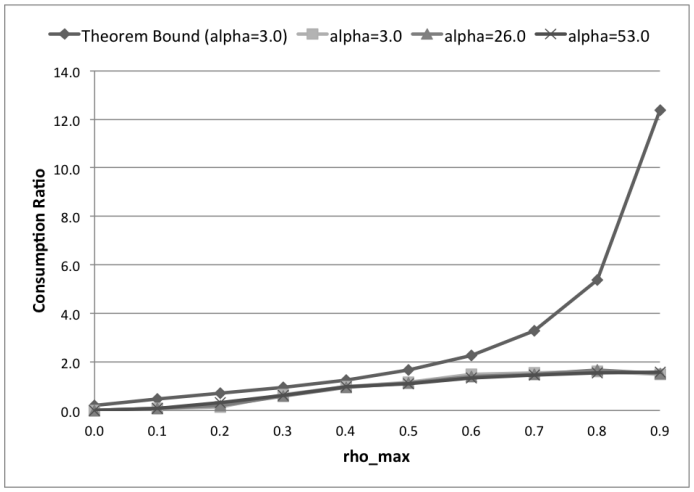


Figure 6.4: Bandwidth Usage: Theorem bound vs. estimated values

Bandwidth usage (Theorem 4 of [105]). This property gives a bound on the bandwidth consumed by legitimate clients in ASV. In particular, Theorem 4 states that, under the assumption of bounded variability in ρ, the ratio of the legitimate bandwidth consumed in ASV to that in the omniscient protocol is bounded above by $\log(\alpha_{\max})/\log(\frac{1}{\rho_{\max}})$. As discussed in [105], the restriction on the variability of ρ is imposed only to simplify manual analysis and the statement of the theorem. Besides giving an independent confirmation of the theorem by model checking analysis, we confirm the conjecture that the upper bound holds even when the restrictions on ρ are lifted. This is achieved by estimating the expectation of the formula $bw(t)$, while fixing $\rho_{\min}$ to a very low value (close to 0.0) and allowing $\rho_{\max}$ to vary from very small values all the way up to almost 1.0. Figure 6.4 plots the estimated ratios at three different attack levels as well as the upper bound given by the theorem at the lowest level of attack ($\alpha = 3.0$). The bounds corresponding to medium and high attack rates are not shown, as both are too high to appear within the figure's scale. The important observation here is that the ratios of the legitimate bandwidth consumed by ASV to that of the omniscient protocol are always below the theorem's bound. The figure also suggests that these ratios change only slightly across different attack conditions.

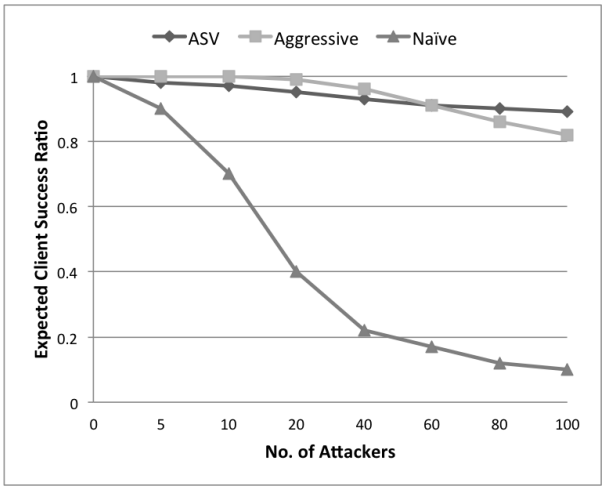
6.3.5 ASV versus Non-adaptive Selective Verification Schemes

In [105], the results of various NS-2 simulations comparing ASV to two non-adaptive selective verification variations under various attack conditions were reported. The non-adaptive schemes are: (i) the *Naive* protocol, in which a client does not increase its replication count with time, and (ii) the *Aggressive* protocol, in which a client sends the maximum number of requests (2^J) at once upon entering the configuration. The simulations reported validated the different trade-offs associated to the adaptive versus the non-adaptive schemes in terms of the ratio of successful connections, the average time-to-service, and the legitimate bandwidth used. We show here that using the ASV model in Maude along with two variants of it, corresponding to the non-adaptive protocols above, we obtain similar results through statistical quantitative analysis with PVeStA, independently confirming, and giving greater strength to, the simulation analyses.

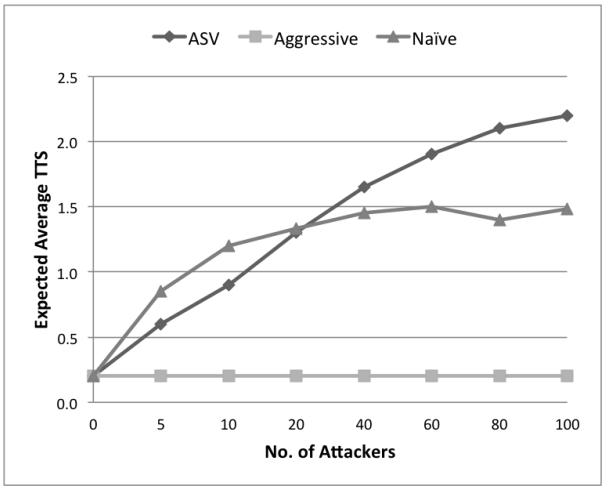
Since the non-adaptive protocols differ from the ASV protocol only in client request behavior, the Naive and Aggressive models are very similar to that of ASV. In fact, their models differ from the ASV model in essentially one rewrite rule, which is the one labeled [CSend]. In the Naive protocol model specification, the [CSend] rule maintains the initial replication count, which is equal to 1, causing the client to send exactly one REQ at every client timeout until connected or failed. On the other hand, the [CSend] rule of the Aggressive model distinguishes two cases. During the first attempt at making a request to the server (indicated by the *retires* attribute being 0), the rule replicates the request 2^J times (the replication count is simply ignored). Otherwise, if the client has already sent its initial set of requests ($0 < \mathit{retries} \leq J$), the client remains silent.

Since we intended to independently confirm the results of the NS-2 simulations of [105], we instantiate the model using essentially the same values for the parameters. That is, we set ρ to 0.08, J to 7, and *Limit* (the simulation duration) to 30.0 time units. The expectation of the connection ratios, average TTS values, and legitimate bandwidth usage are estimated at different attacker rates (given in terms of the number of attackers). The results are shown in Figure 6.5.

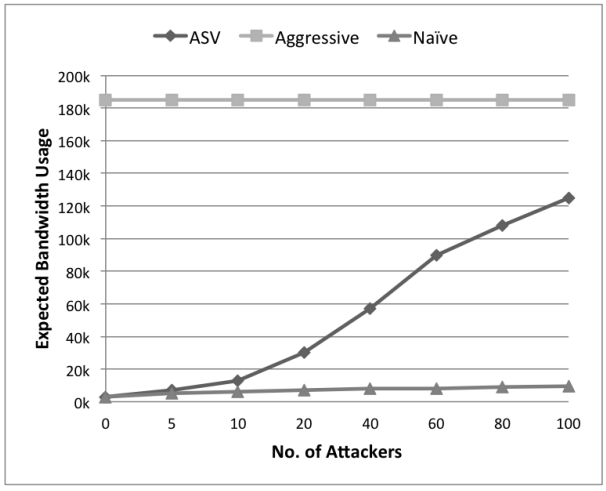
As the figure shows, the results obtained confirm the effectiveness and efficiency of ASV



(a) Connection success ratio



(b) Average TTS



(c) Bandwidth usage

Figure 6.5: Performance of ASV compared to non-adaptive schemes

compared to non-adaptive selective verification schemes. In Figure 6.5(a), ASV is statistically shown to be effective even under high rates of attack, where it outperforms both non-adaptive protocols. ASV is able to achieve this high performance at the expense of some latency inherent in its adaptive behavior (See Figure 6.5(b)), which is higher than that of the Naive protocol during periods of medium to heavy attacks. For legitimate bandwidth consumption, Figure 6.5(c) shows that at low to medium attack levels, ASV is able to maintain low bandwidth consumption levels that are comparable to those of the Naive protocol. Even at higher attack levels, ASV manages to outperform the aggressive protocol by a respectable margin.

CHAPTER 7

AVAILABILITY ANALYSIS OF ORC SERVICES

In general, compositions of services may create new possibilities for DoS attacks by exploiting the newly created inter-dependencies between these services, which constitute an emerging threat to service- and cloud-based software systems. For example, knowledge of an intermediary *forwarding* service within an orchestration may be exploited to dramatically amplify an otherwise limited flooding attack on other services[1]. Another example is a CPU- and memory-exhaustion attack on orchestration processes, described in [113] as an *instantiation flooding* attack on BPEL processes, in which the orchestration process itself is flooded with bogus requests for creating and managing new instances of parts of the orchestration. A third example, which is discussed in detail in [114], illustrates a cross-site scripting attack using the specification of a service composition to destroy the integrity of the target service (and hence deny meaningful service to its clients). While some of these DoS vulnerabilities may be specific to current web service standards, such as BPEL and WS-Security, most of these vulnerabilities fundamentally represent variations on familiar DoS attack opportunities, including most notably distributed network-based flooding attacks, which are the kind of attacks this chapter focuses on.

This chapter presents a formal method for the specification and analysis of availability properties against DoS in distributed service compositions based on: (1) the rewriting logic approach to formal specification and analysis of probabilistic systems using probabilistic rewrite theories and statistical model checking discussed in Chapter 6, (2) the object-based, rewriting logic semantics of Orc, $\mathcal{R}_{Orc}$, introduced in Chapter 4, and (3) the shared channel model and the Adaptive Selective Verification (ASV) protocol briefly reviewed in Section 6.3

[1] See a concrete instance of this kind of DoS attacks in the context of the Session Initiation Protocol (SIP) in [111] and a formal, rewriting logic approach to characterizing it in [112].

of Chapter 6. In particular, we introduce a generalization of the ASV specification as generic wrapper objects, similar in principle to the generic cookie-based DoS protection wrappers in [115], that can be used in a modular way to endow the specification of a communication system with ASV DoS protection. Using a somewhat simplified form of the *onion skin* model [116] specified in rewriting logic [117, 118], we introduce the ASV wrapper specification as object-based, real-time *probabilistic* rewrite theories and discuss the assumptions under which statistical model checking and quantitative analysis of the kind discussed in Chapter 6 can be correctly applied. We then demonstrate how these wrappers can be used to analyze availability properties of services in arbitrary service compositions using service orchestrations in Orc.

7.1 Modular DoS Protection Using the ASV Protocol

The ASV protocol identifies three main roles of nodes in a communication system: (1) a server, which attempts to service incoming requests, (2) a client, which represents a legitimate user requesting services from the server at some predictable, reasonable rate, and (3) an attacker, which poses as a legitimate user and tries to flood the server with fake requests to deny service to clients. As explained in [105], and briefly reviewed in Section 6.3, in the presence of an attack, a legitimate client attempts to adapt to the current level of attack, which is estimated by the prolonged absence of a response from the server, by exponentially replicating its requests (up to a threshold) as the sensed severity of attack increases. At the other end, the server implements a reservoir-sampling algorithm to collect a random sample of the incoming requests and process them at its mean processing rate.

Therefore, when the roles of servers and clients in a communication system are identifiable, the ASV behavior can be naturally applied as a DoS defense layer over nodes in the system against adversaries probabilistically sharing communication channels with legitimate clients according to the shared channel model. This layer of protection can be achieved by defining both *server ASV wrappers*, which implement the ASV server reservoir sampling algorithm for incoming requests, and *client ASV wrappers*, which implement the ASV client adaptive exponential replication strategy. The server ASV wrapper maintains a buffer with which

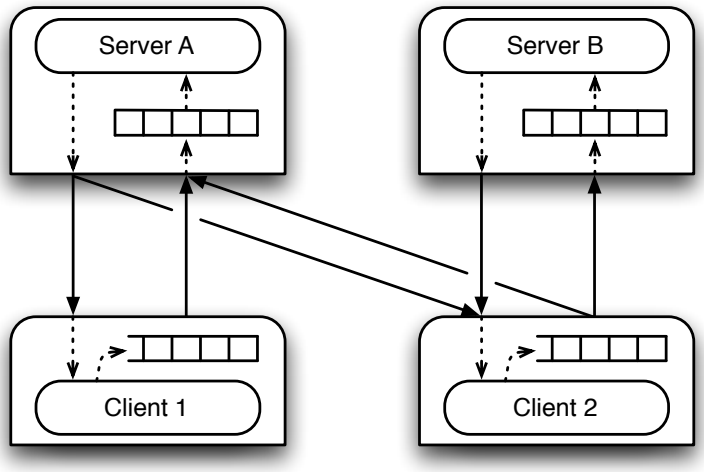


Figure 7.1: An example of an ASV-wrapped communication system

random sampling of incoming requests is performed. Once the wrapper times out, it forwards all the messages in the buffer to the underlying server. Server responses are forwarded to clients as soon as they become available. The Client ASV wrapper, on the other hand, maintains a message queue for outgoing requests. When the underlying client sends out a request, the request is placed in the wrapper queue in preparation for replication. A request remains in the queue until either it is serviced or its retrial span is exhausted. Server responses for pending requests in the queue are forwarded immediately to the client. Figure 7.1 depicts an example communication pattern of ASV-wrapped servers and clients, in which Client 1 requests services from Server A, while Client 2 uses services from both Server A and Server B.

Below, we describe formally the components and behaviors of generic ASV wrappers and how they can be used.

7.2 Assumptions on the Underlying Language

To be able to specify ASV wrappers that are both *generic* and *amenable to statistical analysis* of the kind discussed in Chapter 6, we make a few, reasonable assumptions about the language in which the underlying communication system is specified as follows. We assume that the language is specified as a (possibly probabilistic) rewrite theory $\mathcal{R} = (\Sigma, E \cup A, R, \phi)$.

The equational theory $(\Sigma, E \cup A)$ defines the following sorts and data structures:

1. A sort Configuration, which is constructed by an associative and commutative empty juxtaposition operator with the constant none : $\rightarrow$ Configuration as its identity element:

$$\_\_ : \text{Configuration} \times \text{Configuration} \rightarrow \text{Configuration} \ [\text{assoc comm id: none}]$$

   In addition, we assume two sorts: Object and Msg, which are subsorts of Configuration, for objects and messages, respectively. Intuitively, a term of sort Configuration represents the state of a system as a multiset of objects and messages in transit. In addition, objects within a configuration are assumed uniquely identifiable using terms of sort Oid of object identifiers. These data structures and their properties are already assumed when using Maude's object-based programming facilities [20].

2. Two subsorts, CMsg and SMsg, of the sort Msg, representing, respectively, *client request* and *server response* messages, which can be formed using the following (subsort-overloaded) constructors:

$$\_ \leftarrow \_ : \text{Oid} \times \text{CContent} \rightarrow \text{CMsg} \qquad \_ \leftarrow \_ : \text{Oid} \times \text{SContent} \rightarrow \text{SMsg}$$

   where the sorts CContent and SContent (which are subsorts of the sort Content), model client and server message payloads, respectively. What actually constitutes a term of the sort Content or any of its subsorts is application-specific.

3. A sort MsgID for globally unique message identifiers, which will be used by the ASV wrappers to match server response messages to the corresponding client request messages that prompted them. We assume that messages produced or consumed by the system configuration are encapsulated with a message identifier and a sequence number using the following constructors:

$$[\_, \_, \_] : \text{Nat} \times \text{MsgID} \times \text{CMsg} \rightarrow \text{EncapCContent}$$
$$[\_, \_, \_] : \text{Nat} \times \text{MsgID} \times \text{SMsg} \rightarrow \text{EnacpSContent}$$

where the sorts EncapCContent and EnacpSContent are subsorts of both the sort Msg, as messages in the underlying system state, and the sort Content, as payloads of (wrapped) messages at the level of ASV wrappers. The sequence numbers used in encapsulated messages define a linear order in which these messages can be processed by the wrappers (see below).

In addition to the structural assumptions (1) – (3) above, we also have to make a behavioral assumption about the semantics of the underlying language, so that systems specified in this language do not exhibit any non-deterministic behavior that is *not* probabilistically quantified. In particular, we assume that the semantics of the language is either *deterministic* (optionally specified entirely by the equations E in $\mathcal{R} = (\Sigma, E \cup A, R, \phi)$), or *probabilistic* such that any potential non-determinism is resolved by a probabilistic choice. This implies that (encapsulated) messages (terms of the sort EncapCContent or EnacpSContent) can only be introduced in the system configuration or consumed by objects in the configuration according to the deterministic or probabilistic semantics of the language. Consequently, we may assume two linear orderings: one ordering on the outgoing messages produced by an object in the configuration, and the other linear ordering on the messages to be consumed by an object in the configuration. The orderings are assumed to be maintained by the configuration of each object to be wrapped, and are exposed to the wrappers using the sequence number field of an encapsulated message, which is represented as a natural number. Therefore, a sequence of internal (possibly probabilistic) transitions of an object $\langle O : CID \mid AS \rangle$ in the system may have the form:

$$\begin{aligned} \langle O : CID \mid AS \rangle \; &\rightarrow^{+}_{R/E \cup A} \langle O : CID \mid AS' \rangle \; [N, I_{M_i}, M_i] \\ &\rightarrow^{+}_{R/E \cup A} \langle O : CID \mid AS'' \rangle \; [N, I_{M_i}, M_i] \cdots [N+k, I_{M_{i+k}}, M_{i+k}] \end{aligned}$$

where the order in which the messages are produced is captured by their assigned numbers from the sequence $[N, \cdots, N+k]$. In general, at any given point in time, the configuration may contain a set of *outgoing* messages that are waiting to be forwarded to their target objects ordered by a sub-sequence of the natural numbers $[N^{low}_{out}, \cdots, N^{high}_{out}]$, where N^{low}_{out} denotes the sequence number of the next message to be forwarded, and N^{high}_{out} denotes the

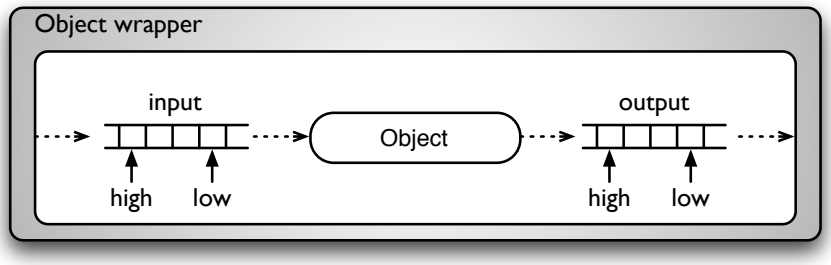


Figure 7.2: Input and output queues of a wrapped object simulated by sequence numbers

sequence number of the next output message to be produced by the configuration.

Similarly, the configuration may contain a set of *incoming* messages that are waiting to be consumed by the target object ordered by a sub-sequence of the natural numbers $[N_{in}^{low}, \cdots, N_{in}^{high}]$, where N_{in}^{low} denotes the sequence number of the next message to be consumed by the object, and N_{in}^{high} denotes the sequence number of the next input message to be forwarded to the object.

As Figure 7.2 illustrates, each pair of sequence numbers, $(N_{in}^{low}, N_{in}^{high})$ and $(N_{out}^{low}, N_{out}^{high})$, defines a *moving window* in an infinite queue of messages, where the *low* sequence number denotes the current index of the front of the queue, and the *high* sequence number denotes the current index of the back of the queue. The queue is empty when the low and high numbers coincide.

7.3 The ASV Wrappers

Given a (probabilistic) rewrite theory $\mathcal{R} = (\Sigma, E \cup A, R, \phi)$ specifying a communication system and satisfying the assumptions above in Section 7.2, the ASV wrappers of $\mathcal{R}$ are specified in a probabilistic rewrite theory $\mathcal{R}_W = (\Sigma_W, E_W \cup A_W, R_W, \phi_W)$ that extends $\mathcal{R}$ with definitions of ASV wrapper objects, declarations of ASV model parameters and interface functions, and with ASV behaviors. This is described next.

Server Wrapper		
$\langle O : SASV \mid$	$conf : C$	the underlying server object configuration
	$reqlist : \bar{M}$	the incoming request buffer
	$reqcnt : P$	the current value of the request count parameter
	$selv : B\rangle$	the selective verification flag
Client Wrapper		
$\langle O : CASV \mid$	$conf : C$	the underlying client object configuration
	$buffer : \bar{M}$	the client request retrial buffer
	$repcnt : K$	the current client replication count
	$gen : O'$	the corresponding client generator
	$rep : B\rangle$	the client replication flag
Client Generator Wrapper		
$\langle O : GASV \mid$	$id : O'$	the object identifier of the client to be generated
	$conf : C\rangle$	the object configuration of the client
Attacker Wrapper		
$\langle O : AASV \mid$	$conf : C$	the underlying attacker object configuration
	$buffer : \bar{M}\rangle$	the attacker's attack buffer

Table 7.1: The ASV wrapper classes

7.3.1 ASV Wrapper Configurations

The theory $\mathcal{R}_W$ declares four classes of ASV wrapper objects: server wrappers, client wrappers, client generator wrappers, and attacker wrappers, which, in some sense, generalize the classes of objects declared in the ASV model described in Section 6.3. The four classes and their fundamental class attributes are summarized in Table 7.1.

Server Wrapper. The server wrapper object maintains in an attribute named *conf* a server object configuration (of the sort Configuration), which is assumed to contain the underlying server object as well as any supporting objects required for the proper behavior of the server as defined by the language in which the system is specified. The server wrapper also includes: (1) a buffer attribute *reqlist* that holds incoming requests between server timeouts, (2) a request count attribute *reqcnt* that is used in determining the probability of accepting an incoming request when the buffer is full, and (3) a selective verification

Boolean flag *selv*, which can be used to switch selective verification on and off. In addition to client request messages, a server wrapper object, with id O, accepts two kinds of internal scheduling messages: (i) a timeout message $O \leftarrow timeout$, which signals a new server time window causing the processing of messages in the buffer, and (ii) an internal poll message $O \leftarrow poll$, which signals a check for possible internal transitions within the underlying configuration and results in encapsulating and lifting up any outgoing server responses to the wrappers level.

Client Wrapper. A client wrapper object maintains a client object configuration, which is assumed to contain the underlying client object and any supporting objects for its proper semantics. Additionally, a client wrapper maintains a client replication buffer attribute *buffer*, which stores a queue of pending client requests that have not yet been serviced. The client buffer is constructed using an associative juxtaposition operator with empty syntax and with *nil* as an identity element. An entry in the buffer is a pair of the form $\{I, CMsg\}$, where I is the current retrial count for the buffered client message *CMsg*. Other client wrapper attributes include the current (global) client replication count *repcnt*, which is required to implement the adaptive protocol, and a Boolean replication flag, which can be used to turn the adaptive replication behavior on or off. In addition to incoming server responses, a client wrapper object, identified by O, accepts two kinds of internal scheduling messages, which are syntactically identical to those described above for server wrappers but have different semantics. These messages are: (i) an internal poll message $O \leftarrow poll$, which signals a check for possible internal transitions within the underlying configuration causing any client request messages in the configuration to be lifted up and stored in the wrapper's request queue, and (ii) a timeout message $O \leftarrow timeout$, which initiates the process of appropriately replicating the next message in the queue, when the queue is non-empty.

Client Generator Wrapper. Like the generator objects of the ASV model described in Section 6.3, the generator wrapper class is a *utility* wrapper that is declared essentially for analysis purposes to model new clients coming in at a specified rate. It maintains the next object identifier and the next configuration of the client object to be generated in the attributes *id* and *conf*, respectively. The client configuration maintained is assumed to contain the underlying client object and any of its supporting objects. The generator

wrapper accepts a message of the form $O \leftarrow \mathit{spawn}$ to schedule the creation of the next client object. Note that, unlike the ASV model in Section 6.3, in which only one generator object is used, there can in general be more than one instance of the generator wrapper class, as we will see in the Orc example in Section 7.4.

Attacker Wrapper. The attacker wrapper is another utility wrapper declared for analysis purposes. The wrapper is a simple class that maintains only two attributes: (1) the attacker configuration in an attribute *conf*, which holds the underlying attacker object configuration defining the attacker behavior, and (2) an attack request queue, having a structure similar to a client request buffer (described above), in an attribute *buffer*. The queue, which holds the attacker's (fake) requests, serves as a launching pad for the attack. An attacker wrapper ignores incoming server responses and accepts the same two kinds of internal messages accepted by the client wrapper with similar semantics.

7.3.2 Parameters and Interface Functions

To support the generic nature of the ASV wrappers, the theory $\mathcal{R}_W$ declares a number of operators representing (partial) functions that capture parameters of the ASV model and define an interface with the underlying communication system. First, the ASV wrappers assume a simple object identification scheme that maps object identifiers O in the underlying configuration to object identifiers at the wrappers layer of the form $f(O)$, where f is one of the four constructor symbols: *sv*, *cv*, *gv*, *av*, for server, client, generator and attacker object wrapper identifiers, respectively. Clearly, uniqueness of wrapper object identifiers is guaranteed by the assumption of uniqueness of wrapped object identifiers in the underlying configuration, which are constructor terms of the sort Oid. This object identification scheme is essential to the proper semantics of the wrappers, as it impacts how messages are processed across the two different layers.

Furthermore, the ASV wrappers define several ASV parameters and interface functions, which are listed in Table 7.2, where the first set of declarations captures the ASV model parameters, and the second set the interface functions. We note that, since we may have multiple server, client or generator wrapper objects in a configuration, some of the ASV

μ : Oid		$\rightarrow$ [Float]	mean processing rate of a server
$\rho_{\min}, \rho_{\max}$: Oid		$\rightarrow$ [Float]	min and max request rates of a client
$\alpha_{\min}, \alpha_{\max}$:		$\rightarrow$ [Float]	min and max attack rates
Timeout : Oid		$\rightarrow$ [Float]	timeout period of a given server or client
JBound : Float $\times$ Float		$\rightarrow$ [Nat]	client retrial span given ρ and α
Delay :		$\rightarrow$ [Float]	message transmission delay
Drop :		$\rightarrow$ [Float]	message drop probability
Limit :		$\rightarrow$ [Float]	time duration of a sample run
get_X^L : Configuration		$\rightarrow$ [Nat]	get the value of a sequence number
set_X^L : Configuration $\times$ Nat		$\rightarrow$ [Configuration]	set the value of a sequence number
eager : Configuration		$\rightarrow$ [Bool]	is a configuration eager?
nextOid : Oid		$\rightarrow$ [Oid]	next client object id to be generated
nextConf : Configuration		$\rightarrow$ [Configuration]	next client configuration to be generated
nextID : MsgID		$\rightarrow$ [MsgID]	next client message id to be used
nextCC : CContent		$\rightarrow$ [CContent]	next client message content to be used

Table 7.2: The ASV wrapper interface functions

model parameters are parametric to wrapper object identifiers, including $\mu(O)$, which defines the mean processing rate of the server wrapped by O, $\rho_{\min}(O)$ and $\rho_{\max}(O)$, which define the minimum and maximum client request rate for the client object wrapped by O, and $timeout(O)$, which defines the server timeout window, if O identifies a server wrapper object, or the client timeout period, if O identifies a client wrapper object.

The interface functions decouple the specification of the wrappers from the specification of the underlying system by hiding some of the system specification details. In particular, the theory $\mathcal{R}_W$ declares a set of four *getter* and four *setter* functions for the message sequence numbers maintained by the configuration, which are of the forms get_X^L and set_X^L, with $X \in \{in, out\}$ for input and output messages to the underlying configuration, and $L \in \{low, high\}$ for the low and high sequence number handlers (as described in Section 7.2). For instance, given a system configuration C, the function call $get_{in}^{high}(C)$ returns the sequence number $N_{in}^{high} + 1$ to be used for the next input message to the configuration, while the call $set_{out}^{low}(C, K)$, with K a natural number, returns the configuration C with its sequence

number N_{out}^{low} of the next output message to be processed by the enclosing wrapper set to K. In addition, the theory declares a function *eager*, which, given a configuration C, evaluates to *true* if and only if the configuration has an enabled transition.

Interface functions also provide added flexibility in the specification and analysis of an ASV-wrapped system by giving further control over its possible behaviors. In particular, the functions *nextOid* and *nextConf* can be used to specify how new client objects are introduced in the configuration by a generator wrapper, while the functions *nextID* and *nextCC* can be used to define how the attacker generates fake requests to the server. The example discussed in Section 7.4 illustrates how these parameters and interface functions can be instantiated.

7.3.3 Wrapper Behaviors

Like the ASV model in rewriting logic described in Section 6.3, the configuration of ASV wrappers maintains a few other objects required for defining the wrappers behaviors, including, most notably, a global scheduler object of the form $\{T \mid S\}$, with T the current global time, and S a list of scheduled, *encapsulated* messages, of the form $[T, M, B]$, where T is the time at which the message is scheduled for delivery, M the encapsulated message, and B a message drop flag for modeling lossy channels. As before, the scheduler object provides two important benefits: (1) it enables resolving any potential non-determinism by enforcing a deterministic order on message delivery, and (2) it provides a flexible mechanism for specifying time and its effects in a configuration and controlling the time length of a simulation of the model using an auxiliary function, *mytick*, as described in Section 6.3.

The behaviors of the different wrappers are explained below, where we show slightly simplified versions of the rewrite rules to improve readability and to highlight the important aspects of wrapper behaviors.

Server Wrapper. When a server wrapper O receives an incoming client message $O \leftarrow [N, ID, CMsg]$, it first checks whether its request buffer stored in the *reqlist* attribute is full or not, where the buffer size is given by $\lfloor \mu(O) \cdot timout(O) \rfloor$. If the buffer is not yet full, the request is simply added to the list. Otherwise, if the buffer has already reached its maximum

capacity, the server tosses a biased coin with success probability $\mathcal{A}(P)$, parametrized by its *reqcnt* attribute P, and uses the outcome of this experiment to decide whether to either replace an existing request selected uniformly at random with the incoming request, or to drop the incoming message altogether. The server wrapper also increments its client request count *reqcnt* in preparation for the next incoming client request. This behavior is modeled by the following probabilistic rewrite rule:

ProcessReq :

$\langle O : SASV \mid reqlist : L, reqcnt : P \rangle \;\; O \leftarrow [N, ID, CMsg] \;\; \{T \mid S\}$

$\rightarrow mytick(\{T \mid S\})$

if the request buffer L is full then

if B then $\langle O : SASV \mid reqlist : replace(L, [N, ID, CMsg], U), reqcnt : P + 1 \rangle$

else $\langle O : SASV \mid reqlist : L, reqcnt : P + 1 \rangle$ fi

else $\langle O : SASV \mid reqlist : add(L, [N, ID, CMsg], reqcnt : P \rangle$ fi

with probability $B := Bernoulli(\mathcal{A}(P)) \;\wedge\; U := Uniform(|L|)$

Once a server wrapper times out, indicated by the self-addressed *timeout* message, the server resets its *reqcnt* counter to the value $\lfloor \mu(O) \cdot timout(O) \rfloor + 1.0$, which is specified using the function *resetP*, and forwards all the messages in its buffer (if any) down to the enclosed configuration using the auxiliary function call $emitMsgs(L, N)$, which emits an ordered set of messages in L starting with the next sequence number N provided by the configuration through the function get_{in}^{high}.

STimeout :

$\langle O : SASV \mid conf : C, reqlist : L, reqcnt : P \rangle \;\; O \leftarrow timeout \;\; \{T \mid S\}$

$\rightarrow \langle O : SASV \mid conf : set_{in}^{high}(C, N + |L|) \; emitMsgs(L, N), reqlist : nil, reqcnt : resetP(O) \rangle$

$O \leftarrow poll \;\; \{T \mid S\}$

if $N := get_{in}^{high}(C)$

The wrapper also updates the next input message sequence number in the configuration appropriately using the function set_{in}^{high}, and schedules an internal poll message $O \leftarrow poll$

to allow for internal transitions in the underlying configuration, which is captured by the following rule:

$$
\begin{aligned}
&\textsc{SInternal1}: \\
&\langle O : SASV \mid conf : C \rangle \;\; O \leftarrow poll \;\; \{T \mid S\} \\
&\rightarrow \langle O : SASV \mid conf : pullUp(C') \rangle \\
&\quad mytick(\{T \mid insert(insert(S, makeSL(C', T)), [timeout(O) + T, (O \leftarrow timeout), 0])\}) \\
&\text{if} \;\; eager(C) \;\wedge\; C \rightarrow C' \;\wedge\; eager(C') \neq true
\end{aligned}
$$

The rule can be applied when the underlying configuration C is *eager*, as defined by the enclosed system specification, implying that an internal transition is possible. In this case, the configuration C is allowed to make one or more transitions all the way to a configuration C' that is *not* eager, where no internal transition is possible. Therefore, when the condition is satisfiable, the server wrapper's configuration C rewrites to the next non-eager configuration in which all outgoing server response messages are removed and submitted to the scheduler using: (1) the auxiliary functions *pullUp*, which removes available outgoing messages from the configuration according to their sequence numbers and appropriately updates the configuration's output sequence number N_{out}^{low}, and (2) the auxiliary function *makeSL*, which constructs a list of scheduled, encapsulated outgoing messages from the configuration given the current global time. The rule also schedules a *timeout* message to be made available after $timeout(O)$ time units. The other case, when the configuration is not eager, is modeled by a rule labeled SInternal2 (not shown), which simply ignores the poll message and schedules a timeout message. We note that, since the poll message is scheduled to be processed immediately after the timeout message, the two transitions defined by these two messages can in principle be combined into one rewrite transition step. However, we have chosen to define the internal server wrapper behavior as two consecutive transition steps to a achieve a simpler and more readable specification.

Client Wrapper. When it is time for the client wrapper to make an internal transition in the underlying configuration C, which is signaled by the self-addressed *timeout* message, and if C is indeed eager, the client wrapper advances C to the next non-eager state C', and

updates its configuration and buffer attributes according to the following rewrite rule:

$$\begin{aligned}\textsc{CInternal1} : \langle O : CASV \mid conf : C,\ buffer : \bar{E}\rangle\ O \leftarrow timeout\ \{T \mid S\}\\ \rightarrow \langle O : CASV \mid conf : pullUp(C'),\ buffer : makeCB(C', T)\ \bar{E}\rangle\ O \leftarrow poll\ \{T \mid S\}\\ \text{if}\ \ eager(C)\ \wedge\ C \rightarrow C'\ \wedge\ eager(C') \neq true\end{aligned}$$

The rule removes any outgoing client request messages from C' while updating the appropriate sequence number handler (namely N^{low}_{out}), using the function *pullUp*, constructs a buffer of these messages and appends them to the queue, using the function *makeCB*, given the current global time. If the configuration is not eager, a corresponding rule CInternal2 applies, in which the *timeout* message is simply ignored. In either case, both rules, CInternal1 and CInternal2, schedule a *poll* message to initiate the client message replication step. If the client wrapper's buffer is non-empty, the CSend1 rule may fire, which has the following form:

$$\begin{aligned}&\textsc{CSend1} :\\ &rate_\rho(O_g, R)\ rate_\alpha(A)\ \{T \mid S\}\\ &\langle O : CASV \mid conf : C,\ buffer : \bar{E}\{J, [N, ID, O_s \leftarrow CC]\}, repcnt : K, gen : O_g\rangle\ O \leftarrow poll\\ &\rightarrow rate_\rho(O_g, R)\ rate_\alpha(A)\\ &\quad \text{if}\ (J = 0)\ \text{then}\\ &\qquad \langle O : CASV \mid conf : C,\ buffer : \{s(J), [N, ID, O_s \leftarrow CC]\}\bar{E}, repcnt : K, gen : O_g\rangle\\ &\qquad mytick(\{T \mid \text{schedule a timeout msg and } sv(O_s) \leftarrow [N, ID, O_s \leftarrow CC]\})\\ &\quad \text{else if}\ (J \leq JBound(R, A))\ \text{then}\\ &\qquad \langle O : CASV \mid conf : C,\ buffer : \{s(J), [N, ID, O_s \leftarrow CC]\}\bar{E}, repcnt : 2 \cdot K, gen : O_g\rangle\\ &\qquad mytick(\{T \mid \text{schedule a timeout msg, replicate } sv(O_s) \leftarrow [N, ID, O_s \leftarrow CC]\ K \text{ times }\})\\ &\quad \text{else}\\ &\qquad \langle O : CASV \mid conf : C,\ buffer : \bar{E}, repcnt : K, gen : O_g\rangle\\ &\qquad mytick(\{T \mid \text{schedule a timeout msg only }\})\ \ \text{fi fi}\end{aligned}$$

The rule considers the next message in the buffer (the head of the queue), having a retrial

count J and targeted to server object O_s. There are three cases. The first case is when J is 0, meaning that the message has not yet been forwarded to the server. In this case, a single copy of the encapsulated message $sv(O_s) \leftarrow [N, ID, O \leftarrow CC]$ is submitted to the scheduler (no replication), and the buffered message is moved to the back of the queue with its retrial count J incremented by 1. The second case is when J is non-zero, but smaller than or equal to the current retrial span limit given by $JBound(R, A)$, a function of the current client rate R and the current attack rate A. In this case, the client wrapper sends K replicated copies of the encapsulated message to the server, with K the value of its *repcnt* attribute, and updates the replication count for the next attempt. Here, again, the message is moved to the back of the message queue with its J incremented. In the third case, when the message's retrial count exceeds the current retrial span limit $JBound(R, A)$, the wrapper gives up on the message and drops it from the queue. In all of the three cases, the wrapper schedules another *timeout* message after $timeout(O)$ time units. The specification also includes a corresponding rule, namely CSend2, for the case when the buffer is empty, which simply ignores the current *poll* message.

When a client wrapper receives a server response corresponding to a pending request in its buffer, which is determined by matching the messages' identifiers, the wrapper consumes the message and forwards it to the underlying configuration while properly updating the incoming message sequence number handlers, using the function *pushDown*, according to the following rule:

CRec1 :
$\langle O : CASV \mid conf : C,\ buffer : \bar{E}_1\{J, [N, ID, CMsg]\}\bar{E}_2, repcnt : K\rangle$
$O \leftarrow [N', ID, SMsg]\ \ \{T \mid S\}$
$\rightarrow \langle O : CASV \mid conf : pushDown(C, [N', ID, SMsg]),\ buffer : \bar{E}_1\bar{E}_2, repcnt : \lceil K/2 \rceil\rangle$
$mytick(\{T \mid S\})$

This rule uses matching modulo associativity and identity to match the ID field of the incoming message with the appropriate buffered message in the queue. The rule drops the matched buffered message from the queue and updates the client replication count by

dividing it by 2 in response to the successful completion of the request. If the client wrapper receives a server response for a request that is no longer in the buffer, implying that the response corresponds either to a request that has already been serviced (a duplicated response message) or to a request that has already failed (a response that is too late), the client wrapper simply ignores the message. This is captured by a corresponding rewrite rule labeled CRec2.

Generator Wrapper. The behavior of the generator is modeled by a single rewrite rule of the form:

SpawnClient :

$\langle O_g : GASV \mid id : O, conf : C \rangle \;\; O_g \leftarrow spawn \;\; rate_\rho(O_g, R) \;\; \{T \mid S\}$

$\rightarrow \langle O_g : GASV \mid id : nextOid(O), conf : nextConf(C) \rangle \;\; rate_\rho(O_g, R)$

$\langle cv(O) : CASV \mid conf : C, \;\; buffer : \phi, repcnt : 1, gen : O_g, \cdots \rangle$

$O \leftarrow poll \;\; \{T \mid$ schedule in S the next spawn message $\}$)

When the $O_g \leftarrow spawn$ message is the next message to be processed, the generator wrapper object O_g creates a new client wrapper object $cv(O)$ that is initialized with the configuration C, an empty buffer, and a replication count of 1. The generator wrapper updates its client object id and client configuration for the next client wrapper to be generated, using the interface function *nextOid* and *nextConf*, and schedules the next *spawn* message.

Attacker Wrapper. The attacker wrapper assumes a similar behavior to client wrappers, in that an attacker wrapper object first, upon receiving a *timeout* message, makes an internal transition (if possible) to pull up client request messages from the underlying configuration and store them in its message queue, according to the rules labeled AInternal1 and AInternal2 (not shown), corresponding, respectively, to client wrapper rules CInternal1 (shown above) and CInternal2. Then, using a subsequent *poll* message, the attacker wrapper forwards a copy of the next message in its (non-empty) queue to the

appropriate server wrapper according to a rule of the following form:

ASend1 :
$\langle O : AASV \mid \mathit{buffer} : \bar{E}\{J, [N, ID, O_s \leftarrow CC]\}\rangle \;\; O \leftarrow \mathit{poll} \;\; \mathit{rate}_\alpha(A) \;\; \{T \mid S\}$
$\rightarrow \langle O : AASV \mid \mathit{buffer} : \{s(J), [N, \mathit{nextID}(ID), O_s \leftarrow \mathit{nextCC}(CC)]\}\bar{E}\rangle \;\; \mathit{rate}_\alpha(A)$
$\mathit{mytick}(\{T \mid$ schedule in S a timeout msg and $\mathit{sv}(O_s) \leftarrow [N, ID, O_s \leftarrow CC]\ \})$

In this rule, the attacker wrapper encapsulates the request message appropriately and submits it to the scheduler. The wrapper also updates the message's identifier (to generate uniquely distinguishable copies of the same request), and optionally the message's content, using the interface functions *nextID* and *nextCC*. A next *timeout* message is also scheduled using the current attack rate A. Another rewrite rule labeled ASend2, which ignores the wrapper's *poll* message, applies when the queue is empty. Finally, the attacker wrapper ignores incoming server responses, regardless of where they come from, which is modeled by a simple rewrite rule labeled ARec.

7.4 Case Study: Availability Analysis in a Service Composition Pattern in Orc

To illustrate how the generic ASV wrappers can be used, we describe a case study in which we analyze service availability in an Orc service composition. The aim of this case study is twofold: (1) to provide a concrete instantiation of ASV wrappers and the type of analysis that can be performed, and (2) to illustrate a formal method for analysis of availability of services in Orc service compositions. Due to the generality and flexibility of both the ASV wrappers, and the Orc language, much more complex case studies can be designed and analyzed in many different ways. The case study presented here was designed for simplicity to emphasize these two goals.

7.4.1 The Underlying Language

Being a concurrent programming model, the semantics of an Orc orchestration expression may naturally be non-deterministic, using either symmetric parallel composition or asymmetric parallel composition (*angelic* and *demonic* types of non-determinism, respectively, as explained in [35]). Consequently, the SOS semantics of Orc as originally defined in [35], and the rewriting semantics introduced in Chapters 3 and 4 cannot be directly used as the semantics of the language in which the underlying system of an ASV wrapper object is specified, if we are to apply the statistical model checking methods described in Chapter 6, which assume that all non-determinism has been removed and the model is purely probabilistic. One possible solution is to develop and use a *probabilistic* semantics of Orc, in which non-determinism is replaced by probabilistic choice (cf. [119]). However, this development is outside the scope of this work, and is, furthermore, *unnecessary*, since the object-level concurrency inherent in the object-based semantics of of Orc, given by $\mathcal{R}_{Orc}$, already provides a semantically equivalent alternative to (symmetric) parallel composition in Orc, as discussed in Section 4.1.4. The non-determinism resulting from object-level concurrency can then be readily eliminated using the wrapper-level scheduler.

Therefore, we use here a *sequential* subset of Orc, denoted $\overset{\rightsquigarrow}{\text{Orc}}$, which is Orc without the parallel composition operators | and <-< . The object-based semantics of $\overset{\rightsquigarrow}{\text{Orc}}$ is captured by a theory $\mathcal{R}_{\overset{\rightsquigarrow}{\text{Orc}}} \subset \mathcal{R}_{Orc}$, a sub-theory of the $\mathcal{R}_{Orc}$ that excludes operator declarations and semantic equations pertaining to parallel compositions[2].

To support the assumed messaging behavior of ASV wrappers, discussed in Section 7.2, the theory $\mathcal{R}_{\overset{\rightsquigarrow}{\text{Orc}}}$ includes sort and operator declarations for messages identifiers, sequence number handlers, and encapsulated messages. A message identifier in $\mathcal{R}_{\overset{\rightsquigarrow}{\text{Orc}}}$ is a pair of the form $id(O, H)$, where O is the Orc object identifier of the caller object and H is the handle name of the site call in O that caused the message. Sequence numbers in an Orc configuration in $\mathcal{R}_{\overset{\rightsquigarrow}{\text{Orc}}}$ are maintained by sequence number handlers of the form $in(N_{in}^{low}, N_{in}^{high})$ for input messages and $out(N_{out}^{low}, N_{out}^{high})$ for output messages, so that each object maintains the appropriate

[2]Currently, only the *untimed* sequential subset of Orc is used. This limitation is compensated for by timing out self-addressed messages at the wrappers layer. Extending wrappers to support real-time behaviors of underlying objects is part of future work.

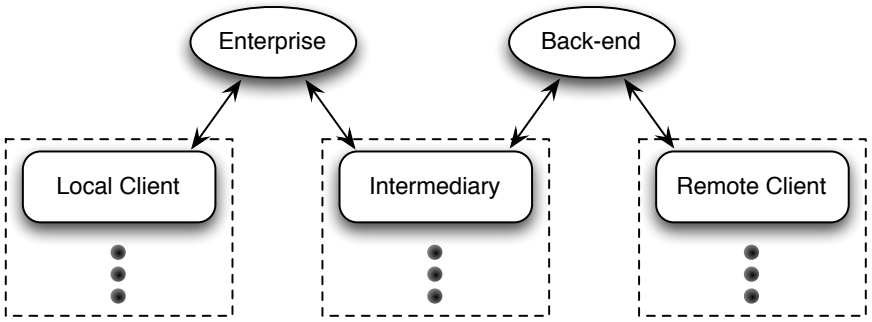


Figure 7.3: The underlying Orc orchestration pattern of the case study

sequence numbers in its configuration when consuming an incoming message or producing and outgoing message, as explained in Section 7.2 above.

7.4.2 The Orc Orchestration Pattern

The Orc orchestration example underlying the communication system to be analyzed is depicted in Figure 7.3, where round rectangles represent Orc expression objects and ellipses represent Orc site objects. The example models a common pattern of communication in service-based systems, in which a group of enterprise clients (*Local Clients* in Figure 7.3) use services provided by a local server in the enterprise (*Enterprise*). The enterprise server is also expected to simultaneously serve other users beyond the boundaries of the enterprise (*Intermediary*), which compose these services with services provided by other back-end service providers (*Back-end*) to complete a given task. Other clients (*Remote Clients*) who do not depend on services from the enterprise, but use services from the back-end server, also exist.

In the Orc program specification, a local client is modeled by an Orc expression object that attempts to make a site call to the enterprise server, represented by a simple site object, with object identifier *ep*, that simply echos back the actual parameter list of the site call. The initial configuration of the ith local client object, given by an operator $localConf(i)$, is of the form:

$$\langle local_i : \mathsf{Expr} \mid env : \emptyset, exp : ep(),\ hdl : \emptyset \rangle \ \ in(0,0) \ \ out(0,0)$$

Since there may be more than one local client Orc expression object present, the enterprise Orc site object may receive multiple, distinct messages corresponding to site calls from local clients. The intermediary Orc expression object, on the other hand, makes three sequential site calls using the Orc expression $ep() \gg be() \gg ep()$, which calls the enterprise server first and waits for a response, calls and waits for the back-end server (with object identifier *be*), and then calls the enterprise server again. The initial state of the ith intermediary expression object has the form:

$$\langle interm_i : \mathsf{Expr} \mid env : E() \triangleq ep() \gg be() \gg ep(), exp : E(),\ hdl : \emptyset \rangle \quad in(0,0) \quad out(0,0)$$

Finally, a remote client is modeled by an Orc expression object that makes a site call to the back-end server, and its specification is similar to that of the local client expression object.

7.4.3 Availability Analysis

In the instantiation of ASV wrappers to $\overset{\rightsquigarrow}{\text{Orc}}$, a server ASV wrapper encloses an Orc site object configuration, whereas a client ASV wrapper encloses an Orc expression object. In addition, a client generator ASV wrapper may optionally be used to generate new instances of (wrapped) Orc expression objects of a certain type, such as local client objects, and an attacker wrapper may be used to simulate an instance of an attack.

In this example, we use three generator wrappers, one for each of the three Orc expression objects: local clients, intermediaries, and remote clients, to model clients requesting services at some specified request rates, and two server wrappers, one for each of the two Orc site objects: *ep* and *be*. Assuming that the initial configurations for the Orc site and expression objects described above are constructed using the respective defined functions $epConf$, $beConf$, $localConf_i$, $intermConf_i$ and $otherConf_i$, the initial wrapper-level configuration (with no attackers) is a term of the following form (the scheduler and other supporting

$\mu(sv(ep)) = 200$	$Timeout(sv(ep)) = 0.4$
$\mu(sv(be)) = 50$	$Timeout(sv(be)) = 0.4$
$\rho(gv(local)) = 0.02$	$Timeout(cv(local)) = 0.4$
$\rho(gv(interm)) = 0.04$	$Timeout(cv(interm)) = 0.4$
$\rho(gv(remote)) = 0.16$	$Timeout(cv(remote)) = 0.4$

Table 7.3: An instantiation of the ASV parameters

objects are not shown):

$$\langle sv(ep) : \mathsf{SASV} \mid conf : epConf, reqcnt : resetP(sv(ep)),\ reqlist : nil, \cdots \rangle$$

$$\langle sv(be) : \mathsf{SASV} \mid conf : beConf, reqcnt : resetP(sv(be)),\ reqlist : nil, \cdots \rangle$$

$$\langle gv(local) : \mathsf{GASV} \mid conf : localConf_0, id : local_0 \rangle$$

$$\langle gv(interm) : \mathsf{GASV} \mid conf : intermConf_0, id : interm_0 \rangle$$

$$\langle gv(remote) : \mathsf{GASV} \mid conf : remoteConf_0, id : remote_0 \rangle$$

To complete the specification of the ASV-wrapped system, we instantiate the ASV model parameters as listed in Table 7.3. For simplicity, we assume no message transmission delays or drops ($Delay = Drop = 0.0$), and a simulation length of 10.0 time units.

In addition, we define the ASV wrapper interface functions given in Table 7.2 as follows. The *eager* function coincides with the *eager* predicate for Orc configurations, defined in Chapter 3, so that a configuration is eager if and only if an instantaneous Orc action is enabled. The definitions of the getter and setter functions for sequence number handlers in an Orc configuration are trivial. The functions *nextOid* and *nextConf* simply increment the appropriate object identifier counters to create fresh new identifiers to be used by the generator wrappers, while the functions *nextID* and *nextCC*, which are used by attackers, generate unique message identifiers and message contents by using fresh handle names. For example, the function *nextOid* has the defining equation $nextOid(local_i) = local_{i+1}$, and the function *nextID* is defined as $nextID(id(O, h_i)) = id(O, h_{i+1})$.

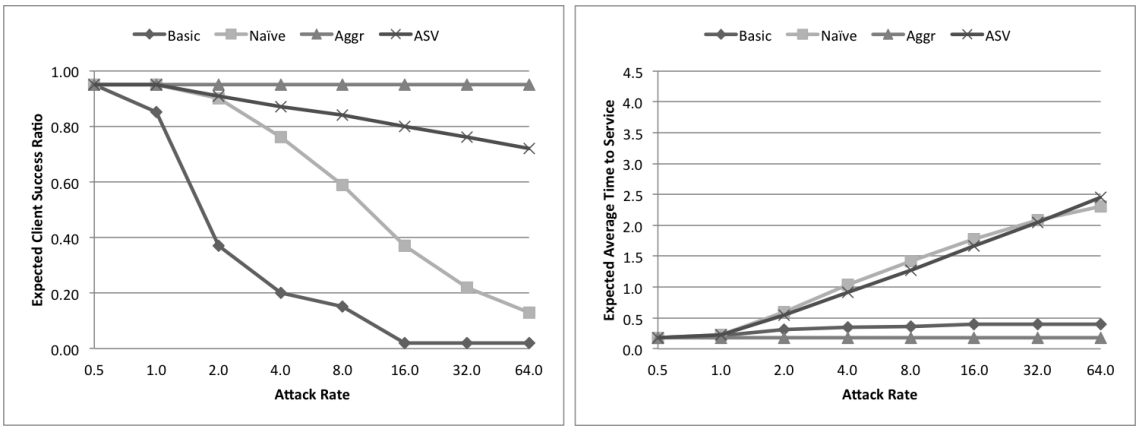
To be able to compare the performance of ASV to other non-adaptive schemes in this

setting, we develop variations of the ASV wrappers corresponding to the *naive* selective verification protocol and the *aggressive* selective verification protocols discussed in Section 6.3.5, in addition to a third variation in which no selective verification is performed, referred to below as the *basic* variant.

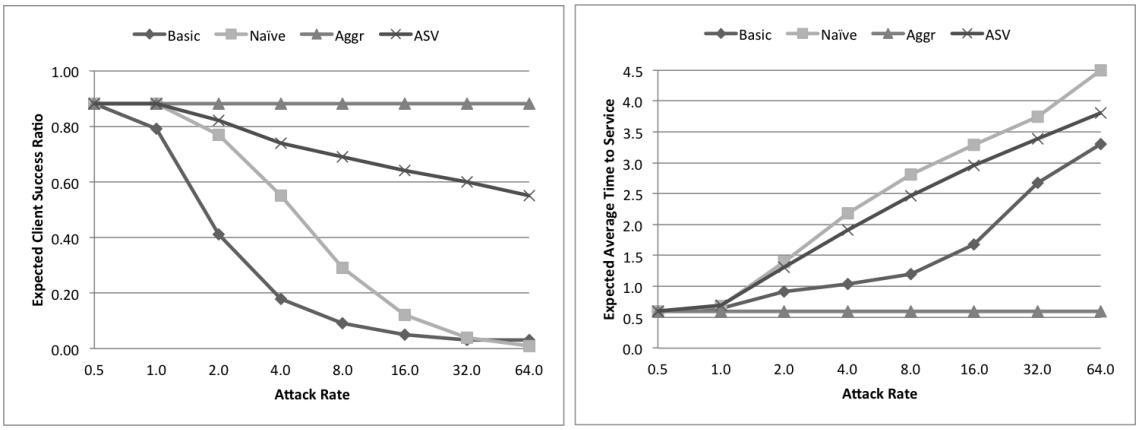
In our analysis, we consider two different attack scenarios with varying attack rates: attacks on the enterprise server *ep*, and attacks on the back-end server *be*. As in the ASV analysis reported in Section 6.3, we analyze the system in terms of three classes of properties: (1) *connection ratio*, which is the number of clients successfully served over the total number of clients, (2) *average time to service*, which is the average amount of time, over all successfully served clients, to get served, and (3) *bandwidth usage*, which is measured by the total number of legitimate client requests reaching the server. These properties have identical specifications as QuaTEx expressions as those presented in Section 6.3.3. Unlike the previous analysis of ASV, where we had only one server and one client generator objects (i.e. one type of multiple, identical client objects), these properties are analyzed here for each of the two server wrappers, and the three client generator wrappers. Throughout the experiments, we assume a 95% confidence interval of size at most 0.05. The analysis results are shown in Figures 7.4–7.7.

Figure 7.4 plots the expected client connection ratio and the average time to service against different attack rates for the three types of clients for the first attack scenario. The connection ratio plots for local and intermediary clients (in Figures 7.4(a) and 7.4(b)) show similar results as those obtained before, in which the ASV protocol performs fairly well against extreme rates of attack. All four variants performed slightly less effectively for intermediary clients than for local clients, which is to be expected, due to the more demanding underlying behavior of intermediary clients. The connection ratio plot for remote clients in Figure 7.4(c) shows the effects of the attack on the enterprise server are only barely noticeable, except for the aggressive protocol, which consistently performs suboptimally due to the consistently high replication count on the slower back-end server. The average time to service results reaffirms latency side-effects of ASV as the severity of attack increases. We note that the service latencies suffered by the intermediary clients are more pronounced because of the fact that an intermediary client requests two sequential services from the enterprise server

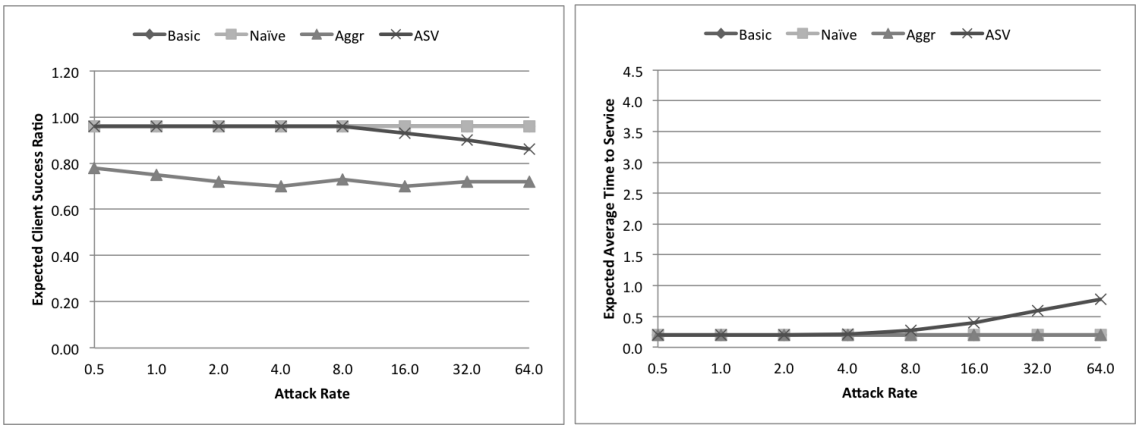
under attack. Figure 7.5 shows a similar, somewhat symmetric, set of results when the back-end server is subjected to the attack. Due to the slower server being attacked, connection ratios are in general lower than in the previous attack scenario. We also note that the average service latency for intermediary clients is less affected by this attack due to their underlying behavior. Finally, legitimate client bandwidth consumption plots in Figures 7.6 and 7.7, show that the ASV behavior performs exceptionally well for both servers, under both attack scenarios. The figures also show that, in moderate to severe attack situations, the bandwidth consumption in the aggressive behavior becomes extremely high compared to the other variants, which is due to the computed (rather than fixed) retrial span using the function *JBound*.



(a) Local Clients

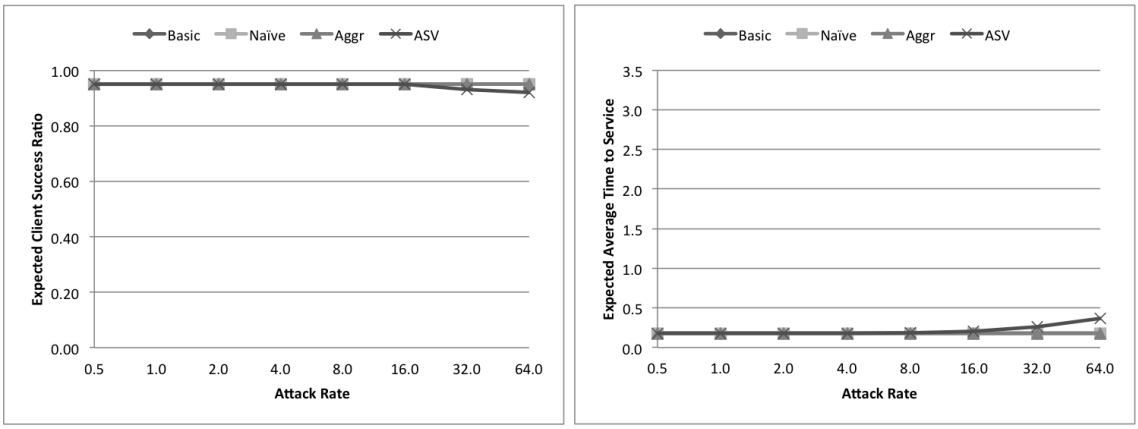


(b) Intermediary Clients

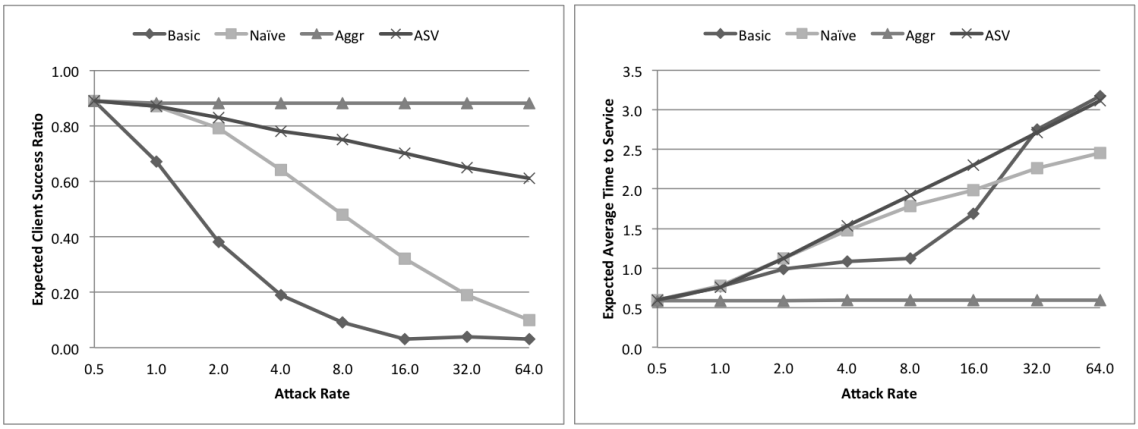


(c) Remote Clients

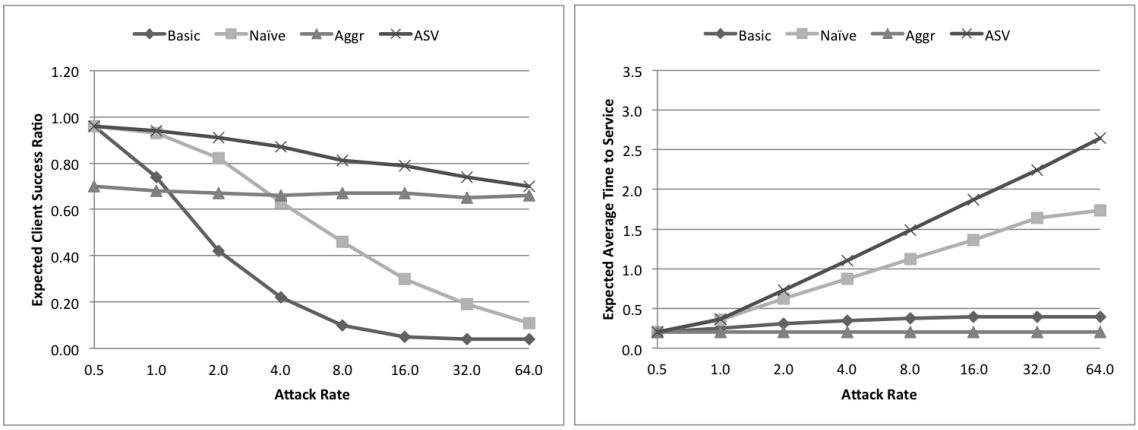
Figure 7.4: The expected connection ratio and average time to service for local, intermediary, and remote clients when attacking the enterprise server



(a) Local Clients

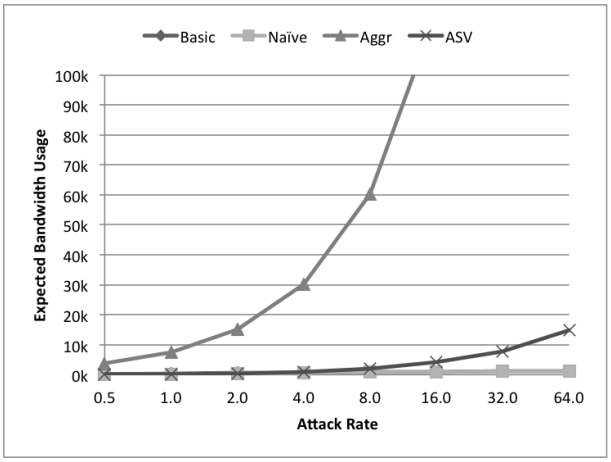


(b) Intermediary Clients

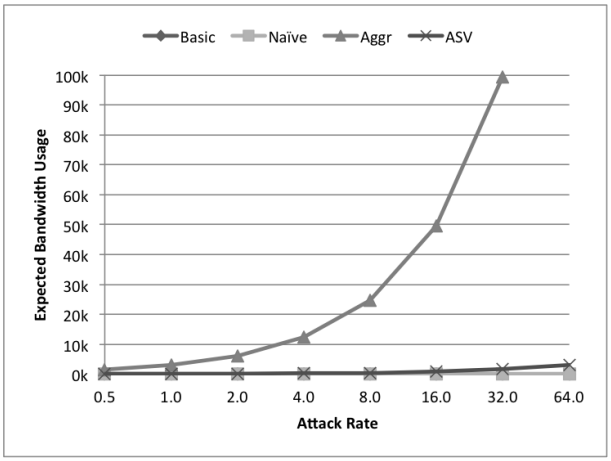


(c) Remote Clients

Figure 7.5: The expected connection ratio and average time to service for local, intermediary, and remote clients when attacking the back-end server

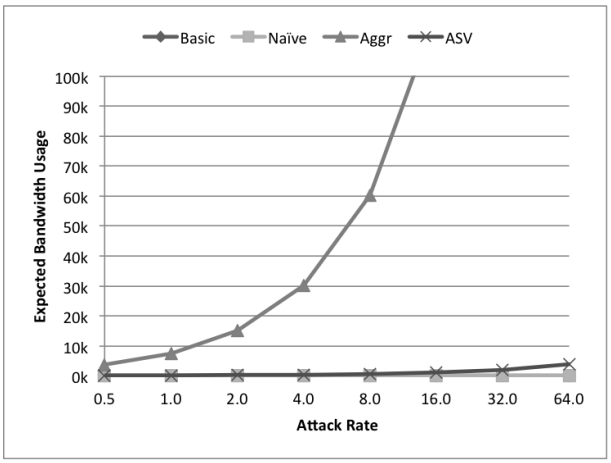


(a) Enterprise server

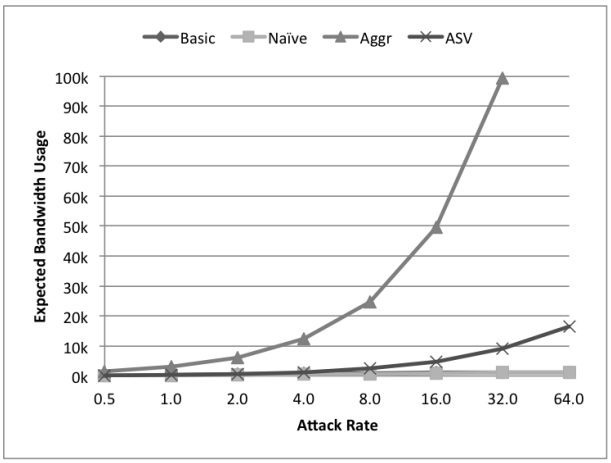


(b) Back-end server

Figure 7.6: Bandwidth usage when attacking the enterprise server



(a) Enterprise server



(b) Back-end server

Figure 7.7: Bandwidth usage when attacking the back-end server

CHAPTER 8

RELATED WORK

In this chapter, we briefly review related work in the different areas to which this dissertation contributes. The intent is not to document every related development in the literature, but rather to highlight fundamentally related, exemplary results. We also provide references to some surveys, where more background information and related cited publications can be found.

8.1 Rewriting Logic Semantics

The fact that rewriting logic is a general semantic framework in which languages and systems can be naturally specified is now well established, as discussed in Meseguer's and Rosu's survey of the *rewriting logic semantics project* [61] as well as in [62], and in the upcoming survey [63]. The generality and flexibility of rewriting logic make it suitable for specifying both deterministic computations (algebraically using equations) and non-deterministic computations (using rewrite rules) within a uniform model, providing a convenient way to control the level of abstraction desired in a specification.

Several recent research projects within the rewriting logic semantics project, which are closely related to the work presented in the thesis, have been conducted (some are discussed in the upcoming survey [63]). For example, a formal framework for the specification and analysis of timing properties in software design based on Real-Time Maude was presented at [120]. The framework uses a flexible intermediate language with timeouts to specify software components in a design, and allows for both static analysis (using abstract interpretations) and dynamic analysis (searching and model checking) of timed properties. Another recent example is a Real-Time-Maude-based tool for specifying and analyzing synchronous, real-time

embedded software systems in Ptolemy II [121, 122]. The tool implements a code generation infrastructure, similar to MOrc's, minimizing exposure to the underlying Maude model. A third example, also for embedded software components, but with emphasis on safety-critical systems, is given in [123], where an object-based, formal semantics of a behavioral subset of AADL in Real-Time Maude is used as a formal analysis back-end for AADL specifications.

8.2 Formal Semantics of Orc

In addition to the operational SOS semantics of Orc in [35] and its timed SOS extension in [81], several denotational formalizations of Orc's semantics, which are more well-suited for reasoning about algebraic properties in the language than for describing the operational behavior of Orc programs, have been developed [84, 82, 124]. Encodings of Orc in some other formal models of concurrency, including encodings in π-calculus [125], Petri nets and the join calculus [126], and networks of timed automata [127] were also given, indirectly providing formal semantics to Orc and highlighting some of its semantic subtleties.

The SOS-based rewriting semantics of Orc, along with some of the operational approaches cited above, has similarities with the various SOS semantics that have been given for different timed process calculi, such as ATP [128] and TLP [129], and real-time extensions to various process calculi, such as extensions of ACP [130, 14], CCS [131], and CSP [132].

8.3 Formal Analysis of Service Compositions

Over the last few years, a considerable amount of foundational research has been conducted to address the problem of formally specifying and analyzing different forms of service compositions, including most importantly service orchestrations, which can be broadly classified into four categories: (1) automata-theoretic, (2) Petri net-based, (3) process-algebraic, and (4) BPEL-oriented approaches. We selectively highlight some of the most relevant results below.

The automata-theoretic approach proposed in [127] leverages available model checking tools for timed automata models, namely Uppaal [133], by modeling the semantics of an Orc

expression as a network of timed automata. Unlike MOrc, the resulting Uppaal-based tool enables formal verification of only a subset of Orc with limited recursion, where the number of threads in an Orc expression is fixed. A fundamentally similar approach, but based on an abstraction of BPEL activities instead of Orc, is used in [134], where an abstracted BPEL process is transformed into a network of Web Service Timed Transition Systems (WSTTS), which are essentially timed automata tailored in design for web service compositions. An implementation of the underlying model enables model checking analysis of timed properties, which are specified in discrete-time Duration Calculus. In comparison to the rewriting-based approach of the thesis, automata-based methods are limited in expressiveness and are, as a result, insufficient for modeling the full generality of service compositions.

Models of service compositions using Petri nets, and their extensions, have been developed. Most of these approaches, such as [135, 136, 137], tend to first define a process calculus in which service composition constructs are specified, and give such constructs formal semantics as Petri nets. Correctness of composition specifications can then be verified using standard reachability analysis methods for Petri net models. A recent, introductory book on modeling business processes using Petri nets is also available [138].

Several other (non-Petri-net-based) process-algebraic approaches to service composition specification were developed. The general theme of these methods is to first specify a process calculus with specialized constructs for the targeted aspects of a service composition, like persistent sessions, error handling (exceptions), or security properties, and then develop their formal semantics in some form of a transition system, on which formal verification is based. Examples of such efforts, many of which were partially supported by the Sensoria project [139], include: (1) Service-Centered Calculus (SCC) [140], an Orc-inspired process calculus of service compositions with persistent sessions; (2) Service Oriented Computing Kernel (SOCK) [141], which defines a layered process calculus for modularly specifying service behaviors, service sessions, and service compositions; (3) the Calculus for Orchestration of Web Services (COWS) [142], a timed process calculus with termination constructs; (4) Event Calculus for Web Services [143], a process calculus with events and event scopes for error handling; and (5) Signal Calculus (SC) [37], a variant of the Ambient Calculus [38] for event-notification-based service coordination. Most of these calculi emphasize expres-

siveness and conciseness by providing constructs that capture different aspects of services, and demonstrate them with examples. It is not clear, however, how mechanizable such formalisms are for performing automatic formal analysis. Bruni [36] provided a fairly recent survey of such process calculus-based approaches, and described one of his own, called the Calculus of Sessions and Pipelines (CaSPiS) [36], which is a calculus for describing service sessions and their interactions.

Other approaches that are based directly on BPEL (and related industrial languages) have also been proposed. Given the fact that BPEL and similar languages are descriptive and verbose, and lack any sort of formal semantics, these approaches essentially try to provide formalizations of (subsets of) these industrial languages in some formal model of computation, like BPEL encodings in Petri Nets [42], the π-calculus [43], Event Calculus [144], and Message Sequence Charts [145]. Alternatively, they may devise new BPEL-inspired *core* languages for service orchestrations that capture some of BPEL's salient features, such as transactions and process termination, including, for example, B*lite* [44], and the BPEL-based process calculi of [146] and [147], focusing on studying correlations between orchestration processes within service choreographies. The manuscript [41] provides a comprehensive (but perhaps fairly outdated) survey of BPEL formalizations. Nevertheless, as a result of BPEL's complexity and expansive feature-set, a comprehensive and practical BPEL-based formal framework for the specification and verification of service orchestrations remains elusive.

8.4 Implementations of Service Composition Languages

There are numerous implementations of service compositions (including implementations in general purpose programming languages, like Java and C#) providing tools and frameworks for designing business processes. Unfortunately, most of these tools are *not* in any way based on formal models against which formal analysis and verification can be carried out, but rely solely on forms of static analysis and traditional non-exhaustive dynamic analysis methods like testing. However, There are a few recent proposals for tools that are based on formal (or at least semi-formal) models. Bruni et. al. [148] described an implementation approach based on abstract state machines (ASMs), in which a formal model of service compositions

is first compiled into the intermediate language of the underlying Service-Oriented Abstract Machine (SOAM), for which a code generator exists. An implementation of Orc, among other service composition formal models, was given using this approach [148]. Compared to our rewriting-based implementation approach in Maude, the ASM approach seems less generic as the underlying SOAM has to be tailored to the specific model to be implemented, which may re-introduce possibilities for implementation errors. Moreover, SOAM's model of services defines specialized constructs for messaging and maintaining multiple program states (corresponding to different services running in parallel) with specialized semantics, which makes it less intuitive than the object-based model with asynchronous message passing employed in our transformation. Finally, it is not clear whether truly distributed implementations can be obtained from their SOAM specifications.

Other approaches to implementations of service compositions, which can be classified under the model-driven software engineering methodology, have also been proposed. These include: (1) VFT [149], which is a tool based on a visual language (with semi-formal semantics) for specifying *semantic* web service compositions in OWL-S [33]; (2) DecSerFlow [150], which is an extensible, graphical language with formal semantics, endowed with LTL model checking; (3) The UML-based specification framework of [151], which is rooted at Hoare's formalism of Communicating Sequential Processes (CSP) [13] to formally check conformance of service compositions with a given choreography specification; and (4) LTSA-WS [152], which is a UML-based specification language, developed as an Eclipse plug-in, in which service composition specifications are translated into labeled transition systems, where verification is performed by trace equivalence.

8.5 Statistical Analysis Methods of Probabilistic Models

Statistical model checking of formal models of probabilistic systems has seen a growing interest in the formal methods community over the last few years, mainly due to its higher scalability and wider applicability, when compared to numerical probabilistic methods. In general, in statistical model checking, the problem of checking whether the probability that a certain property holds is within some specified threshold (expressed as a probabilistic

formula in a probabilistic temporal logic, such as PCTL [106] and CSL [107, 108]) is solved using statistical means, such as statistical hypothesis testing, as described by Sen et. al. [1]. A similar approach based on hypothesis testing is presented in [101, 153], where, instead of pre-computing the sample size and performing a simple hypothesis test as in [1], the authors describe a testing strategy based on *sequential acceptance sampling*, where after each observation, a testing step is performed to decide whether further observations are needed to satisfy the required statistical strength of the result. Although the simple hypothesis testing strategy of [1] is more amenable to parallelization, it is argued in [101] that sequential sampling may reduce generated sample size. Other statistical model-checking algorithms for checking properties of "black-box" systems, in which samples cannot be generated on demand, have been proposed [154, 155].

In statistical quantitative analysis, a quantitative measure of some aspect the probabilistic system is estimated, as demonstrated here and in [2] using QuaTEx and the statistical quantitative analysis algorithm of [2]. A similar algorithm was also described in [104], in which, instead of fixing a sample size N, samples are generated until a standard normality test (e.g. the JB test [156]) succeeds. The standard normal distribution is then used in the computation of the confidence interval.

8.6 Formal Specification and Analysis of Availability

The VeStA tool [72], along with Maude, has been used for statistical model checking and quantitative analysis in several projects, including analysis of TCP SYN floods-based DoS attacks within the shared channel model [102], analysis and redesign of a wireless sensor networks protocol [103], and a few case studies in object-based stochastic hybrid systems [66]. A similar rewriting-based approach using Maude was also described in [104], which was applied to a resource optimization problem in embedded systems.

Other statistical model checking and quantitative analysis tools have been developed. They include PRISM [157], a BDD-based model checker with a simple state-language for system specification, and APMC [158], which is based on a randomized algorithm for approximating probabilities in Discrete Markov Chains and, like PVeStA [57], supports

client-server architectures for parallel computations. Other tools include Rapture [159] (which is also a BDD-based model checker) and Erlangen-Twente Markov Chain Checker $\mathsf{E} \vdash \mathsf{MC}^2$ [160]. Compared to the VeStA/PVeStA and Maude approach, most of these tools sacrifice expressiveness and generality for algorithmic decidability.

There have been several works in the literature to formally analyze DoS attacks. These include Meadow's formal framework for evaluating protocols against DoS attacks [46], an information flow-based framework using the Security Protocols Process Algebra (SPPA) for the detection of DoS vulnerabilities [47], and an observation-equivalence approach based on π-calculus for DoS detection [48]. Other formal approaches and extensions and applications of these approaches have also been developed [49, 50, 51].

8.7 Availability Analysis in Service Compositions

Service availability issues in service compositions in the presence of DoS attacks have only been recently recognized. Jensen and colleagues studied distributed flooding attacks on services in web service compositions in [161, 162], where they highlighted the challenges in maintaining availability, defending against such attacks, and tracing sources of attack in this context. These issues were also described in recent, general surveys of attacks, including DoS attacks, on web services [113, 163] and cloud-based services [164].

Much of the work in the literature addressing availability of services in service-oriented architectures deals with Quality-of-Service (QoS) properties of services, including reliability, latency and bandwidth usage. In general, several different approaches to the analysis of QoS properties of communication networks exist. The approach that is most closely related to our work is based essentially on the use of a probabilistic process calculus for the specification of services and service compositions, and a quantitative temporal logic for QoS properties. An exemplary work using this approach appeared in [165], where the author proposed an extension of Milner's CCS [166] with *costs*, which can be used to represent a QoS property such time delays and drop probabilities. The semantics of the extended calculus, called Quality CCS (QCCS), was given as a cost-extended labelled transition system (Quality Extended Labeled Transition System - QELTS), in which a transition is labeled by a pair (a, c), with a

an action and c the cost associated with the transition. To compare cost-enriched behaviors of processes in QCCS, a notion of quantified bisimulation was also presented. The author also gave a Quantified extension of CTL [167] (QCTL) endowed with costs, with semantics so that satisfaction of QCTL formulas is preserved under quantified bisimulation. A similar approach, given in [168, 169], extends the timed calculus COWS [142] with probabilities, called Stochastic COWS. Analysis is performed by translating processes in Stochastic COWS into Continuous Time Markov Chains (CTMCs), on which probabilistic or statistical verification can be applied using available tools, like PRISM.

CHAPTER 9

CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

This dissertation has presented formal methods and tools for the specification, analysis and implementation of distributed services, based on rewriting logic, it implementation in Maude, and the theory of Orc. A fundamental contention of this work is that in order to be able to provide effective formal tools for the design and analysis of current- and next-generation distributed Internet software systems, the underlying formalism should be both *very expressive*, so that their features can be naturally represented, and *efficiently executable*, which is required for efficient and mechanizable formal analysis, and for seamlessly deriving system implementations from formal specifications.

Below, we provide a summary of the results obtained, while highlighting possible future research directions.

9.1 Formal Specification and Analysis of Orc Service Orchestrations

We have presented a formal semantics in rewriting logic of Orc that provably captures Orc's intended synchronous, real-time semantics, which is a unique contribution to previous efforts to Orc's formal semantics. Two specification styles of Orc's rewriting semantics were first discussed: (1) the SOS-based rewriting semantics, given by the theory $\mathcal{R}^{sos}_{Orc}$, which was obtained directly from ORC's original SOS semantics through a semantics-preserving transformation method, and (2) the reduction rewriting semantics, given by $\mathcal{R}^{red}_{Orc}$, which exploits the distinction between equations and rules in rewriting logic to arrive at a much more efficiently executable specification. Both specifications were shown semantically equivalent through a strong bisimulation theorem, which implies correctness of the (untimed)

reduction rewriting semantics with respect to Orc's original SOS semantics. Through their specifications in Maude (and Real-Time Maude), both rewriting semantics can be immediately used for simulating Orc programs, and exhaustively analyzing their possible behaviors using breadth-first search and model-checking formal analysis tools of Maude. Experimental results, which were obtained by formally analyzing several Orc programs, have demonstrated the execution efficiency advantage of the reduction rewriting semantics over the SOS-based semantics.

The reduction rewriting semantics paved the ground for a more distributed, object-based semantics, $\mathcal{R}_{Orc}$, which extended Orc's semantics to multiple Orc expressions and provided explicit treatment of the environment by modeling Orc sites and asynchronous message passing. A specification of $\mathcal{R}_{Orc}$ in Real-Time Maude was then used as a back-end engine for the MOrc tool, which is a web-based application enabling the formal specification and verification of Orc expressions without exposing its user to the underlying Maude representation. The tool was described in detail with several examples.

There are several possible future research directions that will further advance the results obtained as part of this work. First, the object-based semantics given by $\mathcal{R}_{Orc}$ already improves on the other two semantics by introducing a level of true concurrency (object-level concurrency) as opposed to *simulated* concurrency by interleaving semantics, inherited from the original SOS semantics. However, it would be interesting to further develop an object-based semantics of Orc that is not based on the original SOS semantics, but is instead based on the truly concurrent, distributed semantics of rewriting logic. Recent developments in the rewriting logic semantics project, and, in particular, the K framework [170, 62], may be used as a vehicle towards achieving this semantics. An investigation of how this semantics relates to the original SOS semantics of Orc would also be interesting. Another possible future direction is to incorporate into Orc's rewriting semantics the notion of Orc classes (Professor J. Misra, personal communication, August 2010), which are data structures encapsulating Orc expressions and defining interfaces to external objects. The addition of Orc classes will have the desirable side effect of enabling the specification of behaviors of Orc sites as expressions in Orc. This is particularly useful for MOrc, since custom site behaviors can only be specified internally in Maude. A third possible future development is to improve

on the expressiveness of constraints in MOrc, to allow more general constraint patterns, and MOrc's current post-processing functions, to provide a more user-accessible analysis results.

9.2 Distributed Implementation of Orc Service Orchestrations

We described a general transformation methodology through which a real-time, object-based rewriting specification can be turned with minimal effort into a distributed implementation with physical timing, bridging the gap considerably between formal specifications and actual, deployable implementation. The transformation methodology is fundamentally enabled by Maude's support for external communication with external objects through sockets. We have demonstrated this method by applying it to the real-time, object-based semantics of Orc, obtaining as a result, a distributed implementation, Dist-Orc. A case study was described, in which the Orc specification of an auction application was deployed on a physically distributed network of nodes. Finally, we showed how programs in their implemented form (according to this transformation) can still be subjected to formal analysis by building appropriate abstractions of external objects.

Since the transformation method described above is generic, an interesting possible future research direction is to develop a precise characterization of the class of real-time rewrite theories for which this semantics-preserving transformation can be applied to arrive at a provably correct implementation. This development will provide the necessary theoretical foundation for automating the transformation process, perhaps as a meta-level function in Maude.

Furthermore, while the distributed implementation of Orc, Dist-Orc, currently enables prototyping and experimenting with an Orc program in a distributed environment, it does not provide an easily accessible interface beyond that of the Maude interpreter, which may not be as intuitive and natural as the user of an Orc program would expect when experimenting with or observing its behavior. In addition, the process of setting up the required environment, including management of the ticker objects, can be burdensome since it involves a fairly large number of steps that must be customized for each Orc program (this is

currently partially automated using shell scripts). Therefore, an interesting future development is to extend Dist-Orc with a more user-friendly, visual interface for its applications, using, for example, the IOP+IMaude approach [20] or an external GUI server, such as the GTK+ server [171]. This extension will automate most of the steps required to setup and execute an Orc program, while providing a visual representation of its most relevant aspects.

9.3 Statistical Model Checking Analysis

This dissertation has proposed architecture-independent, map-reduce parallelizations of two statistical analysis algorithms: (1) the statistical model checking algorithm of PCTL/CSL formulas given in [1], and (2) the statistical quantitative analysis algorithm of QuaTEx expressions given in [2]. The parallel algorithms were implemented in PVeStA, which is based on the implementation of the original sequential algorithms in VeStA. Like VeStA, PVeStA supports probabilistic models of systems given either as probabilistic rewrite theories in Maude, or as continuous-time Markov chains in a PRISM-like input syntax. We have also demonstrated the performance gains as a result of parallelization through several statistical analysis tasks of varying complexities, run on two different parallel architectures: a HPC cluster, and a multi-core machine. One important goal of this contribution was to make this increased scalability and efficiency available to expressive probabilistic models of systems, such as probabilistic rewrite theories in Maude. To this effect, we have demonstrated and further developed a rewriting-based approach to the analysis of availability properties of services (specified as QuaTEx expressions) in the presence of DoS attacks in the ASV protocol and its non-adaptive variants. Automatic statistical quantitative analysis results confirmed previously manually proved theorems, in addition to properties shown by statistical analysis of low-level network simulations. In this formal statistical model checking approach, a considerably higher level of assurance can be gained for both analytical properties proved by hand, and for properties suggested by simulation analyses.

An interesting future research direction related to this work is to extend the parallel statistical PCTL/CSL model checking algorithm, and its corresponding implementation in PVeStA, to *nested* probabilistic formulas of the form $\mathcal{P}_{\bowtie p}(\varphi)$, where φ may itself be a

probabilistic formula. While the original sequential algorithm in [1] already supports nested probabilistic formulas, parallelization of the algorithm for nested formulas is not entirely trivial, since nested probabilistic operators will eventually require generating further sample executions along previously generated sample paths. A naive solution is to limit parallelization to the outermost probabilistic operator. Another possible solution is to use techniques from statistical model checking of black-box systems [154, 155].

9.4 Availability Analysis in Service Compositions

By building on the ASV model in rewriting logic, presented in Chapter 6, we proposed a method for modular protection of availability properties of systems using generic ASV wrappers, which are objects that encapsulate lower-layer configurations of objects, mediating communication between sub-configurations of the wrapped system. We provided a detailed description of the design of wrappers, their behaviors, and the assumptions under which statistical model checking and quantitative analysis becomes applicable. As a demonstration of the use of wrappers and an illustration of an expressive and flexible method for analysis of availability in service compositions, we described an instantiation of the ASV wrappers to Orc, given by its object-based semantics in $\mathcal{R}_{Orc}$, and conducted a set of experiments analyzing a simple Orc orchestration pattern.

There is much more to be done in this direction. In particular, to further refine the design of wrappers and demonstrate the generality and effectiveness of the approach, we will need to: (1) further experiment with the Orc instantiation of ASV wrappers using more complex orchestration examples, (2) develop other different instantiations of the wrappers beyond Orc and experiment with them, and (3) extend the specification of wrappers to support timing in underlying system specifications. Another possible research direction is to extend the adaptive behavior of clients in the ASV protocol (specified in the ASV client wrappers) to the area of cloud computing, in which dynamic resource allocation in the could is made adaptive to potential DoS attacks, to try to maintain a level of service comparable to that in the absence of such attacks. The generic wrappers method can be extended to support such adaptive behaviors in server wrappers.

APPENDIX A

PROOFS OF SOME ALGEBRAIC PROPERTIES OF ORC

Equivalence of arbitrary Orc expressions is shown here using a strong bisimulation based on the operational semantics specified by the rules in Figure 2.2. At the risk of abusing notation, the symbol = is used here to denote strong bisimulation.

- $(f \,;\, g) \,;\, h = f \,;\, (g \,;\, h)$. The behavioral transitions of f are the only transitions possible in each side of the identity. So suppose f publishes a value, i.e., $f \stackrel{!v}{\hookrightarrow} f'$. Then, by OtherV, $f \,;\, g \stackrel{!v}{\hookrightarrow} f'$, which, again by OtherV, implies $(f \,;\, g) \,;\, h \stackrel{!v}{\hookrightarrow} f'$. Applying OtherV to the right-hand side of the identity yields $f \,;\, (g \,;\, h) \stackrel{!v}{\hookrightarrow} f'$, which concludes this case.

  Suppose now that f takes a non-publishing action n, i.e., $f \stackrel{n}{\hookrightarrow} f'$. Then, by OtherN, we have:

  $$(f \,;\, g) \,;\, h \stackrel{n}{\hookrightarrow} (f' \,;\, g) \,;\, h \quad \text{and} \quad f \,;\, (g \,;\, h) \stackrel{n}{\hookrightarrow} f' \,;\, (g \,;\, h)$$

  which concludes the proof, assuming that $(f' \,;\, g) \,;\, h = f' \,;\, (g \,;\, h)$.

- $f \,;\, \mathbf{0} \;= f$. There are two cases corresponding to f taking a publishing or a non-publishing action. If f publishes a value, then by OtherV, we have $f \stackrel{!v}{\hookrightarrow} f'$ iff $f \,;\, \mathbf{0} \;\stackrel{!v}{\hookrightarrow} f'$. Otherwise, if f takes a non-publishing action n, then by OtherN, $f \stackrel{n}{\hookrightarrow} f'$ iff $f \,;\, \mathbf{0} \;\stackrel{n}{\hookrightarrow} f' \,;\, \mathbf{0}$. Assuming $f' \,;\, \mathbf{0} \;= f'$, the property is proved.

- $f <\!x\!> \mathbf{0} \;= [\mathbf{stop}/x]f$. We need to show that, for some label l and expression f':

  $$f <\!x\!> \mathbf{0} \;\stackrel{l}{\hookrightarrow} f' <\!x\!> \mathbf{0} \;\text{ iff } [\mathbf{stop}/x]f \stackrel{l}{\hookrightarrow} [\mathbf{stop}/x]f'$$

  By Asym2, we have $f <\!x\!> \mathbf{0} \;\stackrel{l}{\hookrightarrow} f' <\!x\!> \mathbf{0} \;$ iff $f \stackrel{l}{\hookrightarrow} f'$. Therefore, the goal reduces to: $f \stackrel{l}{\hookrightarrow} f'$ iff $[\mathbf{stop}/x]f \stackrel{l}{\hookrightarrow} [\mathbf{stop}/x]f'$, which can be shown by structural induction

on f (this property is called *Substitution Independence* in [83] and is proved there for the timed SOS semantics). Therefore, the property holds assuming $f' <x< \mathbf{0} = [\mathbf{stop}/x]f'$.

- $!v \ ; f = \ !v$. The only possible behavioral transition for both $!v$ and $!v \ ; f$ is the PUBLISH action. Therefore, $!v \xrightarrow{!v} \mathbf{0}$ and, by OTHERV, $!v \ ; f \xrightarrow{!v} \mathbf{0}$, and hence the conclusion.

APPENDIX B

PROPERTIES OF $\mathcal{R}^{SOS}_{ORC}$ AND $\mathcal{R}^{RED}_{ORC}$

In this appendix, we refer to the theories $\mathcal{R}^{sos}_{Orc}$ and $\mathcal{R}^{red}_{Orc}$ respectively by $\mathcal{R}^s$ and $\mathcal{R}^r$, and use $\mathcal{C} \rightarrow_{\mathcal{R}} \mathcal{C}'$ to denote $\mathcal{R} \vdash \mathcal{C} \rightarrow^1 \mathcal{C}'$, for brevity and notational convenience.

B.1 Executability of $\mathcal{R}^{sos}_{Orc}$

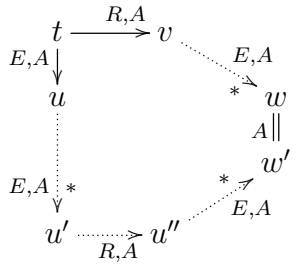
Before going through the proofs of executability of $\hat{\mathcal{R}}^s$, we outline the transformation method applied to the underlying conditional membership equational logic theory $(\Sigma^s, E^s \cup A^s)$ of $\mathcal{R}^s$ into a semantically equivalent conditional *order-sorted* equational logic theory $(\hat{\Sigma}^s, \hat{E}^s \cup A^s)$ in $\hat{\mathcal{R}}^s$ with no kinds or membership assertions, so that the current Maude tools for checking confluence of equations and coherence of rules with respect to equations can be used. In this transformation, a new top sort s_K is introduced for each kind K in Σ. The sort s_K in $\hat{\Sigma}^s$ replaces the kind K in all kind-level operator and variable declarations in Σ^s. In particular, a top sort $s_{[Bool]}$ is introduced and used to represent terms of the kind [Bool]. Then, each membership clause of the form $f(t_1, \cdots, t_2) : s$, with f an operator of a sort $s' \geq s$, is transformed into the equation $s?(f(t_1, \cdots, t_2)) = \mathit{true}$, where $s?$ is a newly introduced (partial) predicate, declared as $s? : s' \rightarrow s_{[Bool]}$. This transformation, which is also applied to $\mathcal{R}^r$ in Appendix B.2, can be straightforwardly shown to preserve the semantics of the original membership equational logic-based rewrite theory.

Proof of Lemma 3 – ground confluence and descent properties of $\hat{\mathcal{R}}^s$. The proof is automatic using Maude's Church-Rosser Checker (CRC) tool [89] and the Maude module `SOS-ORC` corresponding to the rewrite theory $\hat{\mathcal{R}}^s$. In particular, by loading the Full Maude specification of CRC and the Maude module `SOS-ORC`, and by issuing the CRC command `(check Church-Rosser .)`, we obtain the following output:

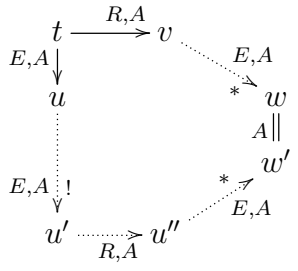
```
...
Church-Rosser checking of SOS-ORC
Checking solution:
The following critical pairs cannot be joined:
  cp for syn04d and cin12a1
    zero = M:SiteName(VL:ValueList).
  ccp for syn04c and cin12b
    zero = (S:Subst ix:IVar)(S:Subst P:AParamList)
    if not hasSV?(P:AParamList)= true .
  ccp for syn04d and cin12a
    zero = M:SiteName(S:Subst P:AParamList)
    if not hasSV?(P:AParamList)= true .
The specification is sort-decreasing.
```

The current version of the tool has a known bug that, in some specific cases, may cause the checker to miss some conjuncts in the conditions of conditional critical pairs, causing the reported (apparently not joinable) critical pairs above (personal communication with Prof. Francisco J. Durán, November 2010). In particular, the first reported critical pair between equations labeled `syn04d` and `cin12a1` should actually be a *conditional* critical pair with the condition `hasSV?(VL:ValueList) = true`, which can be easily shown unfeasible. The other two reported conditional critical pairs are missing the conjunct `hasSV?(P:AParamList) = true` in their conditions, which clearly makes both pairs trivially unfeasible. Finally, the tool correctly reports that the specification is sort-decreasing. □

Proof of Lemma 4 – ground coherence of $\hat{\mathcal{R}}^s$. By definition of ground coherence (see [90]), $\hat{\mathcal{R}}^s$ is ground coherent iff for every ground $\hat{\Sigma}^s$ term t such that $t \to_{E,A} u$ and $t \to_{R,A} v$, we have



where solid and dotted arrows denote, respectively, universal and existential quantification. Likewise, for *local* ground coherence we have,



We first note that Theorem 3 in [90], which states that the notions of coherence and local coherence coincide for the considered class of conditional rewrite theories with *equational* conditions, also applies to rewrite theories with rules having rewrites in their conditions (The proof of Theorem 3 in [90] is essentially the same because the property follows mainly by confluence and termination of the equational rewriting relation $\rightarrow_{E,A}$). Therefore, the problem of checking ground coherence of $\hat{\mathcal{R}}^s$ reduces to checking *local* ground coherence. Consequently, as in [90], we may reason about coherence of $\hat{\mathcal{R}}^s$ by cases on $u \ {}_{E,A}\!\leftarrow t \rightarrow_{R,A} v$, depending on whether they are overlap or non-overlap situations as follows:

Overlap Case. We note that $\hat{\mathcal{R}}^s$ is a top-most theory so that rewrites by rules occur only at the top of an Orc configuration (the top sort is Config) with no function symbols that equationally rewrite above a configuration. Therefore, a (conditional) critical pair may only be the result of an overlap between the left-hand side of an equation in E^s and a non-variable position in the left-hand side of a rewrite rule in R^s (Type I critical pairs in [90]). That is, the overlap case reduces to checking for every rule $l \rightarrow r$ if C in R^s, equation $l' = r'$ if C' in $\hat{E}^s$, non-variable position p in l, and substitution θ such that $\theta(l') = \theta(l|_p)$, that the conditional critical pair $\theta(C') \wedge \theta(C) \implies \theta(l[r']_p) \rightarrow \theta(r)$ is either unfeasible or can be shown joinable. It is easy to see from the equations and rules in $\hat{\mathcal{R}}^s$ that the only possible overlap of this kind is between the equation corresponding to identity (8) in Figure 2.3, namely that $!v \,;\, f = !v$, and the rewrite rule OtherV:

$$\cdot\langle \hat{f} \,;\, g,\ lbl : l \mid r\rangle \rightarrow \langle f',\ lbl : !v' \mid r'\rangle \ \text{ if } \ \cdot\langle \hat{f}, lbl : \epsilon \mid r\rangle \rightarrow \langle f', lbl : !v' \mid r'\rangle$$

The overlap occurs at the top of the expression component of the configuration given by the lhs of the rule OtherV, which results in the following conditional critical pair:

$$\cdot\langle !v, lbl : \epsilon \mid r\rangle \to \langle f', lbl : !v' \mid r'\rangle \implies \cdot\langle !v, lbl : \ l \mid r'\rangle \to \langle f', lbl : !v' \mid r'\rangle$$

The rewrite in the condition $\cdot\langle !v, lbl : \epsilon \mid r\rangle \to \langle f', lbl : !v' \mid r'\rangle$ can only be an application of the Publish rule in R^s (see Figure 3.4) with the substitution $\{f' \mapsto \mathbf{0}, v' \mapsto v, r' \mapsto r\}$, which makes the critical pair joinable by the Publish rule.

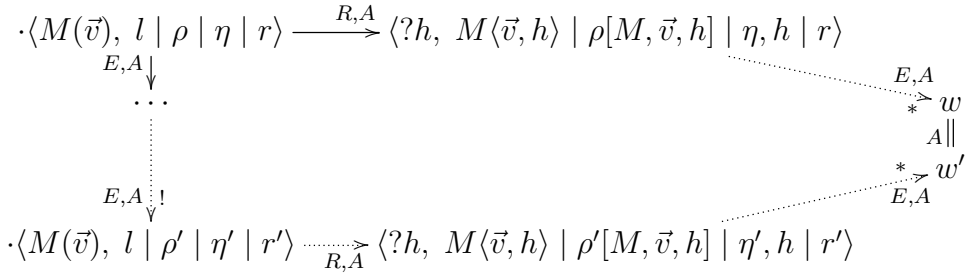
Non-overlap Case. Again, since $\hat{\mathcal{R}}^s$ is top-most, we need only to consider non-overlap situations of $\hat{E}^s$ under R^s. That is, we need to show local ground coherence of situations of the form $\cdot\langle f_1, r_1\rangle \ {}_{E,A}\!\leftarrow \cdot\langle f, r\rangle \to_{R,A} \langle f', r'\rangle$, where rewriting with $\hat{E}^s$ occurs in variable positions in f and/or r. This can be shown by induction on the size of (or the number of applications of the **Replacement** inference rule of rewriting logic [10] in) a proof of the one-step rewrite $\cdot\langle f, r\rangle \to_{R,A} \langle f', r'\rangle$.

The base cases correspond to applications of rules with no rewrites in their conditions, i.e., when the rewrite step $\cdot\langle f, r\rangle \to_{R,A} \langle f', r'\rangle$ is an instance of one of the following rules: SiteCall, SiteRetV, SiteRetStop, Publish, ExprCall or Tick. We will discuss the cases for SiteCall and Tick only, as the other cases are similar. Suppose that the rewrite is proved by SiteCall. Then, the configuration $\cdot\langle f, r\rangle$ is of the form:

$$\cdot\langle M(\vec{v}),\ lbl : l \mid msg : \rho \mid hdl : \eta \mid r\rangle$$

Since $M(\vec{v})$ and any label instance of l are constructor terms, the configuration may rewrite with equations in a substitution of the variables ρ, η, or r. In other words, the configuration may rewrite with $\hat{E}^s$ to $\cdot\langle M(\vec{v}),\ lbl : l \mid msg : \rho' \mid hdl : \eta' \mid r'\rangle$, with ρ', η', and r' of the

respective appropriate sorts. Therefore, we have:

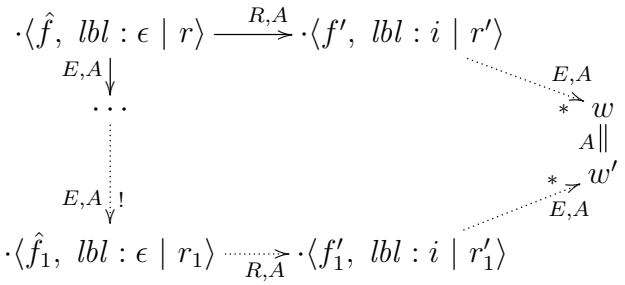


since $h = \mathsf{fresh}(\eta) = \mathsf{fresh}(\eta')$, as the only equation that could apply to an instance of η is the idempotency of handle sets.

For the case of the Tick rule, the configuration is of the form $\cdot\langle f, clk : t \mid r\rangle$ such that $\mathsf{eager}(\langle f, clk : t \mid r\rangle) \neq \mathsf{true}$ and $m = \mathsf{mte}(r)$ is non-zero. Using $\hat{E}^s$, the configuration may rewrite to one of the form $\cdot\langle f', clk : t' \mid r'\rangle$. Note that f in the original configuration cannot be an active expression (because otherwise the configuration would be eager), which implies that f has either the sort IExpr or ZExpr. Since these sorts are minimal, and by the descent property of $\hat{E}^s$, the sort of f' is the same as the sort of f. Moreover, since the original configuration is not eager, the record r does *not* contain a message identified by a handle h that is being used in f (i.e., neither an incoming message $[w, h]$ nor an outgoing message $[M, \vec{v}, h]$). Therefore, the configuration $\cdot\langle f', clk : t' \mid r'\rangle$ is also *not* eager. Finally, since definitions of mte and δ are entirely based on message delays, which must always have identical canonical forms in r and r', it is easy to see that $\mathsf{mte}(r) = \mathsf{mte}(r')$ and $\delta(r) = \delta(r')$, and hence the desired conclusion.

The inductive cases correspond to applications of the remaining rewrite rules in $\hat{\mathcal{R}}^s$, which are the rules with rewrites in their conditions. In particular, suppose that the rewrite step is proved by SymI. Then, the configuration has the form $\cdot\langle \hat{f} \mid \widetilde{g},\ lbl : l \mid r\rangle$, which may rewrite by $\hat{E}^s$ to the term $\cdot\langle \hat{f}_1 \mid \widetilde{g}_1,\ lbl : l \mid r_1\rangle$ (note that $\hat{f}_1$ and $\widetilde{g}_1$ are of the respective appropriate

sorts). By the rewrite condition of SymI and the induction hypothesis, we have



Then, by SymI:

$$\begin{aligned}\cdot\langle \hat{f}_1 \mid \widetilde{g}_1,\ lbl : l \mid r_1\rangle &\rightarrow_{R,A} \langle f'_1 \mid \widetilde{g}_1,\ lbl : i \mid r'_1\rangle \\ &=_A \langle f' \mid \widetilde{g}_1,\ lbl : i \mid r'\rangle\end{aligned}$$

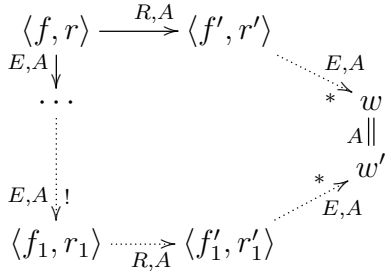
and, thus, the property holds. The other cases are similar. This completes the proof. □

B.2 Executability of $\mathcal{R}^{red}_{Orc}$

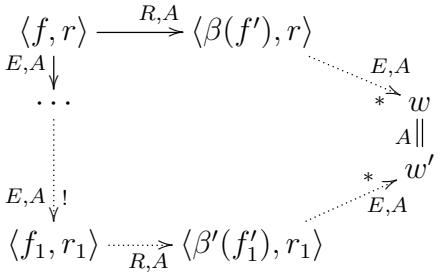
Proof of Lemma 6 – ground confluence and descent properties of $\hat{\mathcal{R}}^r$. As for Lemma 3, the proof of this lemma is almost fully automatic using Maude's Church-Rosser Checker (CRC) and the Maude module `RED-ORC` corresponding to $\hat{\mathcal{R}}^r$. The current version of Maude's CRC tool, however, reports a few additional, superfluous conditional critical pairs to those reported for $\hat{\mathcal{R}}^s$ in the proof of Lemma 3 in Appendix B.1 as a result of a known problem in the tool, as discussed before. The missing conditions, when correctly taken into account, render the conditional critical pairs unfeasible. The tool correctly reports that the Maude module `RED-ORC` is sort-decreasing. □

Proof of Lemma 7 – ground coherence of $\hat{\mathcal{R}}^r$. Let $\hat{\mathcal{R}}^r_\circ = (\hat{\Sigma}^r, \hat{E}^r \cup A^r, R_\circ, \phi^r)$, where $R_\circ = R^r - \{IAction\}$. That is, $\hat{\mathcal{R}}^r_\circ$ is almost identical to $\hat{\mathcal{R}}^r$ except without the IAction rule, which is the only conditional rule in $\hat{\mathcal{R}}^r$ with a rewrite condition. Therefore, ground coherence of $\hat{\mathcal{R}}^r_\circ$ can be shown by Maude's ChC tool. As was argued in the proof of ground coherence of $\hat{\mathcal{R}}^s$ (proof of Lemma 4 in Appendix B.1), by Theorem 3 in [90], it is enough

to show ground *local* coherence of $\hat{\mathcal{R}}^r$ to show ground coherence. That is, we have to show that, for every ground Orc configuration $\langle f, r\rangle$, the following diagram commutes:



We note that, by the ground descent property of $\hat{E}^r$ and the sorts ZExpr and IExpr being minimal, if f is *not* an active expression, then neither is f_1, and, thus, the diagram commutes by ground local coherence of $\hat{\mathcal{R}}^r_\circ$. So suppose f is an active expression. Then, the one-step rewrite must be an instance of the IAction rule. Therefore, we have to show:



where β and β' stand for auxiliary functions for internal actions, namely siteCall, exprCall, publish and publish$^\tau$. Note that, since f is active, f_1 must also be active, and hence the use of the β' in the diagram. Therefore, the problem now reduces to showing that the diagram commutes by applying IAction on $\langle f_1, r_1\rangle$ using the same internal action rule in the condition as the one used in the rewrite step for $\langle f, r\rangle$. It can be easily shown by cases on the equations in $\hat{E}^r$ that sub-expressions in non-frozen positions in any expression g are preserved. That is, if $l \to r$ if C is an equation in $\hat{E}^r$ such that $\theta(l) = g|_q$, for some non-frozen position q, and $g_\circ$ is a sub-term of $g|_q$ then there exists a corresponding non-frozen position q' in $g[\theta r]_q$ with $g_\circ = g[\theta r]_q|_{q'}$. In particular, if f has a basic active expression b as a subterm in a non-frozen position, then b is also a non-frozen subterm of f_1. Therefore, $f_1 \to_{R,A} \beta(f'_1)$, and, hence, the conclusion. □

B.3 Proof of the Equivalence Theorem (Theorem 4)

Proof. ($\Longrightarrow$) By induction on a proof of $\mathsf{smallstep}(\cdot\mathcal{C}) \rightarrow_{\mathcal{R}^s} \mathsf{smallstep}(\cdot\mathcal{C}')$, which is abbreviated below as $(\cdot\mathcal{C}) \rightarrow_{\mathcal{R}^s} (\cdot\mathcal{C}')$. There are six base cases, corresponding to the rules [SiteCall], [Publish], [Def], [SiteRetV], [SiteRetStop], and [Tick] in $\mathcal{R}^s$:

- [SiteCall]: If h is a fresh handle with respect to a handle set η, and

$$\begin{aligned}
&(\cdot\langle M(\vec{v}),\ lbl : l \mid msg : \rho \mid hdl : \eta \mid r\rangle) \\
&\qquad \rightarrow_{\mathcal{R}^s} (\langle ?h,\ lbl : M\langle \vec{v}, h\rangle \mid msg : \rho[M, \vec{v}, h] \mid hdl : \eta, h \mid r\rangle) \\
&\qquad =_{\mathcal{R}^s} (\cdot\langle ?h,\ lbl : \epsilon \mid msg : \rho[M, \vec{v}, h] \mid hdl : \eta, h \mid r\rangle)
\end{aligned}$$

  then, by [IAction] (and [SiteCall]) in $\mathcal{R}^r$, and using the substitution $\{f' \mapsto \gamma, i \mapsto \mathsf{siteCall}(M, \vec{v})\}$, we have

$$\begin{aligned}
&\langle M(\vec{v}),\ lbl : l \mid msg : \rho \mid hdl : \eta \mid r\rangle \\
&\qquad \rightarrow_{\mathcal{R}^r} \langle \mathsf{act}^{\uparrow}(\gamma, \mathsf{siteCall}(M, \vec{v})),\ lbl : l \mid msg : \rho \mid hdl : \eta \mid r\rangle \\
&\qquad =_{\mathcal{R}^r} \langle \mathsf{act}^{\downarrow}(\gamma, ?h),\ lbl : \epsilon \mid msg : \rho[M, \vec{v}, h]) \mid hdl : \eta, h \mid r\rangle \\
&\qquad =_{\mathcal{R}^r} \langle ?h,\ lbl : \epsilon \mid msg : \rho[M, \vec{v}, h] \mid hdl : \eta, h \mid r\rangle
\end{aligned}$$

  The base cases for the remaining internal actions, [Publish] and [Def], are similar.

- [SiteRetV]: Suppose

$$\begin{aligned}
&(\cdot\langle ?h,\ lbl : l \mid msg : \rho[v, h] \mid hdl : \eta, h \mid r\rangle) \\
&\qquad \rightarrow_{\mathcal{R}^s} (\langle !v,\ lbl : h?v \mid msg : \rho \mid hdl : \eta \mid r\rangle) \\
&\qquad =_{\mathcal{R}^s} (\cdot\langle !v,\ lbl : \epsilon \mid msg : \rho \mid hdl : \eta \mid r\rangle)
\end{aligned}$$

Since $?h$ is an inactive expression and $h \in \mathsf{handles}(?h) = \mathit{true}$ is provable from $\mathcal{R}^r$, we use the rule SiteRet in $\mathcal{R}^r$ (with the substitution $\{\bar{f} \mapsto ?h\}$) to get:

$$\begin{aligned}\langle ?h,\ \mathit{lbl} : l \mid \mathit{msg} : \rho[v, h] \mid \mathit{hdl} : \eta, h \mid r\rangle \\ \rightarrow_{\mathcal{R}^r} \langle \mathsf{sret}(?h, v, h),\ \mathit{lbl} : \epsilon \mid \mathit{msg} : \rho \mid \mathit{hdl} : \eta \mid r\rangle \\ =_{\mathcal{R}^r} \langle !v,\ \mathit{lbl} : \epsilon \mid \mathit{msg} : \rho \mid \mathit{hdl} : \eta \mid r\rangle\end{aligned}$$

The base case for [SiteRetStop] is similar.

- [Tick]: Suppose

$$(\cdot\langle \bar{f}, \mathit{clk} : t \mid r\rangle) \rightarrow_{\mathcal{R}^s} (\langle \bar{f}, \mathit{clk} : t + t' \mid \delta(r, t')\rangle)$$

Then, $\mathcal{R}^s$ proves $t' = \mathsf{mte}(\mathcal{C})$, which is non-zero, and $\mathsf{eager}(\mathcal{C}) \neq \mathit{true}$. By Lemma 9, $\mathcal{R}^r$, too, proves that that t' is the maximum time elapse of $\mathcal{C}$ and that $\mathcal{C}$ is not an eager configuration. Therefore, by the Tick rule in $\mathcal{R}^r$, we have:

$$\langle \bar{f}, \mathit{clk} : t \mid r\rangle \rightarrow_{\mathcal{R}^s} \langle \bar{f}, \mathit{clk} : t + t' \mid \delta(r, t')\rangle$$

For the inductive step, there are fourteen cases corresponding to the inductive rules listed in Figure 3.5. We discuss below representative cases for symmetric parallel and sequential compositions. The remaining cases for asymmetric parallel and otherwise compositions are similar.

- [SymI]. Suppose $(\cdot\langle \hat{f} \mid g,\ \mathit{lbl} : l \mid r\rangle) \rightarrow_{\mathcal{R}^s} (\langle f' \mid g,\ \mathit{lbl} : i \mid r'\rangle)$, which is equationally equivalent to $(\cdot\langle f' \mid g,\ \mathit{lbl} : \epsilon \mid r'\rangle)$, then we have

$$(\cdot\langle \hat{f},\ \mathit{lbl} : \epsilon \mid r\rangle) \rightarrow_{\mathcal{R}^s} (\langle f', \mathit{lbl} : i \mid r'\rangle) =_{\mathcal{R}^s} (\cdot\langle f', \mathit{lbl} : \epsilon \mid r'\rangle)$$

By the inductive assumption, this implies $\langle \hat{f},\ \mathit{lbl} : \epsilon \mid r\rangle \rightarrow_{\mathcal{R}^r} \langle f',\ \mathit{lbl} : \epsilon \mid r'\rangle$, by an application of [IAction] in $\mathcal{R}^r$ such that there exists an active base expression as a subexpression of $\hat{f}$. Therefore, $\hat{f} \rightarrow_{\mathcal{R}^r} \mathsf{act}^{\uparrow}(f'', i)$, with $f'' = \hat{f}[p \leftarrow b]$, for some

position p in $\hat{f}$ and b is either *tmp* or $\mathbf{0}$, depending on whether the internal action is a (site or expression) call or a publishing of a value, respectively. By congruence, this implies $\hat{f} \mid g \rightarrow_{\mathcal{R}^r} \mathsf{act}^{\uparrow}(f'', i) \mid g$, which is equal to $\mathsf{act}^{\uparrow}(f'' \mid g, i)$. If the action i is a call, then by the rule [IAction]:

$$\begin{aligned}\langle \hat{f} \mid g,\ lbl : l \mid r \rangle &\rightarrow_{\mathcal{R}^r} \langle \mathsf{act}^{\uparrow}(f'' \mid g, i),\ lbl : l \mid r \rangle \\ &=_{\mathcal{R}^r} \langle \mathsf{act}^{\downarrow}(f'' \mid g, e),\ lbl : \epsilon \mid r' \rangle \\ &=_{\mathcal{R}^r} \langle f' \mid g,\ lbl : \epsilon \mid r' \rangle\end{aligned}$$

where e is a handle expresion if i is a site call, or the instantiation of a body of an appropriate expression declaration if i is an expression call. The remaining case when the action i is a publishing action is similar.

- [SymE]. Let u be a site return label, and suppose

$$(\cdot\langle \bar{f} \mid \bar{g},\ lbl : l \mid r \rangle) \rightarrow_{\mathcal{R}^s} (\langle f' \mid \bar{g},\ lbl : u \mid r' \rangle) =_{\mathcal{R}^s} (\cdot\langle f' \mid \bar{g},\ lbl : \epsilon \mid r' \rangle)$$

Then $(\cdot\langle \bar{f},\ lbl : \epsilon \mid r \rangle) \rightarrow_{\mathcal{R}^s} (\langle f', lbl : u \mid r' \rangle) =_{\mathcal{R}^s} (\cdot\langle f', lbl : \epsilon \mid r' \rangle)$. By the inductive assumption, this implies $\langle \bar{f},\ lbl : \epsilon \mid r \rangle \rightarrow_{\mathcal{R}^r} \langle f',\ lbl : \epsilon \mid r' \rangle$, by an application of [SiteReturn] in $\mathcal{R}^r$ such that there exists a handle base expression of the form $?h$ as a subexpression of $\bar{f}$ at some position p, that a message $[w, h]$ exists in the set of handles in the handles field of r, and that f' is either $\bar{f}[p \leftarrow \mathbf{0}]$ or $\bar{f}[p \leftarrow !v]$, depending on whether the return value is a **stop** value or not, respectively. Therefore, by [SiteReturn], we have:

$$\begin{aligned}\langle \bar{f} \mid g,\ lbl : l \mid r \rangle &\rightarrow_{\mathcal{R}^r} \langle \mathsf{sret}(\bar{f} \mid g, w, h),\ lbl : \epsilon,\ \mid r' \rangle \\ &=_{\mathcal{R}^r} \langle \mathsf{sret}(\bar{f}, w, h) \mid \mathsf{sret}(g, w, h),\ lbl : \epsilon,\ \mid r' \rangle \\ &=_{\mathcal{R}^r} \langle f' \mid g,\ lbl : \epsilon,\ \mid r' \rangle\end{aligned}$$

- [SEQ1V]. Suppose

$$(\cdot\langle \hat{f} >x> g,\ lbl : l \mid r\rangle) \rightarrow_{\mathcal{R}^s} (\langle (f' >x> g) \mid [v/x]g,\ lbl : \tau \mid r'\rangle)$$

which is equationally equivalent to $(\cdot\langle (f' >x> g) \mid [v/x]g,\ lbl : \epsilon \mid r'\rangle)$. Then, $(\cdot\langle \hat{f}, lbl : \epsilon \mid r\rangle) \rightarrow_{\mathcal{R}^s} (\langle f', lbl : !v \mid r'\rangle) =_{\mathcal{R}^s} (\cdot\langle f', lbl : \epsilon \mid r'\rangle)$. By the inductive assumption, this implies $\langle \hat{f}, lbl : \epsilon \mid r\rangle) \rightarrow_{\mathcal{R}^r} (\langle f', lbl : \epsilon \mid r'\rangle$ by an application of [IACTION] in $\mathcal{R}^r$ such that there exists a publishing base expression of the form $!v$ as a subexpression of $\hat{f}$ and that v is not bound in $\hat{f}$. Therefore, $\hat{f} \rightarrow_{\mathcal{R}^r} \mathsf{act}^{\uparrow}(f', \mathsf{publish}(v))$, with $f' = \hat{f}[p \leftarrow \mathbf{0}]$, for some position p in $\hat{f}$. By congruence, this implies, $\hat{f} >x> g \rightarrow_{\mathcal{R}^r} \mathsf{act}^{\uparrow}(f', \mathsf{publish}(v)) >x> g$, which is equal to $\mathsf{act}^{\uparrow}(f' >x> g \mid [v/x]g, \mathsf{publish}^{\tau})$, and, thus, by the rule [IACTION]:

$$\begin{aligned}\langle \hat{f} >x> g,\ lbl : l \mid r\rangle &\rightarrow_{\mathcal{R}^r} \langle \mathsf{act}^{\uparrow}(f' >x> g \mid [v/x]g, \mathsf{publish}^{\tau}),\ lbl : l \mid r\rangle \\ &=_{\mathcal{R}^r} \langle f' >x> g \mid [v/x]g,\ lbl : \epsilon \mid r'\rangle\end{aligned}$$

- [SEQ1NI]. Suppose

$$\begin{aligned}(\cdot\langle \hat{f} >x> g,\ lbl : l \mid r\rangle) &\rightarrow_{\mathcal{R}^s} (\langle f' >x> g,\ lbl : n \mid r'\rangle) \\ &=_{\mathcal{R}^s} (\cdot\langle f' >x> g,\ lbl : \epsilon \mid r'\rangle)\end{aligned}$$

for some internal, non-publishing label n. Then, $(\cdot\langle \hat{f}, lbl : \epsilon \mid r\rangle) \rightarrow_{\mathcal{R}^s} (\langle f', lbl : n \mid r'\rangle) =_{\mathcal{R}^s} (\cdot\langle f', lbl : \epsilon \mid r'\rangle)$. By the induction hypothesis, this implies $\langle \hat{f}, lbl : \epsilon \mid r\rangle) \rightarrow_{\mathcal{R}^r} (\langle f', lbl : \epsilon \mid r'\rangle$ by an application of [IACTION] in $\mathcal{R}^r$ such that $\hat{f}$ has as a subexpression a base expression of one of the folowing forms: (i) a site call expression $M(\vec{v})$, (ii) an expression call expression $E(\vec{p})$, or (iii) a publishing expression $!v$ with v bound in $\hat{f}$. For case (i), $\hat{f} \rightarrow_{\mathcal{R}^r} \mathsf{act}^{\uparrow}(f_{tmp}, \mathsf{siteCall}(M, \vec{v}))$, with $f_{tmp} = \hat{f}[p \leftarrow tmp]$, for some position p in $\hat{f}$. By congruence, this implies, $\hat{f} >x> g \rightarrow_{\mathcal{R}^r} \mathsf{act}^{\uparrow}(f_{tmp}, \mathsf{siteCall}(M, \vec{v})) >x> g$, which is equal to $\mathsf{act}^{\uparrow}(f_{tmp} >x> g, \mathsf{siteCall}(M, \vec{v}))$, and,

thus, by the rule [IAction]:

$$\begin{aligned}\langle \hat{f} >x> g,\ lbl : l \mid r\rangle &\rightarrow_{\mathcal{R}^r} \langle \mathsf{act}^{\uparrow}(f_{tmp} >x> g, \mathsf{siteCall}(M, \vec{v})),\ lbl : l \mid r\rangle \\ &=_{\mathcal{R}^r} \langle \mathsf{act}^{\downarrow}(f_{tmp} >x> g, ?h),\ lbl : \epsilon \mid r'\rangle \\ &=_{\mathcal{R}^r} \langle f' >x> g,\ lbl : \epsilon \mid r'\rangle\end{aligned}$$

Cases (ii) and (iii) can similarly be checked using IAction and the appropriate equations in $\mathcal{R}^r$.

- [Seq1NE]. Let u be a site return label, and suppose

$$\begin{aligned}(\cdot\langle \bar{f} >x> g,\ lbl : l \mid r\rangle &\rightarrow_{\mathcal{R}^s} \langle f' >x> g,\ lbl : u \mid r'\rangle) \\ &=_{\mathcal{R}^s} (\cdot\langle f' >x> g,\ lbl : \epsilon \mid r'\rangle)\end{aligned}$$

Then, $(\cdot\langle \bar{f}, lbl : \epsilon \mid r\rangle \rightarrow_{\mathcal{R}^s} \langle f', lbl : u \mid r'\rangle)$. By the induction hypothesis, this implies $\langle \bar{f}, lbl : \epsilon \mid r\rangle \rightarrow_{\mathcal{R}^r} \langle f', lbl : \epsilon \mid r'\rangle$ by an application of [SiteReturn] in $\mathcal{R}^r$ such that there exists a handle base expression of the form $?h$ as a subexpression of $\bar{f}$ at some position p, that a message $[w, h]$ exists in the set of handles in the handles field of r, and that f' is either $\bar{f}[p \leftarrow \mathbf{0}]$ or $\bar{f}[p \leftarrow !v]$, depending on whether the return value is a **stop** value or not, respectively. Therefore, we have by [SiteReturn]:

$$\begin{aligned}\langle \bar{f} >x> g,\ lbl : l \mid r\rangle &\rightarrow_{\mathcal{R}^r} \langle \mathsf{sret}(\bar{f} >x> g, w, h),\ lbl : \epsilon,\ \mid r'\rangle \\ &=_{\mathcal{R}^r} \langle \mathsf{sret}(\bar{f}, w, h) >x> g,\ lbl : \epsilon,\ \mid r'\rangle \\ &=_{\mathcal{R}^r} \langle f' >x> g,\ lbl : \epsilon,\ \mid r'\rangle\end{aligned}$$

The inductive cases for [Asym2I] and [Asym2E] are, respectively, similar to [SymI] and [SymE]. The cases for [Asym1V] and [OtherV] are similar to the value publishing case of [Seq1V], while the cases for [Asym1NI] and [OtherNI] are similar to the internal non-publishing case of [Seq1NI]. Finally, the cases for the external site return action, namely [Asym1NEa], [Asym1NEb] and [OtherNE] are similar to [Seq1NE].

($\Longleftarrow$) If $\mathcal{C} \rightarrow_{\mathcal{R}^r} \mathcal{C}'$ is an instance of the [TICK] rule (i.e., $\mathcal{C}$ is not an eager configuration), then the implication holds trivially by the corresponding [TICK] rule in $\mathcal{R}^s$ by Lemma 9. So, suppose that the rewrite in the hypothesis is not an instance of the tick rule. Then, we observe that it must be an instance of an instantaneous action, which can either be an internal action (with the [IACTION] rule) or a site return action (using the [SITERETURN] rule). This implies that the expression component f of $\mathcal{C}$ is either active or inactive (i.e., non-zero). To complete the proof, we proceed by induction on f.

If f is a base active expression (i.e. $M(\vec{v})$, $E(\vec{p})$, or $!v$), then the implication holds easily by the equations in $\mathcal{R}^r$, the assumption that $\mathcal{C}$ is closed, and the corresponding base rules ([SITECALL], [DEF], and [PUBLISH]) for these expressions in $\mathcal{R}^s$. Similarly, if f is a base inactive expression, namely the handle expression $?h$, then the implication follows using the corresponding site return rules in $\mathcal{R}^s$ ([SITERETV] and [SITERETSTOP], for the cases of returning an Orc value or a **stop** value, respectively).

Suppose that the expression f is of the form $f \mid g$. If the hypothesis is an instance of an internal action i, then, modulo commutativity, it must be of the form

$$\langle \hat{f} \mid g,\ lbl : l \mid r \rangle \rightarrow_{\mathcal{R}^r} \langle f' \mid g,\ lbl : \epsilon \mid r' \rangle$$

with $\hat{f}$ having as a sub-expression a base active expression at some position p. Then, $\hat{f} \rightarrow_{\mathcal{R}^r} \mathsf{act}^{\uparrow}(f'', i)$, with $f'' = \hat{f}[p \leftarrow b]$, for some position p in $\hat{f}$ and b is either *tmp* or $\mathbf{0}$, depending on whether the internal action is a (site or expression) call or a publishing of a value, respectively. By the equations defining $\mathsf{act}^{\uparrow}$ for internal actions, this implies $\langle \hat{f},\ lbl : \epsilon \mid r \rangle \rightarrow_{\mathcal{R}^r} \langle f',\ lbl : \epsilon \mid r' \rangle$, which by the induction hypothesis implies $(\cdot\langle \hat{f},\ lbl : \epsilon \mid r \rangle) \rightarrow_{\mathcal{R}^s} (\langle f', lbl : i \mid r' \rangle)$, and thus the conclusion holds by the rule SYMI in $\mathcal{R}^s$. If the action is an instance of a site return u, then the hypothesis is of the form:

$$\langle \bar{f} \mid \bar{g},\ lbl : l \mid r \rangle \rightarrow_{\mathcal{R}^r} \langle f' \mid \bar{g},\ lbl : \epsilon \mid r' \rangle$$

with $\bar{f}$ having as a sub-expression a handle expression of the form $?h$, and r having in its messages field an incoming message of the form $[w, h]$, and in its handles field a handle h. By

the [SiteReturn] rule and the equations defining sret for site return actions, this implies $\langle \bar{f},\ lbl : l \mid r\rangle \rightarrow_{\mathcal{R}^r} \langle f',\ lbl : \epsilon \mid r'\rangle$. The induction hypothesis, and the [SymE] rule in $\mathcal{R}^s$ imply the desired conclusion.

Suppose f is of the form $f >x> g$. There are three cases. First, the hypothesis may be an instance of a publishing action, and hence of the form

$$\langle \hat{f} >x> g,\ lbl : l \mid r\rangle) \rightarrow_{\mathcal{R}^r} (\langle (f' >x> g) \mid [v/x]g,\ lbl : \epsilon \mid r'\rangle$$

with $\hat{f}$ having as a sub-expression a base publishing expression $!v$ such that v is not bound in $\hat{f}$. By the [IAction] and [Publish] rules, and the equations defining $\mathsf{act}^{\uparrow}$ for publishing actions, this implies $\langle \hat{f}, lbl : \epsilon \mid r\rangle) \rightarrow_{\mathcal{R}^r} (\langle f', lbl : \epsilon \mid r'\rangle$. By induction, and rule [Seq1V] in $\mathcal{R}^s$, the conclusion holds.

The second case is when the action is a *non-publishing* internal action (i.e. a site call or a τ action). In this case the hypothesis has the form:

$$\langle \hat{f} >x> g,\ lbl : l \mid r\rangle \rightarrow_{\mathcal{R}^r} \langle f' >x> g,\ lbl : \epsilon \mid r'\rangle$$

where, by the [IAction] rule, $\hat{f}$ has as a sub-expression either a site call, an expression call or a publishing expression in a value-binding position. Again, [IAction] and the equations defining $\mathsf{act}^{\uparrow}$ for non-publishing internal actions, imply $\langle \hat{f}, lbl : \epsilon \mid r\rangle) \rightarrow_{\mathcal{R}^r} (\langle f', lbl : \epsilon \mid r'\rangle$. Induction and rule [Seq1NI] in $\mathcal{R}^s$ complete the proof of this case

Finally, the action may be an external site return action, and the hypothesis has the form:

$$\langle \bar{f} >x> g,\ lbl : l \mid r\rangle \rightarrow_{\mathcal{R}^r} \langle f' >x> g,\ lbl : \epsilon \mid r'\rangle$$

which, by the [SiteReturn] rule, the equations defining *sret*, the inductive hypothesis and the $\mathcal{R}^s$ rule [Seq1NE], implies the desired conclusion.

The inductive cases for the asymmetric parallel composition and the otherwise composition are similar and follow by induction and the corresponding rules in $\mathcal{R}^s$. □

APPENDIX C

MAUDE SPECIFICATIONS OF THE REWRITING SEMANTICS AND IMPLEMENTATION OF ORC

The complete specifications in Maude of the rewriting logic semantics of Orc discussed in Chapter 3 and Chapter 4 can be found in supplemental files named as follows:

- `orc-syntax.maude`, which specifies the syntax of Orc and the CINNI substitution instance of Orc.
- `orc-infrastructure.maude`, which specifies the semantic infrastructure, described in Section 3.1.
- `orc-sos-semantics.maude`, which specifies the SOS-based rewriting semantics of Orc, described in Section 3.2.
- `orc-red-semantics.maude`, which specifies the reduction rewriting semantics of Orc, described in Section 3.3.
- `orc-object-semantics.maude`, which specifies the object-based rewriting semantics of Orc, described in Section 4.1.
- `orc-sites.maude`, which specifies a set of basic Orc sites for simulation and analysis.

The complete specifications of the rewriting-based, distributed implementation of Orc in Maude, discussed in Chapter 5 can be found in a supplemental file named `dist-orc.maude`. The specifications of the formal model of the implementation specified in Real-Time Maude can be found in a supplemental file named `dist-orc-model.maude`.

APPENDIX D

MAUDE SPECIFICATIONS OF THE ASV PROTOCOL

The specifications in Maude of the rewriting logic model of the ASV protocol and its variants, discussed in Section 6.3 can be found in supplemental files named as follows:

- `apmaude.maude`, which specifies the general structure of a communication system and the Maude sampler module.
- `common.maude`, which specifies the common infrastructure and behavior across all variants of the protocol.
- `omniscient.maude`, which specifies behaviors specific to the omniscient protocol.
- `asv.maude`, which specifies behaviors specific to the ASV protocol.
- `sv-aggressive.maude`, which specifies behaviors specific to the aggressive, selective verification protocol.
- `sv-naive.maude`, which specifies behaviors specific to the naive, selective verification protocol.

APPENDIX E

MAUDE SPECIFICATIONS OF THE ASV WRAPPERS

The specifications in Maude of the generic ASV protocol wrappers and their variants, discussed in Chapter 7 can be found in supplemental files named as follows:

- `apmaude.maude`, which specifies the general structure of a communication system and the Maude sampler module.
- `wrappers-common.maude`, which specifies the common infrastructure and behavior across all variants of the wrappers.
- `wrappers-asv.maude`, which specifies behaviors specific to the ASV wrappers.
- `wrappers-aggressive.maude`, which specifies behaviors specific to the aggressive, selective verification wrappers.
- `wrappers-naive.maude`, which specifies behaviors specific to the naive, selective verification wrappers.
- `wrappers-nosv.maude`, which specifies behaviors specific to the naive wrappers with no selective verification.
- `orc-syntax.maude`, which specifies the syntax and structure of sequential Orc expressions.
- `orc-semantics.maude`, which specifies the semantics of sequential Orc.
- `orc-analysis.maude`, which specifies the Orc orchestration pattern used in the analysis of Section 7.4.3.

REFERENCES

[1] K. Sen, M. Viswanathan, and G. Agha, "On statistical model checking of stochastic systems," in *Computer Aided Verification (CAV 2005)*, ser. LNCS, vol. 3576. Springer, 2005.

[2] G. Agha, J. Meseguer, and K. Sen, "PMaude: Rewrite-based specification language for probabilistic object systems," *Electronic Notes in Theoretical Computer Science*, vol. 153, no. 2, pp. 213–239, 2006.

[3] M. A. Arbib, *Theories of abstract automata (Prentice-Hall series in automatic computation)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1969.

[4] T. L. Booth, *Sequential Machines and Automata Theory (1st ed.)*. New York, USA: John Wiley and Sons, Inc., 1967.

[5] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994.

[6] Y. Gurevich, "Evolving algebras 1993: Lipari guide," in *Specification and validation methods*, E. Börger, Ed. New York, NY, USA: Oxford University Press, Inc., 1995, pp. 9–36.

[7] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[8] K. Jensen and L. M. Kristensen, *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer, 2009.

[9] G. A. Agha, F. De Cindio, and G. Rozenberg, *Concurrent object-oriented programming and Petri nets: advances in Petri nets*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2001.

[10] J. Meseguer, "Conditional rewriting logic as a unified model of concurrency," *Theor. Comput. Sci.*, vol. 96, no. 1, pp. 73–155, 1992.

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

[12] R. Milner, *Communicating and Mobile Systems : The π-Calculus*. Cambridge University Press, 1999.

[13] C. A. R. Hoare, *Communicating sequential processes*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985.

[14] J. C. M. Baeten and C. A. Middelburg, *Process algebra with timing; Monographs in theoretical computer science*. Berlin; New York: Springer, 2002.

[15] C. Fournet and G. Gonthier, "The reflexive CHAM and the Join calculus," in *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM Press, 1996, pp. 372–385.

[16] ITU-T, "Recommendation Z.100(08/02), languages and general software aspects for telecom. systems - specification and description language (SDL)," August 2002.

[17] OMG, *UML Profile for Schedulability, Performance, and Time Specification, Version 1.1*, January 2005.

[18] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, 1987.

[19] D. Drusinsky, *Modeling and Verification Using UML Statecharts*. Elsevier, 2006.

[20] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All About Maude - A High-Performance Logical Framework*, ser. LNCS. Secaucus, NJ, USA: Springer-Verlag, 2007, vol. 4350.

[21] P. Blackburn, J. F. A. K. v. Benthem, and F. Wolter, *Handbook of Modal Logic, Volume 3 (Studies in Logic and Practical Reasoning)*. New York, NY, USA: Elsevier Science Inc., 2006.

[22] A. Pnueli, "The temporal logic of programs," *Foundations of Computer Science, Annual IEEE Symposium on*, vol. 0, pp. 46–57, 1977.

[23] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[24] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of JML: a behavioral interface specification language for java," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 3, pp. 1–38, 2006.

[25] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Comput. Surv.*, vol. 41, no. 4, pp. 1–36, 2009.

[26] M. Wilding, D. Greve, and D. Hardin, "Efficient simulation of formal processor models," *Form. Methods Syst. Des.*, vol. 18, no. 3, pp. 233–248, 2001.

[27] G. Berry, "Synchronous design and verification of critical embedded systems using SCADE and Esterel," in *Proceedings of the Formal Methods for Industrial Critical Systems*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg Germany: Springer-Verlag, 2008, vol. 4916.

[28] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[29] D. Ballis and T. Kutsia, "WWV'09 - Automated Specification and Verification of Web Systems," Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria, Tech. Rep. 09-10, July 2009.

[30] M. Bravetti and T. Bultan, Eds., *Web Services and Formal Methods - 7th International Workshop, WS-FM 2010, Hoboken, NJ, USA, September 16-17, 2010. Revised Selected Papers*, ser. Lecture Notes in Computer Science, vol. 6551. Springer, 2011.

[31] M. Lumpe and L. Barbosa, Eds., *Formal Aspects of Component Software - 7th International Workshop, FACS 2010, Guimaraes, Portugal, October 14-16, 2010. Revised Selected Papers*, ser. Lecture Notes in Computer Science (to appear). Springer, 2011.

[32] OASIS WSBPEL TC, *Web Services Business Process Execution Language Version 2.0*, April 2007, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf.

[33] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson, *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.

[34] J. Misra, "Computation orchestration: A basis for wide-area computing," in *Proc. of the NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems*, ser. NATO ASI Series, M. Broy, Ed., Marktoberdorf, Germany, 2004.

[35] J. Misra and W. R. Cook, "Computation orchestration: A basis for wide-area computing," *Journal of Software and Systems Modeling*, vol. 6, no. 1, pp. 83–110, March 2007.

[36] R. Bruni, "Calculi for service-oriented computing," in *Proceedings of SFM'09*, ser. LNCS, vol. 5569. Springer, 2009, pp. 1–41.

[37] G. L. Ferrari, R. Guanciale, and D. Strollo, "JSCL: A middleware for service coordination," in *Proceedings of FORTE'06*, ser. Lecture Notes in Computer Science, vol. 4229. Springer, 2006, pp. 46–60.

[38] L. Cardelli and A. D. Gordon, "Mobile ambients," *Theor. Comput. Sci.*, vol. 240, no. 1, pp. 177–213, 2000.

[39] M. Bartoletti, P. Degano, G. Ferrari, and R. Zunino, "Semantics-based design for secure web services," *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 33–49, 2008.

[40] M. Rouached and C. Godart, "Securing web service compositions: Formalizing authorization policies using event calculus," in *Proc. of ICSOC'06: 4th International Conference Service-Oriented Computing*, ser. Lecture Notes in Computer Science, vol. 4294. Springer, 2006, pp. 440–446.

[41] F. van Breugel and M. Koshkina, "Models and verification of BPEL," September 2006, http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf.

[42] N. Lohmann, "A feature-complete Petri net semantics for WS-BPEL 2.0," in *Proceedings of WS-FM'07*, ser. Lecture Notes in Computer Science, vol. 4937. Springer, 2008, pp. 77–91.

[43] R. Lucchi and M. Mazzara, "A pi-calculus based semantics for WS-BPEL," *Journal of Logic and Algebraic Programming*, vol. 70, no. 1, pp. 96–118, 2007.

[44] A. Lapadula, R. Pugliese, and F. Tiezzi, "A formal account of WS-BPEL," *Lecture notes in computer science*, vol. 5052, p. 199, 2008.

[45] S. M. Specht and R. B. Lee, "Distributed denial of service: Taxonomies of attacks, tools, and countermeasures," in *Proceedings of the 17th International Conference on Parallel and Distributed Computing Systems*. ISCA, 2004, pp. 543–550.

[46] C. Meadows, "A cost-based framework for analysis of denial of service in networks," *Journal of Computer Security*, vol. 9, no. 1, pp. 143–164, 2001.

[47] S. Lafrance and J. Mullins, "An information flow method to detect denial of service vulnerabilities," *J. UCS*, vol. 9, no. 11, pp. 1350–1369, 2003.

[48] M. Abadi, B. Blanchet, and C. Fournet, "Just fast keying in the pi calculus," in *Prog. Lang. and Systems, 13th European Symposium on Programming*, ser. LNCS, vol. 2986. Springer, 2004.

[49] C.-F. Yu and V. D. Gligor, "A specification and verification method for preventing denial of service," *IEEE Trans. Softw. Eng.*, vol. 16, no. 6, pp. 581–592, 1990.

[50] A. Mahimkar and V. Shmatikov, "Game-based analysis of denial-of-service prevention protocols," in *IEEE Computer Security Foundations Workshop, (CSFW-18 2005)*. IEEE Computer Society, 2005.

[51] A. E. Goodloe, "A foundation for tunnel-complex protocols," Ph.D. dissertation, University of Pennsylvania, 2008.

[52] M. AlTurki and J. Meseguer, "Real-time rewriting semantics of Orc," in *PPDP '07: Proceedings of the 9th ACM SIGPLAN international symposium on Principles and practice of declarative programming*. New York, NY, USA: ACM Press, 2007, pp. 131–142.

[53] M. AlTurki and J. Meseguer, "Reduction semantics and formal analysis of Orc programs," *Electron. Notes Theor. Comput. Sci.*, vol. 200, no. 3, pp. 25–41, 2008.

[54] M. AlTurki and J. Meseguer, "Rewriting logic semantics of Orc," University of Illinois at Urbana Champaign, Tech. Rep. UIUCDCS-R-2007-2918, November 2007.

[55] M. AlTurki and J. Meseguer, "Dist-Orc: A rewriting-based distributed implementation of Orc with formal analysis," in *The 1st International Workshop on Rewriting Techniques for Real-Time Systems (RTRTS)*, Longyearbyen, Spitsbergen, Norway, April 2010.

[56] M. AlTurki and J. Meseguer, "Dist-Orc: A rewriting-based distributed implementation of Orc with formal analysis," University of Illinois at Urbana Champaign, Tech. Rep., April 2010, http://hdl.handle.net/2142/15414.

[57] M. AlTurki and J. Meseguer, "PVeStA: A parallel statistical model checking and quantitative analysis tool," in *Proceedings of CALCO'11: The 4th International Conference on Algebra and Coalgebra in Computer Science, Winchester, UK*, August 2011.

[58] M. AlTurki, J. Meseguer, and C. A. Gunter, "Probabilistic modeling and analysis of DoS protection for the ASV protocol," *Electron. Notes Theor. Comput. Sci.*, vol. 234, pp. 3–18, 2009.

[59] J. Meseguer, "Rewriting as a unified model of concurrency," in *Proceedings of the Concur'90 Conference, Amsterdam, August 1990.* Springer LNCS 458, 1990, pp. 384–400.

[60] J. Meseguer, "Rewriting logic as a semantic framework for concurrency: a progress report," in *CONCUR '96: Concurrency Theory*, ser. Lecture Notes in Computer Science, U. Montanari and V. Sassone, Eds. Springer Berlin / Heidelberg, 1996, vol. 1119, pp. 331–372.

[61] J. Meseguer and G. Rosu, "The rewriting logic semantics project," *Theor. Comput. Sci.*, vol. 373, no. 3, pp. 213–237, 2007.

[62] T. F. Serbanuta, G. Rosu, and J. Meseguer, "A rewriting logic approach to operational semantics," *Information and Computation*, vol. 207, no. 2, pp. 305 – 340, 2009, special issue on Structural Operational Semantics (SOS).

[63] J. Meseguer and G. Rosu, "The rewriting logic semantics project: A progress report," in *Proc. 18th Intl. Symp. on Fundamentals of Computation Theory (FCT 2011) (to appear)*, ser. LNCS. Oslo, Norway: Springer, August 2011.

[64] P. C. Ölveczky and J. Meseguer, "Specification of real-time and hybrid systems in rewriting logic," *Theoretical Computer Science*, vol. 285, pp. 359–405, August 2002.

[65] K. Sen, N. Kumar, J. Meseguer, and G. Agha, "Probabilistic rewrite theories: Unifying models, logics and tools," University of Illinois at Urbana Champaign, Tech. Rep. UIUCDCS-R-2003-2347, May 2003.

[66] J. Meseguer and R. Sharykin, "Specification and analysis of distributed object-based stochastic hybrid systems," in *Hybrid Systems: Computation and Control (HSCC 2006)*, ser. LNCS, vol. 3927. Springer, 2006.

[67] J. Meseguer, "Membership algebra as a logical framework for equational specification," in *Proc. WADT'97*, ser. LNCS, F. Parisi-Presicce, Ed., vol. 1376. Springer, 1998, pp. 18–61.

[68] R. Bruni and J. Meseguer, "Semantic foundations for generalized rewrite theories," *Theor. Comput. Sci.*, vol. 360, no. 1-3, pp. 386–414, 2006.

[69] P. C. Ölveczky and J. Meseguer, "Semantics and pragmatics of Real-Time Maude," *Higher-Order and Symbolic Computation*, vol. 20, no. 1-2, pp. 161–196, 2007.

[70] P. Viry, "Equational rules for rewriting logic," *Theor. Comput. Sci.*, vol. 285, no. 2, pp. 487–517, 2002.

[71] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton, "Model-checking continuous-time Markov chains," *ACM Trans. Comput. Logic*, vol. 1, no. 1, pp. 162–170, 2000.

[72] K. Sen, M. Viswanathan, and G. A. Agha, "VESTA: A statistical model-checker and analyzer for probabilistic systems," in *Second International Conference on the Quantitative Evaluation of Systems (QEST)*, 2005, pp. 251–252.

[73] H. L. S. Younes and R. G. Simmons, "Probabilistic verification of discrete event systems using acceptance sampling," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, vol. 2404. Springer, 2002, pp. 223–235.

[74] P. C. Ölveczky, "Real-Time Maude 2.3 manual," August 2007, http://heim.ifi.uio.no/~peterol/RealTimeMaude/.

[75] P. C. Ölveczky and J. Meseguer, "Abstraction and completeness for Real-Time Maude," *Electron. Notes Theor. Comput. Sci.*, vol. 176, no. 4, pp. 5–27, 2007.

[76] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns," *Distrib. Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003.

[77] D. Kitchin, A. Quark, W. Cook, and J. Misra, "The Orc programming language," in *Formal techniques for Distributed Systems; Proc. of FMOODS/FORTE*, ser. LNCS, D. Lee, A. Lopes, and A. Poetzsch-Heffter, Eds., vol. 5522. Springer, 2009, pp. 1–25.

[78] D. Kitchin, E. Powell, and J. Misra, "Simulation using orchestration," *Proceedings of AMAST*, Jan 2008.

[79] W. Cook, S. Patwardhan, and J. Misra, "Workflow patterns in Orc," in *Coordination Models and Languages*, ser. Lecture Notes in Computer Science, P. Ciancarini and H. Wiklicky, Eds. Springer Berlin / Heidelberg, 2006, vol. 4038, no. 96, pp. 82–96.

[80] D. Kitchin, A. Quark, and J. Misra, "Quicksort: Combining concurrency, recursion, and mutable data structures," in *Reflections on the Work of C.A.R. Hoare*, ser. History of Computing, A. Roscoe, C. B. Jones, and K. R. Wood, Eds. Springer London, 2010, pp. 229–254.

[81] I. Wehrman, D. Kitchin, W. R. Cook, and J. Misra, "A timed semantics of Orc," *Theor. Comput. Sci.*, vol. 402, no. 2-3, pp. 234–248, 2008.

[82] D. Kitchin, W. R. Cook, and J. Misra, "A language for task orchestration and its semantic properties." in *CONCUR 2006*, ser. Lecture Notes in Computer Science, vol. 4137. Springer, 2006, pp. 477–491.

[83] I. Wehrman, D. Kitchin, W. Cook, and J. Misra, "Properties of the timed operational and denotational semantics of Orc," University of Texas at Austin, Tech. Rep., 2007. [Online]. Available: http://orc.csres.utexas.edu/papers/tcs07-tr.pdf

[84] T. Hoare, G. Menzel, and J. Misra, "A tree semantics of an orchestration language," *Engineering Theories of Software Intensive Systems*, pp. 331–350, 2005.

[85] M.-O. Stehr, "CINNI — A generic calculus of explicit substitutions and its application to λ-, ς- and π-calculi," in *Proceedings Third International Workshop on Rewriting Logic and its Applications, WRLA 2000, Kanazawa, Japan, September 18–20, 2000*, ser. Electronic Notes in Theoretical Computer Science, K. Futatsugi, Ed., vol. 36. Elsevier, 2000, pp. 71–92.

[86] P. D. Mosses, "Modular structural operational semantics." *J. Log. Algebr. Program.*, vol. 60-61, pp. 195–228, 2004.

[87] J. Meseguer and C. Braga, "Modular rewriting semantics of programming languages," *Algebraic Methodology and Software Technology*, pp. 364–378, 2004.

[88] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *Maude Manual (Version 2.6)*, January 2011, http://maude.cs.uiuc.edu/maude2-manual/maude-manual.pdf.

[89] F. Durán and J. Meseguer, "A Church-Rosser checker tool for conditional order-sorted equational Maude specifications," in *Rewriting Logic and Its Applications*, ser. Lecture Notes in Computer Science, P. Ölveczky, Ed. Springer Berlin / Heidelberg, 2010, vol. 6381, pp. 69–85.

[90] F. Durán and J. Meseguer, "A Maude coherence checker tool for conditional order-sorted rewrite theories," in *Rewriting Logic and Its Applications*, ser. Lecture Notes in Computer Science, P. Ölveczky, Ed. Springer Berlin / Heidelberg, 2010, vol. 6381, pp. 86–103.

[91] F. Durán, S. Lucas, and J. Meseguer, "MTT: The Maude termination tool (system description)," in *Automated Reasoning*, ser. Lecture Notes in Computer Science, A. Armando, P. Baumgartner, and G. Dowek, Eds. Springer Berlin / Heidelberg, 2008, vol. 5195, pp. 313–319.

[92] J. Giesl, P. Schneider-Kamp, and R. Thiemann, "AProVE 1.2: Automatic termination proofs in the dependency pair framework," in *Automated Reasoning*, ser. Lecture Notes in Computer Science, U. Furbach and N. Shankar, Eds. Springer Berlin / Heidelberg, 2006, vol. 4130, pp. 281–286.

[93] F. Durán, S. Lucas, C. Marché, J. Meseguer, and X. Urbain, "Proving operational termination of membership equational programs," *Higher-Order and Symbolic Computation, to appear*, vol. 21, no. 1-2, pp. 59–88, 2008.

[94] J. Meseguer, "A logical theory of concurrent objects and its realization in the Maude language," in *Research Directions in Concurrent Object-Oriented Programming*, G. Agha, P. Wegner, and A. Yonezawa, Eds. MIT Press, 1993, pp. 314–390.

[95] L. Lamport, "A fast mutual exclusion algorithm," *ACM Trans. Comput. Syst.*, vol. 5, pp. 1–11, January 1987.

[96] A. Riesco and A. Verdejo, "Distributed applications implemented in Maude with parameterized skeletons," in *Proc. of FMOODS '07*, ser. Lecture Notes in Computer Science, M. M. Bonsangue and E. B. Johnsen, Eds., vol. 4468. Springer, 2007, pp. 91–106.

[97] F. Durán, A. Riesco, and A. Verdejo, "A distributed implementation of Mobile Maude," *Electron. Notes Theor. Comput. Sci.*, vol. 176, no. 4, pp. 113–131, 2007.

[98] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[99] F. Mattern, "Virtual time and global states of distributed systems," in *Proc. Workshop on Parallel and Distributed Algorithms*, C. M. et al., Ed., North-Holland / Elsevier, 1989, (Reprinted in: Z. Yang, T.A. Marsland (Eds.), "Global States and Time in Distributed Systems", IEEE, 1994, pp. 123-133.). pp. 215–226.

[100] C. J. Fidge, "Timestamps in message-passing systems that preserve partial ordering," in *Proceedings of the 11th Australian Computer Science Conference*, Feb. 1988, pp. 56–66.

[101] H. L. S. Younes and R. G. Simmons, "Statistical probabilistic model checking with a focus on time-bounded properties," *Inf. Comput.*, vol. 204, no. 9, pp. 1368–1409, 2006.

[102] G. Agha, C. A. Gunter, M. Greenwald, S. Khanna, J. Meseguer, K. Sen, and P. Thati, "Formal modeling and analysis of DoS using probabilistic rewrite theories," in *International Workshop on Foundations of Computer Security (FCS'05)*. Chicago, IL: IEEE, June 2005.

[103] M. Katelman, J. Meseguer, and J. Hou, "Redesign of the LMST wireless sensor protocol through formal modeling and statistical model checking," in *Proc. of FMOODS '08*, ser. Lecture Notes in Computer Science, vol. 5051. Berlin, Heidelberg: Springer, 2008, pp. 150–169.

[104] M. Kim, M.-O. Stehr, C. L. Talcott, N. D. Dutt, and N. Venkatasubramanian, "A probabilistic formal analysis approach to cross layer optimization in distributed embedded systems," in *Proc. of FMOODS '07*, ser. LNCS, vol. 4468. Springer, 2007.

[105] S. Khanna, S. S. Venkatesh, O. Fatemieh, F. Khan, and C. A. Gunter, "Adaptive selective verification," in *IEEE Conference on Computer Communications (INFOCOM '08)*. Phoenix, AZ: IEEE, April 2008.

[106] H. Hansson and B. Jonsson, "A logic for reasoning about time and reliability," *Formal Aspects of Computing*, vol. 6, no. 5, pp. 512–535, 09 1994.

[107] A. Aziz, V. Singhal, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "It usually works: The temporal logic of stochastic systems," in *7th International Conference On Computer Aided Verification*, P. Wolper, Ed., vol. 939. Liege, Belgium: Springer Verlag, 1995, pp. 155–165.

[108] C. Baier, J.-P. Katoen, and H. Hermanns, "Approximative symbolic model checking of continuous-time markov chains," in *CONCUR'99 Concurrency Theory*, ser. Lecture Notes in Computer Science, J. Baeten and S. Mauw, Eds. Springer Berlin / Heidelberg, 1999, vol. 1664, pp. 781–781.

[109] D. Dolev and A. Yao, "On the security of public key protocols," *Information Theory, IEEE Transactions on*, vol. 29, no. 2, pp. 198–208, 1983.

[110] C. A. Gunter, S. Khanna, K. Tan, and S. S. Venkatesh, "DoS protection for reliably authenticated broadcast," in *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, 2004.

[111] R. Sparks, S. Lawrence, A. Hawrylyshen, and B. Campen, "Addressing an Amplification Vulnerability in Session Initiation Protocol (SIP) Forking Proxies," RFC 5393 (Proposed Standard), Dec. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5393.txt

[112] R. Shankesi, M. AlTurki, R. Sasse, C. A. Gunter, and J. Meseguer, "Model-checking DoS amplification for VoIP session initiation," in *Proc. of ESORICS'09: 14th European Symposium on Research in Computer Security*, ser. Lecture Notes in Computer Science, vol. 5789. Springer, 2009, pp. 390–405.

[113] M. Jensen, N. Gruschka, and R. Herkenhöner, "A survey of attacks on web services," *Computer Science - Research and Development*, vol. 24, no. 4, pp. 185–197, 11 2009.

[114] A. Singhal, T. Winograd, and K. Scarfone, "Guide to secure web services: Recommendations of the national institute of standards and technology," Special Publication 800-95, pp. 800–95, August 2007, http://csrc.nist.gov/publications/nistpubs/800-95/SP800-95.pdf.

[115] R. Chadha, C. A. Gunter, J. Meseguer, R. Shankesi, and M. Viswanathan, "Modular preservation of safety properties by cookie-based DoS-protection wrappers," in *Proc. of FMOODS '08*, ser. Lecture Notes in Computer Science, vol. 5051. Springer, 2008, pp. 39–58.

[116] G. Agha, S. Frølund, R. Panwar, and D. Sturman, "A linguistic framework for dynamic composition of dependability protocols," in *in C. E. Landwehr, B. Randell, and L. Simoncini (editors), Dependable Computing and Fault-Tolerant Systems VIII, pp 345-363, IFIP Transactions, Springer-Verlag*, 1993.

[117] G. Denker, J. Meseguer, and C. Talcott, "Rewriting semantics of meta-objects and composable distributed services," *Electronic Notes in Theoretical Computer Science*, vol. 36, pp. 405–425, 2000.

[118] J. Meseguer and C. L. Talcott, "Semantic models for distributed object reflection," in *ECOOP 2002 - Object-Oriented Programming, 16th European Conference, Malaga, Spain, June 10-14, 2002, Proceedings*, ser. Lecture Notes in Computer Science, vol. 2374. Springer, 2002, pp. 1–36.

[119] G. Norman, C. Palamidessi, D. Parker, and P. Wu, "Model checking probabilistic and stochastic extensions of the π-calculus," *IEEE Transactions on Software Engineering*, vol. 35, no. 2, pp. 209–223, March 2009.

[120] M. AlTurki, D. Dhurjati, D. Yu, A. Chander, and H. Inamura, "Formal specification and analysis of timing properties in software systems," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, M. Chechik and M. Wirsing, Eds., vol. 5503. Springer, 2009, pp. 262–277.

[121] K. Bae, P. C. Ölveczky, T. H. Feng, and S. Tripakis, "Verifying ptolemy ii discrete-event models using real-time maude," in *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings*, ser. Lecture Notes in Computer Science, K. Breitman and A. Cavalcanti, Eds., vol. 5885. Springer, 2009, pp. 717–736.

[122] K. Bae and P. C. Ölveczky, "Extending the real-time maude semantics of ptolemy to hierarchical de models," in *Proceedings First International Workshop on Rewriting Techniques for Real-Time Systems*, ser. EPTCS, P. C. Ölveczky, Ed., vol. 36, 2010, pp. 46–66.

[123] P. Ölveczky, A. Boronat, and J. Meseguer, "Formal semantics and analysis of behavioral aadl models in real-time maude," in *Formal Techniques for Distributed Systems*, ser. Lecture Notes in Computer Science, J. Hatcliff and E. Zucca, Eds. Springer Berlin / Heidelberg, 2010, vol. 6117, pp. 47–62.

[124] S. Rosario, D. Kitchin, A. Benveniste, W. Cook, S. Haar, and C. Jard, "Event structure semantics of Orc," in *WS-FM 2007*, ser. Lecture Notes in Computer Science, vol. 4937. Springer, 2008, pp. 154–168.

[125] W. R. Cook and J. Misra, "A structured orchestration language," July 2005, http://www.cs.utexas.edu/users/wcook/Drafts/OrcCookMisra05.pdf.

[126] R. Bruni, H. Melgratti, and E. Tuosto, "Translating Orc features into Petri nets and the Join calculus," in *Web Services and Formal Methods*, ser. Lecture Notes in Computer Science, M. Bravetti, M. Núñez, and G. Zavattaro, Eds., vol. 4184. Springer, 2006, pp. 123–137.

[127] J. S. Dong, Y. Liu, J. Sun, and X. Zhang, "Verification of computation orchestration via timed automata," in *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, Macao, China, November 1-3, 2006, Proceedings*, ser. Lecture Notes in Computer Science, Z. Liu and J. He, Eds., vol. 4260. Springer, 2006, pp. 226–245.

[128] X. Nicollin and J. Sifakis, "The algebra of timed processes ATP: Theory and application," *Information and Computation*, vol. 114, no. 1, pp. 131–178, 1994.

[129] M. Hennessy and T. Regan, "A process algebra for timed systems," *Inf. Comput.*, vol. 117, no. 2, pp. 221–239, 1995.

[130] J. C. M. Baeten and J. A. Bergstra, "Real time process algebra." *Formal Aspects of Computing*, vol. 3, no. 2, pp. 142–188, 1991.

[131] L. Chen, "An interleaving model for real-time systems," in *TVER '92: Proceedings of the Second International Symposium on Logical Foundations of Computer Science*. London, UK: Springer-Verlag, 1992, pp. 81–92.

[132] S. Schneider, J. Davies, D. M. Jackson, G. M. Reed, J. N. Reed, and A. W. Roscoe, "Timed CSP: Theory and practice," in *Proceedings of the Real-Time: Theory in Practice, REX Workshop*. London, UK: Springer-Verlag, 1992, pp. 640–675.

[133] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on UPPAAL," in *International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM-RT)*, ser. Lecture Notes in Computer Science, M. Bernardo and F. Corradini, Eds. Springer, 2004, vol. 3185, pp. 200–236.

[134] R. Kazhamiakin, P. Pandya, and M. Pistore, "Timed modelling and analysis in web service compositions," in *ARES '06: Proceedings of the First International Conference on Availability, Reliability and Security*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 840–846.

[135] R. Hamadi and B. Benatallah, "A Petri net-based model for web service composition," *Proceedings of the 14th Australasian database conference-Volume 17*, pp. 191–200, 2003.

[136] W. Zhao, Y. Huang, C. Yuan, and L. Wang, "Formalizing business process execution language based on Petri nets," in *Intelligent Systems and Applications (ISA), 2010 2nd International Workshop on*, 22-23 2010, pp. 1 –8.

[137] J. Su, S. Yu, and H. Guo, "Formal description and verification of web service composition based on oopn," in *Advanced Intelligent Computing Theories and Applications. With Aspects of Theoretical and Methodological Issues*, ser. Lecture Notes in Computer Science, D.-S. Huang, D. Wunsch, D. Levine, and K.-H. Jo, Eds. Springer Berlin / Heidelberg, 2008, vol. 5226, pp. 644–652.

[138] W. van der Aalst and C. Stahl, *Modeling Business Processes: A Petri Net Oriented Approach*. Cambridge, MA: MIT Press, 2011.

[139] M. Wirsing, R. De Nicola, S. Gilmore, M. Hölzl, R. Lucchi, M. Tribastone, and G. Zavattaro, "Sensoria process calculi for service-oriented computing," in *Proceedings of TGC'06*, ser. Lecture Notes in Computer Science, vol. 4661, 2007, pp. 30–50.

[140] M. Boreale, R. Bruni, L. Caires, R. D. Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, and D. Sangiorgi, "Scc: a service centered calculus," *Lecture notes in computer science*, vol. 4184, p. 38, 2006.

[141] C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, and G. Zavattaro, "Sock: a calculus for service oriented computing," *Lecture notes in computer science*, vol. 4294, p. 327, 2006.

[142] A. Lapadula, R. Pugliese, and F. Tiezzi, "Cows: A timed service-oriented calculus," *Lecture notes in computer science*, vol. 4711, p. 275, 2007.

[143] C. Guidi, R. Lucchi, and M. Mazzara, "A formal framework for web services coordination," *Electronic Notes in Theoretical Computer Science*, vol. 180, no. 2, pp. 55–70, 2007.

[144] M. Rouached, O. Perrin, and C. Godart, "Towards formal verification of web service composition," *Lecture notes in computer science*, vol. 4102, p. 257, 2006.

[145] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "Model-based verification of web service compositions," in *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, October 2003, pp. 152 – 161.

[146] M. Viroli, "Towards a formal foundation to orchestration languages," *Electronic Notes in Theoretical Computer Science*, vol. 105, pp. 51 – 71, 2004, proceedings of the First International Workshop on Web Services and Formal Methods (WSFM 2004).

[147] F. Abouzaid and J. Mullins, "Formal specification of correlation in WS orchestrations using BP-calculus," *Electronic Notes in Theoretical Computer Science*, vol. 260, pp. 3 – 24, 2010, proceedings of the 5th International Workshop on Formal Aspects of Component Software (FACS 2008).

[148] R. Bruni, R. De Nicola, M. Loreti, and L. Mezzina, "Provably correct implementations of services," in *Trustworthy Global Computing*, ser. Lecture Notes in Computer Science, C. Kaklamanis and F. Nielson, Eds. Springer Berlin / Heidelberg, 2009, vol. 5474, pp. 69–86.

[149] C. Ma, Y. He, N. Xiong, and L. Yang, "Vft: An ontology-based tool for visualization and formalization of web service composition," in *Computational Science and Engineering, 2009. CSE '09. International Conference on*, vol. 1, aug. 2009, pp. 271 –276.

[150] W. van der Aalst and M. Pesic, "Decserflow: Towards a truly declarative service flow language," *Lecture notes in computer science*, vol. 4184, p. 1, 2006.

[151] W. Yeung, "A formal and visual modeling approach to choreography based web services composition and conformance verification," *Expert Systems with Applications*, vol. 38, no. 10, pp. 12 772 – 12 785, 2011.

[152] H. Foster, S. Uchitel, J. Magee, and J. Kramer, "LTSA-WS: a tool for model-based verification of web service compositions and choreography," in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 771–774.

[153] H. Younes, E. Clarke, and P. Zuliani, "Statistical verification of probabilistic properties with unbounded until," in *Formal Methods: Foundations and Applications*, ser. Lecture Notes in Computer Science, J. Davies, L. Silva, and A. Simao, Eds. Springer Berlin / Heidelberg, 2011, vol. 6527, pp. 144–160.

[154] K. Sen, M. Viswanathan, and G. Agha, "Statistical model checking of black-box probabilistic systems," *Computer Aided Verification*, pp. 399–401, 2004.

[155] H. Younes, "Probabilistic verification for "black-box" systems," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, K. Etessami and S. Rajamani, Eds. Springer Berlin / Heidelberg, 2005, vol. 3576, pp. 275–278.

[156] C. Jarque and A. Bera, "A test for normality of observations and regression residuals," *International statistical review*, vol. 55, no. 2, pp. 163–172, 1987.

[157] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: A tool for automatic verification of probabilistic systems," in *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, ser. LNCS, H. Hermanns and J. Palsberg, Eds., vol. 3920. Springer, 2006, pp. 441–444.

[158] T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet, "Approximate probabilistic model checking," in *Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, B. Steffen and G. Levi, Eds. Springer Berlin / Heidelberg, 2004, vol. 2937, pp. 307–329.

[159] B. Jeannet, P. R. D'Argenio, and K. G. Larsen, "RAPTURE: A tool for verifying markov decision processes," in *Tools Day, International Conference on Concurrency Theory, CONCUR'02*, Czech Republic, August 2002, technical Report, Faculty of Informatics at Masaryk University Brno.

[160] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, and M. Siegle, "A markov chain model checker," in *TACAS'00: The 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*. London, UK: Springer-Verlag, 2000, pp. 347–362.

[161] M. Jensen, N. Gruschka, and N. Luttenberger, "The impact of flooding attacks on network-based services," in *Proc. of ARES '08: The International Conference on Availability, Reliability and Security*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 509–513.

[162] M. Jensen and J. Schwenk, "The accountability problem of flooding attacks in service-oriented architectures," in *Proc. of ARES '09: The International Conference on Availability, Reliability and Security*. Washington, DC, USA: IEEE Computer Society, 16-19 2009, pp. 25 –32.

[163] M. Gunestas, M. Mehmet, D. Wijesekera, and A. Singhal, "Forensic web services framework," *IT Professional*, vol. 13, no. 3, pp. 31 –37, may-june 2011.

[164] N. Gruschka and M. Jensen, "Attack surfaces: A taxonomy for attacks on cloud services," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, july 2010, pp. 276 –279.

[165] S. Meng, "QCCS: A formal model to enforce QoS requirements in service composition," in *Theoretical Aspects of Software Engineering, 2007. TASE '07. First Joint IEEE/IFIP Symposium on*, June 2007, pp. 389 –400.

[166] R. Milner, *Communication and concurrency*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[167] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. Cambridge, MA, USA: MIT Press, 1999.

[168] D. Prandi and P. Quaglia, "Stochastic COWS," in *Service-Oriented Computing - ICSOC 2007, Fifth International Conference, Vienna, Austria, September 17-20, 2007, Proceedings*, ser. Lecture Notes in Computer Science, vol. 4749. Springer, 2007, pp. 245–256.

[169] P. Quaglia and S. Schivo, "Approximate model checking of stochastic cows," in *Trustworthly Global Computing - 5th International Symposium, TGC 2010, Munich, Germany, February 24-26, 2010, Revised Selected Papers*, ser. Lecture Notes in Computer Science, M. Wirsing, M. Hofmann, and A. Rauschmayer, Eds., vol. 6084. Springer, 2010, pp. 335–347.

[170] G. Roşu and T. F. Şerbănuţă, "An overview of the K semantic framework," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397 – 434, 2010.

[171] A. Krause, *Foundations of GTK+ Development*. Berkely, CA, USA: Apress, 2007.

AUTHOR'S BIOGRAPHY

Musab Ahmad Al-Turki was born in AlKhobar, Saudi Arabia, on December 25th, 1979. In 2002, He graduated with a first-honors B.Sc. degree in computer science from King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, where he subsequently worked as a graduate assistant teaching introductory computer science courses. In 2003, He joined the department of computer science at the University of Illinois at Urbana-Champaign as a Master's student, where he later received his M.Sc. degree in computer science in formal methods and programming languages in December 2005, after which he immediately started the Ph.D. program in January 2006. Mr. Al-Turki worked as a graduate research assistant at the National Center of Supercomputing Applications in Urbana, Illinois, during the summer of 2006, and as a summer research engineer intern at DOCOMO Labs in Palo Alto, California, during the summer of 2008. He has also been a research assistant at the Formal Methods laboratory in the computer science department at the University of Illinois at Urbana-Champaign since September 2007. Mr. Al-Turki won Saudi Arabia's Higher Education Scholarship Award for graduate studies in 2003 and 2006, and received the prestigious King Abdullah Scholar Award for scientific excellence in 2008 from King Abdullah University of Science and Technology. Following completion of his Ph.D. in August 2011, Mr. Al-Turki will start his academic career as an assistant professor in the Information and Computer Science department at King Fahd University of Petroleum and Minerals.