© 2014 Kyungmin Bae

REWRITING-BASED MODEL CHECKING METHODS

BY

KYUNGMIN BAE

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, 2014

Urbana, Illinois

Doctoral Committee:

Professor José Meseguer, Chair and Director of Research
Professor Gul Agha
Professor Edmund M. Clarke, Carnegie Mellon University
Associate Professor Grigore Roşu

ABSTRACT

Model checking is an automatic technique for verifying concurrent systems [60]. The properties of the system to be verified are typically expressed as temporal logic formulas, while the system itself is formally specified as a certain system specification language, such as computational logics and conventional programming languages. *Rewriting logic* is a highly expressive computational logic for effectively defining a formal executable semantics of a wide range of system specification languages. This dissertation presents new *rewriting-based model checking* methods and tools to effectively verify concurrent systems by means of their rewriting-based formal semantics.

Specifically, this work develops: (i) efficient model checking algorithms and a tool for a suitable property specification language, namely, *linear temporal logic of rewriting* (LTLR) formulas under parameterized fairness; (ii) various infinite-state model checking techniques for LTLR properties, such as equational abstraction, folding abstraction, predicate abstraction, and narrowing-based symbolic model checking; and (iii) the Multirate PALS methodology for making it possible to model check virtually synchronous cyber-physical systems by reducing their system complexity.

To demonstrate rewriting-based model checking, we have developed fully integrated modeling and model checking tools for two widely-used embedded system modeling languages, AADL and Ptolemy II. This approach provides a model-engineering process that combines the advantages of an existing modeling language with automatic rewriting-based model checking.

To my parents and my wife

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor José Meseguer for his sincere support, exemplary guidance, and friendship. His depth of knowledge and creativity has greatly inspired me to become a researcher, and made this work possible. I would also like to extend my appreciation to my committee members: Gul Agha, Edmund Clarke, and Grigore Roşu, for their kindness, valuable comments, and insightful feedbacks.

I would like to thank my research collaborators during my PhD study. Special thanks to Peter Csaba Ölveczky for being my coauthor on many papers, for inviting me to Oslo for a month, and always being a good friend. I am grateful to Santiago Escobar for inviting me to Valencia for two months, and collaborating with me on narrowing-based methods. I am thankful to Abdullah Al-Nayeem and Lui Sha for the fruitful discussions on PALS and AADL. I am grateful to Carolyn Talcott for arranging my visits to SRI, and to Steven Eker for helping me understand the Maude implementation.

I am grateful to the members of my research group for their friendship and many useful discussions: Camilo Rocha, Mu Sun, Mike Katelman, Ralf Sasse, Musab Al-Turki, Stephen Skeirik, Si Liu, Andrew Cholewa, and Fan Yang. I also thank my friends and fellow students for great memories.

Most of all, I thank my wife Sun Hee Lee for her love, friendship, support, and sharing our lives. I also thank my parents, Geum Ju Bae and Seung Hee Kang, my brother, Kyung Woon Bae, and my parents-in-law, Sang Cheol Lee and Soon E Shin, for their support and consistent encouragement.

The work in this dissertation was supported in part by NSF CCF grant 09-05584, NSF CNS grants 07-16638, 08-34709, 09-04749, and 13-19109, AFOSR grant FA8750-11-2-0084, Boeing Corporation grant C8088-557395, Lockheed Martin Corporation, and KFAS scholarship.

TABLE OF CONTENTS

Chapter 1 Introduction 1
1.1 Property Specification Logics 4
1.2 Approximation Methods 6
1.3 Applications to Modeling Languages 8
1.4 Summary of Contributions 9

Part I System and Property Specification Logics 12

Chapter 2 Preliminaries: Rewriting-based System Specifications . . . 13
2.1 Membership Equational Logic 13
2.2 Rewriting Logic 18
2.3 Maude 24

Chapter 3 Linear Temporal Logic of Rewriting 31
3.1 Introduction 31
3.2 Syntax and Semantics 35
3.3 Automata Theoretic LTLR Model Checking 41
3.4 The Maude LTLR Model Checker 46
3.5 Case Study: the Bounded Retransmission Protocol 51
3.6 Concluding Remarks 54

Chapter 4 Model Checking Under Localized Fairness 55
4.1 Introduction 55
4.2 Localized Fairness in Quantified LTLR 59
4.3 Parameterized Fair Model Checking Algorithm 66
4.4 The Maude Fair LTLR Model Checker 75
4.5 Case Study: the Evolving Dining Philosophers 79
4.6 Concluding Remarks 82

Part II Approximation Methods 83

Chapter 5 Infinite-State Model Checking 84
5.1 Introduction 84
5.2 Infinite-State System Examples 88
5.3 Equational Abstraction 93
5.4 Folding Abstraction 101
5.5 Narrowing-based Logical Abstraction 110
5.6 Predicate Abstraction 124
5.7 Concluding Remarks 136

Chapter 6 Multirate PALS 137
6.1 Introduction 137
6.2 Multirate Synchronous Models 141
6.3 Multirate PALS Transformation 149
6.4 Multirate PALS Methodology 161
6.5 Case Study: an Airplane Turning Control System 174
6.6 Concluding Remarks 190

Part III Applications to Modeling Languages 191

Chapter 7 Multirate Synchronous AADL 192
7.1 Introduction 192
7.2 Multirate Synchronous AADL 195
7.3 Real-Time Maude Semantics 205
7.4 The MR-SynchAADL Tool 210
7.5 Case Studies 212
7.6 Concluding Remarks 217

Chapter 8 Ptolemy II Discrete-Event Models 218
8.1 Introduction 218
8.2 Ptolemy II and its DE Model of Computation 221
8.3 The Semantics of Ptolemy II DE Models 226
8.4 Formal Verification in Ptolemy II 244
8.5 Case Studies 246
8.6 Concluding Remarks 251

Chapter 9 Conclusions and Future Work 252
9.1 Summary 252
9.2 Future Work 254

Part IV Appendix 255

Appendix A More LTLR Case Studies and Implementation 256
A.1 More Case Studies 256
A.2 The Model Checker Implementation 273

Appendix B More Details on Multirate PALS 275
B.1 Formalizing Specification of Asynchronous Models 275
B.2 More Details on the Proof 298
B.3 More Details on the Real-Time Maude Framework 303
B.4 The Simplified Asynchronous Model 306

Appendix C More Details on Multirate Synchronous AADL 312
C.1 More Details on the Real-Time Maude Semantics 312
C.2 The Active Standby System Requirements 318
C.3 The Three–Node Active Standby System 320

Appendix D More Details on Ptolemy II DE Models 326
D.1 More Ptolemy II Actors 326
D.2 Real-Time Maude Code Generation 327
D.3 More Details on the Ptolemy II DE Semantics 331

References 353

CHAPTER 1

INTRODUCTION

Computer systems are everywhere, from large-scale infrastructure such as air traffic control systems, to small devices such as smart phones or watches. They have become more integrated with our daily life; for example, modern automobiles often use electronic systems to control braking, speed, airbags, etc. As a consequence, small design errors or software bugs can result in a huge damage or a financial loss, such as rocket explosions or car accidents. It is clearly very important and highly beneficial to verify the safety and correctness of computer systems. Therefore, numerous techniques have been developed in the last few decades, including simulation and testing, static analysis, deductive verification, and model checking.

Model checking [60] refers to a body of *automatic* techniques to verify a *concurrent system* by exhaustively checking any possible behaviors of the system. The system requirements—such as safety, fault tolerance, liveness, etc.—are specified as *temporal logic formulas*, and model checking algorithms verify them by trying to find a *counterexample* that violates one of the system requirements. Unlike simulation and testing, if no counterexamples are found by model checking, then it is guaranteed that the system always satisfies the requirements. This is one of the reasons why model checking is widely used in industry, particularly for safety-critical systems.

To verify a concurrent system by model checking, the first step, called *system specifications*, is to *model* the system with a modeling language or formalism. Generally, two kinds of system specification languages are used by model checking tools: (i) *conventional languages*, such as programming languages (e.g., C, Java, etc.), or software and hardware modeling languages (e.g., Promela, Simulink, Verilog, etc.); and (ii) *mathematical formalisms*, such as process calculi, Petri nets, Boolean programs, automata, etc.

Both types of specification languages have complementary strengths and weaknesses. On the one hand, conventional languages are used directly by modeling engineers to develop the software and hardware products, and have powerful design tools (e.g., debugging and graphical modeling tools). But they sometimes lack a clear mathematical semantics. This implies that model checking may *not* be able to fully guarantee the *correctness* of the system, though it may be helpful for *thoroughly testing* the system. On the other hand, mathematical formalisms have a precise formal semantics. Since mathematical formalisms—particularly those that can be described as *computational* logics—are simple, they allow the use of efficient model checking algorithms. However, they can be comparably harder to use, and may not be able to *directly* express certain complex system features, such as unbounded data structures or distributed objects.[1] This categorization of system specification languages is not only relevant for model checking techniques, but also applies to other formal verification methods.

To overcome such a gap between modeling languages and mathematical formalisms, one promising approach is:

1. using a *very expressive*, yet simple, mathematical formalism in which other modeling languages can be faithfully expressed;

2. defining a *formal mathematical semantics* of a conventional modeling language in the formalism; and

3. verifying a system specified in such a conventional modeling language by means of the formal semantics of the language.

Rewriting logic [132] can be considered as one of such *universal* and highly expressive specification languages, since different models of concurrency (e.g., actors, process calculi, Petri nets, timed and hybrid automata, etc.), programming languages (e.g., C, Java, etc.), and software and hardware modeling languages (e.g., Verilog, AADL, etc.) can be naturally expressed in rewriting logic [137, 140, 141]. There are specification and verification tools for rewriting logic, such as Maude [61], CafeOBJ [71] and ELAN [39]. Rewriting logic specifications are *executable*, so that they can be thoroughly tested with executable system specifications.[2] Figure 1.1 briefly illustrates *rewriting-based formal verification* approaches.

[1]E.g., finite state machines or Boolean programs cannot directly express unbounded data structures, such as (unbounded) integers, lists, and queues.

[2]E.g., the executable rewriting logic semantics of the C language by Ellison and Roşu [82] has been thoroughly tested using the GCC torture suite (GCC itself passes 99.0% of the test cases, and the Ellison-Roşu semantics passes 99.2% of them).

Model		Rewriting Logic		Formal Verification
System specification M	$\Longrightarrow$	Rewrite theory $\mathcal{R}_M$		Simulation Model checking
			$\Longrightarrow$	Theorem proving
Property specification *spec*	$\Longrightarrow$	Logic formula φ_{spec}		$\cdots$

Figure 1.1: Rewriting-based Formal Verification

Rewriting-based formal verification approaches have many conceptual and practical advantages. First, rewriting logic is executable and has a clear mathematical semantics. Second, defining a formal semantics in rewriting logic is definitely much easier than developing new verification tools for the given modeling language.[3] Third, rewriting logic supports various formal analysis methods, including model checking and inductive theorem proving. In order words, giving a formal semantics has the effect of developing both model checking and theorem proving tools for the language at the same time. Because of the high performance of Maude's implementation, the tools thus obtained are actually quite useful and have acceptable performance.[4]

The major goal of this dissertation is to substantially advance the body of rewriting-based *model checking* methods. This can combine the advantages of the three different approaches: (i) model checking is automatic and allows using rich property specification logics, (ii) rewriting logic is simple, formal, expressive, and executable, and (iii) modeling languages come with powerful design tools and well-established model-engineering methods. However, as always, this combination also poses many challenges, including:

1. What are *suitable property specification languages* for rewriting logic specifications? And do they have *efficient model checking algorithms*?
2. How can the *state space explosion problem* be dealt with? How about *infinite-state* concurrent systems?
3. Is it really possible to develop *fully integrated* modeling and verification environments for modeling languages as proposed?

[3]In particular, different language definitional styles, such as denotational semantics and structural operational semantics, and different computation models, such as functional and object-oriented models, can be directly expressed in rewriting logic [140].

[4]This has been demonstrated for semantics-based model checkers for languages such as Java [88], C [82], and Verilog [131], and is demonstrated in this thesis for AADL and Ptolemy II, two widely used modeling languages for embedded systems.

The rest of this chapter reviews previous efforts to tackle these research problems, and shows a high-level overview of our new, both theoretical and practical, techniques presented in this thesis.

1.1 Property Specification Logics

Generally, formal specification involves two specification languages: one for system specifications, and one for property specifications. The concept emerging from these considerations is a *tandem of logics* [136]. That is, we use a pair of logics $(\mathcal{L}_S, \mathcal{L}_P)$ together to specify the system in $\mathcal{L}_S$ and its requirements in $\mathcal{L}_P$, and then verify the satisfaction relation $\mathcal{S} \models \varphi$ between a system specification $\mathcal{S}$ and a property specification φ using verification tools. As discussed in [136], *not* all tandems are *well-matched*: that is, a *mismatch* between $\mathcal{L}_S$ and $\mathcal{L}_P$ can occur because of lack of expressiveness in either of the logics, so that systems of interest or relevant properties cannot be expressed or need to be *encoded* in complicated ways.

In particular, the expressive power of rewriting logic is *not fully exploited* when matched with temporal logics that are either purely state-based (e.g., LTL, CTL and CTL* [60]) or purely event-based (e.g., Hennessy-Milner logic [103] and A-CTL* [145]). The point is that rewriting logic can naturally specify both state-based and event-based aspects of the system, whereas one of these aspects is missing in those temporal logics. That is, the full power of rewriting logic can be best exploited by temporal logics that can deal with both state-based and event-based properties.

The *temporal logic of rewriting* (TLR) [136] was therefore proposed as an expressive temporal logic—where both CTL* and A-CTL* are sublogics of TLR—to give the well-matched tandem between rewriting logic and TLR. The *linear temporal logic of rewriting* (LTLR) is a sublogic of TLR extending linear temporal logic (LTL) in a similar way. It was shown in [136] that the LTLR model checking problem of a rewriting logic specification $\mathcal{R}$ can be reduced to the equivalent *LTL model checking* problem of the *transformed* rewriting logic specification $T(\mathcal{R})$. Since the Maude system already has an LTL model checker [61, 81] implementing efficient LTL model checking algorithms, the first LTLR model checker [20] was implemented using the model transformation T. However, because such a transformation *encodes* event information as part of the state [136], the state space of $T(\mathcal{R})$ could be much bigger than that of the original specification $\mathcal{R}$. An *efficient LTLR model checking algorithm* is needed to resolve this problem.

However, such an algorithm would still leave unanswered the important question of how to efficiently deal with fairness in general, and especially with *parameterized fairness*. Fairness is a very important property for model checking, because many important system requirements are *not* satisfied without appropriate fairness assumptions. For example, the effective data transmission by a fault-tolerant network protocol usually can only be proved under the assumption that the receiving node will receive messages infinitely often if the node is infinitely often enabled to receive them.

In practice, often the necessary fairness assumptions are *parametric* over relevant system entities [134], such as processes, actors, objects, messages, etc. Since fairness conditions can be expressed as temporal logic formulas [32], parameterized fairness conditions correspond to *universally quantified* temporal logic formulas. For example, if a fairness condition of a single process p is expressed as the formula $\mathit{fair}(p)$, then a parameterized version of the fairness condition is expressed as the formula

$$(\forall x)\ \mathit{fair}(x).$$

In rewriting logic specifications, parameterized fairness conditions can be succinctly specified in the notion of *localized fairness* [134].

Although parameterized fairness is very common for many concurrent systems, to the best of our knowledge *no* model checking algorithms were proposed to properly handle parameterized fairness before the work in this thesis. In existing model checking techniques and tools, a total number of fairness instances are either explicitly given (e.g., [32, 60, 108]), or implicitly found by statically analyzing the model (e.g, [158]). However, this *ad-hoc* method is not appropriate for general rewriting-based model checking that does not build in any specific model of computation. In this general setting, parameters are *not* identifiable at the language level, since they can be any kinds of system entities. Further, it may be impossible to determine *a prior* the number of such fairness instances without exploring the entire state space (e.g., consider a system with dynamic process creation).

In short, the temporal logic of rewriting (TLR) under localized fairness was proposed as a suitable property specification language well-matched with rewriting logic [134, 135]. However, efficient model checking algorithms and tools were *not* available for the property specification language; only a model checker for LTL formulas was available in Maude. We present efficient model checking algorithms and a tool for LTLR formulas in Chapter 3, and for parameterized fairness in Chapter 4.

1.2 Approximation Methods

One of the biggest challenges in model checking is how to deal with the *state space explosion* problem [60]. The number of states generated by model checking is in general exponential in the size of the system specification, and therefore it is often unfeasible to verify large system specifications by model checking. In order to cope with the state space explosion problem, many techniques have been developed, including abstractions, symbolic and bounded model checking, partial order reduction, and so on [32, 59, 60].

For rewriting-based model checking, the state space explosion problem is also one of the most difficult problems. Furthermore, the system specified in rewriting logic can be *infinite-state*, because rewriting logic specifications are *parametric* and any algebraic data types can be easily represented in rewriting logic.[5] To verify such infinite-state systems specified in rewriting logic by model checking, two different methods were proposed: equational abstraction [139] and narrowing-based symbolic model checking [86].[6]

Although many infinite-state systems can be successfully verified by using equational abstraction and narrowing-based symbolic model checking, four important problems remained unsolved:

1. Equational abstractions may produce *spurious counterexamples* that violate the given temporal logic formula in the abstract system but *not* in the concrete system.

2. An important class of infinite-state systems, namely, parameterized systems for an unbounded number of processes, could in general not be verified by using *either* of these methods.

3. Both narrowing-based model checking and equational abstraction do *not* guarantee that the resulting abstract systems are *finite-state*.

4. Both narrowing-based model checking and equational abstraction were developed only for LTL properties, and thus needed to be extended for linear temporal logic of rewriting (LTLR) properties.

In Chapter 5, we partly address all of these limitations by further developing each of these approaches, by combining both methods, and by presenting a new rewriting-based abstraction method.

[5] That is, both finite-state systems (e.g., sequencial circuits, Boolean programs, etc.) and infinite-state systems (e.g., pushdown systems, parameterized protocols, etc.) are naturally specified in rewriting logic.

[6] Other state-space reduction methods are also available for rewriting logic, including invisible transitions [89], partial order reduction [90], symmetric reduction [119], etc.

Another important class of concurrent systems are *virtually synchronous* distributed cyber-physical systems. In many embedded computer systems, such as cars and airplanes, their implementation must be asynchronous due to physical and fault-tolerance constraints, but their *logical* design requires that the system components should act together in a virtually synchronous way. Rewriting logic is also a suitable formalism for specifying this kind of *objected-oriented* distributed real-time systems [137, 140]. However, their model checking verification typically becomes unfeasible due to the huge state space explosion caused by the system's concurrency.

The *PALS* (physically asynchronous, logically synchronous) pattern has therefore been developed to reduce the system complexity of a *single-rate* virtually synchronous cyber-physical systems [138, 143]. Roughly speaking, assuming certain performance bounds Γ on the underlying infrastructure, the PALS pattern defines a *model transformation*

$$\mathcal{E} \mapsto \mathcal{A}(\mathcal{E}, \Gamma)$$

that maps a synchronous design $\mathcal{E}$ to a *correct-by construction* distributed implementation $\mathcal{A}(\mathcal{E}, \Gamma)$ that satisfies the same temporal logic properties as $\mathcal{E}$ [138]. This means that the distributed real-time system $\mathcal{A}(\mathcal{E}, \Gamma)$ can be verified by model checking the much simpler synchronous system $\mathcal{E}$.

To apply this PALS methodology for designing and verifying a general class of virtually synchronous distributed cyber-physical systems, there were two major remaining challenges:

1. PALS assumes a *single* logical period during which all components must transition to their next states. But for physical reasons, different components may operate at different rates.

2. Cyber-physical systems often control *physical entities* with continuous dynamics, typically governed by differential equations, but PALS only considers *discrete* real-time systems.

In Chapter 6 these problems are addressed in two ways. First, we define the *Multirate PALS* pattern that generalizes PALS to a wider class of multirate systems. Second, we then explain how the *continuous environments* of such cyber-physical systems can be modeled in the Multirate PALS framework, and also present a general rewriting-based framework to formally specify and verify multirate cyber-physical systems. The Multirate Synchronous AADL modeling language is also defined in Chapter 7 for the same purpose within an industrial modeling standard AADL.

1.3 Applications to Modeling Languages

Following the general research direction of the *rewriting logic semantics project* [140, 141], the formal semantics of many programming and modeling languages have been defined in rewriting logic [137]. Rewriting-based model checking techniques, including those presented in this thesis, can therefore be applied to all of those language. To demonstrate *fully integrated* modeling and verification engineering methods, we consider two modeling languages for real-time embedded systems, Ptolemy II and AADL.

Ptolemy II [79] is a well-established actor-based modeling and simulation tool for embedded systems used in industry, which provides powerful yet intuitive graphical modeling language. Ptolemy II designs are hierarchical models that combine different models of computations, such as finite state machines, data flow, and discrete-event models. In particular, discrete-event (DE) models are widely used for system simulation and embedded software [96, 169]. Although the operational semantics of DE models is defined in [125], Ptolemy II DE models lacked formal verification capabilities.

AADL [91] is an industrial modeling standard for embedded systems that is widely used in avionics, automotive, etc. Because AADL lacks formal semantics, there are a number of approaches to define a formal semantics of AADL; specifically, [147] defines a real-time semantics for a behavior subset of AADL in rewriting logic. However, the state space explosion problem can easily make model checking of AADL models unfeasible, since AADL models consist of *distributed components* that communicate asynchronously with each other. *Multirate Synchronous AADL*, defined in Chapter 7, is a *synchronous* subset of AADL to apply the Multirate PALS methodology for designing and verifying virtually synchronous cyber-physical systems.

In order to develop model-engineering environments that are integrated into the established modeling processes for Ptolemy II DE and Multirate Synchronous AADL models, we follow the three steps in Chapters 7 and 8:

1. defining a formal (real-time) semantics for a targeted subset of each language in rewriting logic;

2. defining a property specification language for each language, using the syntax of the modeling language; and

3. building a tool as a "plugin" of the existing modeling tool, which automatically synthesizes the verification model from a design model and performs model checking of the model *within* the modeling tool.

1.4 Summary of Contributions

The work presented in this dissertation can be classified in three areas, according to the corresponding three research questions discussed above for rewriting-based model checking methods.

Algorithms and Tools. We present the theoretical foundations for LTLR model checking under parameterized fairness. The LTLR model checking problem is characterized as an automata-based approach, which generalizes the SE-LTL model checking technique [49] to LTLR, where SE-LTL can be considered as a sublogic of LTLR [136]. The satisfaction of parameterized fairness conditions can be determined by the notion of *parameter abstraction*, which can be used to deal with an unbounded, dynamically changing, but finite number of generic system entities for fairness.

Based on the theoretical foundations, we present an on-the-fly LTLR model checking algorithm under parameterized fairness conditions. This algorithm is based on the existing on-the-fly algorithms for efficiently model checking *a fixed number of* fairness conditions [76, 122], but is significantly adapted to directly deal with "dynamic" parameters. Experimental results show that this algorithm is comparable to the algorithms used in other explicit-state model checkers (for a fixed number of fairness conditions).

We have developed the Maude Fair LTLR model checker that implements the on-the-fly LTLR model checking algorithm under parameterized fairness at the C++ level as an extension of the Maude system. In addition, our tool provides a convenient and succinct way to specify spatial action patterns and parameterized fairness conditions. The effectiveness of our tool is illustrated with several practical case studies in this dissertation.

Approximation Methods. We significantly further develop equational abstraction and narrowing-based symbolic model checking. First, both narrowing-based symbolic model checking and equational abstraction are extended to LTLR. Second, to deal with spurious counterexamples, we show that equational abstractions can be bisimilar, and *folding abstractions*—that are combined with narrowing-based model checking to achieve finite-state abstractions [86]—can be *faithful* in the sense that they do *not* generate any spurious counterexamples for safety properties. Third, we explain how narrowing-based symbolic model checking can be combined with equational abstractions to further reduce infinite-state systems, which enables us to verify parameterized systems for an unbounded number of processes.

We also present a new rewriting-based method to *automatically* generate a *predicate abstraction*. Predicate abstraction is one of the most widely used abstraction methods that can always generate *finite-state* abstractions; however, automatic predicate abstraction methods were not developed for rewriting logic before. Since constructing *minimal* predicate abstractions is *undecidable* in general for rewriting logic, we also present sound, automatic, rewriting-based, but incomplete method to determine transitions between abstract states for predicate abstractions, which can be used to construct over-approximations of predicate abstractions.

For multirate virtually synchronous cyber-physical systems, we present the *Multirate PALS* pattern, that extends the PALS pattern for single-rate systems, to drastically reduce the system complexity. Similarly, for global period T and performance bounds Γ on the underlying infrastructure, the Multirate PALS pattern defines a model transformation $\mathcal{E} \mapsto \mathcal{MA}(\mathcal{E}, T, \Gamma)$ from a multirate synchronous design $\mathcal{E}$ to its distributed implementation $\mathcal{MA}(\mathcal{E}, T, \Gamma)$. We prove that the distributed system $\mathcal{MA}(\mathcal{E}, T, \Gamma)$ is also *correct-by-construction*, and therefore both $\mathcal{E}$ and $\mathcal{MA}(\mathcal{E}, T, \Gamma)$ satisfy the same temporal logic properties.

We then show how virtually synchronous cyber-physical systems can be specified using Multirate PALS, where the continuous dynamics of their *physical* environments are governed by differential equations. We define a modeling framework for specifying Multirate PALS designs in rewriting logic, and illustrate our methodology with a multirate cyber-physical system, consisting of an airplane maneuvered by a pilot who turns the airplane to a specified angle through a distributed control system. Similarly, we define the *Multirate Synchronous AADL* language as a sublanguage of AADL.

Semantic Integration of Model Checking. We have given a formal rewriting-based semantics to the Multirate Synchronous AADL language, and have developed the *MR-SynchAADL* tool as a plugin of the OSATE AADL modeling environment. The MR-SynchAADL tool is a simulation and model checking tool for Multirate Synchronous AADL based on the rewriting-based semantics. The tool automatically synthesizes a verification model from a Multirate Synchronous AADL model, provides a *requirements specification language* to easily specify the temporal logic requirements of the design model, and performs model checking of the design *within* the OSATE modeling environment. Therefore, this gives a model-engineering process that combines the power of AADL modeling with OSATE, the complexity reduction of Multirate PALS, and rewriting-based model checking.

We have also given a formal semantics to Ptolemy II DE models and have integrated their model checking into the Ptolemy II system, using Ptolemy II's code generation infrastructure. We define a rewriting-based semantics for a significant subset of Ptolemy II DE models, including finite state machine (FSM) actors, composite actors, and modal model actors. This is nontrivial, since Ptolemy II models use unbounded data types, its operational semantics is based on fixed-point computations, and Ptolemy II supports a generic expression language. We define a property specification language to easily specify the temporal logic requirements. The entire model checking processes can be done within Ptolemy II, and therefore it provides a model-engineering process that combines the convenience of Ptolemy II modeling with rewriting-based model checking.

Part I

System and Property Specification Logics

CHAPTER 2

PRELIMINARIES: REWRITING-BASED SYSTEM SPECIFICATIONS

Rewriting logic [132] is a generic system specification formalism in which many concurrent systems are naturally described, including actors, process calculi, Petri nets, and the semantics of concurrent programming languages, such as C and Java [137, 140]. In rewriting logic specifications, each state of a system is specified as an algebraic data type axiomatized by *an equational theory*, and each concurrent transition between states is axiomatized by *rewrite rules*. Rewriting logic specifications are *executable* under reasonable assumptions, so that we can apply model checking techniques to verify any possible concurrent behaviors of the system.

This chapter briefly introduces rewriting logic, the underlying system specification formalism used in this thesis. Section 2.1 explains membership equational logic and some preliminary notions on term rewriting. Section 2.2 describes rewriting logic, and Section 2.3 illustrates Maude—a language and tool for rewriting logic—with simple concurrent system examples.

2.1 Membership Equational Logic

Equational logic is generally a sublogic of first-order logic with equality that has only function symbols and no other predicate symbols. Theories in equational logic typically model *algebraic structures*, such as numbers, sets, multisets, strings, trees, etc. *Membership equational logic* (MEL) [133] is an expressive version of equational logic, with extra membership predicates, that subsumes many-sorted and order-sorted equational logics. We follow the classical notation and terminology from [11, 37] for term rewriting.

2.1.1 Signatures and Terms

A MEL *signature* is a tuple $\Sigma = (K, \sigma, S)$ with K a set of *kinds*, $S = \{S_k\}_{k \in K}$ a K-kinded family of disjoint sets of *sorts*, and $\sigma = \{\Sigma_{w,k}\}_{(w,k) \in K^* \times K}$ a many-kinded signature in which each $\Sigma_{k_1 \dots k_n, k}$ for $n \geq 0$ is a set of n-ary function symbols of the form $f : k_1 \times \cdots \times k_n \to k$ with domain $k_1 \times \cdots \times k_n$ and range k.[1] A 0-ary function symbol $c : nil \to k$ is often called a *constant symbol* (of kind k). The kind of a sort s is denoted by $[s]$.

For a set $\mathcal{X}$ of K-kinded variables,[2] $\mathcal{T}_\Sigma(\mathcal{X}) = \{\mathcal{T}_\Sigma(\mathcal{X})_k\}_{k \in K}$ denotes the K-kinded set of Σ-*terms*, inductively defined by:

- for each variable $x \in \mathcal{X}$ of kind k: $x \in \mathcal{T}_\Sigma(\mathcal{X})_k$; and
- for each function symbol $f \in \Sigma_{k_1 \dots k_n, k}$: $f(t_1, \dots, t_n) \in \mathcal{T}_\Sigma(\mathcal{X})_k$, if $t_1 \in \mathcal{T}_\Sigma(\mathcal{X})_{k_1}, t_2 \in \mathcal{T}_\Sigma(\mathcal{X})_{k_2}, \dots, t_n \in \mathcal{T}_\Sigma(\mathcal{X})_{k_n}$.

Similarly, $\mathcal{T}_\Sigma = \{\mathcal{T}_{\Sigma,k}\}_{k \in K}$ denotes the K-kinded set of *ground* Σ-terms containing *no* variables. Throughout this thesis, we assume that $\mathcal{T}_{\Sigma,k} \neq \emptyset$ for each kind $k \in K$ (i.e. there exists a ground term for each $k \in K$).

Example 2.1. *The signature $\Sigma_\mathbb{N}$ of the natural numbers can be composed of two kinds $K = \{[\mathsf{Nat}], [\mathsf{Bool}]\}$, sets of sorts $S_{[\mathsf{Nat}]} = \{\mathsf{Nat}, \mathsf{Zero}, \mathsf{NzNat}\}$ and $S_{[\mathsf{Bool}]} = \{\mathsf{Bool}\}$, and function symbols*

$$\Sigma_{nil,[\mathsf{Nat}]} = \{0\}, \qquad \Sigma_{[\mathsf{Nat}],[\mathsf{Nat}]} = \{s\}, \quad \Sigma_{nil,[\mathsf{Bool}]} = \{\mathit{true}, \mathit{false}\},$$
$$\Sigma_{[\mathsf{Nat}][\mathsf{Nat}],[\mathsf{Nat}]} = \{+\}, \quad \Sigma_{[\mathsf{Nat}][\mathsf{Nat}],[\mathsf{Bool}]} = \{<\}.$$

For example, $s(0) < (0 + s(0)) \in \mathcal{T}_{\Sigma,[\mathsf{Bool}]}$ is a ground term of kind $[\mathsf{Bool}]$, and $x + ((s(0) + s(y)) + s(s(0))) \in \mathcal{T}_\Sigma(\mathcal{X})_{[\mathsf{Nat}]}$ is a term of kind $[\mathsf{Nat}]$, where $x, y \in \mathcal{X}$ are variables of kind $[\mathsf{Nat}]$.

Given a term $t \in \mathcal{T}_\Sigma(\mathcal{X})$, $\mathit{vars}(t) \subseteq \mathcal{X}$ denotes the set of variables in t. *Positions* in a term t are denoted as strings of nonzero natural numbers that indicate argument positions of function symbols in t. A *subterm* $t|_p$ of a term t at a position p is inductively defined by:

$$t|_\varepsilon = t \quad \text{and} \quad f(t_1, \dots, t_n)|_{i.p} = t_i|_p,$$

where ε is the empty string. The *replacement* in a term t of such a subterm at position p by another term u is denoted by $t[u]_p$.

[1] We assume that if $f \in \Sigma_{k_1 \dots k_n, k} \cap \Sigma_{k'_1 \dots k'_n, k'}$, then $(k_1, \dots, k_n) = (k'_1, \dots, k'_n)$ implies $k = k'$, to avoid ambiguous terms of the same form but of different kinds.

[2] That is, $\mathcal{X} = \{\mathcal{X}_k\}_{k \in K}$ with disjoint sets $\mathcal{X}_k$ of variables.

A *substitution* $\theta : \mathcal{X} \to \mathcal{T}_\Sigma(\mathcal{X})$ is a function that maps variables to terms of the same kind and $\theta(x) \neq x$ for only finitely many x's. The *domain* of a substitution θ is the finite set $dom(\theta) = \{x \in \mathcal{X} \mid \theta(x) \neq x\}$. The restriction of a substitution θ to $Y \subseteq \mathcal{X}$ is the substitution $\theta|_Y : \mathcal{X} \to \mathcal{T}_\Sigma(\mathcal{X})$ such that $\theta|_Y(x) = \theta(x)$ if $x \in Y$, and $\theta|_Y(x) = x$ otherwise. The substitution instance $\theta(t)$ of t is a term obtained from t by *simultaneously* replacing each occurrence of variable $y \in \mathcal{X}$ in t with $\theta(y)$.

Example 2.2. *Consider the* Mel *signature* $\Sigma_\mathbb{N}$ *in Example 2.1. For the term* $t = x+((s(0)+y)+0)$*, the subterm* $t|_{2.1}$ *is* $s(0)+y$*, and the replacement* $t[z]_{2.1}$ *is* $x + (z + 0)$*. For the substitution* $\theta = \{x \mapsto s(y), y \mapsto (x + 0)\}$*,* $\theta(t) = s(y) + ((s(0) + (x + 0)) + 0)$*.*

2.1.2 Equational Theories and Algebras

There are two kinds of atomic formulas in membership equational logic: an *equation* $t = t'$, where $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})_k$; and a *membership* $t : s$, stating that t has sort s, where $t \in \mathcal{T}_\Sigma(\mathcal{X})_k$ and $s \in S_k$. A Mel *theory* is a pair (Σ, E) with Σ a Mel-signature and E a finite set of Mel *sentences*, either a conditional equation or a conditional membership of the respective forms:

$$(\forall \mathcal{X})\ t = t' \ \textbf{if} \bigwedge_{i \in I} u_i = v_i \wedge \bigwedge_{j \in J} w_j : s_j, \quad (\forall \mathcal{X})\ t : s \ \textbf{if} \bigwedge_{i \in I} u_i = v_i \wedge \bigwedge_{j \in J} w_j : s_j.$$

A subsort declaration $s_1 < s_2$ can be used to shorten the Mel sentence $(\forall x)\ x : s_2$ **if** $x : s_1$. Similarly, an operator declaration $f : s_1 \times \cdots \times s_n \to s$ corresponds to $(\forall\, x_1, \ldots, x_n)\ f(x_1, \ldots, x_n) : s$ **if** $x_1 : s_1 \wedge \cdots \wedge x_n : s_n$.

Example 2.3. *The* Mel *theory* $(\Sigma_\mathbb{N}, E_\mathbb{N})$ *of the natural numbers can be defined by the* Mel *signature* $\Sigma_\mathbb{N}$ *in Example 2.1 and the set* $E_\mathbb{N}$ *of the following (abbreviated)* Mel *sentences:*

$$\begin{array}{llll} \mathit{true} : \mathsf{Bool}, & \mathit{false} : \mathsf{Bool}, & 0 : \mathsf{Zero}, & s : \mathsf{Nat} \to \mathsf{NzNat}, \\ \mathsf{Zero} < \mathsf{Nat}, & \mathsf{NzNat} < \mathsf{Nat}, & & (\forall x)\ x + 0 = x, \end{array}$$

$$\begin{array}{cc} (\forall x, y)\ x + s(y) = s(x + y), & (\forall x, y)\ x + y = y + x, \\ (\forall x, y, z)\ (x + y) + z = x + (y + z), & (\forall x)\ 0 < s(x) = \mathit{true}, \\ (\forall x)\ x < 0 = \mathit{false}, & (\forall x, y)\ s(x) < s(y) = x < y. \end{array}$$

Intuitively, terms in sorts are well-defined, whereas terms without sorts are either "undefined" values, or expressions, such as $0 + s(0)$, that are not yet "computed," but that will evaluate to well-sorted terms.

A Σ-*algebra* A consists of a *carrier* set A_k for each kind k, a subset $A_s \subseteq A_k$ for each sort $s \in S_k$, and a function $f_A : A_{k_1} \times \cdots \times A_{k_n} \to A_k$ for each function symbol $f \in \Sigma_{k_1 \ldots k_n, k}$. For a valuation $a : \mathcal{X} \to A$ assigning a value in carrier set A_k to each variable $x \in \mathcal{X}$ of kind k, its homomorphic extension $\bar{a} : \mathcal{T}_\Sigma(\mathcal{X}) \to A$ is inductively defined by:

- $\bar{a}(x) = a(x)$ for each $x \in \mathcal{X}$, and
- $\bar{a}(f(t_1, \ldots, t_n)) = f_A(\bar{a}(t_1), \ldots, \bar{a}(t_n))$ for each $f \in \Sigma_{k_1 \ldots k_n, k}$.

A Σ-*algebra* A is a *model* of (Σ, E) iff each sentence $\phi \in E$ is satisfied in A for any valuation a. An equation $t = t'$ is satisfied on A for a valuation a, denoted by $(A, a) \models t = t'$, iff $\bar{a}(t) = \bar{a}(t')$. Similarly, $(A, a) \models t : s$ iff $\bar{a}(t) \in A_s$. For a MEL sentence, $(A, a) \models \alpha$ **if** $\bigwedge_{i \in I} u_i = v_i \wedge \bigwedge_{j \in J} w_j : s_j$, where α is either $t = t'$ or $t : s$, iff $(A, a) \models \alpha$ whenever $(A, a) \models u_i = v_i$ and $(A, a) \models w_j : s_j$ for each $i \in I$ and $j \in J$.

Example 2.4. *A model of the* MEL *theory* $(\Sigma_\mathbb{N}, E_\mathbb{N})$ *in Example 2.3 can be defined by the* Σ*-algebra* A *with:*

- *the carrier sets* $A_{[\mathsf{Nat}]} = \mathbb{N}$ *and* $A_{[\mathsf{Bool}]} = \{\bot, \top\}$*;*
- *the subsets* $A_{\mathsf{Nat}} = A_{[\mathsf{Nat}]}$*,* $A_{\mathsf{Zero}} = \{0\}$*,* $A_{\mathsf{NzNat}} = A_{[\mathsf{Nat}]} - \{0\}$*, and* $A_{\mathsf{Bool}} = A_{[\mathsf{Bool}]}$*;*
- *the functions* $0_A = 0$*,* $\mathit{true}_A = \top$*,* $\mathit{false}_A = \bot$*,* $s_A(n) = n + 1$*,* $+_A(n, m) = n +_\mathbb{N} m$*, and* $<_A (n, m) = \textbf{if } n <_\mathbb{N} m \textbf{ then } \top \textbf{ else } \bot \textbf{ fi}$.

where $+_\mathbb{N}$ *and* $<_\mathbb{N}$ *are the standard addition and order for* $\mathbb{N}$.

2.1.3 Initial and Free Algebras

Membership equational logic is sound and complete in the sense that for any MEL sentence ϕ, a proof of $E \vdash \phi$ can be derived by the deduction rules in [133] iff $(A, a) \models \phi$ for any model A of (Σ, E) and valuation a. A MEL theory (Σ, E) induces a congruence[3] relation $=_E$ on terms defined by the equivalence $t =_E t' \iff E \vdash t = t'$. Let $\mathcal{T}_{\Sigma/E,k}$ be the set of E-equivalence classes $[t]_E = \{t' \in \mathcal{T}_{\Sigma,k} \mid t =_E t'\}$ of ground terms of kind k, and let $\mathcal{T}_{\Sigma/E}(\mathcal{X})_k$ be the set of E-equivalence classes $\mathcal{T}_{\Sigma/E}(\mathcal{X})_k = \{[t]_E \mid t \in \mathcal{T}_\Sigma(\mathcal{X})_k\}$.

[3]That is, if $t_1 =_E t'_1$, $t_2 =_E t'_2$ ..., $t_n =_E t'_n$, then $f(t_1, \ldots, t_n) =_E f(t'_1, \ldots, t'_n)$.

A Mel theory (Σ, E) has a "standard" model up to isomorphism, namely, the *initial algebra* $\mathcal{T}_{\Sigma/E}$, from which there exists a unique homomorphism[4] to any model of (Σ, E) [133]. The initial algebra $\mathcal{T}_{\Sigma/E}$ of (Σ, E) consists of:

- a carrier set $\mathcal{T}_{\Sigma/E,k}$ for each kind k;
- a subset $\mathcal{T}_{\Sigma/E,s} = \{[t]_E \in \mathcal{T}_{\Sigma/E,k} \mid E \vdash t : s\}$ for each sort $s \in S_k$; and
- a function $f_{\mathcal{T}_{\Sigma/E,k}} : \mathcal{T}_{\Sigma/E,k_1} \times \cdots \times \mathcal{T}_{\Sigma/E,k_n} \to \mathcal{T}_{\Sigma/E,k}$ for each operator $f \in \Sigma_{k_1 \ldots k_n,k}$ such that $f_{\mathcal{T}_{\Sigma/E,k}}([t_1]_E, \ldots, [t_n]_E) = [f(t_1, \ldots, t_n)]_E$.

Likewise, a Mel theory (Σ, E) has the *free algebra* $\mathcal{T}_{\Sigma/E}(\mathcal{X})$, where for any model A of (Σ, E), a valuation $a : \mathcal{X} \to A$ can be uniquely extended to a homomorphism $\bar{a} : \mathcal{T}_{\Sigma/E}(\mathcal{X}) \to A$ [133], which consists of:

- a carrier set $\mathcal{T}_{\Sigma/E}(\mathcal{X})_k$ for each kind k;
- a subset $\mathcal{T}_{\Sigma/E}(\mathcal{X})_s = \{[t]_E \in \mathcal{T}_{\Sigma/E}(\mathcal{X})_k \mid E \vdash (\forall \mathcal{X})\ t : s\}$ for each sort $s \in S_k$; and
- a function $f_{\mathcal{T}_{\Sigma/E}(\mathcal{X})_k} : \mathcal{T}_{\Sigma/E}(\mathcal{X})_{k_1} \times \cdots \times \mathcal{T}_{\Sigma/E}(\mathcal{X})_{k_n} \to \mathcal{T}_{\Sigma/E}(\mathcal{X})_k$ for each $f \in \Sigma_{k_1 \ldots k_n,k}$ such that $f_{\mathcal{T}_{\Sigma/E}(\mathcal{X})_k}([t_1]_E, \ldots, [t_n]_E) = [f(t_1, \ldots, t_n)]_E$.

2.1.4 Order-Sorted Equational Logic

An order-sorted signature is a triple $\Sigma = (S, \leq, \sigma)$ with $(S, \leq)$ a poset of sorts and $\sigma = \{\Sigma_{w,s}\}_{(w,s) \in S^* \times S}$ a many-sorted signature. An order-sorted equational theory is a pair (Σ, E) with Σ an order-sorted signature and E a set of conditional equations of the form $(\forall \mathcal{X})\ t = t'$ **if** $\bigwedge_{i \in I} u_i = v_i$. As hinted at in Section 2.1.2, an order-sorted equational theory (Σ, E) can be transformed into an Mel theory as follows [133]:

- a kind k for each connected component in $(S, \leq)$;
- $(\forall x)\ x : s_2$ **if** $x : s_1$ for each $s_1 \leq s_2$ in $(S, \leq)$; and
- $(\forall x_1, \ldots, x_n)\ f(x_1, \ldots, x_n) : s$ **if** $x_1 : s_1 \wedge \cdots \wedge x_n : s_n$ for each function symbol $f \in \Sigma_{s_1 \ldots s_n, s}$.

However, it is sometimes useful to explicitly consider order-sorted theories instead of Mel theories, because order-sorted unification algorithms are well-developed (e.g., see Chapter 5).

[4] $h : A \to B$ is a homomorphism iff $h(f_A(a_1, \ldots, a_n)) = f_B(h(a_1), \ldots, h(a_n))$ for each $f \in \Sigma_{k_1 \ldots k_n,k}$ and $a_i \in A_{k_i}$, $1 \leq i \leq n$, and $h(a) \in B_s$ for each sort $s \in S_k$ and $a \in A_s$.

2.2 Rewriting Logic

Rewriting logic basically extends (membership) equational logic by adding *rewrite* predicates of the form $t \longrightarrow t'$, where $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})_k$, stating that a term t can "evolve" to t'. A *rewrite theory* $\mathcal{R} = (\Sigma, E, R)$ is a formal specification of a concurrent system [132], where (Σ, E) is a MEL theory specifying the system's states, and R is a finite set of (universally quantified) *labeled conditional rewrite rules*, each of which has the form:

$$l : q \longrightarrow r \textbf{ if } \bigwedge_{i \in I} u_i = v_i \ \wedge \ \bigwedge_{j \in J} w_j : s_j \ \wedge \ \bigwedge_{m \in M} t_m \longrightarrow t'_m,$$

specifying the system's concurrent transitions, where l is a *label*, and $u_i = v_i$ and $w_j : s_j$ are atomic formulas in membership equational logic.

Example 2.5 (Dining Philosophers)**.** *There are N philosophers sitting at a circular table who are thinking, waiting, or eating. A chopstick is placed between each pair of adjacent philosophers. A thinking philosopher wakes up. A waiting philosopher grabs one adjacent chopstick and eats when having both. After eating, a philosopher places the chopsticks back and thinks.*

In rewriting logic, a philosopher can be expressed by a term $p(id, status)$ of sort Philo *with $status \in \{think, wait0, wait1, eat\}$, and a chopstick can be represented by $c(id)$ of sort* Chopstick*. A system state can be expressed by a $\|$-separated* multiset *of philosophers and chopsticks, with the empty multiset $none$ (for example, $p(0, wait0) \parallel c(0) \parallel p(1, think) \parallel c(1)$). That is, the* MEL *theory (Σ, E) (see Example 2.10 for the Maude specification) contains:*

- *kinds:* [Conf] *and* [Status]*,*
- *sorts: $S_{[\mathsf{Status}]} = \{\mathsf{Status}\}$ and $S_{[\mathsf{Conf}]} = \{\mathsf{Philo}, \mathsf{Chopstick}, \mathsf{Conf}\}$,*
- *function symbols: $\Sigma_{[\mathsf{Conf}][\mathsf{Conf}],[\mathsf{Conf}]} = \{\|\}$, $\Sigma_{nil,[\mathsf{Conf}]} = \{none\}$, $\Sigma_{nil,[\mathsf{Status}]} = \{think, wait0, wait1, eat\}$, $\Sigma_{[\mathsf{Nat}][\mathsf{Status}],[\mathsf{Conf}]} = \{p\}$, and $\Sigma_{[\mathsf{Nat}],[\mathsf{Conf}]} = \{c\}$.*
- *and the (abbreviated)* MEL *sentences:*

$$\begin{array}{lll} p : \mathsf{Nat\ Status} \to \mathsf{Philo}, & c : \mathsf{Nat} \to \mathsf{Chopstick}, & none : \mathsf{Conf}, \\ \| : \mathsf{Conf\ Conf} \to \mathsf{Conf}, & think : \mathsf{Status}, & wait0 : \mathsf{Status}, \\ wait1 : \mathsf{Status}, & eat : \mathsf{Status}, & \mathsf{Philo} < \mathsf{Conf}, \\ \mathsf{Chopstick} < \mathsf{Conf}, & (\forall x)\ x \parallel none = x, & (\forall x, y)\ x \parallel y = y \parallel x, \\ (\forall x, y, z)\ (x \parallel y) \parallel z = x \parallel (y \parallel z). & & \end{array}$$

If the functions $\texttt{lc}(i)$ *and* $\texttt{rc}(i)$*, respectively, return the chopstick's id on the left and the right of the philosopher* i*, and* $\texttt{adj}(i,j)$ *returns true iff* $j = \texttt{lc}(i)$ *or* $j = \texttt{rc}(i)$*,*[5] *then the behavior can be defined by the rules:*

$$\begin{aligned}
wake &: \texttt{p}(i, \texttt{think}) \longrightarrow \texttt{p}(i, \texttt{wait0}) \\
grabF &: \texttt{p}(i, \texttt{wait0}) \parallel \texttt{c}(j) \longrightarrow \texttt{p}(i, \texttt{wait1}) \quad \textbf{\textit{if}}\ \texttt{adj}(i,j) = true \\
grabS &: \texttt{p}(i, \texttt{wait1}) \parallel \texttt{c}(j) \longrightarrow \texttt{p}(i, \texttt{eat}) \quad \textbf{\textit{if}}\ \texttt{adj}(i,j) = true \\
stop &: \texttt{p}(i, \texttt{eat}) \longrightarrow \texttt{p}(i, \texttt{think}) \parallel \texttt{c}(\texttt{lc}(i)) \parallel \texttt{c}(\texttt{rc}(i))
\end{aligned}$$

For example, from the state $\texttt{p}(0, \texttt{wait0}) \parallel \texttt{c}(0) \parallel \texttt{p}(1, \texttt{think}) \parallel \texttt{c}(1)$*, one next state by the grabF rule is* $\texttt{p}(0, \texttt{wait1}) \parallel \texttt{p}(1, \texttt{think}) \parallel \texttt{c}(1)$*.*

2.2.1 Rewrite Relations and Proof Terms

Rewriting logic has the deduction rules in [45, 132] that infer all concurrent computations in the system specified by $\mathcal{R}$. That is, a term t' is reachable from a term t iff $\mathcal{R} \vdash \lambda : (\forall \mathcal{X})\ t \longrightarrow t'$ can be derived, where a *proof term* λ describes how t' has been computed from t. A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ induces the *rewrite relation* $\longrightarrow^{*}_{\mathcal{R}}$ on $\mathcal{T}_{\Sigma/E}(\mathcal{X})$ such that $\lambda : [t]_E \longrightarrow^{*}_{\mathcal{R}} [t']_E$ iff $\mathcal{R} \vdash \lambda : (\forall \mathcal{X})\ t \longrightarrow t'$, which is generally classified as follows [132]:

- a *zero-step rewrite* $\quad t : [t]_E \longrightarrow^{0}_{\mathcal{R}} [t]_E$, where $\longrightarrow^{0}_{\mathcal{R}} \subseteq \longrightarrow^{*}_{\mathcal{R}}$.
- a *sequential rewrite* $\quad (\lambda_1; \lambda_2; \ldots; \lambda_n) : [t_1]_E \longrightarrow^{*}_{\mathcal{R}} [t_n]_E$ iff for each $1 \leq i < n$, $\lambda_i : [t_i]_E \longrightarrow^{*}_{\mathcal{R}} [t_{i+1}]_E$ holds.
- a *one-step rewrite*

  $$t[l(\theta, \beta_1, \ldots, \beta_{|M|})]_p : [t]_E \longrightarrow^{1}_{\mathcal{R}} [t']_E, \qquad \text{where } \longrightarrow^{1}_{\mathcal{R}} \subseteq \longrightarrow^{*}_{\mathcal{R}},$$

  iff there exist a substitution θ, a position p, and a rule $l : q \longrightarrow r$ **if** $\bigwedge_{i \in I} u_i = v_i \ \wedge\ \bigwedge_{j \in J} w_j : s_j \ \wedge\ \bigwedge_{m \in M} t_m \longrightarrow t'_m \in R$ such that

  $$\begin{aligned}
  & t|_p =_E \theta(q), \qquad t' = t[\theta(r)]_p, \quad \textstyle\bigwedge_{i \in I} \theta(u_i) =_E \theta(v_i), \\
  & \textstyle\bigwedge_{j \in J} E \vdash (\forall \mathcal{X})\ \theta(w_j) : s_j, \qquad \textstyle\bigwedge_{m \in M} \beta_m : [\theta(t_m)]_E \longrightarrow^{*}_{\mathcal{R}} [\theta(t'_m)]_E.
  \end{aligned}$$

  That is, a term $t[\theta(q)]_p$ containing $\theta(q)$ is rewritten to $t[\theta(r)]_p$ in which $\theta(q)$ has been replaced by $\theta(r)$, provided that the condition holds.

[5] That is, Σ contains: $\Sigma_{[\mathsf{Nat}],[\mathsf{Nat}]} = \{\texttt{lc}, \texttt{rc}\}$ and $\Sigma_{[\mathsf{Nat}][\mathsf{Nat}],[\mathsf{Bool}]} = \{\texttt{adj}\}$, and E contains: $(\forall i)\ \texttt{lc}(i) = i$, $(\forall i)\ \texttt{rc}(i) = s(i)$ **if** $s(i) < N$, $(\forall i)\ \texttt{rc}(i) = 0$ **if** $s(i)$ *equals* N, and $(\forall i,j)\ \texttt{adj}(i,j) = (j$ *equals* $\texttt{lc}(i))$ *or* $(j$ *equals* $\texttt{rc}(i))$.

- a *concurrent rewrite*

$$\begin{aligned} & f(\lambda_1, \ldots, \lambda_n) : [f(t_1, \ldots, t_n)]_E \longrightarrow^*_{\mathcal{R}} [f(t'_1, \ldots, t'_n)]_E \\ \iff & \lambda_i : [t_i]_E \longrightarrow^*_{\mathcal{R}} [t'_i]_E, \quad \text{for each } 1 \leq i \leq n. \end{aligned}$$

Example 2.6. *Consider $\mathcal{R} = (\Sigma, E, R)$ for the dining philosophers problem in Example 2.5. For $N = 2$, by the rule grapF, we have the one-step rewrite:*

$$\begin{aligned} & grapF(\{i \mapsto 0, j \mapsto 0\}) \parallel p(1, \mathit{think}) \parallel c(1) : \\ & \quad [p(0, \mathit{wait0}) \parallel c(0) \parallel p(1, \mathit{think}) \parallel c(1)]_E \longrightarrow^1_{\mathcal{R}} [p(0, \mathit{wait1}) \parallel p(1, \mathit{think}) \parallel c(1)]_E. \end{aligned}$$

2.2.2 Concurrent and Interleaving Models

A standard model of a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ is the *initial concurrent model*[6] $\mathcal{T}_{\mathcal{R}} = \{\mathcal{T}_{\mathcal{R},k}\}_{k \in K}$, where $\mathcal{T}_{\mathcal{R},k} = (\mathcal{T}_{\Sigma/E,k}, \longrightarrow^*_{\mathcal{R}})$ [132]. That is, each $\mathcal{T}_{\mathcal{R},k}$ is a transition system whose states are elements of the k-component of the initial algebra $\mathcal{T}_{\Sigma/E,k}$, and whose transitions, labeled by proof terms, are given by the rewrite relation $\longrightarrow^*_{\mathcal{R}}$. Similarly, the *free concurrent model*[7] of $\mathcal{R}$ is defined by $\mathcal{T}_{\mathcal{R}}(\mathcal{X}) = \{\mathcal{T}_{\mathcal{R}}(\mathcal{X})_k\}_{k \in K}$, where $\mathcal{T}_{\mathcal{R}}(\mathcal{X})_k = (\mathcal{T}_{\Sigma/E}(\mathcal{X})_k, \longrightarrow^*_{\mathcal{R}})$.

Several concurrent computations, specified by different proof terms, can be equivalent to each other. Indeed, any proof term λ, except for zero-step proof terms, is always equivalent to an "interleaving" description, namely, a sequential composition $\lambda_1; \ldots; \lambda_n$ of *one-step proof terms* [132, 135]. That is, $\longrightarrow^*_{\mathcal{R}}$ is equivalent to the reflexive and transitive closure of $\longrightarrow^1_{\mathcal{R}}$.

Example 2.7. *For $\mathcal{R}$ in Example 2.5 with $N = 2$, the concurrent rewrite that applies the rule grapF to the two philosophers at the same time:*

$$\begin{aligned} & grapF(\{i \mapsto 0, j \mapsto 0\}) \parallel grapF(\{i \mapsto 1, j \mapsto 1\}) : \\ & \quad [p(0, \mathit{wait0}) \parallel c(0) \parallel p(1, \mathit{wait0}) \parallel c(1)]_E \longrightarrow^*_{\mathcal{R}} [p(0, \mathit{wait1}) \parallel p(1, \mathit{wait1})]_E. \end{aligned}$$

is equivalent to the sequential composition of the following one-step rewrites that apply the rule grapF to one philosopher at a time:

$$\begin{aligned} & grapF(\{i \mapsto 0, j \mapsto 0\}) \parallel p(1, \mathit{wait0}) \parallel c(1) : \\ & \quad [p(0, \mathit{wait0}) \parallel c(0) \parallel p(1, \mathit{wait0}) \parallel c(1)]_E \longrightarrow^1_{\mathcal{R}} [p(0, \mathit{wait1}) \parallel p(1, \mathit{wait0}) \parallel c(1)]_E, \\ & p(0, \mathit{wait1}) \parallel grapF(\{i \mapsto 1, j \mapsto 1\}) : \\ & \quad [p(0, \mathit{wait1}) \parallel p(1, \mathit{wait0}) \parallel c(1)]_E \longrightarrow^1_{\mathcal{R}} [p(0, \mathit{wait1}) \parallel p(1, \mathit{wait1})]_E. \end{aligned}$$

---

[6] There exists a unique homomorphism from $\mathcal{T}_{\mathcal{R}}$ to any concurrent model $\mathcal{A} = (A, \rightarrow^*_{\mathcal{A}})$ of $\mathcal{R}$ [132], where A is a model of the MEL theory (Σ, E) and $\rightarrow^*_{\mathcal{A}} \subseteq A^2$ is a binary relation that satisfies the rewrite rules in R and the algebraic laws of true concurrency in [132].

[7] For any concurrent model $\mathcal{A} = (A, \rightarrow^*_{\mathcal{A}})$ of $\mathcal{R}$, $a : \mathcal{X} \rightarrow A$ can be uniquely extended to a homomorphism $\bar{a} : \mathcal{T}_{\mathcal{R}}(\mathcal{X}) \rightarrow \mathcal{A}$, where $[t]_E \longrightarrow^*_{\mathcal{R}} [t']_E$ implies $\bar{a}(t) \rightarrow^*_{\mathcal{A}} \bar{a}(t')$.

Therefore, we have the *initial interleaving model* $\mathcal{T}^1_\mathcal{R} = \{\mathcal{T}^1_{\mathcal{R},k}\}_{k\in K}$, where $\mathcal{T}^1_{\mathcal{R},k} = (\mathcal{T}_{\Sigma/E,k}, \longrightarrow^1_\mathcal{R})$, whose reflexive and transitive closure generates the initial concurrent model $\mathcal{T}_\mathcal{R}$. Similarly, the *free interleaving model* of $\mathcal{R}$ is defined by $\mathcal{T}^1_\mathcal{R}(\mathcal{X}) = \{\mathcal{T}^1_\mathcal{R}(\mathcal{X})_k\}_{k\in K}$, where $\mathcal{T}^1_\mathcal{R}(\mathcal{X})_k = (\mathcal{T}_{\Sigma/E}(\mathcal{X})_k, \longrightarrow^1_\mathcal{R})$. For model checking verification of a rewrite theory $\mathcal{R}$, we focus on these interleaving models of $\mathcal{R}$ throughout this thesis.

2.2.3 Localized Fairness

Fairness means that if a certain kind of choice is sufficiently often provided, then it is sufficiently often taken [168]. For example, *strong fairness* means that if a given choice is available infinitely often, then it is taken infinitely often. Similarly, *weak fairness* means that if the choice is continuously available beyond a certain point, then it is taken infinitely often. Fairness assumptions are often necessary to verify many important system properties, since without fairness, unrealistic counterexamples can be produced.

Example 2.8. *For the dining philosophers model in Example 2.5, when* $N = 2$*, without fairness assumptions, there exists the following (unrealistic) infinite behavior in which the philosopher* 1 *does not take any action forever:*

$$\begin{array}{ccc}[p(0, \mathit{think}) \parallel c(0) \parallel p(1, \mathit{think}) \parallel c(1)]_E & \longrightarrow^1_\mathcal{R} & [p(0, \mathit{wait0}) \parallel c(0) \parallel p(1, \mathit{think}) \parallel c(1)]_E \\ \uparrow^1_\mathcal{R} & & \downarrow^1_\mathcal{R} \\ [p(0, \mathit{eat}) \parallel p(1, \mathit{think})]_E & {}^1_\mathcal{R}\longleftarrow & [p(0, \mathit{wait1}) \parallel p(1, \mathit{think}) \parallel c(1)]_E.\end{array}$$

*This behavior indeed violates the weak fairness condition: "*if a philosopher can continuously wake up beyond a certain point, then the philosopher must wake up infinitely often.*" However, there still exists the unrealistic behavior:*

$$\begin{array}{ccc}[p(0, \mathit{think}) \parallel c(0) \parallel p(1, \mathit{wait0}) \parallel c(1)]_E & \longrightarrow^1_\mathcal{R} & [p(0, \mathit{wait0}) \parallel c(0) \parallel p(1, \mathit{wait0}) \parallel c(1)]_E \\ \uparrow^1_\mathcal{R} & & \downarrow^1_\mathcal{R} \\ [p(0, \mathit{eat}) \parallel p(1, \mathit{wait0})]_E & {}^1_\mathcal{R}\longleftarrow & [p(0, \mathit{wait1}) \parallel p(1, \mathit{wait0}) \parallel c(1)]_E,\end{array}$$

*which can be eliminated by assuming the strong fairness condition: "*if a philosopher can grab a chopstick infinitely often, then the philosopher must grab a chopstick infinitely many times.*"*

In rewriting logic it may not be enough to give fairness of rewrite rules: rather, we often need *parameterized* fairness *localized* to specific entities in the system. For example, in Example 2.8, both counterexamples *fairly* perform all the four rewrite rules but only for the philosopher 0, whereas the necessary fairness conditions are parameterized to *each* philosopher.

This idea was captured by the notion of *localized fairness* [134]. That is, a transition specified by a rule $l : q \longrightarrow r$ **if** C is *parametric* on the variables in q, and fairness for the rule l can then be *localized* to a subset $\{x_{j_1}, \ldots, x_{j_k}\}$ of $vars(q)$. For instance, the fairness conditions in Example 2.8 are localized to the single variable i denoting the id of a philosopher. One of the major contribution of this thesis is to provide a novel model checking algorithm for such parameterized fairness conditions (see Section 4.3).

2.2.4 Executability Conditions

For a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, a one step rewrite $[t]_E \longrightarrow^1_{\mathcal{R}} [t']_E$ may be *undecidable* in general. Therefore, we require that $\mathcal{R}$ satisfies additional executability conditions under which $[t]_E \longrightarrow^1_{\mathcal{R}} [t']_E$ can be decided in a finite number of steps. A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ is called *computable* iff E and R are finite and the following conditions hold [136].

First, E can be decomposed into a disjoint union $E_o \cup B$ with B a set of *structural axioms* (such as associativity, commutativity, and identity), so that equational deduction is performed *modulo* B. Also, $t =_B t'$ is decidable, and furthermore, there exists a *matching algorithm modulo* B, which, given a subject term t and a pattern term u, produces a finite and complete set of B-matching substitutions θ such that $t =_B \theta(u)$, or failing otherwise.

Second, E_o is oriented into *sort-decreasing*,[8] and *ground terminating*,[9] *confluent*,[10] and *coherent*[11] rewrite rules modulo B [70]. This implies that for $E_o/B = (\Sigma, B, E_o)$, each ground term $t \in \mathcal{T}_\Sigma$ has a *unique* B-equivalence class $[can_{E_o/B}(t)]_B \in \mathcal{T}_{\Sigma/B}$, called the E_o/B-*canonical form* of t, such that $[t]_B \longrightarrow^*_{E_o/B} [can_{E_o/B}(t)]_B$ and $[can_{E_o/B}(t)]_B$ cannot be further rewritten by $\longrightarrow^1_{E_o/B}$. That is, $=_E$ is decidable for ground terms, since we have:

$$t =_E t' \iff can_{E_o/B}(t) =_B can_{E_o/B}(t').$$

Moreover, this condition makes the *canonical term algebra* $Can_{\Sigma,E_o/B}$, given by a carrier set $Can_{\Sigma,E_o/B,k} = \{[can_{E_o/B}(t)]_B \mid t \in \mathcal{T}_{\Sigma,k}\}$ for each kind k, isomorphic to the initial algebra $\mathcal{T}_{\Sigma/E}$, where $E = E_o \cup B$.

[8] For any substitution θ and equation ($t = t'$ **if** *condition*) $\in E_o$, the least sort of $[\theta(t)]_A$ is greater than or equal to the least sort of $[\theta(t')]_A$.

[9] There exists no infinite E_o/B-rewrite sequence of ground B-equivalence classes $[t_0]_B \longrightarrow^1_{E_o/B} [t_1]_B \longrightarrow^1_{E_o/B} [t_2]_B \longrightarrow^1_{E_o/B} [t_3]_B \longrightarrow^1_{E_o/B} \cdots$.

[10] For ground terms $t, t_1, t_2 \in \mathcal{T}_\Sigma$, if $[t]_B \longrightarrow^*_{E_o/B} [t_1]_B$ and $[t]_B \longrightarrow^*_{E_o/B} [t_2]_B$, then there exists $t' \in \mathcal{T}_\Sigma$ such that $[t_1]_B \longrightarrow^*_{E_o/B} [t']_B$ and $[t_2]_B \longrightarrow^*_{E_o/B} [t']_B$.

[11] For ground terms $t_1, t'_1, t_2 \in \mathcal{T}_\Sigma$, if $t_1 \longrightarrow^1_{E_o/\emptyset} t'_1$ and $t_1 =_B t_2$, then there exists $t'_2 \in \mathcal{T}_\Sigma$ such that $t_2 \longrightarrow^1_{E_o/\emptyset} t'_2$ and $t'_1 =_B t'_2$.

Finally, R is *ground coherent* with E_o modulo B [166]. Together with the above conditions, this means that for $R/B = (\Sigma, B, R)$, if a one-step rewrite $\lambda : [t]_B \longrightarrow^1_{R/B} [t']_B$ holds for ground terms $t, t' \in \mathcal{T}_\Sigma$, then there exists a corresponding one-step rewrite $\lambda' : [can_{E_o/B}(t)]_B \longrightarrow^1_{R/B} [t'']_B$ with a *canonical* one-step proof term λ' such that $can_{E_o/B}(t') =_B can_{E_o/B}(t'')$. Therefore, provided the conditions C in rewrite rules $l : q \longrightarrow r$ **if** C only involve equations and memberships and do not have extra variables, $\longrightarrow^1_{\mathcal{R}}$ is decidable for ground terms, since we have $\lambda : [t]_E \longrightarrow^1_{\mathcal{R}} [t']_E$ iff

$$\lambda' : [can_{E_o/B}(t)]_B \longrightarrow^1_{R/B} [t'']_B \ \wedge \ can_{E_o/B}(t') =_B can_{E_o/B}(t'').$$

Similarly, this condition yields "canonical" models of $\mathcal{R} = (\Sigma, E_o \cup B, R)$. In particular, the *canonical interleaving model* $Can^1_{\mathcal{R}} = \{Can^1_{\mathcal{R},k}\}_{k \in K}$ with

$$Can^1_{\mathcal{R},k} = (Can_{\Sigma,E_o/B,k}, \longrightarrow^1_{\mathcal{R}})$$

is isomorphic to the initial interleaving model $\mathcal{T}^1_{\mathcal{R}}$ [45, 132], where the relation $\longrightarrow^1_{\mathcal{R}}$ is defined by the equivalence:

$$\lambda : [can_{E_o/B}(t)]_B \longrightarrow^1_{\mathcal{R}} [can_{E_o/B}(t')]_B \iff \lambda : [can_{E_o/B}(t)]_B \longrightarrow^1_{R/B} [t']_B.$$

Let $\mathit{CanProofTerms}^1(\mathcal{R})$ denote the set of all canonical one-step proof terms in $\mathcal{R}$. Then, an *infinite computation* of kind k is given by a pair of two functions (π, γ) with $\pi : \mathbb{N} \to Can_{\Sigma,E_o/B,k}$ and $\gamma : \mathbb{N} \to \mathit{CanProofTerms}^1(\mathcal{R})$ such that $\gamma(n) : \pi(n) \longrightarrow^1_{\mathcal{R}} \pi(n+1)$ is a canonical one-step rewrite for each $n \in \mathbb{N}$ [136]. Pictorially:

$$\pi(0) \xrightarrow{\gamma(0)}_{\mathcal{R}} \pi(1) \xrightarrow{\gamma(1)}_{\mathcal{R}} \pi(2) \xrightarrow{\gamma(2)}_{\mathcal{R}} \cdots.$$

These computability requirements are quite natural in practical system specifications [137]. Given a rewrite theory $\mathcal{R} = (\Sigma, E_o \cup B, R)$, the first condition holds for B any combination of associativity, commutativity, and identity axioms. For a Maude specification of $\mathcal{R}$, the tool can automatically check sort-decreasingness of E_o, and can guarantee B-coherence of E_o by a simple theory transformation [61]. Finally, the Maude Church-Rosser, Termination, and Coherence tools can check the rest of the executability conditions [61, 62]. For example, using these tools we can easily show that the Maude specification of the dining philosophers problem in Example 2.5 (see Example 2.10 in Section 2.3.2) satisfies the executability conditions.

2.3 Maude

Maude [61] is a declarative language and high-performance tool to support the formal specification and analysis of concurrent systems in rewriting logic. The language can directly specify MEL theories and rewrite theories, and the tool provides a number of formal analysis methods, such as simulation, reachability analysis, and LTL model checking. This section summarizes the syntax of the Maude language in a nutshell (see [61] for more details), illustrated with simple concurrent system examples.

2.3.1 Functional Modules

In Maude, a MEL theory (Σ, E) is specified by a *functional module* M, declared with the syntax: '`fmod` M `is` (Σ, E) `endfm`'. Sorts are declared by the keywords `sort` and `sorts` followed by identifiers, and subsort relations are declared by the keywords `subsort` and `subsorts`:

`subsort` s_1 `<` s_2 `.`

`subsorts` $s_{1,1}, \ldots, s_{1,n_1}$ `<` $\cdots$ `<` $s_{m,1}, \ldots, s_{m,n_m}$ `.`

A kind $[s]$ is automatically inferred for each connected component of a sort s in the poset $(S, \leq)$ of sorts. An operator f, with $s_1 \ldots s_n$ the sorts of its arguments and s its range sort, is declared with the syntax:

`op` f `:` $s_1 \ldots s_n$ `->` s `.`

`ops` f_1 f_2 $\ldots$ f_m `:` $s_1 \ldots s_n$ `->` s `.`

where the keyword `ops` can simultaneously define several operators of the same type. As explained in Section 2.1.4, it also declares the membership axiom $(\forall\, x_1, \ldots, x_n)\ f(x_1, \ldots, x_n) : s$ **if** $x_1 : s_1 \wedge \cdots \wedge x_n : s_n$. A *partial function* with no such axioms (i.e., a function $f : [s_1] \times \cdots \times [s_n] \to [s]$ at the kind level) can be declared by using the arrow '`~>`' instead of '`->`'.

Operators can have user-definable syntax with underbars '`_`', marking each of the argument positions (for example, `_+_` and `if_then_else_fi`). An operator f can be declared to have equational *attributes*, such as `comm`, `assoc`, and `id:` t, stating that f satisfies, respectively, the commutativity $(f(x, y) = f(y, x))$, associativity $(f(f(x, y), z) = f(x, f(y, z)))$, and identity $(f(x, t) = x)$ axioms. An operator can be declared to be a *constructor*, by the `ctor` attribute, that defines the carrier of a sort. A variable can be either explicitly declared by the keywords `var` and `vars` followed by variable identifiers, or can be introduced on-the-fly using the syntax *var* : *sort*. A comment is preceded by '`***`' or '`---`' and lasts till the end of the line.

Equations are declared by using the keywords `eq` and—for conditional equations—`ceq`, and memberships are declared by the keywords `mb` and—for conditional memberships—`cmb`:

`eq` $u = v$ `.` `ceq` $u = v$ `if` *condition* `.`
`mb` $u : s$ `.` `cmb` $u : s$ `if` *condition* `.`

An equation $f(t_1, \ldots, t_n) = t$ with the `owise` (for "otherwise") attribute can be applied to a term $f(\ldots)$ only if no other equation with left-hand side $f(u_1, \ldots, u_n)$ can be applied.[12] An equational condition $u = v$ can be declared as a *matching equation*, written u `:=` v, which instantiates the variables in the *pattern* u by a substitution θ such that $[\theta(u)]_E = [v]_E$.

Example 2.9. *The* Mel *theory* $(\Sigma_{\mathbb{N}}, E_{\mathbb{N}})$ *in Example 2.3 for the natural numbers is specified by the following functional module in Maude, including additional Boolean and comparison operators:*

```
fmod NATURAL-NUMBERS is
  sorts Bool Zero NzNat Nat .
  subsorts Zero NzNat < Nat .
  vars X Y : [Nat] .                 var B : Bool .

  op 0 : -> Zero [ctor] .            op s : Nat -> NzNat [ctor] .
  ops true false : -> Bool [ctor] .
  op _+_ : Nat Nat ~> Nat [comm assoc].
  op _<_ : Nat Nat ~> Bool .
  op _equals_ : Nat Nat ~> Bool [comm] .
  op not : Bool ~> Bool .
  op _or_ : Bool Bool ~> Bool [comm] .
  op _and_ : Bool Bool ~> Bool [comm] .

  eq X + 0 = X .             eq X + s(Y) = s(X + Y) .
  eq 0 < s(X) = true .       eq X < 0 = false .
  eq s(X) < s(Y) = X < Y .   eq X equals Y = not((X < Y) or (Y < X)) .
  eq not(true) = false .     eq not(false) = true .
  eq true or B = true .      eq false or false = false .
  eq false and B = false .   eq true and true = true .
endfm
```

For example, the following Maude reduce *command evaluates the given term and returns the result (i.e., its canonical form):*

```
Maude> red s(0) + s(s(0)) equals s(s(s(0))) .
result Bool: true
```

[12] A specification with `owise` equations can be transformed to an equivalent system without such equations [61].

2.3.2 System Modules

A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ is specified by a *system module* in Maude, declared with the syntax

$\texttt{mod}\ \mathcal{R}\ \texttt{is}\ (\Sigma, E, R)\ \texttt{endm}$

where rewrite rules in R are declared by using the keywords `rl` and—for conditional rules—`crl` as follows:

$\texttt{rl [}l\texttt{]: }u\texttt{ => }v\texttt{ .}\qquad \texttt{crl [}l\texttt{]: }u\texttt{ => }v\texttt{ if }\textit{condition}\texttt{ .}$

For both functional and system modules, a module inclusion relation can be declared with the keywords `including` and `protecting`. If a functional module M_1, specifying a MEL theory (Σ_1, E_1), *includes* another function module M_2 that specifies (Σ_2, E_2), then

$$(\Sigma_2, E_2) \subseteq (\Sigma_1, E_1).$$

Similarly, if a system module $\mathcal{R}_1 = (\Sigma_1, E_1, R_1)$ *includes* a system module $\mathcal{R}_2 = (\Sigma_2, E_2, R_2)$ (resp., a functional module $M_2 = (\Sigma_2, E_2)$), then

$$(\Sigma_2, E_2, R_2) \subseteq (\Sigma_1, E_1, R_1), \quad (\Sigma_2, E_2) \subseteq (\Sigma_1, E_1).$$

The `protecting` keyword asserts that the semantics of the submodule is preserved. A functional module M_1 *protects* a functional module M_2 iff $(\Sigma_2, E_2) \subseteq (\Sigma_1, E_1)$ and for each sort $s \in \Sigma_2$, both initial algebras $\mathcal{T}_{\Sigma_1/E_1,s}$ and $\mathcal{T}_{\Sigma_2/E_2,s}$ have the same set of elements.[13] Similarly, a system module $\mathcal{R}_1$ *protects* $\mathcal{R}_2$ iff $(\Sigma_2, E_2, R_2) \subseteq (\Sigma_1, E_1, R_1)$, (Σ_1, E_1) protects (Σ_2, E_2), and the reachability relation of $\mathcal{R}_2$ is preserved in $\mathcal{R}_2$.[14]

In the Maude tool, the *rewrite* command simulates *one behavior* of the system from the initial state t in k rewrite steps:

$\texttt{rew [}k\texttt{] t .}$

The *search* command analyzes *all possible behaviors* by using a breadth-first strategy to search for n states that are reachable from the initial state t, match the search *pattern*, and satisfy the search *condition*:

$\texttt{search [}n\texttt{] }t\texttt{ =>* }\textit{pattern}\texttt{ such that }\textit{condition}\texttt{ .}$

If the arrow `=>!` is used instead of `=>*` in the command, then it searches for terminating states that cannot be further rewritten by rewrite rules.

[13] For each sort $s \in \Sigma$, the unique homomorphism $\mathcal{T}_{\Sigma_2/E_2,s} \to \mathcal{T}_{\Sigma_1/E_1,s}$ induced by the theory inclusion $(\Sigma_2, E_2) \subseteq (\Sigma_1, E_1)$ is bijective.

[14] For two ground Σ_2-terms $t, t' \in \mathcal{T}_{\Sigma_2}$, $[t]_{E_2} \longrightarrow_{\mathcal{R}_2} [t']_{E_2} \iff [t]_{E_1} \longrightarrow_{\mathcal{R}_1} [t']_{E_1}$.

Example 2.10. *The rewrite theory $\mathcal{R}$ for the dining philosophers problem in Example 2.5 with $N = 2$ is specified by the following modules:*

```
fmod DINING-PHILOS-FUNCS is
  protecting NATURAL-NUMBERS .
  sorts Status .      ops think wait0 wait1 eat : -> Status [ctor] .
  op #N : ~> Nat .    --- a total number of philosophers
  ops lc rc : Nat ~> Nat .
  op adj : Nat Nat ~> Bool .
  vars I J : Nat .    ceq rc(I) = s(I) if (s(I) < #N) = true .
  eq lc(I) = I .      ceq rc(I) = 0 if (s(I) equals #N) = true .
  eq adj(I,J) = (J equals lc(I)) or (J equals rc(I)) .
endfm

mod DINING-PHILOS is
  including DINING-PHILOS-FUNCS .
  sorts Philo Chopstick Conf .
  subsorts Philo Chopstick < Conf .
  op p : Nat Status -> Philo [ctor] .
  op c : Nat -> Chopstick [ctor] .    op none : -> Conf [ctor] .
  op _||_ : Conf Conf -> Conf [ctor comm assoc id: none] .
  eq #N = s(s(0)) .   vars I J : Nat .      var C : Conf .

  *** defining the system behavior
  rl [wake]: p(I,think) => p(I,wait0) .
 crl [grabF]: p(I,wait0) || c(J) => p(I,wait1)  if adj(I,J) = true .
 crl [grabS]: p(I,wait1) || c(J) => p(I,eat)    if adj(I,J) = true .
  rl [stop]: p(I,eat) => p(I,think) || c(lc(I)) || c(rc(I)) .
endm
```

For example, the following rewrite *command executes the system from the initial state and returns a state reachable in three rewrite steps:*

```
Maude> rew [3] p(0,think) || c(0) || p(s(0),think) || c(s(0)) .
result Conf: p(0, eat) || p(s(0),think)
```

To find a deadlock *state from which no one-step rewrites exists, we can use the Maude* search *command with the arrow* **=>!** *as follows:*

```
Maude> search p(0,think) || c(0) || p(s(0),think) || c(s(0)) =>! C .

Solution 1 (state 15)
C:Conf --> p(0, wait-1) || p(s(0), wait-1)

No more solutions.
```

2.3.3 Object-Oriented Modules

Rewriting logic is particularly useful to formally specify concurrent systems in an object-oriented style. Concurrent object system are typically modeled by a set of *objects* that can interact with each other either synchronously, or asynchronously by sending and receiving *messages*. As already mentioned, other object-oriented models for concurrent systems, such as actors [5], can be naturally specified in rewriting logic [132, 137].

In Maude, each state of a system, called a *configuration*, is modeled as a term of sort Configuration that has a structure of a *multiset* made up of objects and messages. Multiset union for configurations is denoted by a juxtaposition operator (empty syntax) that is declared associative and commutative and has `none` as its identity as follows [61]:

```
sorts Object Msg Configuration .
subsorts Object Msg < Configuration .
op none : -> Configuration [ctor] .
op __ : Configuration Configuration -> Configuration
                              [ctor assoc comm id: none] .
```

In Full Maude—a language extension of Maude—one can declare *classes* and *messages* in a *object-oriented module* [61], declared with the syntax:[15]

$$\texttt{(omod } \mathcal{R} \texttt{ is } (\Sigma, E, R) \texttt{ endom)}$$

Object-oriented modules are just syntactic sugar, and are automatically transformed into system modules in Maude [61]. A *class* declaration

$$\texttt{class } C \texttt{ | } att_1 : s_1 \texttt{,} \ldots \texttt{,} att_n : s_n \texttt{ .}$$

declares a class C with attributes $att_1, \ldots, att_n$ of sorts $s_1, \ldots, s_n$. An *object* of class C is represented as a term of sort Object and has the form:

$$\texttt{< O : C | } att_1 : val_1 \texttt{,} \ldots \texttt{,} att_n : val_n \texttt{ >},$$

where O is the object's identifier of sort Oid, C is the class identifier of sort Cid, and $val_1, \ldots, val_n$ are its attribute values of sort $s_1, \ldots, s_n$. A *subclass*, introduced with the keyword `subclass`, inherits all the attributes of its superclasses. A *message* is a term of sort Msg, where the declaration

$$\texttt{msg } m : s_1 \ldots s_n \texttt{ -> Msg .}$$

defines the syntax of the message $m(v_1, \ldots, v_n)$ and the sorts $s_1, \ldots, s_n$ of its parameters $v_1, \ldots, v_n$.

[15] In Full Maude, module declarations and commands must be enclosed by parentheses.

Example 2.11. *The Maude specification in Example 2.10 for the dining philosophers problems, when $N = 2$, can be alternatively specified by:*

```
(omod OO-DINING-PHILOS is
  including DINING-PHILOS-FUNCS .          vars I J : Nat .
  class Philo | status : Status .          msg c : Nat -> Msg .
  subsort Nat < Oid .                      eq #N = s(s(0)) .
  rl [wake]:   < I : Philo | status : think >
          =>   < I : Philo | status : wait0 > .
 crl [grabF]:  < I : Philo | status : wait0 > c(J)
           =>  < I : Philo | status : wait1 >  if adj(I,J) = true .
 crl [grabS]:  < I : Philo | status : wait1 > c(J)
           =>  < I : Philo | status : eat >    if adj(I,J) = true .
  rl [stop]:   < I : Philo | status : eat >
          =>   < I : Philo | status : think > c(lc(I)) c(rc(I)) .
endom)
```

Example 2.12 (Client-Server Communication)**.** *There are a number of clients and servers, with status either idle or busy, where each server can serve many clients but each client communicates with one server. An idle client* `C` *sends a query* `N` *to a server* `S` *and becomes busy. If the server* `S` *is idle, then* `S` *becomes busy and returns the answer* `f(S,N)` *of the query using a function* `f`*, only known to the server itself. A busy client can receive an answer and become idle, and a busy server can become idle at any time. This system can be specified by the following object-oriented module:*[16]

```
(omod OO-CLIENT-SERVER is
  including NAT .
  class Node | status : Status .       msg m : Oid Oid Nat -> Msg .
  class Client | val : Nat, to : Oid . class Server .
  subclass Server Client < Node .      sorts Status .
  ops idle busy : -> Status [ctor] .   op f : Oid Nat -> Nat .
  vars C S : Oid .                     var N : Nat .
  rl [reqs]: < C : Client | status : idle, val : N, to : S >
          => < C : Client | status : busy > m(S,C,N) .
  rl [repl]: < S : Server | status : idle > m(S,C,N)
          => < S : Server | status : busy > m(C,S,f(S,N)) .
  rl [recv]: < C : Client | status : busy > m(C,S,N)
          => < C : Client | status : idle, val : N > .
  rl [idle]: < S : Server | status : busy >
          => < S : Server | status : idle > .
endom)
```

[16] By convention, attributes of objects can be omitted from (one side of) a rewrite rule if they are not relevant, e.g., the attributes `to` and `val` in the *recv* rule [61].

2.3.4 Real-Time Maude

Real-Time Maude [149] is a language and tool for real-time systems that extends Maude [61]. A Real-Time Maude *timed module* specifies a *real-time rewrite theory* $\mathcal{R} = (\Sigma, E, R)$ [148], where:

- (Σ, E) contains an equational subtheory $(\Sigma_{TIME}, E_{TIME}) \subseteq (\Sigma, E)$ that satisfies the *TIME* axioms in [148] specifying sort Time for the time domain (which can be discrete or dense). The supersort TimeInf extends the sort Time with an "infinity" value `INF`.
- The rules in R are decomposed into:
  - "ordinary" rewrite rules specifying the system's *instantaneous* (i.e., zero-time) local transitions, and
  - *tick rules*, that model the elapse of time in a system, of the form

    $$l : \{t\} \xrightarrow{u} \{t'\} \textbf{ if } \textit{condition},$$

    where t and t' are of sort System, u is of sort Time denoting the *duration* of the rewrite, and `{_}` is a built-in constructor of sort GlobalSystem. In Real-Time Maude, tick rules, together with their durations, are specified with the syntax:

    `crl [`l`]: {`t`} => {`t'`} in time` u `if` *condition*.

The initial state must be reducible to a term $\{t_0\}$, for t_0 a ground term of sort Configuration (which is a subsort of System) in an object-oriented style, using the equations in the specification. The form of the tick rules then ensures uniform time elapse in all parts of the system.

Real-Time Maude provides formal analysis methods, including simulation, reachability analysis, and metric LTL and timed CTL model checking [149]. For example: (i) the *timed rewrite* command `(tfrew` t `in time <=` τ `.)` simulates *one behavior* of the system within time τ from the initial state t, (ii) the *timed search* command `(tsearch [`n`]` t `=>*` *pattern* `such that` *condition* `in time <=` τ `.)` analyzes *all possible behaviors* to search for n states that are reachable from the initial state t within time τ, match the search *pattern*, and satisfy the search *condition*, and (iii) the *timed* LTL model checking command `(mc` t `|=t` φ `in time <=` τ `.)` checks whether the LTL formula φ holds from the initial state t *within time* τ, extending Maude's LTL model checking commands.

CHAPTER 3

LINEAR TEMPORAL LOGIC OF REWRITING

The *linear temporal logic of rewriting* (LTLR) extends linear temporal logic (LTL) by adding *spatial action patterns* that describe patterns of rewrite events. This chapter[1] presents the foundation and design of the LTLR model checker, developed at the C++ level by extending the existing LTL model checker within the Maude system. LTLR generalizes various state-based and event-based logics, so that "mixed" properties involving both states and events, such as fairness properties, can be naturally expressed in LTLR. This greater expressiveness is gained without compromising performance, since the LTLR algorithm minimizes the extra costs in handling events.

3.1 Introduction

As mentioned in Chapter 1, model checking of a concurrent system involves two different tasks: (i) system specifications to formally model the system in a system specification language $\mathcal{L}_S$, and (ii) property specifications to define the requirements in a temporal logic $\mathcal{L}_P$, where $(\mathcal{L}_S, \mathcal{L}_P)$ is called a *tandem* of logics [136]. Given a system specification $S \in \mathcal{L}_S$ and a property specification $\varphi \in \mathcal{L}_P$, model checking tools verify the satisfaction relation

$$S \models \varphi.$$

For example, using rewriting logic as a system specification language, the Maude LTL model checker [61] can verify an LTL property φ of a rewrite theory $\mathcal{R}$, where each atomic proposition p in φ is related to each state $[t]_E$ of $\mathcal{R}$ by means of the satisfaction relation $[t]_E \models p$.

[1]This chapter is based on the papers [17, 22], joint work with José Meseguer.

However, such a satisfaction relation $S \models \varphi$ is *not* always definable in a natural way when there is a *mismatch* between $\mathcal{L}_S$ and $\mathcal{L}_P$ [136]. The mismatch problems are typically caused by lack of expressiveness in one of the logics. For example, there exists a mismatch between rewriting logic and temporal logics that are either only state-based (e.g., LTL and CTL* [60]) or only event-based (e.g., Hennessy-Milner logic [103] and A-CTL* [145]). Rewriting logic can specify *both* state-based and event-based aspects of a system but one of these aspects is missing in those temporal logics.[2]

In practice this problem can be partially resolved by "cooking" of the system specification to *encode* the model features not directly expressible in the property logic. For instance, to deal with the event-based aspects of a system using a state-based temporal logic, we may encode action information in a modified version of the state and introduce appropriate atomic state propositions that can detect certain actions having taken place. However, this encoding often makes the system specification unnecessarily complex, and further increases the size of the state space.

The *temporal logic of rewriting* (TLR) [136] is an expressive temporal logic that adds *atomic event propositions* to CTL*. Moreover, not only atomic events (corresponding to rule applications), but also *spatial action patterns* indicating *where* in the state structure, and *for which instances* a given action took place are expressible in TLR. When used together with rewriting logic (RWL), the (RWL, TLR) tandem can avoid such mismatch problems as explained in [136]. But how usable and well-supported by algorithms and tools is the (RWL, TLR) tandem in practice?

To answer the question, this chapter presents a model checking algorithm and a tool for the (RWL, LTLR) tandem, a subtandem of (RWL, TLR). The *linear temporal logic of rewriting* (LTLR) is a sublogic of TLR that extends LTL by just adding spatial action patterns. Since LTL is one of the simplest and most widely used state-based logics, the good features of LTL are also shared by LTLR. Furthermore, since Maude already has an efficient LTL model checker, it has been possible to reuse some of the components of its C++ implementation in the new LTLR model checker.

---

[2] E.g., consider the rewriting logic specification of the dining philosophers problem in Example 2.5. Since there exists no event information in the state, the satisfaction of the fairness properties in Example 2.8 cannot be easily defined using the LTL semantics.

3.1.1 Main Contributions

First, this chapter presents the *automata-theoretic foundations* for an LTLR model checking algorithm. Specifically, model checking an LTLR formula φ for a rewrite theory $\mathcal{R}$ is equivalent to deciding a language emptiness problem for a Büchi automaton, obtained as a "special" synchronous product of the Büchi automaton $\mathscr{B}_{\neg\varphi}$ for the negation of φ and a labeled Kripke structure $\bar{\mathcal{K}}$ naturally associated to $\mathcal{R}$, that recognizes a *union trace* of $\bar{\mathcal{K}}$ given by a sequence of sets of state propositions and spatial action patterns.

Second, this chapter extends the original notion of spatial action patterns in [136], denoted by $SP(\mathcal{R})$, into any form of spatial action patterns that can be *equationally* defined with respect to one-step proof terms, in a way similar to defining state propositions for LTL model checking in rewriting logic. In this characterization, $SP(\mathcal{R})$ is just a special case, since the syntax and semantics of $SP(\mathcal{R})$ can also be defined by using equations. This is particularly useful when a system requirement contains a complex event proposition that cannot be expressed using simple action patterns.

Third, this chapter presents the Maude LTLR model checker, based on the automata-theoretic foundations and the extended notion of spatial action patterns, and implemented at C++ level as an extension of the Maude system. This implementation uses an *intrinsic* model checking algorithm, using the automata-theoretic foundations, that does *not* incur any state space blowup. In contrast, a previous LTLR model checker implementation, documented in [20], used a theory transformation $\mathcal{R} \mapsto T(\mathcal{R})$ that encoded event information as part of the system state [136], but the state space of the transformed theory $T(\mathcal{R})$ could be much bigger than that of the original theory $\mathcal{R}$. Substantial effort has also gone into designing and implementing a *user interface*. This is nontrivial because:

- the syntax of spatial action patterns, typically *not* a part of the system specification, must be made available to the user; and
- the possibility of *extending* the basic LTLR, which corresponds to the extended notion of spatial action patterns, should also be supported by the tool to make the system more flexible.

A case study of the bounded retransmission protocol, involving *user-defined* spatial action patterns, shows the expressiveness and effectiveness of our tool. More case studies can also be found in Chapter 4 and Appendix A.1.

3.1.2 Related Work

The first implementation of an LTLR model checker was presented in [20]. However, the implementation reused the Maude LTL model checker *as given*, without any changes in its algorithm. It relied on a *theory and formula transformation*, so that the LTLR model checking problem was transformed into an equivalent LTL model checking problem for a transformed Maude specification. As mentioned above, while useful for experimental purposes, this solution was not optimal for event-based properties, because rewrite events had to be encoded in the state of the transformed Maude specification, leading to a considerable increase of the state space.

The family of TLR logics is introduced in [136]. Besides LTLR, the most general one of these logics is TLR*, which generalizes the state-based logic CTL*. Many well-known state-based logics, such as LTL, CTL, and CTL* [60, 130], and event-based logics, such as Hennessy-Milner's logic [103] or De Nicola and Vaandrager's A-CTL* [145], can be viewed as special cases of TLR* [135]. There are a number of approaches to combine state-based and event-based formulas; we refer to [22, 136] for an overview and comparison with TLR. In particular, the state/event-based logic SE-LTL proposed in [49, 50], where each transition is labeled by a *single* event, can also be considered as a special case of LTLR. Our automata-theoretic LTLR model checking algorithm generalizes the SE-LTL model checking algorithm in [49] to allow transitions labeled by a set (or a "pattern") of atomic events.

3.1.3 Structure of the Chapter

This chapter is organized as follows. Section 3.2 describes the syntax and semantics of LTLR, including spatial action patterns and labeled Kripke structures, and explains how a rewrite theory $\mathcal{R}$ can be associated with its underlying labeled Kripke structure $\bar{\mathcal{K}}$ to define the semantics of an LTLR property φ with respect to $\mathcal{R}$. Section 3.3 presents an automata theoretic foundation for LTLR model checking, generalizing the standard automata theoretic technique for LTL model checking. Section 3.4 explains the design of the Maude LTLR model checker, including its user interface. Section 3.5 illustrates our tool with a case study, showing the effectiveness of the tool. Finally, Section 3.6 gives some concluding remarks.

3.2 Syntax and Semantics

The *linear temporal logic of rewriting* (LTLR) is a state/event extension of LTL with *spatial action patterns*. Recall that the syntax of LTL is defined by $\varphi ::= p \mid \neg\varphi \mid \varphi\wedge\varphi' \mid \bigcirc\varphi \mid \varphi\mathbf{U}\varphi'$, where p is an atomic state proposition [60]. The only syntactic difference of LTLR is that an LTLR formula may include spatial action patterns $\delta_1, \ldots, \delta_n$ as well as state propositions $p_1, \ldots, p_m$, and therefore may describe properties involving both states and events.

Definition 3.1 (The LTLR Syntax)**.** *Given a set of state propositions AP and a set of spatial action patterns ACT, the syntax of LTLR is defined by:*

$$\varphi ::= p \mid \delta \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi\mathbf{U}\varphi, \quad \textit{where } p \in AP \textit{ and } \delta \in ACT.$$

Other logical and temporal operators can be defined by using equivalences, e.g., $\varphi \vee \varphi' \equiv \neg(\neg\varphi \wedge \neg\varphi')$, $\varphi \to \varphi' \equiv \neg\varphi \vee \varphi'$, $\Diamond\varphi \equiv \mathit{true}\,\mathbf{U}\,\varphi$, $\Box\varphi \equiv \neg\Diamond\neg\varphi$, $\varphi\mathbf{R}\varphi' \equiv \neg(\neg\varphi\mathbf{U}\neg\varphi')$, and $\varphi\mathbf{W}\varphi' \equiv (\varphi\mathbf{U}\varphi') \vee \Box\varphi$.

3.2.1 Spatial Action Patterns

Spatial action patterns are the action atoms of LTLR that describe actions of a rewrite theory $\mathcal{R}$. In a concurrent system specified by $\mathcal{R}$, actions of the system can be considered as rewrite events triggered by the rules in $\mathcal{R}$, characterized by one-step proof terms. A spatial action pattern δ defines a set of one-step proof terms λ that match the pattern δ.

We define a simple language for spatial action patterns that can express a wide range of atomic actions. Disregarding rewrite condition proof terms, a one-step proof term[3] $t[l(\theta, \beta_1, \ldots, \beta_m)]_p$ is represented as a triple

$$\texttt{\{} \quad t[\Box]_p \quad \texttt{|} \quad {'l} \quad \texttt{:} \quad {'x_1} \setminus u_1 \ \texttt{;}\ \ldots\ \texttt{;}\ {'x_m} \setminus u_m \quad \texttt{\}}$$

of a *context term* $t[\Box]_p$ that contains a hole $\Box$ at position p, a *quoted* rule label $'l$, and a substitution $\theta = \{x_1 \mapsto u_1, \ldots, x_m \mapsto u_m\}$ expressed as a semicolon-separated assignment set, where $'x_i$ is a quoted identifier for a variable x_i, $1 \leq i \leq m$. We use quoted identifiers for labels and variable names, since $\mathcal{R}$ may contain other constants with the same names.

[3] A one-step proof term $t[l(\theta, \beta_1, \ldots, \beta_m)]_p$ indicates that a rule with label l has been applied with substitution θ at position p of the state t, where $\beta_1, \ldots, \beta_m$ denote proof terms for the rewrite conditions of the rule (see Section 2.2.1). In this thesis, $\beta_1, \ldots, \beta_m$ are *not* used to define spatial action patterns (that is, two one-step proof terms $t[l(\theta, \beta_1, \ldots, \beta_m)]_p$ and $t[l(\theta, \beta_1', \ldots, \beta_{m'}')]_p$ correspond to the same set of spatial action patterns).

Definition 3.2. *Given a rewrite theory $\mathcal{R}$, $SP(\mathcal{R})$ is the set of spatial action patterns that have one of the forms [19, 136]:*

$$\{'l\} \quad \{'l : \theta\} \quad \{u[\Box]_q \mid 'l\} \quad \{u[\Box]_q \mid 'l : \theta\} \quad top\{'l\} \quad top\{'l : \theta\}$$

where $'l$ is a quoted label, θ is a substitution of the form $'x_1\backslash u_1; \ldots ; 'x_m\backslash u_m$, $u[\Box]_q$ is a context term, and $u, u_1, \ldots, u_m \in \mathcal{T}_\Sigma$ are ground terms in $\mathcal{R}$. In particular, $BP(\mathcal{R}) \subseteq SP(\mathcal{R})$ is the set of spatial action patterns either of the form $\{'l\}$ or $\{'l : \theta\}$, called basic action patterns.

A spatial action pattern in $SP(\mathcal{R})$ can be viewed as a *partial description* of a one-step proof term that specifies a set of more specific one-step proof term instances. A basic action pattern $\{'l\}$ describes a rule labeled l that can be applied *anywhere*. A basic action pattern $\{'l : \theta\}$ allows l to also be applied anywhere, but constrains the variable instantiation related to rule l to be an extension of the substitution θ. Spatial action patterns $\{u[\Box]_q \mid 'l\}$ and $\{u[\Box]_q \mid 'l : \theta\}$ describe one-step rewrites related to the basic action patterns $\{'l\}$ and $\{'l : \theta\}$, respectively, where l is applied at position p of the term u. Similarly, spatial action patterns $top\{'l\}$ and $top\{'l : \theta\}$ cover the cases where l is applied at the top of the term. Notice that a one-step proof term λ is a spatial action pattern matching itself.

Definition 3.3. *The matching relation $\models$ between a one-step proof term and a spatial action pattern in $SP(\mathcal{R})$ is then formalized as follows, where $\theta \subseteq_E \vartheta$ iff for each $'x\backslash u \in \theta$, there exists $'x\backslash v \in \vartheta$ such that $u =_E v$, and $t[\Box]_p =_E u[\Box]_q$ iff $t[x]_p =_E u[x]_q$ for an appropriate kinded variable x:*

$$\begin{array}{rcll}
\{t[\Box]_p \mid 'l : \vartheta\} & \models & \{'l\} & \\
\{t[\Box]_p \mid 'l : \vartheta\} & \models & \{'l : \theta\} & \iff \quad \theta \subseteq_E \vartheta \\
\{t[\Box]_p \mid 'l : \vartheta\} & \models & \{u[\Box]_q \mid 'l\} & \iff \quad t[\Box]_p =_E u[\Box]_q \\
\{t[\Box]_p \mid 'l : \vartheta\} & \models & \{u[\Box]_q \mid 'l : \theta\} & \iff \quad t[\Box]_p =_E u[\Box]_q \ \wedge \ \theta \subseteq_E \vartheta \\
\{\Box \mid 'l : \vartheta\} & \models & top\{'l\} & \\
\{\Box \mid 'l : \vartheta\} & \models & top\{'l : \theta\} & \iff \quad \theta \subseteq_E \vartheta.
\end{array}$$

The syntax and semantics of $SP(\mathcal{R})$ can be defined by using equations in membership equational logic, which can be *automatically* generated from $\mathcal{R}$ as explained in Section 3.4.1. This equational semantics enables us to easily define more general kinds of spatial action patterns.

3.2.2 Labeled Kripke Structures

To compare LTLR with LTL we first recall the semantics of LTL that is defined with respect to a *Kripke structure* [60].

Definition 3.4. *A* Kripke structure *is a* 4*-tuple* $\mathcal{K} = (S, AP, \mathcal{L}, \rightarrow_{\mathcal{K}})$ *with* S *a set of* states*,* AP *a set of* atomic state propositions*,* $\mathcal{L} : S \rightarrow 2^{AP}$ *a* state-labeling function*, and* $\rightarrow_{\mathcal{K}} \subseteq S \times S$ *a total*[4] transition relation*.*

A *path* $\pi : \mathbb{N} \rightarrow S$ is an infinite sequence with $\pi(i) \rightarrow_{\mathcal{K}} \pi(i+1)$ for each $i \in \mathbb{N}$. An LTL formula φ is satisfied from a set of initial states $S_0 \subseteq S$ in $\mathcal{K}$, denoted by $\mathcal{K}, S_0 \models \varphi$, iff for each path π with $\pi(0) \in S_0$, the path satisfaction relation $\mathcal{K}, \pi \models \varphi$ holds, which is inductively defined as follows, where π^i is the suffix of π starting at $\pi(i)$:

$$
\begin{aligned}
\mathcal{K}, \pi \models p &\iff p \in \mathcal{L}(\pi(0)) \\
\mathcal{K}, \pi \models \neg\varphi &\iff \mathcal{K}, \pi \not\models \varphi \\
\mathcal{K}, \pi \models \bigcirc\varphi &\iff \mathcal{K}, \pi^1 \models \varphi \\
\mathcal{K}, \pi \models \varphi \wedge \varphi' &\iff \mathcal{K}, \pi \models \varphi \;\wedge\; \mathcal{K}, \pi \models \varphi' \\
\mathcal{K}, \pi \models \varphi \,\mathbf{U} \varphi' &\iff (\exists j \geq 0)\, \mathcal{K}, \pi^j \models \varphi' \;\wedge\; (\forall 0 \leq i < j)\, \mathcal{K}, \pi^i \models \varphi.
\end{aligned}
$$

Similarly, the semantics of LTLR is defined with respect to a *labeled Kripke structure*, a natural extension of a Kripke structure with transition labels [17, 49]. A labeled Kripke structure is a Kripke structure whose transition is also labeled by a (possibly empty) *set* Λ of atomic events, which enables us to describe an event pattern as well as just an atomic event.

Definition 3.5. *A* labeled Kripke structure *(LKS) is defined by a* 5*-tuple* $\bar{\mathcal{K}} = (S,\ AP,\ \mathcal{L},\ ACT,\ \rightarrow_{\bar{\mathcal{K}}})$*, where:*

- S *is a set of* states*;*
- AP *is a set of* state propositions*;*
- $\mathcal{L} : S \rightarrow 2^{AP}$ *is a* state-labeling function*;*
- ACT *is a set of* atomic events *(e.g., spatial action patterns); and*
- $\rightarrow_{\bar{\mathcal{K}}} \subseteq S \times 2^{ACT} \times S$ *is a total* labeled transition relation*.*

A *path* (π, α) is a pair of functions $\pi : \mathbb{N} \rightarrow S$ and $\alpha : \mathbb{N} \rightarrow 2^{ACT}$ such that $\pi(i) \xrightarrow{\alpha(i)}_{\bar{\mathcal{K}}} \pi(i+1)$ for each $i \in \mathbb{N}$. We denote by $(\pi, \alpha)^i$ the suffix of (π, α) beginning at position $i \in \mathbb{N}$ (that is, $(\pi, \alpha)^i = (\pi \circ s^i, \alpha \circ s^i)$).

[4] That is, each state $s \in S$ has a next state $s' \in S$ with $s \rightarrow_{\mathcal{K}} s'$.

Definition 3.6 (The LTLR Semantics)**.** *An LTLR formula φ is satisfied from a set of initial state $S_0 \subseteq S$ in an LKS $\bar{\mathcal{K}}$, denoted by $\bar{\mathcal{K}}, S_0 \models \varphi$, iff for each path (π, α) of $\bar{\mathcal{K}}$ such that $\pi(0) \in S_0$, the path satisfaction relation $\bar{\mathcal{K}}, (\pi, \alpha) \models \varphi$ holds, which is inductively defined as follows:*

$$
\begin{aligned}
\bar{\mathcal{K}}, (\pi, \alpha) \models p \quad &\iff p \in \mathcal{L}(\pi(0)) \\
\bar{\mathcal{K}}, (\pi, \alpha) \models \delta \quad &\iff \delta \in \alpha(0) \\
\bar{\mathcal{K}}, (\pi, \alpha) \models \neg\varphi \quad &\iff \bar{\mathcal{K}}, (\pi, \alpha) \not\models \varphi \\
\bar{\mathcal{K}}, (\pi, \alpha) \models \varphi \wedge \varphi' \quad &\iff \bar{\mathcal{K}}, (\pi, \alpha) \models \varphi \ \wedge \ \bar{\mathcal{K}}, (\pi, \alpha) \models \varphi' \\
\bar{\mathcal{K}}, (\pi, \alpha) \models \bigcirc\varphi \quad &\iff \bar{\mathcal{K}}, (\pi, \alpha)^1 \models \varphi \\
\bar{\mathcal{K}}, (\pi, \alpha) \models \varphi \, \mathbf{U} \varphi' \quad &\iff (\exists j \geq 0)\ \bar{\mathcal{K}}, (\pi, \alpha)^j \models \varphi' \wedge \\
&\qquad (\forall 0 \leq i < j)\ \bar{\mathcal{K}}, (\pi, \alpha)^i \models \varphi.
\end{aligned}
$$

Notice that the only difference between the LTLR and the LTL semantics is the semantics of *spatial action patterns*. Specifically, $\bar{\mathcal{K}}, (\pi, \alpha) \models \delta$ holds iff the first transition $\pi(0) \xrightarrow{\alpha(0)}_{\bar{\mathcal{K}}} \pi(1)$ corresponds to the action pattern δ.

3.2.3 Relating LKSs and Rewrite Theories

This section explains how to associate to a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ a corresponding labeled Kripke structure $\bar{\mathcal{K}} = (S, AP, \mathcal{L}, ACT, \rightarrow_{\bar{\mathcal{K}}})$. Basically, in order to define the semantics of AP and ACT, we need to have: (i) the *state satisfaction relation* $[t]_E \models p$ for each state $[t]_E$ and state proposition $p \in AP$; and (ii) the *action satisfaction relation* $\lambda \models \delta$ for each one-step proof term λ and spatial action pattern $\delta \in ACT$.

These satisfaction relations for $\mathcal{R} = (\Sigma, E, R)$ can be defined by means of equations D in a *support equational theory* $\mathcal{P} = (\Pi, D)$, including:

- A kind $k \in \Sigma$ for states, sort $\mathsf{ProofTerm}$ for one-step proof terms, sort Prop for state propositions, and sort Action for spatial action patterns;
- Two boolean constants *true* and *false* of sort Bool in Π such that $\mathit{true} \neq_{E \cup D} \mathit{false}$, and for each ground term $t \in \mathcal{T}_{\Sigma \cup \Pi, \mathsf{Bool}}$ of sort Bool, either $t =_{E \cup D} \mathit{true}$ or $t =_{E \cup D} \mathit{false}$ holds;
- A signature for one-step proof terms, including context terms, quoted identifiers, and substitutions; and
- Operators $_\models_ : k\ \mathsf{Prop} \rightarrow \mathsf{Bool}$ and $_\models_ : \mathsf{ProofTerm}\ \mathsf{Action} \rightarrow \mathsf{Bool}$.

A state proposition is then defined as a term of sort $\mathsf{Prop} \in \Pi$, using function symbols in Π of the form $p : s_1 \ldots s_n \to \mathsf{Prop}$. For each ground state proposition $p(u_1, \ldots, u_n) \in \mathcal{T}_{\Sigma\cup\Pi,\mathsf{Prop}}$ and state $[t]_E \in \mathcal{T}_{\Sigma\cup\Pi,k}$, the state satisfaction relation is given by:

$$[t]_E \models p(u_1, \ldots, u_n) \quad\iff\quad t \models p(u_1, \ldots, u_n) \;=_{E\cup D}\; \mathit{true}.$$

Likewise, a spatial action pattern is defined as a term of sort $\mathsf{Action} \in \Pi$, using function symbols in Π of the form $\delta : s_1 \ldots s_m \to \mathsf{Action}$. For each ground spatial action pattern $\delta(v_1, \ldots, v_m) \in \mathcal{T}_{\Sigma\cup\Pi,\mathsf{Action}}$ and one-step proof term λ, the action satisfaction relation is given by:

$$\lambda \models \delta(v_1, \ldots, v_m) \quad\iff\quad \lambda \models \delta(v_1, \ldots, v_m) \;=_{E\cup D}\; \mathit{true}.$$

Note that spatial action patterns in $SP(\mathcal{R})$ can be *automatically* generated (see Section 3.4.1). Finally, since the set of states $\mathcal{T}_{\Sigma/E,k}$ should not be disturbed by $\mathcal{P} = (\Pi, D)$, the extended theory $(\Sigma \cup \Pi, E \cup D)$ should *protect* (Σ, E) (that is, $\mathcal{T}_{\Sigma\cup\Pi/E\cup D,s} \simeq \mathcal{T}_{\Sigma/E,s}$ for each sort $s \in \Sigma$).

Example 3.1. *For the dining philosophers model in Example 2.10, the state proposition $enabled(wake(i))$—meaning that the philosopher i can wake up—and the spatial action pattern $wake(i)$—meaning that the philosopher i has woken up—can be defined by the following equations in Maude:*

```
var CF : Conf .        vars I J : Nat .      var R : RuleName .
var SUB : StateSubstitution .                var CXT : StateContext .

op wake : Nat -> Action [ctor] .
eq {CXT | 'wake : 'I \ I ; SUB} |= wake(I) = true .
eq {CXT | R : SUB} |= wake(I) = false [owise] .

op enabled : Action -> Prop [ctor] .
eq (p(I,think) || CF) |= enabled(wake(I))= true .
eq CF |= enabled(wake(I)) = false [owise] .
```

Notice that the spatial action pattern $wake(i)$ is equivalent to the spatial action pattern $\{'wake : 'I \backslash i\}$ in $SP(\mathcal{R})$.

We now associate to a rewrite theory $\mathcal{R}$ an LKS $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}$, where state propositions and spatial action patterns are specified by a support equational theory $\mathcal{P}$. Since $\mathcal{R}$ may contain a *deadlock state* from which *no* one-step rewrites exists, $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}$ adds the self loop $[t]_E \xrightarrow{\{\mathtt{deadlock}\}}_{\bar{\mathcal{K}}} [t]_E$ for each deadlock state $[t]_E$, labeled by a predefined event `deadlock`.

Definition 3.7. *Given a rewrite theory* $\mathcal{R} = (\Sigma, E, R)$*, a support equational theory* $\mathcal{P} = (\Pi, D)$*, and a kind* $k \in \Sigma$ *of states, the corresponding LKS is*

$$\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}} = (\mathcal{T}_{\Sigma/E,k},\ AP,\ \mathcal{L}_{\mathcal{P}},\ ACT,\ \rightarrow_{\bar{\mathcal{K}}}),$$

where $AP = \mathcal{T}_{\Sigma\cup\Pi,Prop}$*,* $ACT = \mathcal{T}_{\Sigma\cup\Pi,Action} \cup \{\texttt{deadlock}\}$*, and:*

- $\mathcal{L}_{\mathcal{P}}([t]_E)$ *is the set of state propositions satisfied in state* $[t]_E$*, i.e.:*

$$\mathcal{L}_{\mathcal{P}}([t]_E) = \{p \in AP \mid (t \models p) =_{E\cup D} true\}.$$

- $\rightarrow_{\bar{\mathcal{K}}}$ *is the total transition relation, labeled with either events in ACT or a deadlock event* `deadlock` *such that* $[t]_E \xrightarrow{\Lambda}_{\bar{\mathcal{K}}} [t']_E$ *iff:*

$$\lambda : [t]_E \longrightarrow^1_{\mathcal{R}} [t']_E, \textit{and } \Lambda = \{\delta \in ACT \mid (\lambda \models \delta) =_{E\cup D} true\}; \textit{or}$$
$$t = t', \Lambda = \{\texttt{deadlock}\}, \textit{and } [t]_E \textit{ cannot be rewritten by } \longrightarrow^1_{\mathcal{R}}.$$

Example 3.2. *Consider the dining philosophers model in Example 2.10. If* $AP = \{enabled(wake(0))\}$ *and* $ACT = \{wake(0), \{'wake\}, \texttt{deadlock}\}$*, where* $enabled(wake(0))$ *and* $wake(0)$ *are defined in Example 3.1, then there exists the following path* (π, α) *in the corresponding LKS* $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}$*:*

$$[p(0, \texttt{think}) \parallel c(0) \parallel p(1, \texttt{think}) \parallel c(1)]_E \qquad \mathcal{L}_{\mathcal{P}}(\pi(0)) = \{enabled(wake(0))\}$$
$$\downarrow \{\{'wake\}\}$$
$$[p(0, \texttt{think}) \parallel c(0) \parallel p(1, \texttt{wake0}) \parallel c(1)]_E \qquad \mathcal{L}_{\mathcal{P}}(\pi(1)) = \{enabled(wake(0))\}$$
$$\downarrow \emptyset$$
$$[p(0, \texttt{think}) \parallel c(0) \parallel p(1, \texttt{wake1})]_E \qquad \mathcal{L}_{\mathcal{P}}(\pi(2)) = \{enabled(wake(0))\}$$
$$\downarrow \{wake(0), \{'wake\}\}$$
$$[p(0, \texttt{wait0}) \parallel c(0) \parallel p(1, \texttt{wake1})]_E \qquad \mathcal{L}_{\mathcal{P}}(\pi(3)) = \emptyset$$
$$\downarrow \emptyset$$
$$\{\texttt{deadlock}\} \circlearrowright [p(0, \texttt{wait1}) \parallel p(1, \texttt{wake1})]_E \qquad \mathcal{L}_{\mathcal{P}}(\pi(i)) = \emptyset,\ \textit{for } i \geq 4.$$

For an rewrite theory $\mathcal{R} = (\Sigma, E, R)$ and its support equational theory $\mathcal{P} = (\Pi, D)$, in order to compute the corresponding LKS $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}$ in which LTLR formulas can be decided by model checking, the good executability properties of sort-decreasingness, and ground termination, coherence and confluence modulo B should also be satisfied by the extended equational theory $(\Sigma \cup \Pi, E \cup D)$, provided that $\mathcal{R}$ is executable and $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}$ has a finite set of reachable states from a given initial state $[t]_E$.

3.3 Automata Theoretic LTLR Model Checking

This section presents the automata-theoretic foundation for an LTLR model checking algorithm and its associated computational complexity. We make use of the standard automata theoretic LTL model checking approach to characterize the LTLR model checking problem.

3.3.1 Preliminaries on LTL Model Checking

Let us first recall the automata theoretic LTL model checking method. Given a Kripke structure $\mathcal{K} = (S, AP, \mathcal{L}, \rightarrow_{\mathcal{K}})$, the *trace* $\mathit{Traces}(\pi)$ of a path π is an infinite sequence of state labels such that $\mathit{Traces}(\pi)(i) = \mathcal{L}(\pi(i))$ for each $i \in \mathbb{N}$. The automata-based verification of an LTL formula φ then uses the Büchi automaton $\mathscr{B}_{\neg\varphi}$ associated to the negated formula $\neg\varphi$, and checks the emptiness of the synchronous product $\mathcal{K}[S_0] \times \mathscr{B}_{\neg\varphi}$ to determine whether $\mathscr{B}_{\neg\varphi}$ accepts any trace of $\mathcal{K}$ from a set of initial states $S_0 \subseteq S$ [32].

Definition 3.8. *A Büchi automaton is a 5-tuple $\mathscr{B} = (Q, Q_0, P, \delta, F)$, where Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, P is an alphabet of transition labels, $\delta \subseteq Q \times P \times Q$ is a labeled transition relation, and $F \subseteq Q$ is a set of accepting states.*

If $[\mathbb{N} \rightarrow P]$ denotes the set of all functions from $\mathbb{N}$ to P, the *language* accepted by $\mathscr{B}$ is the subset $L(\mathscr{B}) \subseteq [\mathbb{N} \rightarrow P]$ of infinite runs of $\mathscr{B}$ starting from an initial state $q_0 \in Q_0$ that visit accepting states in F infinitely often.

Definition 3.9. *Given a Kripke structure $\mathcal{K} = (S, AP, \mathcal{L}, \rightarrow_{\mathcal{K}})$, a set of initial states $S_0 \subseteq S$ of $\mathcal{K}$, and a Büchi automaton $\mathscr{B} = (Q, Q_0, 2^{AP}, \delta, F)$ with an alphabet 2^{AP}, the* synchronous product *of $\mathcal{K}$ and $\mathscr{B}$ is the Büchi automaton $\mathcal{K}[S_0] \times \mathscr{B} = (S \times Q,\ S_0 \times Q_0,\ 2^{AP},\ \delta_{\mathcal{K}},\ S \times F)$, where:*

$$(s, b) \xrightarrow{\mathcal{L}(s)} (s', b') \in \delta_{\mathcal{K}} \iff s \rightarrow_{\mathcal{K}} s' \ \wedge\ b \xrightarrow{\mathcal{L}(s)} b' \in \delta.$$

Such a product automaton $\mathcal{K}[S_0] \times \mathscr{B}$ accepts an infinite run of $\mathscr{B}$ that is also a trace of $\mathcal{K}$ starting from a state in S_0. The essence of LTL model checking is expressed by the following theorem:

Theorem 3.1 (LTL Model Checking [60])**.** *Given a Kripke structure $\mathcal{K}$ and an LTL formula φ, there is a Büchi automaton $\mathscr{B}_{\neg\varphi}$ with size $O(2^{|\varphi|})$ such that $\mathcal{K}, S_0 \models \varphi \iff L(\mathcal{K}[S_0] \times \mathscr{B}_{\neg\varphi}) = \emptyset$ for a set of initial states S_0.*

3.3.2 Automata-Based Verification of LTLR Formulas

We can convert the LTLR model checking problem for a rewrite theory $\mathcal{R}$ into an LTL model checking problem by theory transformations that encode rewriting events in states [20, 136]. In spite of the simplicity of this method, there is the problem of a blowup in the number of states: if $\mathcal{R}$ has n states and m transitions, the associated Kripke structure has $O(nm)$ states [136]. We can avoid such a state-space blowup by directly constructing an LKS.

This section shows that model checking an LTLR formula φ for a rewrite theory $\mathcal{R}$ is equivalent to deciding a language emptiness problem for a Büchi automaton, obtained from the Büchi automaton $\mathscr{B}_{\neg\varphi}$ for the negation of φ and the associated LKS $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}$. Our approach is closely related to the automata-theoretic solution for model checking SE-LTL properties proposed in [49], where SE-LTL is subsumed by LTLR [136]. The SE-LTL model checking algorithm in [49] assumes that each transition of an LKS is labeled by a *single* event, while our algorithm has been generalized to handle the case when a transition is labeled by a *set* of events (e.g., Example 3.2).

The model checking problem of an LTLR formula φ for a rewrite theory $\mathcal{R}$ can be reduced to the satisfiability of φ on its associated LKS $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}$. Given an LKS $\bar{\mathcal{K}} = (S, AP, \mathcal{L}, ACT, \rightarrow_{\bar{\mathcal{K}}})$, the *trace* of a path (π, α) is the pair of functions $(\mathit{Traces}(\pi), \alpha)$ such that for each $i \in \mathbb{N}$:[5]

$$\mathit{Traces}(\pi)(i) = \mathcal{L}(\pi(i)).$$

Basically, we need to determine whether an LTLR formula φ recognizes every trace of $\bar{\mathcal{K}}$ from a set of initial states $S_0 \subseteq S$.

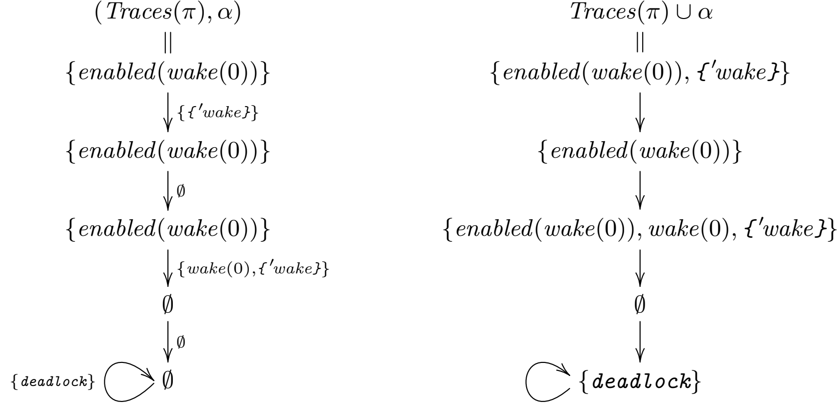
Definition 3.10. *A* union trace *of a path (π, α) in an LKS $\bar{\mathcal{K}}$ is a function $\mathit{Traces}(\pi) \cup \alpha : \mathbb{N} \to AP \cup ACT$ such that for each $i \geq 0$:*

$$(\mathit{Traces}(\pi) \cup \alpha)(i) = \mathit{Traces}(\pi)(i) \cup \alpha(i).$$

If AP and ACT are disjoint, then there exists a one-to-one correspondence between a trace $(\mathit{Traces}(\pi), \alpha)$ and its union trace $\mathit{Traces}(\pi) \cup \alpha$. In such union traces there is no difference between events and state propositions. Hence, we can check if a union trace $\mathit{Traces}(\pi) \cup \alpha$ is accepted by a Büchi automaton $\mathscr{B}_{\neg\varphi}$ for the negated formula $\neg\varphi$, using the same Büchi automata construction as in the LTL case, where the alphabet of $\mathscr{B}_{\neg\varphi}$ is the power set $2^{AP \uplus ACT}$ of the disjoint union $AP \uplus ACT$.

[5] Notice that $\mathit{Traces}(\pi)$ is a trace of a path π in a usual Kripke structure.

Example 3.3. *The following are the trace and its union trace of the path* (π, α) *in Example 3.2 for the dining philosophers example:*

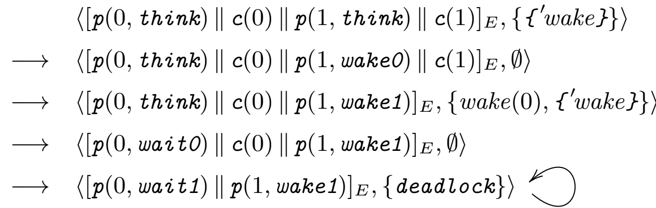


In fact, union traces of an LKS $\bar{\mathcal{K}}$ induce an equivalent Kripke structure $\mathcal{D}(\bar{\mathcal{K}})$, whose states are pairs of a state and a transition label.

Definition 3.11. *Given an LKS* $\bar{\mathcal{K}} = (S, AP, \mathcal{L}, ACT, \rightarrow_{\bar{\mathcal{K}}})$*, its associated Kripke structure is* $\mathcal{D}(\bar{\mathcal{K}}) = (\mathcal{D}(S),\ AP \cup ACT,\ \mathcal{D}(\mathcal{L}),\ \rightarrow_{\mathcal{D}(\bar{\mathcal{K}})})$*, where:*

- $\mathcal{D}(S) = \{\langle s, \Lambda\rangle \in S \times 2^{ACT} \mid \exists s' \in S.\ s \xrightarrow{\Lambda}_{\bar{\mathcal{K}}} s'\}$
- $\mathcal{D}(\mathcal{L})(\langle s, \Lambda\rangle) = \mathcal{L}(s) \cup \Lambda$
- $\langle s, \Lambda\rangle \rightarrow_{\mathcal{D}(\bar{\mathcal{K}})} \langle s', \Lambda'\rangle$ *iff* $s \xrightarrow{\Lambda}_{\bar{\mathcal{K}}} s'$ *and* $\langle s', \Lambda'\rangle \in \mathcal{D}(S)$*.*

Example 3.4. *The path* (π, α) *in Example 3.2 for the dining philosophers example corresponds to the following path* $\pi \cup \alpha$ *in* $\mathcal{D}(\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}})$*:*



where

$$\mathcal{D}(\mathcal{L}_{\mathcal{P}})((\pi \cup \alpha)(0)) = \{enabled(wake(0)),\ \{'wake\}\},$$
$$\mathcal{D}(\mathcal{L}_{\mathcal{P}})((\pi \cup \alpha)(1)) = \{enabled(wake(0))\},$$
$$\mathcal{D}(\mathcal{L}_{\mathcal{P}})((\pi \cup \alpha)(2)) = \{enabled(wake(0)), wake(0),\ \{'wake\}\},$$
$$\mathcal{D}(\mathcal{L}_{\mathcal{P}})((\pi \cup \alpha)(3)) = \emptyset, \quad \text{and} \quad \mathcal{D}(\mathcal{L}_{\mathcal{P}})((\pi \cup \alpha)(i)) = \{\mathtt{deadlock}\}, \quad \text{for } i \geq 4.$$

Notice that the union trace $Traces(\pi) \cup \alpha$ *in* $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}$ *is identical to the corresponding trace* $Traces(\pi \cup \alpha)$ *in* $\mathcal{D}(\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}})$*.*

It is clear that each trace of $\mathcal{D}(\bar{\mathcal{K}})$ is a union trace of $\bar{\mathcal{K}}$. There exists a one-to-one correspondence between a path (π, α) of $\bar{\mathcal{K}}$ and a path $\pi \cup \alpha$ of $\mathcal{D}(\bar{\mathcal{K}})$, where $(\pi \cup \alpha)(i) = \langle \pi(i), \alpha(i) \rangle$ for each $i \in \mathbb{N}$. Further, $\bar{\mathcal{K}}$ and $\mathcal{D}(\bar{\mathcal{K}})$ are equivalent in the sense of the satisfiability of a formula as follows.

Lemma 3.1. *Given an LKS $\bar{\mathcal{K}} = (S, AP, \mathcal{L}, ACT, \rightarrow_{\bar{\mathcal{K}}})$, an LTLR formula φ over AP and ACT, and a set of initial states $S_0 \subseteq S$, if any spatial action pattern in φ is regarded as a state proposition of $\mathcal{D}(\bar{\mathcal{K}})$, then:*

$$\bar{\mathcal{K}}, S_0 \models \varphi \iff \mathcal{D}(\bar{\mathcal{K}}), \mathcal{D}(S)\restriction_{S_0} \models \varphi,$$

where $\mathcal{D}(S)\restriction_{S_0} = (S_0 \times 2^{ACT}) \cap \mathcal{D}(S)$.

Proof. It suffices to show that $\bar{\mathcal{K}}, (\pi, \alpha) \models \varphi$ iff $\mathcal{D}(\bar{\mathcal{K}}), \pi \cup \alpha \models \varphi$ for each (π, α) of $\bar{\mathcal{K}}$. We can prove this by structural induction on φ. If φ is a spatial action pattern δ, then $\bar{\mathcal{K}}, (\pi, \alpha) \models \delta$ iff $\delta \in \alpha(0)$. Equivalently, $\delta \in \mathcal{L}(\pi(0)) \cup \alpha(0) = \mathcal{D}(\mathcal{L})(\langle \pi(0), \alpha(0) \rangle)$, since $AP \cap ACT = \emptyset$. Thus, $\mathcal{D}(\bar{\mathcal{K}}), \pi \cup \alpha \models \delta$. Similarly, if φ is a state proposition p, then $\bar{\mathcal{K}}, (\pi, \alpha) \models p$ iff $p \in \mathcal{L}(\pi(0))$ iff $p \in \mathcal{L}(\pi(0)) \cup \alpha(0) = \mathcal{D}(\mathcal{L})(\langle \pi(0), \alpha(0) \rangle)$ iff $\mathcal{D}(\bar{\mathcal{K}}), (\pi, \alpha) \models p$, since $AP \cap ACT = \emptyset$. The other cases follow easily from the induction hypothesis. For example, $\bar{\mathcal{K}}, (\pi, \alpha) \models \bigcirc\varphi$ iff $\bar{\mathcal{K}}, (\pi, \alpha)^1 \models \varphi$ by definition, $\bar{\mathcal{K}}, (\pi, \alpha)^1 \models \varphi$ iff $\mathcal{D}(\bar{\mathcal{K}}), (\pi \cup \alpha)^1 \models \varphi$ by induction hypothesis, and finally, $\mathcal{D}(\bar{\mathcal{K}}), (\pi \cup \alpha)^1 \models \varphi$ iff $\mathcal{D}(\bar{\mathcal{K}}), \pi \cup \alpha \models \bigcirc\varphi$ by definition. □

In order to determine whether a Büchi automaton $\mathscr{B}_{\neg\varphi}$ accepts a union trace of an LKS $\bar{\mathcal{K}}$ from a set of initial states S_0 of $\bar{\mathcal{K}}$, we avoid the use of $\mathcal{D}(\bar{\mathcal{K}})[\mathcal{D}(S)\restriction_{S_0}] \times \mathscr{B}_{\neg\varphi}$ directly, since it can produce a state-space blowup.[6] Instead, we define a special state/event synchronous product $\bar{\mathcal{K}}[S_0] \otimes \mathscr{B}_{\neg\varphi}$, which advances to the next state only if both state labels and event labels are accepted by the current transition of $\mathscr{B}_{\neg\varphi}$.

Definition 3.12. *Given an LKS $\bar{\mathcal{K}} = (S, AP, \mathcal{L}, ACT, \rightarrow_{\bar{\mathcal{K}}})$, a set of initial states $S_0 \subseteq S$, and a Büchi automaton $\mathscr{B} = (Q, Q_0, 2^{AP \cup ACT}, \delta, F)$, the* state/event product *of $\bar{\mathcal{K}}$ and $\mathscr{B}$ is the Büchi automaton*

$$\bar{\mathcal{K}}[S_0] \otimes \mathscr{B} = (S \times Q, S_0 \times Q_0, 2^{AP \cup ACT}, \delta_{\bar{\mathcal{K}}}, S \times F)$$

such that $(s, b) \xrightarrow{\mathcal{L}(s) \cup \Lambda} (s', b') \in \delta_{\bar{\mathcal{K}}}$ iff $s \xrightarrow{\Lambda}_{\bar{\mathcal{K}}} s'$ and $b \xrightarrow{\mathcal{L}(s) \cup \Lambda} b' \in \delta$.

The following lemma shows that the language emptiness problem for the state/event product $\bar{\mathcal{K}} \otimes \mathscr{B}$ is equivalent to that for $\mathcal{D}(\bar{\mathcal{K}}) \times \mathscr{B}$.

---

[6] If an LKS $\bar{\mathcal{K}}$ has n states and m transitions, its associated Kripke structure $\mathcal{D}(\bar{\mathcal{K}})$ has $O(n + m)$ states and m transitions.

Lemma 3.2. *Given an LKS $\bar{\mathcal{K}} = (S, AP, \mathcal{L}, ACT, \rightarrow_{\bar{\mathcal{K}}})$, a set of initial states $S_0 \subseteq S$, and a Büchi automaton $\mathscr{B} = (Q, Q_0, 2^{AP \cup ACT}, \delta, F)$, we have $L(\bar{\mathcal{K}}[S_0] \otimes \mathscr{B}) = \emptyset \iff L(\mathcal{D}(\bar{\mathcal{K}})[\mathcal{D}(S)\restriction_{S_0}] \times \mathscr{B}) = \emptyset$.*

Proof. Let $\rho = (\mathcal{L}(s_0), b_0)(\mathcal{L}(s_1), b_1)(\mathcal{L}(s_2), b_2)\ldots$ be a run of the state/event product $\bar{\mathcal{K}}[S_0] \otimes \mathscr{B}$. By definition:

- $\mathcal{L}(s_0)\mathcal{L}(s_1)\mathcal{L}(s_2)\ldots$ is a trace of $\bar{\mathcal{K}}$,
- $b_0 b_1 b_2 \ldots \in L(\mathscr{B})$, and
- $s_i \xrightarrow{\Lambda_i}_{\bar{\mathcal{K}}} s_{i+1}$ and $b_i \xrightarrow{\mathcal{L}(s_i) \cup \Lambda_i} b_{i+1}$ for each $i \geq 0$.

Clearly, the union trace $(\mathcal{L}(s_0) \cup \Lambda_0)(\mathcal{L}(s_1) \cup \Lambda_1)(\mathcal{L}(s_2) \cup \Lambda_2)\ldots$ becomes a trace of $\mathcal{D}(\bar{\mathcal{K}})$ by definition, where $\langle s_0, \Lambda_0 \rangle \in \mathcal{D}(S)\restriction_{S_0}$, and therefore $(\mathcal{L}(s_0) \cup \Lambda_0, b_0)(\mathcal{L}(s_1) \cup \Lambda_1, b_1)(\mathcal{L}(s_2) \cup \Lambda_2, b_2)\ldots \in L(\mathcal{D}(\bar{\mathcal{K}})[\mathcal{D}(S)\restriction_{S_0}] \times \mathscr{B})$. Conversely, let $\hat{\rho} = (\mathcal{L}(s_0) \cup \Lambda_0, b_0)(\mathcal{L}(s_1) \cup \Lambda_1, b_1)(\mathcal{L}(s_2) \cup \Lambda_2, b_2)\ldots$ be a run of $\mathcal{D}(\bar{\mathcal{K}})[\mathcal{D}(S)\restriction_{S_0}] \times \mathscr{B}$. By definition:

- $(\mathcal{L}(s_0) \cup \Lambda_0)(\mathcal{L}(s_1) \cup \Lambda_1)(\mathcal{L}(s_2) \cup \Lambda_2)\ldots$ is a trace of $\mathcal{D}(\bar{\mathcal{K}})$,
- $b_0 b_1 b_2 \ldots \in L(\mathscr{B})$, and
- $(s_i, \Lambda_i) \rightarrow_{\mathcal{D}(\bar{\mathcal{K}})} (s_{i+1}, \Lambda_{i+1})$ and $b_i \xrightarrow{\mathcal{L}(s_i) \cup \Lambda_i} b_{i+1}$ for each $i \geq 0$.

Notice that $s_i \xrightarrow{\Lambda_i}_{\bar{\mathcal{K}}} s_{i+1}$ for each $i \geq 0$ by construction of $\mathcal{D}(\bar{\mathcal{K}})$. Hence, $(\mathcal{L}(s_0), b_0)(\mathcal{L}(s_1), b_1)(\mathcal{L}(s_2), b_2)\ldots \in L(\bar{\mathcal{K}}[S_0] \otimes \mathscr{B})$ by definition. □

As a consequence, for an LTLR formula φ, an LKS $\bar{\mathcal{K}}$, and a set of initial states S_0 of $\bar{\mathcal{K}}$, if $\mathscr{B}_{\neg\varphi}$ is a Büchi automaton for $\neg\varphi$ constructed in exactly the same way as in the LTL case, then $\bar{\mathcal{K}}, S_0 \models \varphi$ iff $L(\bar{\mathcal{K}}[S_0] \otimes \mathscr{B}_{\neg\varphi}) = \emptyset$. Therefore, by the LKS construction associated to a rewrite theory $\mathcal{R}$:

Theorem 3.2. *Given an LTLR formula φ, a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, a support equational theory $\mathcal{P}$, a state kind $k \in \Sigma$, and an initial state $[t]_E$:*

$$\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}, \{[t]_E\} \models \varphi \quad \iff \quad L(\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}[\{[t]_E\}] \otimes \mathscr{B}_{\neg\varphi}) = \emptyset.$$

The cost of model checking an LTLR formula φ on a labeled Kripke structure $\bar{\mathcal{K}}$ is $O((n+m) \cdot 2^{|\varphi|})$, where $\bar{\mathcal{K}}$ has n states and m transitions. The m factor here is added since each transition needs to test the spatial action pattern in φ, where $\bar{\mathcal{K}}$ may have several transitions with different labels between two states. If there are no spatial action patterns in the formula φ, then the cost is $O(n \cdot 2^{|\varphi|})$, exactly the same as for LTL model checking.

3.4 The Maude LTLR Model Checker

This section explains the design and user interface of the Maude LTLR model checker, illustrated with the Maude specification in Example 2.10 for the dining philosophers problem. Our tool provides an extensible way to define *generic* spatial action patterns, besides the patterns in $SP(\mathcal{R})$. The tool is available at `http://maude.cs.illinois.edu/tools/tlr`.

3.4.1 Support Equational Theories

The Maude LTLR model checker extends the existing Maude LTL model checker that declares a support equational theory for LTL model checking in the predefined functional module `SATISFACTION`:

```
fmod SATISFACTION is
  protecting BOOL .            sorts State Prop .
  op _|=_ : State Prop -> Bool .
endfm
```

In the Maude LTLR model checker, the action satisfaction operator $\models$ is also declared in the predefined functional module `ACTION-SATISFACTION`, which defines sort Action for spatial action patterns:

```
fmod ACTION-SATISFACTION is
  including PROOF-TERM .       including SATISFACTION .
  sort Action .                op _|=_ : ProofTerm Action -> Bool .
  eq P:ProofTerm |= P:ProofTerm = true .
endfm
```

The triple representation $\{t[\square]_p \mid {'l} : \theta\}$ of a one-step proof term $t[l(\theta)]_p$ is declared in the predefined functional module `PROOF-TERM`. A quoted rule label $'l$ is a constant of sort RuleName, which includes sort Qid for any quoted identifier constants of the form $'id$. The constant `deadlock` of sort ProofTerm denotes the deadlock event:[7]

```
fmod PROOF-TERM is
  protecting QID .                      including STATE-CONTEXT .
  including STATE-SUBSTITUTION .
  sorts ProofTerm RuleName .            subsort Qid < RuleName .
  op unlabeled : -> RuleName [ctor] .
  op deadlock : -> ProofTerm [ctor] .
  op {_|_:_} : StateContext RuleName StateSubstitution -> ProofTerm
                                                               [ctor] .
endfm
```

[7]Recall that $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}$ adds a self loop $[t]_E \xrightarrow{\{\text{deadlock}\}}_{\bar{\mathcal{K}}} [t]_E$ for a deadlock state $[t]_E$.

Each spatial action pattern is then defined by equations using the above constructs. For example, the syntax and semantics of the spatial action patterns in $SP(\mathcal{R})$ are both specified in the predefined functional module `SPATIAL-ACTION-PATTERN`, where sort ActionPattern denotes action patterns in $SP(\mathcal{R})$ and sort BasicActionPattern denotes basic action patterns:

```
fmod SPATIAL-ACTION-PATTERN is
  including ACTION-SATISFACTION .
  sorts BasicActionPattern ActionPattern .
  subsorts BasicActionPattern ProofTerm < ActionPattern < Action .
  var CXT : StateContext .          var R : RuleName .
  var SUB SUB' : StateSubstitution .

  op {_}       : RuleName -> BasicActionPattern .
  op {_:_}     : RuleName StateSubstitution -> BasicActionPattern .
  op {_|_}     : StateContext RuleName -> ActionPattern .
  op top{_}    : RuleName -> ActionPattern .
  op top{_:_}  : RuleName StateSubstitution -> ActionPattern .

  eq {CXT | R : SUB}         |= {R}               = true .
  eq {CXT | R : SUB ; SUB'}  |= {R : SUB}         = true .
  eq {CXT | R : SUB}         |= {CXT | R}         = true .
  eq {CXT | R : SUB ; SUB'}  |= {CXT | R : SUB}   = true .
  eq {[]  | R : SUB}         |= top{R}            = true .
  eq {[]  | R : SUB ; SUB'}  |= top{R : SUB}      = true .
endfm
```

A substitution is represented as a semicolon-separated set of assignments of the form $'x\backslash u$. Since variables in rewrite rules can have any sort, an assignment operator `__` : Qid $s \to$ StateAssignment should be defined for any variable of sort s in the left-sides of the rules. For that purpose, the operator `__` is declared as *polymorphic* in its second argument by using the attribute `poly(2)` and the predefined sort Universal, so that the instance operator of each sort s in $\mathcal{R}$ is automatically declared by Maude:

```
fmod STATE-SUBSTITUTION is
  protecting QID .        sorts StateSubstitution StateAssignment .
  subsort StateAssignment < StateSubstitution .
  op _\_ : Qid Universal -> StateAssignment [ctor poly(2)] .
  op none : -> StateSubstitution [ctor] .
  op _;_ : StateSubstitution StateSubstitution -> StateSubstitution
                                    [ctor comm assoc id: none] .
  eq A:StateAssignment ; A:StateAssignment = A:StateAssignment .
endfm
```

Similarly, the hole symbol $\Box$ in context terms can have any of the sorts of left-sides of the rules, because a rewrite can happen at any position in a state term. However, unlike assignment operators, in this case we *cannot* use the polymorphic operator declaration

```
op [] : -> Universal [ctor poly(0)] .
```

since then the extended theory will *not* protect the original model. Instead, a context term of the form $t[\Box]_p$ is by default *partially* declared as follows, where the constant `[]` denotes the *hole* symbol for sort State:

```
fmod STATE-CONTEXT is
  including SATISFACTION .          sort StateContext .
  op [] : -> StateContext [ctor] .
  op noContext : -> StateContext [ctor] .
endfm
```

For a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, a full signature of context terms that *protects* (Σ, E) can be automatically generated by a theory transformation. Let $\Omega \subseteq \Sigma$ be the subsignature of *constructors* in which every ground term in canonical form is an Ω-term. For each rule $(l : q \to r \textbf{ if } cond) \in R$, the signature $Context[\mathcal{R}]$ extends Ω by adding incrementally:

- a hole constant $\Box$ with a new sort $\mathsf{Context\$}s$, where q has sort s;
- for each operator $f : b_1 \ldots b_m \to b$ in Ω, a set of operators:

$$\begin{aligned} f &: \mathsf{Context\$}b_1\ b_2\ b_3\ \ldots\ b_m \to \mathsf{Context\$}b \\ f &: b_1\ \mathsf{Context\$}b_2\ b_3\ \ldots\ b_m \to \mathsf{Context\$}b \\ f &: b_1\ b_2\ \mathsf{Context\$}b_3\ \ldots\ b_m \to \mathsf{Context\$}b \\ &\ldots \\ f &: b_1\ b_2\ b_3\ \ldots\ \mathsf{Context\$}b_m \to \mathsf{Context\$}b \end{aligned}$$

  where $\mathsf{Context\$}b_1, \ldots, \mathsf{Context\$}b_m$ and $\mathsf{Context\$}b$ are new sorts related to each sort in the operator declaration (these operators guarantee that each context term should contain only one hole symbol); and

- a subsort relation $\mathsf{Context\$}s_1 < \mathsf{Context\$}s_2$ if s_1 is a subsort of s_2, and $\mathsf{Context\$}s < \mathsf{StateContext}$ if s is a subsort of `State`.

The signature $Context[\mathcal{R}]$ defines *new* sorts of context terms for all operators in Ω. For example, if a term u has sort $s \in \Sigma$, a context term $u[\Box]_p$ has sort $\mathsf{Context\$}s$. Note that $Context[\mathcal{R}]$ protects $\mathcal{R}$, since any new constants and operators are introduced with new sorts that are not included in $\mathcal{R}$.

3.4.2 The Model Checker Interface

The main functionality of the Maude LTLR model checker is defined in the predefined functional module `LTLR-MODEL-CHECKER`. An LTLR formula has sort Formula, and is constructed by state propositions of sort Prop, spatial action patterns of sort Action, and temporal logic operators such as `~` (negation), `/\`, `\/`, `->` (implication), `[]` ("always"), `<>` ("eventually"), `U` ("until"), and `O` ("next"). For LTLR model checking the function

```
modelCheck :  State Formula ~> ModelCheckResult
```

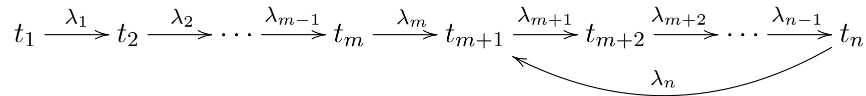
takes an initial state and an LTLR formula, and returns either `true`—if the formula is satisfied—or a counterexample, provided that the number of reachable states from the initial state is finite, where:

```
sort ModelCheckResult .
subsort Bool < ModelCheckResult .
op counterexample : TransList TransList -> ModelCheckResult .
```

A counterexample for an LTLR formula is an infinite path consisting of two transition lists, where the first one is a finite prefix and the second one is a cycle that gives the rest of the infinite path:

```
sorts Transition TransList .
subsort Transition < TransList .
op {_,_} : State ProofTerm -> Transition [ctor] .
op nil : -> TransList [ctor] .
op __ : TransList TransList -> TransList [ctor assoc id: nil] .
```

Each transition is a pair $\{t, \lambda\}$, representing a one-step rewrite from the state t to a next state with the one-step proof term λ. For example, the term `counterexample`$(\{t_1, \lambda_1\}\, \{t_2, \lambda_2\} \cdots \{t_m, \lambda_m\},\ \ \{t_{m+1}, \lambda_{m+1}\} \cdots \{t_n, \lambda_n\})$, where $m < n$, represents the infinite path:



Finally, the module expression `CONTEXT[M]` for a system module `M`—only available in the Full-Maude interface—generates the declarations for context terms using the theory transformation $\mathit{Context}[\mathcal{R}]$. Such context terms are essential for some spatial action patterns, such as top`{`$'l$`}` in $\mathit{SP}(\mathcal{R})$. If a signature for context terms is not given, the constant `noContext` will be *internally* used, instead of actual context terms, to indicate that context terms could not be created during model checking.

For the dining philosophers model in Example 2.10, we can declare the state proposition *eating*(i)—meaning that the philosopher i is eating—in the following function module, which also includes the predefined module SPATIAL-ACTION-PATTERN for spatial action patterns in $SP(\mathcal{R})$:

```
mod DINING-PHILOS-PROP is
  protecting DINING-PHILOS .
  including LTLR-MODEL-CHECKER .
  including SPATIAL-ACTION-PATTERN .
  subsort Conf < State .     var CF : Conf .     vars I : Nat .

  op eating : Nat -> Prop [ctor] .
  eq p(I, eat) || CF |= eating(I) = true .

  op init : -> State .        *** the initial state
  eq init = p(0,think) || c(0) || p(1,think) || c(1) .
endm
```

By model checking the formula $(\Box\neg\texttt{deadlock}) \to \Diamond \mathit{eating}(0)$, we can find a counterexample in which only the philosopher 1 performs actions:[8]

```
Maude> red modelCheck(init, []~ deadlock -> <> eating(0)) .
result ModelCheckResult: counterexample(
{c(0) || c(1) || p(0,think) || p(1,think), {'wake : 'I\0}},
{c(0) || c(1) || p(0,wait0) || p(1,think), {'wake : 'I\1}}
{c(0) || c(1) || p(0,wait0) || p(1,wait0), {'grabF : 'I\1 ; 'J\0}}
{c(1) || p(0,wait0) || p(1,wait1), {'grabS : 'I\1 ; 'J\1}}
{p(0,wait0) || p(1,eat), {'stop : 'I\1}})
```

To avoid such unrealistic situation, we need suitable fairness assumptions. For the dinning philosophers problem with 2 philosophers, we need the two fairness assumptions in the following model checking command:

```
Maude> red modelCheck(init,
   ((<>[] enabled({'wake : 'I\0}) -> []<> {'wake : 'I\0}) /\
    ([]<> enabled({'grabF : 'I\0}) -> []<> {'grabF : 'I\0}))
    -> ([]~ deadlock -> <> eating(0))) .
result Bool: true
```

In our tool the state proposition *enabled*(δ) for a spatial action pattern δ is automatically declared (see Section 4.2). Notice that the deadlock freedom property $\Box\neg$**deadlock** is also necessary to prove *eating*(0), since there exists a deadlock state in this system.

[8] Context terms are not displayed in the counterexample since no signature for context terms was provided. Such a signature can be provided by using the module expression CONTEXT[DINING-PHILOS] in the Full Maude interface (e.g., see Section A.1.4).

3.5 Case Study: the Bounded Retransmission Protocol

This section shows how complex requirements of a concurrent system can be naturally expressed in LTLR. The bounded retransmission protocol is an extension of the alternating bit protocol where a limit is placed on the number of message transmissions [1]. Descriptions of this protocol such as those given in [139] are quite complex due to the use of a state-based logic, which forces the specification to *encode* action information in the state. With LTLR this encoding completely disappears, leading to a much simpler protocol specification. For example, a previous state-based rewriting logic specification had 35 rewrite rules, but we specify here the same model with only 14 rules and with an easier to understand state representation.

The bounded retransmission protocol is described as follows. At the sender side the protocol requests a sequence of data elements $d_1, \ldots, d_n$ (action REQ) and communicates one of the confirmations:

- SOK: the file has been transferred successfully;
- SNOK: the file has not been transferred completely; and
- SDNK: the file may not have been transferred completely.

At the receiver side the protocol marks each correctly received datum with one of the indications

- RFST: the delivered datum is the first one;
- RINC: the datum is an intermediate one;
- ROK: the datum is the last one and the file is completed; and
- RNOK: the connection with the sender is broken.

The specification in [139] is adapted from the *untimed* model in [1], and our specification substantially simplifies it. The configuration is a tuple

$$\langle \mathit{Status}_S, \mathit{Flag}_S, \mathit{Channel}_S, \mathit{Channel}_R, \mathit{Flag}_R, \mathit{Status}_R \rangle,$$

where Status_S is the status of the sender, Status_R is the status of the receiver, Flag_S and Flag_R are Boolean values used by the sender and the receiver for synchronization purposes, and $\mathit{Channel}_S$ and $\mathit{Channel}_R$ are the two ordered lossy channels through which the sender and the receiver communicate. Each message is one of 0, 1, *first*, *last*, where *first* denotes the first datum and *last* the last datum. Messages typically contain both data and sequence bits, but such message data is abstracted away in this specification.

```
fmod BRP-SYNTAX is
  protecting BOOL .
  sorts Conf Sender Receiver Msg MsgL .     subsort Msg < MsgL .
  op <_,_,_,_,_,_> : Sender Bool MsgL MsgL Bool Receiver
                   -> Conf [ctor] .
  op idle : -> Sender [ctor] .              --- sender's status
  ops set snd acc : Msg -> Sender [ctor] .
  op wait : -> Receiver [ctor] .            --- receiver's status
  op rec : Msg -> Receiver [ctor] .
  ops 0 1 first last : -> Msg [ctor] .      --- messages
  op nil : -> MsgL [ctor] .
  op _;_ : MsgL MsgL -> MsgL [ctor assoc id: nil] .
endfm
```

The sender's status is one of `idle`, `snd(`α`)` and `acc(`α`)`, where `snd(`α`)` means that the sender is sending a message α, and `acc(`α`)` indicates that the sender gets an acknowledgement of α. The receiver can have status `wait` or `rec(`α`)`, where `rec(`α`)` denotes that the receiver gets a message α.

The behavior of the protocol is specified by rewrite rules as follows. The client-side behavior is specified by the four rules *req*, *snd*, *acc*, and *los*. In the *req* rule, the auxiliary status `set(`α`)` denotes that the sender is about to send a message α, and it is equationally reduced to the state with status `snd(`α`)` with one α sent. In the *acc* rule, the sender's status is changed to `acc(`α`)` only if the message α has arrived in the receiver's channel.

```
 rl [req]: < idle, A, nil, nil, false, R >
       =>  < set(first), false, nil, nil, false, R > .
 rl [snd]: < snd(M), A, K, L, T, R >
       =>  < snd(M), A, K ; M, L, T, R > .
crl [acc]: < snd(M), A, K, M' ; L, T, R >
       =>  < S, A, K, L, T, R >
   if S := (if M == M' then acc(M) else snd(M) fi) .
crl [los]: < snd(M), A, K, nil, T, R >
       =>  < idle, true, K, nil, T, R >    if M =/= first .
 eq < set(M), A, K, L, T, R > = < snd(M), A, K ; M, L, T, R > .
```

The following rules labeled with *sel* describe the nondeterministic choice of a next message. The transfer of data is finished when the sender accepts the message `last` and the sender's status is initialized to `idle`.

```
rl [sel]: acc(first) => set(0) .
rl [sel]: acc(first) => set(last) . rl [sel]: acc(last)  => idle .
rl [sel]: acc(0) => set(1) .        rl [sel]: acc(0) => set(last) .
rl [sel]: acc(1) => set(0) .        rl [sel]: acc(1) => set(last) .
```

The server-side behavior is specified by the three rules *rec*, *ign*, and *nil*. In the *rec* rule, when a received datum is `first`, the server flag is set to true.

```
crl [rec]: < S, false, M ; K, L, T, R >
        =>  < S, false, K, L ; M, B, rec(M) >
   if R =/= rec(M)  /\    B := (if M == first then true else T fi) .
 rl [ign]: < S, A, M ; K, L, T, rec(M) >
        =>  < S, A, K, L ; M, T, rec(M) > .
crl [nil]: < S, A, nil, L, T, rec(M) >
        =>  < S, A, nil, L, false, wait >
   if M == last or A == true .
```

This protocol has an infinite number of states due to unbounded channels. However, we can define the finite-state equational abstraction [139] (see Section 5.3) by adding extra equations, and by adding extra rewrite rules to keep the system coherent, where adjacent duplicate messages in the channels are merged into a single message:

```
 eq < S, A, KL ; M ; M ; K, L, T, R >
  = < S, A, KL ; M ; K,     L, T, R > .
 eq < S, A, K, KL ; M ; M ; L, T, R >
  = < S, A, K, KL ; M ; L,     T, R > .

crl [acc]: < snd(M), A, K, M' ; L, T, R >
        =>  < S, A, K, M' ; L, T, R >
   if S := if M == M' then acc(M) else snd(M) fi .
crl [rec]: < S, false, M ; K, L, T, R >
        =>  < S, false, M ; K, L ; M, B, rec(M) >
   if R =/= rec(M)  /\    B := (if M == first then true else T fi) .
 rl [ign]: < S,  A, M ; K, L, T, rec(M) >
        =>  < S,  A, M ; K, L ; M, T, rec(M) > .
```

The bounded retransmission protocol should satisfy the following system requirements expressed in LTLR:

1. $\Box(\mathit{req} \rightarrow \bigcirc(\neg \mathit{req}\ \mathbf{W}\ (\mathit{sok} \vee \mathit{snok} \vee \mathit{sdnk})))$: a request Req must be followed by a confirmation before the next request;

2. $\Box(\mathit{rfst} \rightarrow (\neg \mathit{req}\ \mathbf{W}\ (\mathit{rok} \vee \mathit{rnok})))$: an Rfst indication must be followed by one of the two indications Rok or Rnok before the beginning of a new transmission (new request of a sender);

3. $\Box(\mathit{req} \rightarrow (\neg \mathit{sok}\ \mathbf{W}\ \mathit{rok}))$: an Sok confirmation must be preceded by an Rok indication; and

4. $\Box(\mathit{req} \rightarrow (\neg \mathit{rnok}\ \mathbf{W}\ (\mathit{snok} \vee \mathit{sdnk})))$: an Rnok indication must be preceded by an Snok or Sdnk confirmation (abortion).

Events occurring in these formulas can be defined by (conditional) equations in the following module BRP-CHECK:

```
mod BRP-CHECK is
  protecting BRP-ABS .        including LTLR-MODEL-CHECKER .
  subsort Conf < State .      var M : Msg .
  var CXT : StateContext .    var SUB : StateSubstitution .
  ops req sok snok sdnk rfst rinc rok rnok : -> Action .
  eq {CXT | 'req : SUB}              |= req = true .
  eq {CXT | 'acc : 'M \ last ;
                   'M' \ last ; SUB} |= sok = true .
 ceq {CXT | 'los : 'M \ M ; SUB}     |= snok = true  if M =/= last .
  eq {CXT | 'los : 'M \ last ; SUB}  |= sdnk = true .
  eq {CXT | 'rec : 'M \ first ; SUB} |= rfst = true .
 ceq {CXT | 'rec : 'M \ M ; SUB}     |= rinc = true
    if M == 0 or M == 1 .
  eq {CXT | 'rec : 'M \ last ; SUB}  |= rok  = true .
 ceq {CXT | 'nil : 'M \ M ; SUB}     |= rnok = true  if M =/= last .
endm
```

The following model checking command verifies the four LTLR properties (122 system states and 328 transitions explored by this commend, while the previous implementation [20] generates 283 states and 1034 transitions):

```
Maude> red modelCheck(< idle, false, nil, nil, false, wait >,
                       ([](req -> O(~ req W(sok \/ snok \/ sdnk))))
                    /\ ([](rfst -> (~ req W(rok \/ rnok))))
                    /\ ([](req -> (~ sok W rok)))
                    /\ ([](req -> (~ rnok W(snok \/ sdnk))))) .
result Bool : true
```

3.6 Concluding Remarks

This chapter has shown that rewriting logic and LTLR are useful together as a tandem of logics with tool support, equipped with an efficient model checking algorithm, a suitable specification language, and an intuitive user interface. After explaining the syntax and semantics of LTLR, we have presented the automata-theoretic foundation of the Maude LTLR model checker, and explained its support of language extensions for spatial action patterns. The Maude LTLR model checker has been implemented at the C++ level as an extension of the Maude system. As illustrated with several case studies, the user interface of the Maude LTLR model checker provides a convenient and succinct way of specifying spatial action patterns.

CHAPTER 4

MODEL CHECKING UNDER LOCALIZED FAIRNESS

This chapter[1] presents a model checking algorithm to verify LTLR properties under parameterized fairness assumptions. Fairness is an essential property in model checking verification, but often the needed fairness assumptions cannot be expressed as *propositional* temporal logic formulas because they are *parametric*, that is, they correspond to *universally quantified* temporal logic formulas. Such universal quantification is succinctly captured by the notion of *localized fairness*; for example, fairness is localized to the object name parameter in object fairness conditions. We have implemented our algorithm within the Maude Fair LTLR model checker, the first tool we are aware of that can deal with parameterized fairness assumptions.

4.1 Introduction

Fairness is an essential property in model checking, because many system requirements do not hold unless appropriate fairness assumptions about the system behavior are provided. LTLR is particularly relevant to fairness, because fairness is a *mixed* property involving both state propositions and events. Fairness cannot be directly specified in either purely state-based or event-based logics, but it can be expressed in a simple and general way in LTLR. For example, in its simplest form, strong fairness declares that some system transitions infinitely often *enabled* are infinitely often *taken*, and weak fairness says that some transitions continuously *enabled* after a certain point are infinitely often *taken*. Enabledness is a state-based property, but the taking of a transition is a paradigmatic example of an event.

[1]This chapter is based on the papers [18, 19, 22], joint work with José Meseguer.

To verify an LTLR property φ under fairness assumptions $\psi_1, \dots, \psi_n$, we can model check the implication $(\psi_1 \wedge \cdots \wedge \psi_n) \rightarrow \varphi$, *but not efficiently.* The problem is that LTLR model checking is based on associating a Büchi automaton to the negation of the given formula (see Section 3.3.2); however, constructing its associated Büchi automaton incurs an exponential blowup which may be very large because of the fairness assumptions $\psi_1, \dots, \psi_n$ in the formula $(\psi_1 \wedge \cdots \wedge \psi_n) \rightarrow \varphi$ [165]. For this reason, various model checkers, such as PAT [158] and Maria [122], *build in* fairness assumptions into their model checking algorithms.

However, what we often need in practice is not the fairness of *transitions*, but a form of *parameterized* fairness in which fairness is *localized* to a possibly infinite family of transition *instances*. A good example is that of *object fairness*, where fairness is required for each object instance.[2] This idea was captured in rewriting logic with the notion of *localized fairness* [134]. As explained in Section 2.2.3, a transition specified by a rule $l : q \longrightarrow r$ **if** C is *parametric* on the set X of variables in the rule, and fairness for the rule l can be *localized* to a subset $\{x_{j_1}, \dots, x_{j_k}\}$ of X. For example, in object fairness we localize it to the single variable o parameterizing the identity of the object involved in the transition.

This is conceptually clear, but difficult to support at the model checking level for at least three reasons: (i) fairness localized to variables $x_{j_1}, \dots, x_{j_k}$ is actually a universally quantified *first-order* temporal logic formula of the form $\forall(x_{j_1}, \dots, x_{j_k})\ \varphi$, whereas model checking works at the *propositional* level; (ii) even if a system is finite-state, the number of actual *instances* of the variables $x_{j_1}, \dots, x_{j_k}$ may be impossible to determine *a priori* without exploring the entire state space (for example, object systems with dynamic object creation); and (iii) even if one could determine all the instances, the total number of fairness conditions generated for all instances may be large, making the verification of properties under such localized fairness by direct model checking of $(\psi_1 \wedge \ldots \wedge \psi_n) \rightarrow \varphi$ particularly hopeless. To the best of our knowledge no solution to the problem of model checking properties under such parameterized fairness assumptions was known until we proposed and demonstrated a model checking algorithm for it in [18].

[2]E.g., in Example 2.5, the objects are philosophers having generic transitions (identified by the *grabF* and *grabS* rules) to pick up a chopstick. It is not enough for such a transition to be fair: we must require it to be fair when instantiated *for each philosopher*.

4.1.1 Main Contributions

First, this chapter presents a framework to verify LTLR properties under parameterized fairness conditions, given by generalized strong and weak fairness formulas of the forms:

$$(\forall \overline{x})\ \Box\Diamond\Phi \to \Box\Diamond\Psi, \qquad (\forall \overline{x})\ \Diamond\Box\Phi \to \Box\Diamond\Psi,$$

where the number of system entities over which the parametrization ranges can be unbounded[3] and may change during execution. Our framework is based on the notion of *parameter abstraction* to make explicit the fact that, even though the domain of entities or parameters is infinite, only a *finite* number of parameters are meaningful in a single state for fairness purposes. For example, in concurrent object systems with dynamic object creation, meaningful parameters are the objects in the state, and strong/weak fairness is vacuously satisfied for the objects not existing in a system.[4]

Next, this chapter presents an on-the-fly LTLR model checking algorithm that can handle universally quantified fairness formulas using parameter abstraction. This algorithm is based on the emptiness checking algorithms for a Streett automaton associated to the strong fairness conditions [76, 122], but significantly adapted to directly deal with "dynamic" parameters. Its computational complexity is linear in the number of fairness instances (recall that the standard method that takes fairness as a premise of a property is exponential in the number of strong fairness conditions).

Finally, the Maude LTLR model checker, introduced in Chapter 3, has been extended to support verification of LTLR properties under localized fairness assumptions. The work to implement this Maude Fair LTLR model checker has been substantial and has involved non-trivial design decisions: both the C++ implementation of the fair model checking algorithm and the standard LTLR model checking algorithm are supported by the tool. This is because, if no fairness assumptions are made, the second algorithm offers greater efficiency. A convenient and succinct way to specify localized fairness conditions by means of *rule attributes* has also been provided. To the best of our knowledge, this is the first model checker that can verify temporal logic properties under parameterized fairness assumptions.

[3]For finite-state systems the number is finite, but it may be impossible to determine such a number from the initial state without exploring the entire state space.

[4]E.g., $\mathit{enabled}(o)$ becomes false for all states if an object o does not exist in the system. Therefore, $\Diamond\Box\mathit{enabled}(o) \to \Box\Diamond\mathit{execute}(o)$ is vacuously satisfied.

4.1.2 Related Work

Parameterization has long been considered as a way to describe fairness of concurrent systems. The theorem proving of liveness properties commonly involves parameterized fairness conditions, e.g., [150]. Fairness for modeling languages is often parameterized, such as object/process fairness [98] and actor fairness [5]. However, such fairness notions are parameterized *only* by specific entities, depending on the system modeling language. Localized fairness [134] was introduced as a unified notion to express different variants of fairness, depending on the chosen system granularity level, but generalized versions of strong and weak fairness were not discussed in [134]. This chapter extends localized fairness to incorporate generalized strong/weak fairness involving generic spatial action patterns, and answers the question of how to model check LTLR properties under such localized fairness.

The typical method to model check a temporal logic property φ under parameterized fairness is to build the conjunction of corresponding instances of fairness, and to apply either a standard model checking algorithm for the reformulated formula $\mathit{fair} \to \varphi$, or a specialized model checking algorithm to directly handle fairness, such as [76, 104, 122, 115]. The first approach is inadequate for fairness in general, since the computational complexity is exponential in the number of strong fairness conditions, while the other is linear. Also, compiling such a formula $\mathit{fair} \to \varphi$ into a Büchi automaton is often not feasible in reasonable time [165]. There are several tools using the specialized algorithms, such as PAT [158] and Maria [122]. Our tool is related to the second approach, but it does not require pre-translation of parameterized fairness, and can handle *dynamic* fairness instances.

4.1.3 Structure of the Chapter

This chapter is organized as follows. Section 4.2 first introduces localized fairness specifications and presents a logical framework for parameterized fairness, including parameter abstraction. Section 4.3 then describes the on-the-fly model checking algorithm under parameterized fairness, based on Streett automata. Section 4.4 explains the user interface for conveniently specifying localized fairness in the Maude Fair LTLR model checker, and shows some experimental results. Finally, Section 4.5 presents a case study, and Section 4.6 presents some concluding remarks.

4.2 Localized Fairness in Quantified LTLR

Fairness properties of a rewrite theory $\mathcal{R}$ can be easily expressed by patterns of rewrite events, i.e., by spatial action patterns in LTLR. Given a *ground* spatial action pattern δ, the strong fairness condition and the weak fairness condition are respectively expressed by the LTLR formulas

$$\Box\Diamond \mathit{enabled}(\delta) \to \Box\Diamond\delta \qquad \Diamond\Box \mathit{enabled}(\delta) \to \Box\Diamond\delta$$

where a special state proposition $\mathit{enabled}(\delta)$ is *operationally*[5] defined using the function symbol $\mathit{enabled} : \mathsf{Action} \to \mathsf{Prop}$ as follows.

Definition 4.1. *Given a computable rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R)$ and a spatial action pattern δ, the state proposition $\mathit{enabled}(\delta)$ holds in exactly those states $[can_{E/B}(t)]_B$ from which there is a canonical one-step rewrite $\lambda : [can_{E/B}(t)]_A \longrightarrow_{\mathcal{R}} [can_{E/B}(t')]_A$ such that the spatial action pattern δ corresponds to λ (i.e., $\lambda \models \delta$).*

4.2.1 Localized Fairness Specifications

Localized fairness specifications make it possible to define *parameterized fairness* conditions associated to spatial action patterns containing variables.

Definition 4.2. *Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, its* localized fairness specification *is a pair of finite sets $(\mathcal{J}, \mathcal{F})$, whose elements are* parametric spatial action patterns *of the form*

$$\delta(y_1, \dots, y_k) \in \mathcal{J} \cup \mathcal{F}.$$

The set $\mathcal{J}$ stands for parameterized weak fairness conditions and $\mathcal{F}$ stands for parameterized strong fairness conditions.

The localized fairness condition specified by $\delta(y_1, \dots, y_m) \in \mathcal{J} \cup \mathcal{F}$ means that for each *ground* instance $\theta(\delta(y_1, \dots, y_m))$ of $\delta(y_1, \dots, y_m)$ with a ground substitution θ, the corresponding one-step rewrite satisfies the desired weak or strong fairness requirements. Thanks to the expressive power of spatial action patterns, this localized fairness specification is quite general, so that many different notions of fairness, including object/process fairness [98] and actor fairness [5], can all be expressed in a unified way [134].

[5] That is, for efficiency reasons, the meaning of $\mathit{enabled}(\delta)$ is defined at the meta-level, *not* using equations. In our tool, the *enabled* function is implemented at the C++ level as a special function symbol in Core Maude.

A localized fairness specification $(\mathcal{J}, \mathcal{F})$ of a rewrite theory $\mathcal{R}$ defines a *fair infinite paths* in the corresponding LKS $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}$, where the spatial action patterns in $\mathcal{J} \cup \mathcal{F}$ are defined by the support equational theory $\mathcal{P}$.

Definition 4.3. *An infinite path (π, α) in $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}$ is $\mathcal{J}, \mathcal{F}$–*fair *iff each ground instance of every localized fairness condition in $\mathcal{J} \cup \mathcal{F}$ is satisfied on the path (π, α) in $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}$, that is:*

- *for each $\delta(\overline{y}) \in \mathcal{J}$ and ground substitution θ, where $\overline{y} = (y_1, \ldots, y_m)$, the weak fairness condition of $\theta(\delta(\overline{y}))$ holds, i.e.:*

$$\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}, (\pi, \alpha) \models \Diamond\Box \mathit{enabled}(\theta(\delta(\overline{y}))) \to \Box\Diamond\theta(\delta(\overline{y})),$$

- *for each $\delta(\overline{y}) \in \mathcal{F}$ and ground substitution θ, where $\overline{y} = (y_1, \ldots, y_m)$, the strong fairness condition of $\theta(\delta(\overline{y}))$ holds, i.e.:*

$$\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}, (\pi, \alpha) \models \Box\Diamond \mathit{enabled}(\theta(\delta(\overline{y}))) \to \Box\Diamond\theta(\delta(\overline{y})).$$

An LTLR formula φ is then *fairly* satisfied in $\mathcal{R}$ from an initial state $[t]_E$ under a localized fairness specification $(\mathcal{J}, \mathcal{F})$, denoted by

$$\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}, [t]_E \models_{\mathcal{J} \cup \mathcal{F}} \varphi,$$

iff $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}, (\pi, \alpha) \models \varphi$ holds for each $\mathcal{J}, \mathcal{F}$-fair infinite path (π, α) starting from $[t]_E$ such that $\pi(0) = [t]_E$.

Each localized fairness condition $\delta(\overline{y}) \in \mathcal{J} \cup \mathcal{F}$ can be expressed by an equivalent *universally quantified* LTLR formula of the form $\forall \overline{y}\ \varphi$, where φ is quantifier-free, and $\mathit{vars}(\varphi) \subseteq \overline{y}$. If $[\overline{y} \to \mathcal{T}_\Sigma]$ denotes the set of all ground substitutions $\theta : \overline{y} \to \mathcal{T}_\Sigma$, then the satisfaction of $\forall \overline{y}\ \varphi$ is defined by:

$$\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}, (\pi, \alpha) \models \forall \overline{y}\ \varphi \iff (\forall \theta \in [\overline{y} \to \mathcal{T}_\Sigma])\ \bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}, (\pi, \alpha) \models \theta(\varphi).$$

The strong and weak localized fairness conditions for a parametric spatial action pattern $\delta(\overline{y}) \in \mathcal{J} \cup \mathcal{F}$ can then be respectively expressed by the following universally quantified LTLR formulas:

$$\forall \overline{y}\ \Box\Diamond \mathit{enabled}(\delta(\overline{y})) \to \Box\Diamond\delta(\overline{y}), \qquad \forall \overline{y}\ \Diamond\Box \mathit{enabled}(\delta(\overline{y})) \to \Box\Diamond\delta(\overline{y}),$$

where the meaning of the parametric state proposition $\mathit{enabled}(\delta(\overline{y}))$ is again operationally defined so that a ground instance $\mathit{enabled}(\theta(\delta(\overline{y})))$ is satisfied exactly on those states $[\mathit{can}_{E/B}(t)]_A$ such that $\theta(\delta(\overline{y}))$ is enabled.

Example 4.1. *Consider the parameterized fairness conditions for the dining philosophers problem in Example 2.8:*

- *if a philosopher can continuously wake up beyond a certain point (by the wake rule), then the philosopher must wake up infinitely often; and*
- *if a philosopher can grab a chopstick infinitely often (by the grabF and grabS rules), then the philosopher must grab it infinitely often.*

Using the related spatial action patterns $\texttt{\{}'wake : 'i\backslash i\texttt{\}}$, $\texttt{\{}'grabF : 'i\backslash i\texttt{\}}$, *and* $\texttt{\{}'grabS : 'i\backslash i\texttt{\}}$, *where the variable* i *denotes each philosopher's identity, the fairness conditions can be expressed as the localized specification:*

$$\mathcal{J} = \{\ \texttt{\{}'wake : 'i\backslash i\texttt{\}}\ \}, \quad \mathcal{F} = \{\ \texttt{\{}'grabF : 'i\backslash i\texttt{\}}, \texttt{\{}'grabS : 'i\backslash i\texttt{\}}\ \}.$$

Example 4.2 (Fault-Tolerant Client-Server Communication [136])**.** *There are a number of clients and servers, where each client `C` sends a query `N` to a server `S` to receive an answer, and the server returns the answer `f(S,C,N)` of the query using a function `f`. The communication environment is faulty in the sense that messages can be duplicated or lost.*

The configuration is a multiset of clients, servers, and messages, with the empty multiset `null`. A client is represented as a term `[C,S,N,W]` with `C` the client's name, `S` a server's name, `N` a number representing a query, and `W` either a number representing an answer or `nil` if the answer has not yet been received. A server is represented as a term `[S]` with the name `S`, and a message is represented as a term `I <- {J,N}` with `I` the receiver's name, `J` the sender's name, and `N` a number. The following rewriting rules define the behavior of the system:

```
rl [req]  : [C,S,N,nil]            =>  [C,S,N,nil] S <-{C,N} .
rl [reply]: S <-{C,N} [S]          =>  [S]  C <-{S,f(S,C,N)} .
rl [rec]  : C <-{S,M} [C,S,N,nil]  =>  [C,S,N,M] .
rl [dupl] : I <-{J,M}              =>  I <-{J,M} I <-{J,M} .
rl [loss] : I <-{J,M}              =>  null  .
```

The fairness assumptions needed to prove the liveness property $\Diamond\texttt{\{}'rec\texttt{\}}$*— meaning that some client will eventually receive an answer—are: (i) weak fairness of the rule req* for each client `C`*, (ii) strong fairness of the rule reply* for each server `S` and client `C`*, and (iii) strong fairness of the rule rec* for each client `C`*, and are naturally expressed as the localized fairness specification:*

$$\mathcal{J} = \{\ \texttt{\{}'req : 'C\backslash C\texttt{\}}\ \}, \qquad \mathcal{F} = \{\ \texttt{\{}'reply : 'S\backslash S ; 'C\backslash C\texttt{\}}, \texttt{\{}'rec : 'C\backslash C\texttt{\}}\ \}.$$

4.2.2 Parameterized Labeled Kripke Structures

For a universally quantified LTLR formula $\forall \overline{y}\ \varphi$, the state propositions and the spatial action patterns in φ are parametric on the relevant entities, e.g., process names, messages, or other data structures. Therefore, we allow a *parametric* state proposition $p(x_1, \ldots, x_n) \in \mathit{AP}(\mathcal{X})$ and a *parametric* spatial action pattern $\delta(y_1, \ldots, y_m) \in \mathit{ACT}(\mathcal{X})$ over an infinite set of variables $\mathcal{X}$.

Definition 4.4. *A labeled Kripke structure $\bar{\mathcal{K}} = (S, \mathit{AP}(\mathcal{C}), \mathcal{L}, \mathit{ACT}(\mathcal{C}), \rightarrow_{\bar{\mathcal{K}}})$ is* parameterized *over a set of parameters $\mathcal{C}$ iff there exist a set $\mathit{AP}(\mathcal{X})$ of parametric state propositions and a set $\mathit{ACT}(\mathcal{X})$ of parametric spatial action pattern over an infinite set of variables $\mathcal{X}$ such that $\mathcal{X} \cap \mathcal{C} = \emptyset$, where:*

$$\mathit{AP}(\mathcal{C}) = \{p(a_1, \ldots, a_n) \mid a_1, \ldots, a_n \in \mathcal{C}, p(x_1, \ldots, x_n) \in \mathit{AP}(\mathcal{X})\},$$
$$\mathit{ACT}(\mathcal{C}) = \{\delta(b_1, \ldots, b_m) \mid b_1, \ldots, b_m \in \mathcal{C}, \delta(x_1, \ldots, x_m) \in \mathit{ACT}(\mathcal{X})\}.$$

There is an implicit relation between a set $\mathcal{C}$ of parameters and a set S of states derived from an LKS $\bar{\mathcal{K}}$ in terms of a definable set. Let $[\overline{x} \to \mathcal{C}]$ denote the set of all substitutions $\theta : \overline{x} \to \mathcal{C}$.

Definition 4.5. *Given $\bar{\mathcal{K}} = (S, \mathit{AP}(\mathcal{C}), \mathcal{L}, \mathit{ACT}(\mathcal{C}), \rightarrow_{\bar{\mathcal{K}}})$, the* definable *set of a parametric state proposition $p(\overline{x}) \in \mathit{AP}(\mathcal{X})$ for a state $s \in S$ is*

$$\mathcal{D}_s(p(\overline{x})) = \{\theta \in [\overline{x} \to \mathcal{C}] \mid \theta(p(\overline{x})) \in \mathcal{L}(s)\},$$

that is, the set of substitutions θ that make their instances $\theta(p(\overline{x}))$ satisfied in s. Similarly, the definable *set of a parametric spatial action pattern $\delta(\overline{x}) \in \mathit{ACT}(\mathcal{X})$ for a transition $s \xrightarrow{\Lambda}_{\bar{\mathcal{K}}} s'$ is*

$$\mathcal{D}_\Lambda(\delta(\overline{x})) = \{\theta \in [\overline{x} \to \mathcal{C}] \mid \theta(\delta(\overline{x})) \in \Lambda\}.$$

We define the set of *universally quantified LTLR formulas* over $\mathit{AP}(\mathcal{X})$, $\mathit{ACT}(\mathcal{X})$, and $\mathcal{C}$ as the set of formulas of the form $\forall \overline{x}\ \varphi$ with $\mathit{vars}(\varphi) \subseteq \mathcal{X}$, where φ is a propositional formula over $\mathit{AP}(\mathcal{C} \cup \mathcal{X})$ and $\mathit{ACT}(\mathcal{C} \cup \mathcal{X})$.

Definition 4.6. *Given a parameterized LKS $\bar{\mathcal{K}}$ over a set of parameters $\mathcal{C}$, for a path (π, α) and a universally quantified LTLR formula $\forall \overline{x}\ \varphi$:*

$$\bar{\mathcal{K}}, (\pi, \alpha) \models \forall \overline{x}\ \varphi \quad \iff \quad (\forall \theta \in [\overline{x} \to \mathcal{C}])\ \bar{\mathcal{K}}, (\pi, \alpha) \models \theta(\varphi)$$

For a set of initial states $S_0 \subseteq S$, $\bar{\mathcal{K}}, S_0 \models \forall \overline{x}\ \varphi$ iff $\bar{\mathcal{K}}, (\pi, \alpha) \models \forall \overline{x}\ \varphi$ for each path (π, α) starting from $\pi(0) \in S_0$.

Although the size of the parameter set $\mathcal{C}$ can be infinite, in practice, the number of parameters $\mathcal{C}$ that *occur* in a state is typically finite. For example, in a concurrent object system that dynamically creates a new object for each step, the total number of objects—in the system's infinite state space—would be infinite, but the number of processes in a *single* state is always finite. In particular, for a transition $s \xrightarrow{\Lambda}_{\bar{\mathcal{K}}} s'$, if the sets $\mathcal{L}(s)$, $\mathcal{L}(s')$, and Λ are finite, then the definable sets for any state propositions and spatial action patterns are finite. This is captured by the *finite instantiation property.*

Definition 4.7. *A parameterized LKS* $\bar{\mathcal{K}} = (S, AP(\mathcal{C}), \mathcal{L}, ACT(\mathcal{C}), \rightarrow_{\bar{\mathcal{K}}})$ *over a set of parameters* $\mathcal{C}$ *satisfies the* finite instantiation property (FIP) *iff for every transition* $s \xrightarrow{\Lambda}_{\bar{\mathcal{K}}} s'$*, the sets* $\mathcal{D}_s(p(\overline{x}))$ *for each* $p(\overline{x}) \in AP(\mathcal{X})$ *and* $\mathcal{D}_\Lambda(\delta(\overline{x}))$ *for each* $\delta(\overline{x}) \in ACT(\mathcal{X})$ *are always finite.*

In general, a spatial action pattern $\delta(\overline{y})$ for a rewrite theory $\mathcal{R}$ may *not* satisfy FIP. For example, consider the spatial action pattern $ge(N)$ defined by the conditional equation in Maude:

```
var CXT : StateContext .     var SUB : StateSubstitution .
var R : RuleName .           var N : Nat .

cep {CXT | R : SUB} |= ge(N) = true if N > 0 .
```

Obviously, there exist infinitely many relevant parameters for any transition, namely, all natural numbers. In this case, the variable N is *unbounded* with respect to the pattern $\{CXT \mid R : SUBST\}$ for the spatial action pattern $ge(N)$, so that it can be instantiated to any values that satisfy the condition.

However, for a *finite-state* rewrite theory $\mathcal{R}$ and any basic action pattern of the form $l(\overline{y}) = \{'l : 'y_1\backslash y_1; \ldots ; 'y_m\backslash y_m\}$, both $l(\overline{y})$ and its corresponding enabled proposition $enabled(l(\overline{y}))$ satisfy FIP.

Definition 4.8. *A rewrite theory* $\mathcal{R} = (\Sigma, E, R)$ *is called* finite-state *iff the set of reachable states* $Reach_{\mathcal{R}}([t]_E) = \{[u]_E \in \mathcal{T}_{\Sigma,k} \mid [t]_E \longrightarrow^*_{\mathcal{R}} [u]_E\}$ *for each initial state* $[t]_E \in \mathcal{T}_{\Sigma,k}$ *is finite.*

Each one-step rewrite satisfies only one ground instance of $l(\overline{y})$, since each ground instance $\theta(l(\overline{y}))$ is satisfied by a one-step proof term $t[l(\theta)]$. Similarly, each ground instance $\theta(enabled(l(\overline{y})))$ is satisfied in a state $[t]_E$ iff there exists a one-step rewrite from $[t]_E$ satisfying $\theta(l(\overline{y}))$. Since a finite-state rewrite theory has only finitely many one-step rewrites from a state, each state $[t]_E$ of $\mathcal{R}$ satisfies only finitely many ground instances of $enabled(l(\overline{y}))$.

4.2.3 Sufficient Conditions for FIP

Furthermore, for a finite-state rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R)$, there exist simple conditions for parametric state propositions and spatial action patterns to satisfy FIP as follows (formally stated in Theorem 4.1 below).

1. For any equation involving parametric state propositions (resp., spatial action patterns), the variables in the proposition pattern should be bounded to the state pattern (resp., the proof term pattern).

2. To avoid an infinite number of ground proposition instances due to the equations in E, the patterns in such equations should already be in E/B-canonical form, and furthermore be strongly irreducible.

Definition 4.9. *A term t is called* strongly irreducible *with respect to E modulo B iff for every canonical ground substitution $\theta : \mathcal{X} \to Can_{\Sigma,E/B}$, the term $\theta(t)$ is in E/B-canonical form, i.e., $\theta(t) =_B can_{E/B}(\theta(t))$.*

Note that these conditions are *not* necessary conditions: even if there are unbounded variables in parametric propositions, the conditions in related equations may restrict the number of relevant parameters to be finite. For example, consider the spatial action pattern $ge2(N)$ given by the equation $\texttt{\{}CXT \mid R : SUBST\texttt{\}} \models ge2(N) = true$ **if** $N \geq 0 \wedge N < 10$. There is only a finite number of relevant parameters $0, \ldots, 9$, although N is unbounded.

Any spatial action pattern $\delta(\overline{y}) \in SP(\mathcal{R})$ satisfies these two conditions whenever the pattern $\delta(\overline{y})$ is strongly irreducible, and therefore satisfies FIP. Every equation for $SP(\mathcal{R})$ in Section 3.4.1 has a strongly irreducible proof term pattern and no unbounded variables. For instance, in the equations

```
eq {CXT | R : SUB ; SUB'}    |= {R : SUB}          = true .
eq {CXT | R : SUB ; SUB'}    |= {CXT | R : SUB}    = true .
```

the pattern $\texttt{\{}CXT \mid R : SUB; SUB'\texttt{\}}$ includes all the variables in the equations and is strongly irreducible. Moreover, if $\mathcal{R}$ is finite-state, then $enabled(\delta(\overline{y}))$ also satisfies FIP for strongly irreducible $\delta(\overline{y}) \in SP(\mathcal{R})$. By definition, each ground instance $\theta(enabled(\delta(\overline{y})))$ is satisfied in a state $[t]_{E \cup B}$ iff there is a canonical one-step rewrite $\gamma : [can_{E/B}(t)]_B \longrightarrow_{\mathcal{R}} [can_{E/B}(t')]_B$ such that $\gamma \models \theta(\delta(\overline{y}))$. Since $\delta(\overline{y})$ satisfies FIP, there is only a finite number of such substitutions θ, and a finite-state rewrite theory has only finitely many one-step rewrites from each state. Consequently:

Corollary 4.1. *Given a finite-state and computable rewrite theory $\mathcal{R}$, a* strongly irreducible *spatial action pattern $\delta(\overline{y})$ in $SP(\mathcal{R})$ and its enabled proposition $enabled(\delta(\overline{y}))$ satisfy FIP.*

We now precisely specify the sufficient conditions for FIP in the following theorem. Two terms $t, u \in \mathcal{T}_{\Sigma}(\mathcal{X})$ are $E \cup B$-*unifiable* iff there is a ground substitution $\theta : \mathcal{X} \to \mathcal{T}_{\Sigma}$ such that $\theta(t) =_{E \cup B} \theta(u)$.

Theorem 4.1 (Sufficient Conditions for FIP)**.** *Given a finite-state rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R)$ and a support equational theory $\mathcal{P} = (\Pi, D)$, where both $\mathcal{R}$ and $(\Sigma \cup \Pi, E \cup B \cup D, R)$ are computable:*

- *a parametric state proposition $p \in \mathcal{T}_{\Sigma\cup\Pi}(\mathcal{X})_{\mathsf{Prop}}$ satisfies FIP, if for each matching equation $(t \models p' = t'$ **if** $cond) \in D$ such that p and p' are $E \cup D \cup B$-unifiable, both t and p are strongly irreducible with respect to $E \cup D$ modulo B, and $vars(p') \subseteq vars(t)$.*
- *a parametric spatial action pattern $\delta \in \mathcal{T}_{\Sigma\cup\Pi}(\mathcal{X})_{\mathsf{Action}}$ satisfies FIP, if for each matching equation $(\gamma \models \delta' = t'$ **if** $cond) \in D$ such that δ and δ' are $E \cup D \cup B$-unifiable, both γ and δ are strongly irreducible with respect to $E \cup D$ modulo B, and $vars(\delta') \subseteq vars(\gamma)$.*

Proof. Let us first consider a parametric state proposition $p \in \mathcal{T}_{\Sigma\cup\Pi}(\mathcal{X})_{\mathsf{Prop}}$. Since $(\Sigma \cup \Pi, E \cup B \cup D, R)$ is computable, we can always consider terms in $E \cup D$-canonical form modulo B. By definition, each ground instance $\theta(p) \in \mathcal{T}_{\Sigma\cup\Pi}$ is satisfied in a state $[can_{E/B}(u)]_B$ iff:

$$can_{E/B}(u) \models can_{E\cup D/B}(\theta(p)) \quad =_{E\cup D\cup B} \quad can_{E\cup D/B}(true),$$

which holds *only if* there are a matching equation $(t \models p' = t'$ **if** $cond) \in D$ and a canonical substitution $\vartheta : vars(t) \cup vars(p') \to Can_{\Sigma\cup\Pi, E\cup D/B}$ with $can_{E\cup D/B}(\vartheta(t)) =_B can_{E\cup D/B}(u)$ and $can_{E\cup D/B}(\vartheta(p')) =_B can_{E\cup D/B}(\theta(p))$.

By the assumptions, the theory $(\Sigma \cup \Pi, E \cup B \cup D)$ protects $(\Sigma, E \cup B)$, and $vars(p') \subseteq vars(t)$. Therefore, we have $\vartheta \in [vars(t) \to Can_{\Sigma,E/B}]$. Also, since t is strongly irreducible with respect to $E \cup D$ modulo B, we have $can_{E\cup D/B}(\vartheta t) =_B can_{E\cup D/B}(u) \iff \vartheta(can_{E/B}(t)) =_B can_{E/B}(u)$. Because we assume that $\mathcal{R}$ is computable, there exists only a *finite* number of such B-matching substitutions ϑ for each $[can_{E/B}(u)]_B$.

Next, since p is strongly irreducible with respect to $E \cup D$ modulo B, $can_{E\cup D/B}(\vartheta p') =_B can_{E\cup D/B}(\theta p) \iff can_{E\cup D/B}(\vartheta p') =_B \theta(can_{E\cup D/B}(p))$. Again, since $(\Sigma\cup\Pi, E\cup A\cup D, R)$ is computable, there exists a *finite* number of such B-matching substitutions θ for each $can_{E\cup D/B}(\vartheta p')$. Consequently, there exists a finite number of relevant canonical ground substitutions θ with respect to a parametric state proposition p for each state $[u]_{E\cup B}$, i.e., p satisfies FIP. The case for a parametric spatial action pattern δ is similar. $\square$

4.3 Parameterized Fair Model Checking Algorithm

As explained in Section 3.3.2, the model checking problem for an LTLR formula φ on a rewrite theory $\mathcal{R}$ can be characterized by automata-theoretic techniques applied to the associated LKS $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}$ together with the Büchi automaton $\mathscr{B}_{\neg\varphi}$ for the negated formula $\neg\varphi$.

However, the model checking algorithm for a localized fairness condition on $\mathcal{R}$, namely, a universally quantified LTLR formula $\forall \overline{x}\ \varphi$, is nontrivial, since such a variable quantification can range over an infinite set $\mathcal{C}$ of actual parameters: for each variable $y \in \overline{x}$, the set of ground terms having the variable's sort in $\mathcal{R}$. We cannot directly use the LKS $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}$ for model checking $\forall \overline{x}\ \varphi$. However, the satisfaction relation for $\forall \overline{x}\ \varphi$ can be efficiently determined on a finite LKS satisfying the finite instantiation property (FIP).

4.3.1 Parameter Abstraction

By definition, if an LKS $\bar{\mathcal{K}}$ over a set of parameters $\mathcal{C}$ satisfies FIP, then for each transition $s \xrightarrow{\Lambda}_{\bar{\mathcal{K}}} s'$, there are only finitely many "relevant" ground substitutions that appear in the finite definable sets $\mathcal{D}_s(p(\overline{x}))$ and $\mathcal{D}_\Lambda(\delta(\overline{x}))$. Therefore, we can define an abstraction of each substitution $\theta : \overline{x} \to \mathcal{C}$ with respect to the definable sets, by collapsing the cofinite[6] complement set of each definable set into the abstract substitution $\bot_{\overline{x}} : \overline{x} \to \{\bot\}$ with a fresh new constant $\bot$, which intuitively denotes any "irrelevant" parameters that *never* appear in such a definable set.

Example 4.3. *For a parametric state proposition $p(\overline{x}) \in AP(\mathcal{X})$ and a state $s_0 \in S$, the abstraction $\varrho_{s_0,p(\overline{x})}(\theta)$ of each substitution θ is defined by: $\varrho_{s_0,p(\overline{x})}(\theta) = \bot_{\overline{x}}$ if $\theta \notin \mathcal{D}_{s_0}(p(\overline{x}))$, and $\varrho_{s_0,p(\overline{x})}(\theta) = \theta$ if $\theta \in \mathcal{D}_{s_0}(p(\overline{x}))$. The extended parameter set $\mathcal{C}_\bot = \mathcal{C} \cup \{\bot\}$ then implies the parameterized LKS*

$$\bar{\mathcal{K}}_\bot = (S, AP(\mathcal{C}_\bot), \mathcal{L}, ACT(\mathcal{C}_\bot), \to_{\bar{\mathcal{K}}})$$

that naturally extends $\bar{\mathcal{K}} = (S, AP(\mathcal{C}), \mathcal{L}, ACT(\mathcal{C}), \to_{\bar{\mathcal{K}}})$. In this case, we can easily see that for any substitution θ and path (π, α) such that $\pi(0) = s_0$:

$$\bar{\mathcal{K}}, (\pi, \alpha) \models \theta(p(\overline{x})) \iff \bar{\mathcal{K}}_\bot, (\pi, \alpha) \models \varrho_{s_0,p(\overline{x})}(\theta)(p(\overline{x}))$$

since $\bot$ is a fresh new constant and therefore $\bot_{\overline{x}}(p(\overline{x})) \notin \mathcal{L}(s_0)$ (recall that $\mathcal{D}_s(p(\overline{x})) = \{\theta \in [\overline{x} \to \mathcal{C}] \mid \theta(p(\overline{x})) \in \mathcal{L}(s)\}$ by definition).

[6] A set is *cofinite* iff the complement of the set is finite.

This abstraction function can be extended to any LTLR formula using a natural partial ordering $\preceq \subseteq [\overline{x} \to \mathcal{C}_\bot]^2$ in the abstract domain $[\overline{x} \to \mathcal{C}_\bot]$.

Definition 4.10. *For two abstract substitutions $\theta_1, \theta_2 \in [\overline{x} \to \mathcal{C}_\bot]$, let:*

$$\theta_1 \preceq \theta_2 \iff \theta_1(x) = \bot \textit{ or } \theta_1(x) = \theta_2(x) \textit{ for each } x \in \overline{x}.$$

If θ_1 and θ_2 has a common upper bound (that is, $\theta_1 \preceq \theta$ and $\theta_2 \preceq \theta$ for some $\theta \in [\overline{x} \to \mathcal{C}_\bot]$), then their least upper bound $\theta_1 \vee \theta_2$ is defined as follows, where $c \vee \bot = \bot \vee c = c \vee c = c$ for each $c \in \mathcal{C}$:

$$(\theta_1 \vee \theta_2)(x) = \theta_1(x) \vee \theta_2(x) \textit{ for each } x \in \overline{x},$$

For two abstract substitutions θ_1 and θ_2 with possibly different domains, where $\theta_1(x) \vee \theta_2(x)$ is defined for $x \in dom(\theta_1) \cap dom(\theta_2)$, the combined substitution $\theta_1 \oplus \theta_2$ is defined by:

$$\theta_1 \oplus \theta_2(x) = \begin{cases} \theta_1(x) & \textit{if } x \in dom(\theta_1) - dom(\theta_2) \\ \theta_2(x) & \textit{if } x \in dom(\theta_2) - dom(\theta_1) \\ \theta_1(x) \vee \theta_2(x) & \textit{otherwise.} \end{cases}$$

The *parameter abstraction* $\varrho_{(\pi,\alpha),\varphi}(\theta) \in [vars(\varphi) \to \mathcal{C}_\bot]$ of a substitution θ can then be defined by extending the base cases for $p(\overline{x})$ and $\delta(\overline{x})$ as follows.

Definition 4.11. *Given a parameterized LKS $\bar{\mathcal{K}}$ over a set of parameters $\mathcal{C}$, the* abstraction function $\varrho_{(\pi,\alpha),\varphi} : [vars(\varphi) \to \mathcal{C}_\bot] \to [vars(\varphi) \to \mathcal{C}_\bot]$ *for an LTLR formula φ and a path (π, α) is inductively defined by:*

$$\begin{aligned}
\varrho_{(\pi,\alpha),p(\overline{x})}(\theta) &= \textbf{if } \theta \in \mathcal{D}_{\pi(0)}(p(\overline{x})) \textbf{ then } \theta \textbf{ else } \bot_{\overline{x}} \textbf{ fi} \\
\varrho_{(\pi,\alpha),\delta(\overline{x})}(\theta) &= \textbf{if } \theta \in \mathcal{D}_{\alpha(0)}(\delta(\overline{x})) \textbf{ then } \theta \textbf{ else } \bot_{\overline{x}} \textbf{ fi} \\
\varrho_{(\pi,\alpha),\neg\varphi}(\theta) &= \varrho_{(\pi,\alpha),\varphi}(\theta) \\
\varrho_{(\pi,\alpha),\varphi_1 \wedge \varphi_2}(\theta) &= \varrho_{(\pi,\alpha),\varphi_1}(\theta|_{vars(\varphi_1)}) \oplus \varrho_{(\pi,\alpha),\varphi_2}(\theta|_{vars(\varphi_2)}) \\
\varrho_{(\pi,\alpha),\bigcirc\varphi}(\theta) &= \varrho_{(\pi,\alpha)^1,\varphi}(\theta) \\
\varrho_{(\pi,\alpha),\varphi_1 \mathbf{U} \varphi_2}(\theta) &= \bigvee_{i \geq 0} \varrho_{(\pi,\alpha)^i,\varphi_1}(\theta|_{vars(\varphi_1)}) \;\oplus\; \bigvee_{j \geq 0} \varrho_{(\pi,\alpha)^j,\varphi_2}(\theta|_{vars(\varphi_2)})
\end{aligned}$$

Notice that both $\varrho_{(\pi,\alpha),\varphi_1 \wedge \varphi_2}(\theta)$ and $\varrho_{(\pi,\alpha),\varphi_1 \mathbf{U} \varphi_2}(\theta)$ are well-defined since $\varrho_{(\pi,\alpha),\varphi}(\theta) \preceq \theta$ always holds by construction. For a universally quantified LTLR formula $\forall \overline{x}\ \varphi$, the satisfaction relation of φ on a path (π, α) for an abstract substitution $\vartheta = \varrho_{(\pi,\alpha),\varphi}(\theta)$ is naturally defined in the extended LKS $\bar{\mathcal{K}}_\bot = (S, AP(\mathcal{C}_\bot), \mathcal{L}, ACT(\mathcal{C}_\bot), \to_{\bar{\mathcal{K}}})$, where $\mathcal{C}_\bot = \mathcal{C} \cup \{\bot\}$.

Lemma 4.1. *Given a parameterized LKS $\bar{\mathcal{K}}$ over a set of parameters $\mathcal{C}$, a universally quantified LTLR formula $\forall \overline{x}\ \varphi$, and a path (π, α):*

$$(\forall \theta \in [\overline{x} \to \mathcal{C}]) \quad \bar{\mathcal{K}}, (\pi, \alpha) \models \theta(\varphi) \iff \bar{\mathcal{K}}_{\perp}, (\pi, \alpha) \models \varrho_{(\pi,\alpha),\varphi}(\theta)(\varphi).$$

Proof. We prove the following generalized version of the lemma by structural induction on φ: for any substitution $\vartheta \in [\overline{x} \to \mathcal{C}_{\perp}]$ with $\varrho_{(\pi,\alpha),\varphi}(\theta) \preceq \vartheta \preceq \theta$, $\bar{\mathcal{K}}, (\pi, \alpha) \models \theta(\varphi) \iff \bar{\mathcal{K}}_{\perp}, (\pi, \alpha) \models \vartheta(\varphi)$.

- $\varphi = p(\overline{x})$: $\bar{\mathcal{K}}, (\pi, \alpha) \not\models \theta(p(\overline{x}))$ iff $\theta \notin \mathcal{D}_{\pi(0)}(p(\overline{x}))$ iff $\varrho_{(\pi,\alpha),p(\overline{x})}(\theta) = \perp_{\overline{x}}$, and for any substitution $\perp_{\overline{x}} \preceq \vartheta \prec \theta$, $\bar{\mathcal{K}}_{\perp}, (\pi, \alpha) \not\models \vartheta(p(\overline{x}))$, since $\perp$ is a fresh new constant and therefore $\perp_{\overline{x}}(p(\overline{x})) \notin \mathcal{L}(\pi(0))$.
- $\varphi = \delta(\overline{x})$: $\bar{\mathcal{K}}, (\pi, \alpha) \not\models \theta(\delta(\overline{x}))$ iff $\theta \notin \mathcal{D}_{\alpha(0)}(\delta(\overline{x}))$ iff $\varrho_{(\pi,\alpha),\delta(\overline{x})}(\theta) = \perp_{\overline{x}}$, and for any substitution $\perp_{\overline{x}} \preceq \vartheta \prec \theta$, $\bar{\mathcal{K}}_{\perp}, (\pi, \alpha) \not\models \vartheta(\delta(\overline{x}))$.
- $\varphi = \neg\varphi'$: $\bar{\mathcal{K}}, (\pi, \alpha) \models \theta(\neg\varphi')$ iff $\bar{\mathcal{K}}, (\pi, \alpha) \not\models \theta(\varphi')$. By definition of ϱ, $\varrho_{(\pi,\alpha),\neg\varphi'}(\theta) = \varrho_{(\pi,\alpha),\varphi'}(\theta)$. By induction hypothesis, $\bar{\mathcal{K}}, (\pi, \alpha) \not\models \theta(\varphi')$ iff $\bar{\mathcal{K}}_{\perp}, (\pi, \alpha) \not\models \vartheta(\varphi')$ iff $\bar{\mathcal{K}}_{\perp}, (\pi, \alpha) \models \vartheta(\neg\varphi')$.
- $\varphi = \bigcirc\varphi'$: $\bar{\mathcal{K}}, (\pi, \alpha) \models \theta(\bigcirc\varphi')$ iff $\bar{\mathcal{K}}, (\pi, \alpha)^1 \models \theta(\varphi')$. By definition, $\varrho_{(\pi,\alpha),\bigcirc\varphi'}(\theta) = \varrho_{(\pi,\alpha)^1,\varphi'}(\theta)$. By induction hypothesis, $\bar{\mathcal{K}}, (\pi, \alpha)^1 \models \theta(\varphi')$ iff $\bar{\mathcal{K}}_{\perp}, (\pi, \alpha)^1 \models \vartheta(\varphi')$ iff $\bar{\mathcal{K}}_{\perp}, (\pi, \alpha) \models \vartheta(\bigcirc\varphi')$.
- $\varphi = \varphi_1 \wedge \varphi_2$: $\bar{\mathcal{K}}, (\pi, \alpha) \models \theta(\varphi_1 \wedge \varphi_2)$ iff $\bar{\mathcal{K}}, (\pi, \alpha) \models \theta|_{vars(\varphi_1)}(\varphi_1)$ and $\bar{\mathcal{K}}, (\pi, \alpha) \models \theta|_{vars(\varphi_2)}(\varphi_2)$. Let $\overline{v}_1 = vars(\varphi_1)$ and $\overline{v}_2 = vars(\varphi_2)$. By definition of $\varrho_{(\pi,\alpha),\varphi_1\wedge\varphi_2}$, we can easily see that:

  $$\varrho_{(\pi,\alpha),\varphi_1}(\theta|_{\overline{v}_1}) \preceq \varrho_{(\pi,\alpha),\varphi_1\wedge\varphi_2}(\theta)|_{\overline{v}_1}, \quad \varrho_{(\pi,\alpha),\varphi_2}(\theta|_{\overline{v}_2}) \preceq \varrho_{(\pi,\alpha),\varphi_1\wedge\varphi_2}(\theta)|_{\overline{v}_2}.$$

  That is, for any $\varrho_{(\pi,\alpha),\varphi_1\wedge\varphi_2}(\theta) \preceq \vartheta \preceq \theta$, $\varrho_{(\pi,\alpha),\varphi_1}(\theta|_{\overline{v}_1}) \preceq \vartheta|_{\overline{v}_1} \preceq \theta|_{\overline{v}_1}$ and $\varrho_{(\pi,\alpha),\varphi_2}(\theta|_{\overline{v}_2}) \preceq \vartheta|_{\overline{v}_2} \preceq \theta|_{\overline{v}_2}$ hold. Therefore, by induction hypothesis, $\bar{\mathcal{K}}, (\pi, \alpha) \models \theta|_{\overline{v}_1}(\varphi_1)$ and $\bar{\mathcal{K}}, (\pi, \alpha) \models \theta|_{\overline{v}_2}(\varphi_2)$, iff $\bar{\mathcal{K}}, (\pi, \alpha) \models \vartheta|_{\overline{v}_1}(\varphi_1)$ and $\bar{\mathcal{K}}, (\pi, \alpha) \models \vartheta|_{\overline{v}_2}(\varphi_2)$, iff $\bar{\mathcal{K}}, (\pi, \alpha) \models \vartheta(\varphi_1 \wedge \varphi_2)$.
- $\varphi = \varphi_1 \mathbf{U} \varphi_2$: Let $\overline{v}_1 = vars(\varphi_1)$ and $\overline{v}_2 = vars(\varphi_2)$. Similarly, by definition of $\varrho_{(\pi,\alpha),\varphi_1\mathbf{U}\varphi_2}$, we can easily see that for any $i \geq 0$:

  $$\varrho_{(\pi,\alpha)^i,\varphi_1}(\theta|_{\overline{v}_1}) \preceq \varrho_{(\pi,\alpha),\varphi_1\mathbf{U}\varphi_2}(\theta)|_{\overline{v}_1}, \quad \varrho_{(\pi,\alpha)^i,\varphi_2}(\theta|_{\overline{v}_2}) \preceq \varrho_{(\pi,\alpha),\varphi_1\mathbf{U}\varphi_2}(\theta)|_{\overline{v}_2}.$$

  That is, for any $\varrho_{(\pi,\alpha),\varphi_1\mathbf{U}\varphi_2}(\theta) \preceq \vartheta \preceq \theta$, $\varrho_{(\pi,\alpha)^i,\varphi_1}(\theta|_{\overline{v}_1}) \preceq \vartheta|_{\overline{v}_1} \preceq \theta|_{\overline{v}_1}$ and $\varrho_{(\pi,\alpha)^i,\varphi_2}(\theta|_{\overline{v}_2}) \preceq \vartheta|_{\overline{v}_2} \preceq \theta|_{\overline{v}_2}$ hold. Therefore, by induction hypothesis, $\bar{\mathcal{K}}, (\pi, \alpha)^i \models \theta|_{\overline{v}_1}(\varphi_1)$ iff $\bar{\mathcal{K}}_{\perp}, (\pi, \alpha)^i \models \vartheta|_{\overline{v}_1}(\varphi_1)$, and $\bar{\mathcal{K}}, (\pi, \alpha)^j \models \theta|_{\overline{v}_2}(\varphi_2)$ iff $\bar{\mathcal{K}}_{\perp}, (\pi, \alpha)^j \models \vartheta|_{\overline{v}_2}(\varphi_2)$, for each $i, j \geq 0$. And this implies that $\bar{\mathcal{K}}, (\pi, \alpha) \models \theta(\varphi_1 \mathbf{U} \varphi_2)$ iff $\bar{\mathcal{K}}_{\perp}, (\pi, \alpha) \models \vartheta(\varphi_1 \mathbf{U} \varphi_2)$. □

On the other hand, as a dual of $\varrho_{(\pi,\alpha),\varphi}$, we can also define a concretization function of an abstraction substitution $\vartheta \in [\overline{x} \to \mathcal{C}_\perp]$. Let the "glueing" $I_1 \odot I_2$ of two sets I_1 and I_2 of concrete substitutions be defined by

$$I_1 \odot I_2 = \{\theta \mid \theta|_{dom(I_1)} \in I_1,\, \theta|_{dom(I_2)} \in I_2\}.$$

Definition 4.12. *Given a parameterized LKS $\bar{\mathcal{K}}$ over a set of parameters $\mathcal{C}$, if $[\overline{x} \to \mathcal{C}]_{\succeq\vartheta}$ denotes the set $\{\theta \in [\overline{x} \to \mathcal{C}] \mid \theta \succeq \vartheta\}$, then the* concretization *function $I_{(\pi,\alpha),\varphi} : [vars(\varphi) \to \mathcal{C}_\perp] \to 2^{[vars(\varphi) \to \mathcal{C}]}$ for an LTLR formula φ and a path (π, α) is inductively defined by:*

$$\begin{aligned}
I_{(\pi,\alpha),p(\overline{x})}(\vartheta) &= \textbf{if } \vartheta \in \mathcal{D}_{\pi(0)}(p(\overline{x})) \textbf{ then } \vartheta \textbf{ else } [\overline{x} \to \mathcal{C}]_{\succeq\vartheta} - \mathcal{D}_{\pi(0)}(p(\overline{x})) \textbf{ fi} \\
I_{(\pi,\alpha),\delta(\overline{x})}(\vartheta) &= \textbf{if } \vartheta \in \mathcal{D}_{\alpha(0)}(\delta(\overline{x})) \textbf{ then } \vartheta \textbf{ else } [\overline{x} \to \mathcal{C}]_{\succeq\vartheta} - \mathcal{D}_{\alpha(0)}(\delta(\overline{x})) \textbf{ fi} \\
I_{(\pi,\alpha),\neg\varphi}(\vartheta) &= I_{(\pi,\alpha),\varphi}(\vartheta) \\
I_{(\pi,\alpha),\varphi_1 \wedge \varphi_2}(\vartheta) &= I_{(\pi,\alpha),\varphi_1}(\vartheta|_{vars(\varphi_1)}) \odot I_{(\pi,\alpha),\varphi_2}(\vartheta|_{vars(\varphi_2)}) \\
I_{(\pi,\alpha),\bigcirc\varphi}(\vartheta) &= I_{(\pi,\alpha)^1,\varphi}(\vartheta) \\
I_{(\pi,\alpha),\varphi_1 \mathbf{U} \varphi_2}(\vartheta) &= \bigcap_{i \geq 0} I_{(\pi,\alpha)^i,\varphi_1}(\vartheta|_{vars(\varphi_1)}) \;\odot\; \bigcap_{j \geq 0} I_{(\pi,\alpha)^j,\varphi_2}(\vartheta|_{vars(\varphi_2)})
\end{aligned}$$

It is easy to check that for each concrete substitution $\theta \in I_{(\pi,\alpha),\varphi}(\vartheta)$, $\vartheta \preceq \theta$ and $\varrho_{(\pi,\alpha),\varphi}(\vartheta) = \varrho_{(\pi,\alpha),\varphi}(\theta)$. The abstraction of a concrete substitution does always exist, but there may be *no* concretization for some abstract substitution. For instance, given a path (π, α) such that $\mathcal{D}_{\pi(i)}(p(x)) = \{i\}$ for each $i \geq 0$, if $\mathcal{C} = \mathbb{N}$, then $\varrho_{(\pi,\alpha),\Diamond p(x)}(\theta) = \theta$ for any $\theta \in [\{x\} \to \mathbb{N}]$, but $I_{(\pi,\alpha),\Diamond p(x)}(\perp_x) = \varnothing$. However, for a *finite* LKS satisfying FIP that has only a finite set of states and a finite set of transitions, each abstract substitution has a corresponding concrete substitution (provided $\mathcal{C}$ is an infinite set).

Lemma 4.2. *Given a* finite *parameterized LKS $\bar{\mathcal{K}}$ satisfying FIP over a set of parameters $\mathcal{C}$, a universally quantified LTLR formula $\forall\overline{x}\ \varphi$, and an abstract substitution $\vartheta \in [\overline{x} \to \mathcal{C}_\perp]$, for each path (π, α), $I_{(\pi,\alpha),\varphi}(\vartheta) \neq \varnothing$.*

Proof. It suffices to show, by structural induction on φ, that for each variable $x \in vars(\theta)$, $I_{(\pi,\alpha),\varphi}(\vartheta)|_{\{x\}}$ is *cofinite* if $\vartheta(x) = \perp$, and the singleton $\{\vartheta(x)\}$ otherwise. When $\varphi = p(\overline{x})$ or $\varphi = \delta(\overline{x})$, it is obvious by definition since $\bar{\mathcal{K}}$ satisfies FIP. The case of $\varphi = \varphi_1 \wedge \varphi_2$ comes from the fact that the intersection of two cofinite sets is cofinite. For $\varphi_1 \mathbf{U} \varphi_2$, it is enough to mention that: (i) the set of suffixes $\{(\pi, \alpha)^i \mid i \geq 0\}$ is finite when $\bar{\mathcal{K}}$ is finite, and (ii) a finite intersection of cofinite sets is cofinite. The other cases are clear by definition and the induction hypothesis. ☐

Consequently, for a *finite* LKS $\bar{\mathcal{K}}$ satisfying FIP, we can determine the satisfaction of $\forall\overline{x}\ \varphi$ by using a (possibly small) finite set $\mathscr{R}$ of substitutions.

Theorem 4.2. *Given a* finite *parameterized LKS* $\bar{\mathcal{K}}$ *satisfying FIP over a set of parameters* $\mathcal{C}$*, a universally quantified LTLR formula* $\forall\overline{x}\ \varphi$*, and a path* (π,α)*, for any set* $\varrho_{(\pi,\alpha),\varphi}([\overline{x}\to\mathcal{C}]) \subseteq \mathscr{R} \subseteq [\overline{x}\to\mathcal{C}_\perp]$ *of substitutions:*

$$\bar{\mathcal{K}},(\pi,\alpha) \models \forall\overline{x}\ \varphi \iff (\forall\vartheta\in\mathscr{R})\ \bar{\mathcal{K}}_\perp,(\pi,\alpha)\models\vartheta(\varphi).$$

Proof. First, $\bar{\mathcal{K}},(\pi,\alpha) \models \forall\overline{x}\ \varphi$ iff $\bar{\mathcal{K}},(\pi,\alpha)\models\theta(\varphi)$ for each $\theta\in[\overline{x}\to\mathcal{C}]$, and by Lemma 4.1, iff $\bar{\mathcal{K}}_\perp,(\pi,\alpha)\models\vartheta(\varphi)$ for each $\vartheta\in\varrho_{(\pi,\alpha),\varphi}([\overline{x}\to\mathcal{C}])$. That is, $\bar{\mathcal{K}},(\pi,\alpha)\models\forall\overline{x}\ \varphi$ iff $(\forall\vartheta\in\varrho_{(\pi,\alpha),\varphi}([\overline{x}\to\mathcal{C}]))\ \bar{\mathcal{K}}_\perp,(\pi,\alpha)\models\vartheta(\varphi)$. Next, by Lemma 4.2, if $\vartheta\in[\overline{x}\to\mathcal{C}_\perp]-\varrho_{(\pi,\alpha),\varphi}([\overline{x}\to\mathcal{C}])$, then there exists a concrete substitution $\theta\in[\overline{x}\to\mathcal{C}]$ such that $\varrho_{(\pi,\alpha),\varphi}(\vartheta)=\varrho_{(\pi,\alpha),\varphi}(\theta)$, which implies $\bar{\mathcal{K}}_\perp,(\pi,\alpha)\models\vartheta(\varphi)$ iff $\bar{\mathcal{K}},(\pi,\alpha)\models\theta(\varphi)$. ☐

Such a *path-realized* set $\mathscr{R}$ satisfying $\varrho_{(\pi,\alpha),\varphi}([\overline{x}\to\mathcal{C}])\subseteq\mathscr{R}\subseteq[\overline{x}\to\mathcal{C}_\perp]$ can be easily constructed from a given path (π,α). For two sets I_1 and I_2 of substitutions, let $I_1\oplus I_2=\{\theta_1\oplus\theta_2 \mid \theta_1\in I_1,\theta_2\in I_2\}$ (recall that $\theta_1\oplus\theta_2$ is the combined substitution of θ_1 and θ_2, as defined in Definition 4.10).

Definition 4.13. *Given a parameterized LKS* $\bar{\mathcal{K}}$ *over a set of parameters* $\mathcal{C}$ *and a universally quantified LTLR formula* $\forall\overline{x}\ \varphi$*, the* path-realized *set* $\mathscr{R}_{(\pi,\alpha),\varphi}\subseteq[vars(\varphi)\to\mathcal{C}_\perp]$ *for a path* (π,α) *is defined by:*

$$\begin{aligned}
\mathscr{R}_{(\pi,\alpha),p(\overline{x})} &= \mathcal{D}_{\pi(0)}(p(\overline{x}))\cup\{\perp_{\overline{x}}\}\\
\mathscr{R}_{(\pi,\alpha),\delta(\overline{x})} &= \mathcal{D}_{\alpha(0)}(\delta(\overline{x}))\cup\{\perp_{\overline{x}}\}\\
\mathscr{R}_{(\pi,\alpha),\neg\varphi} &= \mathscr{R}_{(\pi,\alpha),\varphi}\\
\mathscr{R}_{(\pi,\alpha),\varphi_1\wedge\varphi_2} &= \mathscr{R}_{(\pi,\alpha),\varphi_1}\oplus\mathscr{R}_{(\pi,\alpha),\varphi_2}\\
\mathscr{R}_{(\pi,\alpha),\bigcirc\varphi} &= \mathscr{R}_{(\pi,\alpha)^1,\varphi}\\
\mathscr{R}_{(\pi,\alpha),\varphi_1\mathbf{U}\varphi_2} &= \bigcup_{i\geq 0}\mathscr{R}_{(\pi,\alpha)^i,\varphi_1}\ \oplus\ \bigcup_{j\geq 0}\mathscr{R}_{(\pi,\alpha)^j,\varphi_2}
\end{aligned}$$

Notice that $\mathscr{R}_{(\pi,\alpha),\varphi}$ is guaranteed to be finite if the underlying LKS $\bar{\mathcal{K}}$ is *finite* and satisfies FIP. Since $\mathscr{R}_{(\pi,\alpha),\varphi}$ is the aggregation of all possible values of $\varrho_{(\pi,\alpha),\varphi}$, by Theorem 4.2, we have the following *localization lemma*, stating that $\mathscr{R}_{(\pi,\alpha),\varphi}$ is a *complete* set of abstract substitutions.

Lemma 4.3. *Given a* finite *parameterized LKS* $\bar{\mathcal{K}}$ *satisfying FIP over* $\mathcal{C}$*, a universally quantified LTLR formula* $\forall\overline{x}\ \varphi$*, and a path* (π,α)*:*

$$(\forall\theta\in[\overline{x}\to\mathcal{C}],\exists\vartheta\in\mathscr{R}_{(\pi,\alpha),\varphi})\ \bar{\mathcal{K}},(\pi,\alpha)\models\theta(\varphi) \iff \bar{\mathcal{K}}_\perp,(\pi,\alpha)\models\vartheta(\varphi).$$

4.3.2 Automata-based Characterization

The satisfiability of a universally quantified LTLR formula $\forall\overline{x}\ \varphi$ for a finite LKS $\bar{\mathcal{K}}$ satisfying FIP is now reduced to the satisfiability of $\vartheta\varphi$ on the LKS $\bar{\mathcal{K}}_{\bot}$ for each path-realized substitution $\vartheta \in \mathscr{R}_{(\pi,\alpha),\varphi}$. To model check a propositional temporal formula ϕ under parameterized fairness assumptions $\forall\overline{y}_1\ \psi_1, \dots, \forall\overline{y}_m\ \psi_m$, we can just consider the formula $(\widehat{\psi_1} \wedge \cdots \wedge \widehat{\psi_m}) \to \phi$, where $\widehat{\psi_i}$ is the conjunction of all path-realized instances of ψ_i. This method can also be applied to verify a more general class of parameterized LTLR formulas $\forall\overline{x}\ \varphi$. However, it is *not* on-the-fly, since constructing such a formula requires to traverse the entire (reachable) state space. Furthermore, the exponential blowup in generating the Büchi automaton for the formula $(\widehat{\psi_1} \wedge \cdots \wedge \widehat{\psi_m}) \to \phi$ can easily make such a generation unfeasible.

We can however have an efficient *on-the-fly* algorithm to model check a formula φ under parameterized fairness assumptions of the forms

$$(\forall\overline{y})\ \Box\Diamond\Phi \to \Box\Diamond\Psi, \qquad (\forall\overline{y})\ \Diamond\Box\Phi \to \Box\Diamond\Psi,$$

where Φ and Ψ are Boolean formulas with no temporal operators and their atoms are state propositions or spatial action patterns. The satisfaction of such a parameterized fairness formula $\forall\overline{y}\ \psi$ does not vary if we skip finitely many steps of a path. Therefore, by Lemma 4.3, we can consider only the set $\mathscr{R}^{inf}_{(\pi,\alpha),\psi}$ of *infinitely often* path-realized substitutions, given by:

$$\mathscr{R}^{inf}_{(\pi,\alpha),\psi} = \{\vartheta \in \mathscr{R}_{(\pi,\alpha),\psi} \mid \vartheta \in \mathscr{R}_{(\pi,\alpha)^i,\psi} \text{ for infinitely many } i \in \mathbb{N}\},$$

which is identical to $\mathscr{R}_{(\pi,\alpha)^N,\psi}$ for a sufficiently large number $N \in \mathbb{N}$ by which all substitutions with finite occurrences are skipped. Accordingly:

Corollary 4.2. *Given a finite LKS $\bar{\mathcal{K}}$ satisfying FIP over $\mathcal{C}$, a parameterized fairness formula $\forall\overline{y}\ \psi$, and a path (π,α), for any $\mathscr{R}^{inf}_{(\pi,\alpha),\psi} \subseteq \mathscr{R} \subseteq [\overline{x} \to \mathcal{C}_{\bot}]$, $\bar{\mathcal{K}}, (\pi,\alpha) \models \forall\overline{x}\ \psi$ iff $(\forall\vartheta \in \mathscr{R})\ \bar{\mathcal{K}}_{\bot}, (\pi,\alpha) \models \vartheta(\psi)$.*

Therefore, from a set of parameterized fairness formulas, we can construct an equivalent set $\mathcal{G}$ of *propositional* fairness formulas, by instantiating each parameterized fairness formula $\forall\overline{x}\ \psi$ with the path-realized substitutions in $\mathscr{R}^{inf}_{\psi}$, the union of $\mathscr{R}^{inf}_{(\pi,\alpha),\psi}$ for each path (π,α) in $\bar{\mathcal{K}}$. Since a weak fairness formula $\Diamond\Box\Phi \to \Box\Diamond\Psi$ can be expressed as an equivalent strong fairness formula $\Box\Diamond\mathit{True} \to \Box\Diamond(\neg\Phi \vee \Psi)$, we can regards $\mathcal{G}$ as a set of strong fairness formulas. Such strong fairness conditions can be incorporated into the acceptance conditions of a transition-based *Streett* automaton.

Definition 4.14. *A* Streett automaton *is a 5-tuple* $\mathcal{S} = (Q, Q_0, P, \Delta, \mathcal{F})$, *where* Q *is a finite set of states,* $Q_0 \subseteq Q$ *is a set of initial states,* P *is an alphabet of transition labels,* $\Delta \subseteq Q \times P \times Q$ *is a labeled transition relation, and* $\mathcal{F} \subseteq 2^{\Delta \times \Delta}$ *is an acceptance condition.*

A run $\sigma = q_0 \xrightarrow{l_0} q_1 \xrightarrow{l_1} q_2 \xrightarrow{l_2} \cdots$ is *accepted* by $\mathcal{S}$, where $q_0 \in Q_0$, iff for each pair $(G, H) \in \mathcal{F}$, whenever σ has transitions in G infinitely many times, σ has transitions in H infinitely many times.

Definition 4.15. *Given an LKS* $\bar{\mathcal{K}} = (S, AP, \mathcal{L}, ACT, \rightarrow_{\bar{\mathcal{K}}})$*, a set of initial states* $S_0 \subseteq S$ *of* $\bar{\mathcal{K}}$*, and a set of propositional strong fairness formulas* $\mathcal{G} = \{\Box\Diamond\Phi_i \rightarrow \Box\Diamond\Psi_i \mid 1 \leq i \leq f\}$*, we can construct the fair Streett automaton* $\mathcal{S}^{\mathcal{G}}(\bar{\mathcal{K}}[S_0]) = (S, S_0, 2^{AP \uplus ACT}, \Delta, \mathcal{F}^{\mathcal{G}})$ *such that:*[7]

$$\Delta = \{s \xrightarrow{\mathcal{L}(s)\uplus\Lambda} s' \mid s \xrightarrow{\Lambda}_{\bar{\mathcal{K}}} s'\}$$
$$\mathcal{F}^{\mathcal{G}} = \{(\Delta^{\Phi_i}, \Delta^{\Psi_i}) \mid 1 \leq i \leq f\}, \quad \textit{where } \Delta^{\Phi} = \{s \xrightarrow{B} s' \in \Delta \mid B \models \Phi\}.$$

Each path (π, α) of an LKS $\bar{\mathcal{K}}$ is in one-to-one correspondence with the run $\pi(0) \xrightarrow{\mathcal{L}(\pi(0))\uplus\alpha(0)} \pi(1) \xrightarrow{\mathcal{L}(\pi(1))\uplus\alpha(1)} \pi(2) \xrightarrow{\mathcal{L}(\pi(2))\uplus\alpha(2)} \cdots$ of the Streett automaton $\mathcal{S}^{\mathcal{G}}(\bar{\mathcal{K}}[S_0])$. Also, (π, α) satisfies all fairness conditions in $\mathcal{G}$ iff the corresponding run of (π, α) is accepted by $\mathcal{S}^{\mathcal{G}}(\bar{\mathcal{K}}[S_0])$. Therefore, to verify an LTLR formula φ under parameterized fairness conditions, if $\mathscr{B}_{\neg\varphi} = (Q, Q_0, 2^{AP \uplus ACT}, \delta, F)$,[8] then we can make use of the product Streett automaton $\mathcal{S}^{\mathcal{G}}(\bar{\mathcal{K}}[S_0]) \times \mathscr{B}_{\neg\varphi} = (S \times Q, S_0 \times Q_o, 2^{AP \uplus ACT}, \Delta_\delta, \mathcal{F}^{\mathcal{G}}_F)$, where $(s, b) \xrightarrow{\mathcal{L}(s)\uplus\Lambda} (s', b') \in \Delta_\delta$ iff $s \xrightarrow{\mathcal{L}(s)\uplus\Lambda} s' \in \Delta$ and $b \xrightarrow{\mathcal{L}(s)\uplus\Lambda} b' \in \delta$, and $\mathcal{F}^{\mathcal{G}}_F = \{(G \times \delta, H \times \delta) \mid (G, H) \in \mathcal{F}^{\mathcal{G}}\} \cup \{(\Delta \times \delta, \Delta \times F)\}$.

Theorem 4.3. *Given a finite LKS* $\bar{\mathcal{K}}$ *satisfying FIP, an LTLR formula* φ*, and a set* $\mathscr{G}$ *of parameterized fairness formulas, there exists a Streett automaton* $\mathcal{S} = \mathcal{S}^{\mathcal{G}}(\bar{\mathcal{K}}[S_0]) \times \mathscr{B}_{\neg\varphi}$ *such that* $L(\mathcal{S}) = \varnothing \iff \bar{\mathcal{K}}, S_0 \models_{\mathcal{G}} \varphi$, $\mathcal{G} = \{\vartheta(\psi) \mid \forall\bar{y}\ \psi \in \mathscr{G},\ \vartheta \in \mathscr{R}^{inf}_{\psi}\}$*, and* $|\mathcal{S}| = O(|\bar{\mathcal{K}}| \cdot 2^{|\varphi|})$.

It is worth noting that without parameter abstraction, a naive selection of such instantiated fairness conditions does not guarantee the equivalence with the parameterized fairness conditions. For example, in a concurrent object system, the fairness formula $(\forall x)\ \Box\Diamond\neg \mathit{enabled}(x) \rightarrow \Box\Diamond \mathit{execute}(x)$ is always false, since $\mathit{execute}(o)$ is false for any "nonexistent" object o in the system. But using only the *existing* objects in the system, their instantiated formulas can all be true if such objects are always enabled.

[7] $B \models \Phi$ is defined inductively as follows: $B \models p$ iff $p \in B$, $B \models \delta$ iff $\delta \in B$, $B \models \neg\Phi$ iff $B \not\models \Phi$, and $B \models \Phi_1 \wedge \Phi_2$ iff $B \models \Phi_1$ and $B \models \Phi_2$, where $p \in AP$ and $\delta \in ACT$.

[8] We here consider a *transition-based* Büchi automaton $\mathscr{B}$ [66] in which $F \subseteq \delta$ is a set of accepting *transitions*, instead of accepting states.

```
findFairSCC(Q, Q_0, Δ)
2  while there is a reachable state q ∈ Q from Q_0 that has not been visited do
3    𝔖 := computeNextSCC(Q, q, Δ);
4    if fairnessSatisfied(𝔖) then
5      return 𝔖
6    else if 𝔖 is maximal and contains bad transitions then
7      Q^𝔖 := the set of states in 𝔖;
8      Λ^𝔖 := the set of bad transitions for unsatisfied acceptance conditions in 𝔖;
9      Q_0^𝔖 := the set of states that occur in Λ^𝔖;
10     mark each state in Q^𝔖 as unvisited;
11     return findFairSCC(Q^𝔖, {q} ∪ Q_0^𝔖, Δ − Λ^𝔖) unless ⊥
12   end if
13 end while;
14 return ⊥;
```

Figure 4.1: Streett Emptiness Checking Algorithm for $\mathcal{S} = (Q, Q_0, P, \Delta, \mathcal{F})$

4.3.3 On-The-Fly Model Checking Algorithm

This section presents an *on-the-fly* model checking algorithm to verify an LTLR formula φ of an LKS $\bar{\mathcal{K}}$ under parameterized fairness assumptions, based on checking the emptiness of the product Streett automaton

$$\mathcal{S} = \mathcal{S}^{\mathcal{G}}(\bar{\mathcal{K}}[S_0]) \times \mathscr{B}_{\neg\varphi}.$$

To check the emptiness of the Streett automaton $\mathcal{S}$, the basic idea is to find a reachable strongly connected component (SCC) satisfying all the acceptance conditions of $\mathcal{S}$ [83]. An acceptance condition (Φ_i, Ψ_i) is satisfied in a SCC $\mathfrak{S}$ iff whenever $\mathfrak{S}$ contains a transition $s_1 \xrightarrow{B} s_2$ such that $B \models \Phi_i$, there exists some transition $s'_1 \xrightarrow{B'} s'_2 \in \mathfrak{S}$ such that $B' \models \Psi_i$. If some (Φ_i, Ψ_i) is not satisfied in $\mathfrak{S}$, then there exists some *bad* transitions of $\mathfrak{S}$ that satisfy $\Phi_i \wedge \neg\Psi_i$ and therefore prevent the satisfaction of (Φ_i, Ψ_i).

The emptiness checking algorithm specified in Fig. 4.1 is to find a SCC with no bad transitions. The *computeNextSCC*(Q, q, Δ) function in Line 3 identifies each SCC $\mathfrak{S}$ in the graph (Q, Δ) containing the state q, which can be implemented by using any *on-the-fly* algorithm to find a SCC, such as Tarjan's algorithm [161] or Couvreur's algorithm [66]. If $\mathfrak{S}$ satisfies all the acceptance conditions (Line 4), then we can generate a counterexample given by a fair cycle from $\mathfrak{S}$ using breadth-first search [122]. Otherwise, $\mathfrak{S}$ must include some bad transitions. If $\mathfrak{S}$ is a *maximal* SCC (MSCC) so that there exists no other SCC strictly containing $\mathfrak{S}$, then the whole $\mathfrak{S}$ is traversed again, *except for* the bad transitions $\Lambda^{\mathfrak{S}}$ (Line 11), which leads to dividing $\mathfrak{S}$ into multiple smaller subcomponent with no such bad transitions.

Theorem 4.4. *Given a Streett Automaton $\mathcal{S} = (Q, Q_0, P, \Delta, \mathcal{F})$, assuming the correctness of an underlying SCC finding algorithm, if there exists a reachable nonempty SCC satisfying $\mathcal{F}$, then findFairSCC(Q, Q_0, Δ) finds it.*

Proof. Suppose that there exists a nonempty SCC $\mathfrak{T} \in (Q, \Delta)$ satisfying all the acceptance conditions in $\mathcal{F}$ and being reachable from Q_0. Because the underlying SCC finding algorithm is correct, $\mathfrak{T}$ is a subcomponent of a MSCC $\mathfrak{S} = (Q^{\mathfrak{S}}, \Delta^{\mathfrak{S}})$ given by *computeNextSCC*(Q, q, Δ) with a reachable state q from Q_0. Furthermore, since $\mathfrak{T}$ does not contain any bad transitions $\Lambda^{\mathfrak{S}}$ of $\mathfrak{S}$, we have $\mathfrak{T} \subseteq (Q^{\mathfrak{S}}, \Delta^{\mathfrak{S}} - \Lambda^{\mathfrak{S}})$. This means that whenever we meet *findFairSCC*$(Q^{\mathfrak{S}}, \{q\} \cup Q_0^{\mathfrak{S}}, \Delta - \Lambda^{\mathfrak{S}})$ in Line 11, $\mathfrak{T}$ is contained in $(Q^{\mathfrak{S}}, \Delta^{\mathfrak{S}} - \Lambda^{\mathfrak{S}})$ and is reachable from $\{q\} \cup Q_0^{\mathfrak{S}}$. □

In order to make this algorithm on-the-fly under parameterized fairness conditions, we have to check *fairnessSatisfied*$(\mathfrak{S})$ in Line 4 using only states and transitions in $\mathfrak{S}$, without generating all instantiated fairness conditions for the entire system. Given parameterized fairness formulas $\forall \overline{x}_i\ \psi_i$, with $\psi_i = \Box\Diamond\Phi_i \to \Box\Diamond\Psi_i$,[9] since Φ_i and Ψ_i have no temporal operators:

$$\mathscr{R}_{(\pi,\alpha)^k,\Phi_i} = \bigoplus_{p(\overline{y})\in\Phi_i} (\mathcal{D}_{\pi(k)}(p(\overline{y})) \cup \{\bot_{\overline{y}}\}) \oplus \bigoplus_{\delta(\overline{y})\in\Phi_i} (\mathcal{D}_{\alpha(k)}(\delta(\overline{y})) \cup \{\bot_{\overline{y}}\}).$$

Therefore, for any path (π, α) whose infinite suffix is included in $\mathfrak{S}$, the infinitely often path-realized set $\mathscr{R}^{inf}_{(\pi,\alpha),\psi_i}$ is a subset of the set $\mathscr{R}_{\mathfrak{S},\psi_i}$:

$$\bigcup_{s \xrightarrow{\Lambda}_{\bar{\mathcal{K}}} s' \in \mathfrak{S}} \left(\begin{array}{l} \bigoplus_{p(\overline{y})\in\Phi_i} (\mathcal{D}_s(p(\overline{y})) \cup \{\bot_{\overline{y}}\}) \oplus \bigoplus_{\delta(\overline{y})\in\Phi_i} (\mathcal{D}_\Lambda(\delta(\overline{y})) \cup \{\bot_{\overline{y}}\}) \oplus \\ \bigoplus_{p(\overline{y})\in\Psi_i} (\mathcal{D}_s(p(\overline{y})) \cup \{\bot_{\overline{y}}\}) \oplus \bigoplus_{\delta(\overline{y})\in\Psi_i} (\mathcal{D}_\Lambda(\delta(\overline{y})) \cup \{\bot_{\overline{y}}\}) \end{array} \right)$$

Thanks to Corollary 4.2, we only need to check fairness instances of ψ_i by $\mathscr{R}_{\mathfrak{S},\psi_i}$ to determine the satisfaction of $\forall \overline{x}_i\ \psi_i$. That is, *fairnessSatisfied*$(\mathfrak{S})$ can be computed on-the-fly by using only states and transitions in $\mathfrak{S}$.

Given a finite LKS $\bar{\mathcal{K}}$ satisfying FIP, an LTLR formula φ, and a set $\mathscr{G}$ of parameterized fairness formula, the time complexity of the algorithm is $O(|\mathscr{G}| \cdot f \cdot |\bar{\mathcal{K}}| \cdot 2^{|\varphi|})$, where $f = \max_{\mathfrak{S},\psi_i\in\mathscr{G}} \mathscr{R}_{\mathfrak{S},\psi_i}$ is the maximum number of the path-realized substitutions for a parameterized fairness formula of a single MSCC in the system, bounded by the total number of infinitely often path-realized substitutions. The space complexity is also exponential in $|\varphi|$, since in the worst case the whole state space can be a single SCC maintained by the underlying Streett emptiness checking algorithm.

[9] Recall that $\Diamond\Box\Phi_i \to \Box\Diamond\Psi_i \equiv \Box\Diamond\, True \to \Box\Diamond(\neg\Phi_i \vee \Psi_i)$.

4.4 The Maude Fair LTLR Model Checker

This section presents the Maude Fair LTLR model checker,[10] developed as an extension of the Maude LTLR model checker in Chapter 3. Our tool has comparable performance to other explicit-state model checkers such as SPIN [108] and PAT [158], and is the first tool we are aware of which can model check LTLR properties under localized fairness assumptions.

4.4.1 The Model Checker Interface

The Maude Fair LTLR model checker extends the predefined system module `LTLR-MODEL-CHECKER` (see Section 3.4.2) that defines the main functionality of the LTLR model checker. In particular, for LTLR model checking under localized fairness assumptions, the function

```
modelCheckFair : State Formula FairnessSet ~> ModelCheckResult
```

takes an initial state, an LTLR formula, and a semicolon-separated set of strong/weak fairness conditions, and returns either `true`—if the formula is satisfied—or a counterexample. Each fairness condition is a (parametric) term of sort Fairness of the forms:

$$\texttt{fair}(\delta(\overline{y})),\quad \texttt{just}(\delta(\overline{y})),\quad \texttt{fair} : \Phi(\overline{y}) \texttt{ => } \Psi(\overline{z}),\quad \texttt{just} : \Phi(\overline{y}) \texttt{ => } \Psi(\overline{z}),$$

where $\delta(\overline{y})$ is a spatial action pattern, and $\Phi(\overline{y})$ and $\Psi(\overline{z})$ are any Boolean formulas *without* temporal operators, involving (possibly parametric) state propositions and spatial action patterns. Fairness conditions $\texttt{fair}(\delta(\overline{y}))$ and $\texttt{just}(\delta(\overline{y}))$, respectively, are shorthands for the fairness formulas

$$(\forall\overline{y})\ \Box\Diamond enabled(\delta(\overline{y})) \rightarrow \Box\Diamond\delta(\overline{y}) \quad (\forall\overline{y})\ \Diamond\Box enabled(\delta(\overline{y})) \rightarrow \Box\Diamond\delta(\overline{y}).$$

For example, the following command returns the model checking result of the formula $\Box\neg deadlock \rightarrow \Diamond eating(0)$ under the given parameterized fairness assumptions for the dining philosophers example in Section 3.4.2:

```
Maude> red modelCheckFair(init, []~ deadlock -> <> eating(0),
                          just({'wake : 'I \ I:Nat}) ;
                          fair({'grabF : 'I \ I:Nat}) ;
                          fair({'grabS : 'I \ I:Nat})) .
result Bool: true
```

[10] The tool is available at `http://maude.cs.illinois.edu/tools/tlr`.

Likewise, fairness conditions `fair` : $\Phi(\overline{y})$ `=>` $\Psi(\overline{z})$ and `just` : $\Phi(\overline{y})$ `=>` $\Psi(\overline{z})$, respectively, are shorthands for the strong and weak fairness formulas:

$$(\forall \overline{y} \cup \overline{z})\ \Box\Diamond\Phi(\overline{y}) \to \Box\Diamond\Psi(\overline{z}), \qquad (\forall \overline{y} \cup \overline{z})\ \Diamond\Box\Phi(\overline{y}) \to \Box\Diamond\Psi(\overline{z}).$$

They can be useful when some fairness conditions cannot be expressed by rule annotations.[11] The Maude syntax for fairness conditions is as follows:

```
sorts Fairness FairnessType .
ops fair just : -> FairnessType .
op _:_=>_ : FairnessType Formula Formula -> Fairness [ctor] .

ops fair just : Action -> Fairness .
eq fair(A:Action) = fair : enabled(A:Action) => A:Action .
eq just(A:Action) = just : enabled(A:Action) => A:Action .

sort FairnessSet .           subsort Fairness < FairnessSet .
op noFairness : -> FairnessSet [ctor] .
op _;_ : FairnessSet FairnessSet
      -> FairnessSet [ctor comm assoc id: noFairness] .
```

In our tool, localized fairness assumptions with *basic action patterns* can be simply declared by rule annotations, and can be applied for LTLR model checking using the Full Maude interface. A localized fairness specification $(\mathcal{J}, \mathcal{F})$ of a system module is given by a *metadata* attribute `metadata "..."` for each rule, which contains a semicolon-separated list of fairness items. For a rule $l : q \longrightarrow r$ **if** *cond*, a fairness item $\mathit{just}(x_1, \ldots, x_n)$ declares the basic action pattern $\texttt{\{}'l : 'x_1 \backslash x_1; \ldots ; 'x_n \backslash x_n\texttt{\}}$ in the weak fairness specification $\mathcal{J}$, where $x_1, \ldots, x_n \in \mathit{vars}(q)$. Similarly, a fairness item $\mathit{fair}(x_1, \ldots, x_n)$ declares the basic action pattern $\texttt{\{}'l : 'x_1 \backslash x_1; \ldots ; 'x_n \backslash x_n\texttt{\}}$ in the strong fairness specification $\mathcal{F}$. For example, the localized fairness specification $(\mathcal{J} = \{\texttt{\{}'\mathit{wake} : 'I \backslash I\texttt{\}}\}, \mathcal{F} = \{\texttt{\{}'\mathit{grabF} : 'I \backslash I\texttt{\}}, \texttt{\{}'\mathit{grabS} : 'I \backslash I\texttt{\}}\})$ for the dining philosophers model in Example 4.1 can be succinctly represented by the metadata rule attributes as follows:

```
 rl [wake]: p(I,think) => p(I,wait0)        [metadata "just(I)"] .
crl [grabF]: p(I,wait0) || c(J) => p(I,wait1)
   if adj(I,J) = true                       [metadata "fair(I)"] .
crl [grabS]: p(I,wait1) || c(J) => p(I,eat)
   if adj(I,J) = true                       [metadata "fair(I)"] .
 rl [stop]: p(I,eat) => p(I,think) || c(lc(I)) || c(rc(I)) .
```

[11] We may have objects a, b, c, d, and e, but we may only want to specify fairness requirements for a, c, and e, but not for b and d. Or $\Phi(\overline{y})$ and $\Psi(\overline{z})$ may not correspond to the fairness requirements for a rule application and may specify other fairness properties.

The Full Maude interface provides several commands for LTLR model checking under localized fairness specifications. Given a state t, an LTLR formula φ, and a localized fairness specification $(\mathcal{J}, \mathcal{F})$ declared using rule attributes, the parameterized-fair model checking command

(`pfmc` t `|=` φ `.)`

model checks φ from the initial state t under the parameterized fairness assumptions $(\mathcal{J}, \mathcal{F})$. For the dining philosophers example, the following command returns the model checking result of $\Box\neg \mathit{deadlock} \rightarrow \Diamond \mathit{eating}(0)$ under the above localized fairness specification:

```
Maude> (pfmc init |= []~ deadlock -> <> eating(0) .)
ltlr model check under localized fairness in DINING-PHILOS-PROP:
  init |= []~ deadlock -> <> eating(0)
result Bool :
  true
```

There also exists a model checking command (`mc` t `|=` φ `.)` that does not consider a localized fairness specification by rule annotations.

In addition, each model checking command allows the user to specify additional fairness conditions as follows, which can be useful when some fairness conditions cannot be expressed by rule annotations:

(`pfmc` t `|=` φ `under` *FairnessSet* `.)`
(`mc` t `|=` φ `under` *FairnessSet* `.)`

Notice that the Full Maude command (`mc` t `|=` φ `under` *FairnessSet* `.)` and the Core Maude command '`red modelCheck(t,`φ`,`*FairnessSet*`) .`' are equivalent to each other. E.g., for the dining philosophers example:

```
Maude> (mc init |= []~ deadlock -> <> eating(0)  under
         just({'wake : 'I \ I:Nat}) ;
         fair({'grabF : 'I \ I:Nat}) ;
         fair({'grabS : 'I \ I:Nat}) .)
ltlr model check in DINING-PHILOS-PROP :
  init |= <> answer(b)
under fairness :
  just({'wake : 'I \ I:Nat}) ;
  fair({'grabF : 'I \ I:Nat}) ;
  fair({'grabS : 'I \ I:Nat})
result Bool :
  true
```

Fairness	N	Weak Only (False)		Strong/Weak (True)	
		States	Time	States	Time
MAUDE	6	913	< 0.1	5777	1.8
	7	2418	0.1	24475	11.5
	8	11092	0.9	103681	77.6
PAT	6	1596	1.0	18101	3.9
	7	5718	5.1	69426	16.1
	8	21148	33.5	260998	79.0
SPIN	6	672	< 0.1	> 30 minutes	
	7	2765	0.2		
	8	9404	0.8		

Table 4.1: Dining philosophers for the property $\Box\neg\mathit{deadlock} \to \Diamond\mathit{eating}(1)$

4.4.2 Experimental Results

In this section we compare the performance of the Maude Fair LTLR model checker with two other explicit-state model checkers PAT [158] and SPIN [108]. Since they only support unparameterized fairness, the comparison with those tools used a model with unparameterized fairness assumptions. We used the classical dining philosophers problem described in Example 2.5, which requires both strong and weak fairness conditions to verify the liveness property $\Box\neg\mathit{deadlock} \to \Diamond\mathit{eating}(1)$. This model is specified as the Maude model in Example 2.10, as the following Promela model for SPIN:

```
bit fork[N];
proctype phil(int id) {
    think: atomic {fork[id] == 0 -> fork[id] = 1; }
    one:   atomic {fork[(id+1)%N] == 0 -> fork[(id+1)%N] = 1; }
    eat:   fork[(id+1)%N] = 0; fork[id] = 0; goto think;
}
```

and as the following CSP model for PAT (borrowed from the PAT manual):

```
Phil(i) = get.i.(i+1)%N -> get.i.i -> eat.i
       -> put.i.(i+1)%N -> put.i.i -> Phil(i);
Fork(x) = get.x.x -> put.x.x -> Fork(x)
       [] get.(x-1)%N.x -> put.(x-1)%N.x -> Fork(x);
College() = ||x:{0..N-1}@(Phil(x) || Fork(x));
```

Table 4.1 shows the verification results, where "N" denotes the number of philosophers and "Time" is the running time in seconds. The experiment was conducted on an Intel Core 2 Duo 2.66 GhZ with 8GB RAM running Mac OS X 10.6. We can observe that in the weak-fairness cases, our tool is comparable to SPIN, and for the strong/weak fairness cases, it shows similar performance as PAT. For SPIN, we had to encode strong fairness conditions into the LTL formula since SPIN only supports weak fairness.

4.5 Case Study: the Evolving Dining Philosophers

This section shows how annotated rewrite theories can naturally specify localized fairness assumptions (see Appendix A.1 for more examples). The evolving dining philosophers problem [117] is similar to the classical dining philosophers problem, described in Example 2.5. However, in the evolving version, a philosopher can join or leave the table, so that the number of philosophers can change dynamically. Therefore, it can be difficult to specify exact fairness conditions *before* exploring the entire (reachable) state space, because we cannot know how many philosophers will appear, but the fairness conditions depend on *each* philosopher in the system.

In this model, a philosopher is represented as a term `ph(I,S,C)` with `I` the philosopher's id, `S` the philosopher's status, and `C` the number of chopsticks held. A philosopher holding two chopsticks is considered to be always *eating*. Likewise, a chopstick with id `I` is represented as a term `stk(I)`. The configuration of the system is described by a *set* of philosophers and chopsticks, built by an associative-commutative set union operator `_;_`. The state of the system is represented as a triple `< P, N, CF >` of sort `PState`, where `P` is a global counter regarding the dynamic behavior of the system, `N` is the current number of philosophers, and `CF` is a *set* of philosophers and chopsticks. The signature is defined by the following functional module:

```
fmod PHILO-SYNTAX is
  protecting NAT .          sorts Philo Status Chopstick .
  op ph : Nat Status Nat -> Philo [ctor] .
  ops think hungry : -> Status [ctor] .
  op stk : Nat -> Chopstick [ctor] .
  sorts Conf PState .       subsorts Philo Chopstick < Conf .
  op none : -> Conf [ctor] .
  op _;_ : Conf Conf -> Conf [ctor comm assoc id: none] .
  op <_,_,_> : Nat Nat Conf -> PState [ctor] .
endfm
```

With a fixed number `N` of philosophers, the behavior of philosophers is given by the following system module that also declares the localized fairness specification ($\mathcal{J} = \{\{'wake : 'I \backslash I\}\},\ \mathcal{F} = \{\{'grab : 'I \backslash I\}\}$), meaning that: (i) if a philosopher `I` is continuously able to wake up, then the philosopher must wake up infinitely often, and (ii) if a philosopher `I` can grab a chopstick infinitely often, then the philosopher must grab a chopstick infinitely many times. The function `left(I)` returns the chopstick's id on the left-hand side of the philosopher `I`, and the function `right(I,N)` returns the chopstick's id on the right-hand side of the philosopher `I`:

```
mod PHILO-STATIC is
  protecting PHILO-SYNTAX .
  vars P N C : Nat .          var I J : NzNat .     var CF : Conf .
  op left : Nat -> Nat .      op right : Nat Nat -> Nat .
  eq left(I) = I .            eq right(I, N) = s(I rem N) .
  rl [wake]: ph(I,think,0) => ph(I,hungry,0)  [metadata "just(I)"] .
 crl [grab]: < P, N, ph(I,hungry,C) ; stk(J) ; CF >
         => < P, N, ph(I,hungry,C + 1) ; CF >
    if J == left(I) or J == right(I,N)        [metadata "fair(I)"] .
  rl [stop]: < P, N, ph(I,hungry,2) ; CF >
         => < P, N, ph(I,think,0) ;
                    stk(left(I)) ; stk(right(I,N)) ; CF > .
endm
```

We now specify the dynamic behavior of philosophers in the evolving dining philosopher problem. Although there is no limit to the number of philosophers in the original problem, we can give an unpredictable bound using the Collatz conjecture [67]. The counter `P` in the state symbolizes a philosophical problem, and philosophers keep thinking the problem by changing the number `P` to: (i) $3 \cdot \texttt{P} + 1$ for `P` odd, or (ii) `P`/2 for `P` even. For the current number `N` of philosophers, new philosophers can join the group only if `P` is divided by $4 \cdot \texttt{N}$, or `N` is a multiple of 3.[12] No more philosophers can join after the number `P` eventually goes to 1. We assume that only the last philosopher can leave the group for simplicity. To keep consistency, whenever a philosopher joins or leaves the table, the related chopsticks should not be held by another philosopher.

```
mod PHILO-DYNAMIC is
  protecting PHILO-STATIC .  vars P N C : Nat .
  op collatz : Nat -> Nat .  vars I J : NzNat .    var CF : Conf .
  eq collatz(P)
   = if P rem 2 == 0 then (P quo 2) else (3 * P + 1) fi .
  crl [solve]: < P, N, ph(I,think,0) ; CF >
          =>  < collatz(P), N, ph(I,think,0) ; CF >   if P > 1 .
  crl [join] : < P, N,    ph(N,think,0) ; CF >
          => < P, s(N), ph(N,think,0) ;
                        ph(N + 1,think,0) ; stk(N + 1) ; CF >
     if (P rem (4 * N) == 0) or (N rem 3 == 0) .
  crl [leave]: < P, N, CF ; ph(N,think,0) ; stk(N) >
          => < P, N - 1, CF >     if N > 2 .
endm
```

[12]These conditions make the total number of philosophers more unpredictable. The second condition is needed because any number `P` in a Collatz sequence cannot be a multiple of 3 unless `P` is the initial number.

The following system module `PHILO-CHECK` declares the state proposition `eating(I)`, where `eating(I)` is satisfied if the philosopher `I` is eating. The initial state `init` describes the case of 2 philosophers with the global counter 97, which generates the longest Collatz sequence (taking 118 steps before reaching 1) for any starting number less than 100.

```
mod PHILO-CHECK is
  protecting PHILO-DYNAMIC .        including SATISFACTION .
  subsort PState < State .
  vars P N : Nat .    var I : NzNat .      var CF : Conf .

  op eating : Nat -> Prop [ctor] .
  eq < P, N, ph(I,hungry,2) ; CF > |= eating(I) = true .

  op init : -> State .
  eq init
   = < 97, 2, ph(1,think,0) ; stk(1) ; ph(2,think,0) ; stk(2) > .
endm
```

We are interested in verifying the formula $\Box(\neg \mathit{deadlock} \rightarrow \Diamond \mathit{eating}(1))$. Without fairness assumptions, the model checker generates the following counterexample in which only the philosopher 2 performs actions while the other philosopher keeps idle and no new philosopher joins:

```
Maude> (mc init |=  [] ~ deadlock -> <> eating(1) .)
ltlr model check in PHILO-CHECK :
  init |= []~ deadlock -> <> eating(1)
result ModelCheckResult :
  counterexample(
    {< 97,2,stk(1); stk(2); ph(1,think,0); ph(2,think,0)>,
     {'solve : 'I \ 1 ; 'N \ 2 ; ...}}
    {< 292,2,stk(1); stk(2); ph(1,think,0); ph(2,think,0)>,
     {'solve : 'I \ 1 ; 'N \ 2 ; ...}}
    ...
    {< 1,2,stk(1); stk(2); ph(1,think,0); ph(2,think,0)>,
     {'wake : 'I \ 1}}
    ,
    {< 1,2,stk(1); stk(2); ph(1,hungry,0); ph(2,think,0)>,
     {'wake : 'I \ 2}}
    {< 1,2,stk(1); stk(2); ph(1,hungry,0); ph(2,hungry,0)>,
     {'grab : 'I \ 2 ; 'J \ 1 ; ...}}
    {< 1,2,stk(2); ph(1,hungry,0); ph(2,hungry,1)>,
     {'grab : 'I \ 2 ; 'J \ 2 ; ...}}
    { < 1,2,ph(1,hungry,0); ph(2,hungry,2)>,
     {'stop : 'I \ 2 ; 'N \ 2 ; ...}})
```

However, when we assume the localized fairness specification given in the rule attributes, the model checker can verify the formula:

```
Maude> (pfmc init |=  [] ~ deadlock -> <> eating(1) .)
ltlr model check under localized fairness in PHILO-CHECK :
  init |= []~ deadlock -> <> eating(1)
result Bool :
  true
```

There are at most 7 philosophers in the reachable state space from the initial state, so that a total of 14 ground fairness conditions are instantiated by realized substitutions. However, we cannot know how many fairness conditions would be required to prove the formula before exploring the state space. Furthermore, the previous model checkers in Maude cannot verify the formula with those 14 fairness conditions in a reasonable time, because of the exponential blowup when generating the Büchi automaton for the negation of the formula that explicitly contains those fairness conditions in the antecedent of an implication.

4.6 Concluding Remarks

This chapter has addressed the need of model checking under parameterized fairness assumptions. Such parameterized fairness assumptions occur very often in practice, but up to now verification under such assumptions has *not* been supported by existing model checking techniques and tools. We have presented a logical framework and an efficient on-the-fly algorithm for model checking LTLR properties under parameterized fairness assumptions. We have implemented this parameterized-fair model checking algorithm in the Maude Fair LTLR model checker, which shows reasonable performance when compared with other existing model checkers that support fairness. In particular, our tool can deal with fairness conditions for dynamic systems in which the number of relevant parameter entities cannot be predicted. Furthermore, the user interface provides a convenient and succinct way of specifying localized fairness. In the following chapter, we show how an infinite-state system can be verified using rewriting-based model checking techniques, as already hinted at in several case studies.

Part II

Approximation Methods

CHAPTER 5

INFINITE-STATE MODEL CHECKING

This chapter[1] presents a number of abstraction methods for model checking infinite-state systems specified as rewrite theories, particularly for LTLR formulas. *Equational* abstractions define quotients of the system by adding extra equations. *Folding* abstractions systemically collapse the state space of the system according to simulation preorders, and they do *not* generate any spurious counterexamples for safety properties. Narrowing-based *logical* abstractions *symbolically* represent the system's state space by means of terms with logical variables. *Predicate* abstractions construct *finite-state* abstractions of the system using state predicates, and can be automatized for rewrite theories by semantic unification and variant narrowing. These abstraction methods can be used in combination to effectively reduce an infinite-state system into a finite-state abstraction.

5.1 Introduction

Concurrent systems are often infinite-state systems for two reasons: (i) they may include unbounded data types, such as integers and stacks, so that the number of reachable states can be infinite; and (ii) they can be parameterized by certain system entities, such as processes or inputs, and therefore they define an *infinite family* of different systems. Rewriting logic is a well-suited formalism for specifying both types of infinite systems, as illustrated by several case studies in Chapters 3 and 4. In particular, the verification of infinite-state systems is important for software systems and protocol designs, since they are typically infinite-state.

[1]Some of the ideas presented in this chapter are based on the papers [14, 21, 23], joint work with José Meseguer and Santiago Escobar.

To automatically verify a temporal logic property φ of an infinite-state system $\mathcal{S}$ by model checking, the algorithms for finite-state systems, such as those in Chapters 3 and 4, cannot be directly used; instead, two approaches can be generally applied:

1. *symbolic techniques*, which describe infinite sets of states in a symbolic form, such as regular languages and logic formulas in a decidable logic, to verify the satisfaction of φ in $\mathcal{S}$; and

2. *abstraction techniques*, such as existential abstractions and predicate abstractions, which collapse $\mathcal{S}$ into a *finite-state* system $\mathcal{A}$ in such a way that the satisfaction $\mathcal{A} \models \varphi$ implies the satisfaction $\mathcal{S} \models \varphi$.

These two techniques have complementary strengths. On the one hand, a symbolic technique can directly verify the infinite-state system $\mathcal{S}$. However, they may not always terminate, and may only be applicable under some restrictions on $\mathcal{S}$ and φ; to overcome these problems, symbolic techniques are often combined with abstraction techniques. On the other hand, an abstraction technique allows the use of efficient finite-state model checking algorithms, but may generate *spurious counterexamples* in $\mathcal{A}$ that do *not* correspond to any behaviors in the original system $\mathcal{S}$, showing that $\mathcal{A} \not\models \varphi$, while in fact $\mathcal{S} \models \varphi$ holds. In such cases, *abstraction refinement methods* [55, 105] can be used to find a less abstract, yet still finite, abstraction $\mathcal{A}'$ where we may show $\mathcal{A}' \models \varphi$ if indeed $\mathcal{S} \models \varphi$ holds.

Both types of infinite-state model checking techniques have been recently developed for rewrite theories. First, in *narrowing-based* symbolic methods [86, 142], infinite sets of states are represented by terms $t(x_1, \ldots, x_n)$ with *logical variables* $x_1, \ldots, x_n$, so that the pattern $t(x_1, \ldots, x_n)$ describes the (typically infinite) set of all its ground instances. Then, *narrowing*—that generalizes term rewriting by performing unification instead of matching—defines transitions between two logical states. Because such a logical state space can still be infinite, they are combined with *folding abstractions* to collapse a pattern $t(x_1, \ldots, x_n)$ into a more general pattern $u(y_1, \ldots, y_m)$ such that $t(x_1, \ldots, x_n)$ is a substitution instance of $u(y_1, \ldots, y_m)$. Second, *equational abstraction* [139] provides a way to define existential abstraction [56, 60] for a rewrite theory $\mathcal{R} = (\Sigma, E, R)$. A set of extra equations G defines the equivalence relation $\equiv_G$ on states such that $[t]_E \equiv_G [t']_E$ iff $t =_{E \cup G} t'$, implying the abstraction function $\alpha : [t]_E \mapsto [t]_{E \cup G}$, where the abstract system is simply the rewrite theory $\mathcal{R}/G = (\Sigma, E \cup G, R)$. This chapter further develops these previous efforts, and also presents new methods, to use rewriting logic for LTLR model checking of infinite-state systems.

5.1.1 Main Contributions

First, this chapter extends both narrowing-based symbolic model checking and equational abstraction to linear temporal logic of rewriting (LTLR). As explained in Chapter 3, state-based temporal logics, such as LTL, are not expressive enough to deal with properties involving events, but LTLR is a perfect match (at the level of property specification) for rewriting logic (at the level of system specification). Extending those methods requires taking into account simulation relations between concrete and abstract *transitions*, as well as between states.

Second, this chapter shows that equational abstractions can be *bisimilar* under certain conditions, and that folding abstractions can be *faithful* in the sense that they do *not* generate any spurious counterexamples for safety properties. Moreover, this chapter generalizes folding abstractions to any simulation relations on (labeled) Kripke structures, whereas [86] considers only matching relations between logical states. Folding abstractions are closed under composition, and *strictly* more general than bisimulations since they are *not* faithful for general temporal properties.

Third, this chapter explains how narrowing-based model checking can be combined with equational abstraction. Even when folding abstractions are applied, narrowing may generate an infinite number of logical states. Therefore, equational abstractions are applied to further reduce such an infinite logical state space. This is supported by the recently developed method to automatically derive *unification algorithms modulo equational theories* by variant narrowing [87]. In case the state space is still infinite, this chapter presents a *bounded logical model checking algorithm* that can model check a system up to a given depth and that can detect if a finite state space exists within the specified depth.

Fourth, this chapter presents a new method to *automatically* generate a *predicate abstraction* for a rewrite theory $\mathcal{R}$. Predicate abstraction is one of the most widely used abstraction methods; however, except for [151] based on interactive theorem proving, predicate abstraction of rewrite theories has remained undeveloped. We systematically exploit rewriting and unification techniques to *fully automate* the predicate abstraction procedure, although it may produce an over-approximation. Since unification problems modulo equations are only semi-decidable in general [12], this chapter also presents effective unsatisfiability checking procedure, which, although incomplete, is automatic and can be used in practice to prove the *unsatisfiability* of unification problems module equations.

5.1.2 Related Work

Narrowing-based symbolic model checking is related to other infinite-state model checking techniques that symbolically represent the system's state space, such as regular languages [2, 41], tree automata [100, 146], string or multiset grammars [40, 164], Presburger arithmetic [46], constraint logic programming [69], etc. Similarly, they can be combined with abstraction techniques, e.g., [42, 47, 99]. We refer to [14, 86] for a comparison with narrowing-based symbolic model checking. Maude-NPA [85] uses similar approaches based on narrowing and folding, but is a more specialized tool supporting reachability analysis for cryptographic protocols. To the best of our knowledge, our work proposes the first *symbolic* model checking method to verify state/event-based properties of infinite-state systems.

Equational abstractions and folding abstractions can be viewed as part of a broader class of abstractions techniques, such as [56, 65, 114], which can sometimes collapse an infinite-state system into a finite-state one, and are related to abstraction techniques for parameterized systems [58, 152]. Usual abstraction techniques do not typically provide bisimulations between the abstract and concrete systems, and when they do provide them, they rely on manual proofs, instead than on simple criteria (such as those given in Theorem 5.2) for bisimilar equational abstractions.

Predicate abstraction was first introduced in [102], and has been widely applied to both conventional programming languages (e.g., [33, 167, 57, 77]), and formal specifications (e.g., [68, 156, 120]). For rewrite theories, predicate abstraction has *not* been much developed, except for the quite different semi-automatic method in [151], which generated proof obligations for an interactive theorem prover. Finally, our method to check unsatisfiability of equational constraints for disunification is related to the techniques in [9, 74] for checking unfeasibility of conditional critical pairs in the context of proving confluence of conditional rewrite rules.

There also exist many infinite-state model checking methods to exploit an order relation $\preccurlyeq$, e.g., [3, 84, 95]. Those methods typically assume that $\preccurlyeq$ is well quasi-ordered (which implies well-foundedness of $\preccurlyeq$), whereas we do not impose such conditions on $\preccurlyeq$. Indeed, matching modulo equations, used for folding in narrowing-based model checking, is *not* well-founded in general. Nevertheless, the use of well quasi-ordered relations can guarantee termination of infinite-state model checking procedures for safety properties, while our approach uses combinations of different abstractions to achieve termination of infinite-state model checking procedures.

5.1.3 Structure of this Chapter

This chapter is organized as follows. Section 5.2 presents rewriting logic specifications of infinite-state systems that are used later in this chapter to illustrate our methods. Section 5.3 presents equational abstractions of rewrite theories for LTLR formulas, and shows simple criteria for equational abstractions to be bisimilar. Section 5.4 presents folding abstractions that can be used to verify LTLR properties of infinite-state systems, and proves that folding abstractions can be faithful for safety properties. Section 5.5 explains how LTLR properties can be model checked symbolically using narrowing, and shows that narrowing-based model checking can be used in combination with folding and equational abstractions. Section 5.6 presents an automatic method to generate a predicate abstraction of a conditional rewrite theory based on unification modulo equations. Finally, Section 5.7 presents some concluding remarks.

5.2 Infinite-State System Examples

This section presents rewriting logic specifications of three infinite-states concurrent systems, used to illustrate our methods throughout this chapter. These specifications are given by, slightly restricted, *topmost order-sorted* rewrite theories, because our narrowing-based symbolic model checking and predicate abstraction methods require rewrite theories being topmost, and because the underlying E-unification procedure [87] currently only supports order-sorted equational logic. However, such restrictions are not necessary for equational abstractions and folding abstractions.

Definition 5.1. *A rewrite theory $\mathcal{R}$ is* topmost *iff there exists a sort* State *at the top of one of the connected component of $(S, \leq)$ such that: (i) for each rule $l : q \longrightarrow r$* **if** *condition, both q and r have the top sort* State*; and (ii) no operator in Σ has* State *or any of its subsorts as an argument sort.*

In a topmost rewrite theory $\mathcal{R} = (\Sigma, E, R)$, all rewrites with rules in R must take place at the top of the term. Many concurrent systems, including object-oriented systems and communication protocols, can be specified by topmost rewrite theories [137, 142]. As a matter of fact, most of the rewriting logic specifications appeared in this thesis are topmost, or can be easily transformed into topmost rewrite theories. Therefore, being expressible as a topmost rewrite theory is not a strong restriction in practice.

5.2.1 Lamport's Bakery Algorithm

We presents a topmost order-sorted rewrite theory $\mathcal{R}$ specifying Lamport's bakery algorithm for mutual exclusion of an arbitrary number of processes (adapted from [86]), and its corresponding LKS $\bar{\mathcal{K}}(\mathcal{R},k)_{\mathcal{P}}$ for a state kind k and an associated equational theory $\mathcal{P}$ that defines state propositions AP and spatial action patterns ACT. Each state of the system has the form

$$n\ ;\ m\ ;\ [i_1, d_1] \ldots [i_k, d_k],$$

given by the operator _;_;_ : Nat Nat ProcSet $\rightarrow$ State, where n is the current number in the bakery's number dispenser, m is the number currently being served, and $[i_1, d_1] \ldots [i_k, d_k]$ is a set of customer processes, each with a name i_l and in a *mode* d_l. A mode can be *idle* (not yet picked a number), $wait(n)$ (waiting with number n), or $crit(n)$ (being served with number n). The behavior is specified by the following *topmost* rewrite rules:

```
vars N M I J K L : Nat .            var PS : ProcSet .

rl [wake]: N ; M ; [I,idle] PS     =>  s N ; M ; [I,wait(N)] PS .
rl [crit]: N ; M ; [I,wait(M)] PS  =>    N ; M ; [I,crit(M)] PS .
rl [exit]: N ; M ; [I,crit(M)] PS  =>    N ; s M ; [I,idle] PS .
```

where natural numbers of sort Nat are modeled as multisets of s with the multiset union operator __ : Nat Nat $\rightarrow$ Nat (empty syntax), satisfying laws of commutativity and associativity, and the empty multiset 0 (for example, $0 = 0$, $1 = \mathsf{s}$, and $3 = \mathsf{s\,s\,s}$).

For the mutual exclusion $\Box ex?$, meaning that *at most one process can enter the critical section*, the state proposition is defined by the following equations, where the variable WS of sort ProcWaitSet stands for a set of processes whose status is either *idle* or $wait(n)$:

```
eq N ; M ; WS |= ex? = true .
eq N ; M ; [I,crit(K)] WS |= ex? = true .
eq N ; M ; [I,crit(K)] [J,crit(L)] PS |= ex? = false .
```

The sort ProcWaitSet can be defined in an order-sorted signature as follows, where sort ModeWait denotes modes *idle* and $wait(n)$:

```
sorts ProcWait ProcWaitSet Proc ProcSet .
subsorts ProcWait < Proc ProcWaitSet < ProcSet .
op [_,_] : Nat ModeWait -> ProcWait .  op [_,_] : Nat Mode -> Proc .
op none : -> ProcWaitSet [ctor] .
op __ : ProcWaitSet ProcWaitSet -> ProcWaitSet [assoc comm id: none] .
op __ : ProcSet ProcSet -> ProcSet [assoc comm id: none] .
```

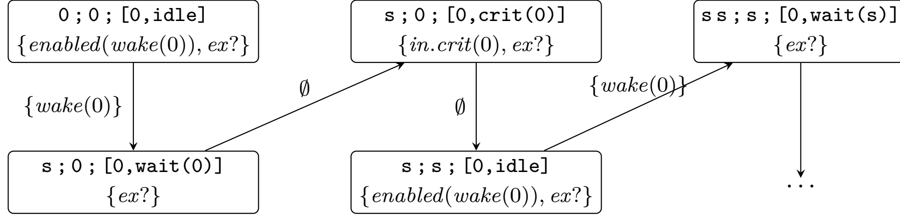


Figure 5.1: An infinite path from the initial state `0 ; 0 ; [0,idle]` in the LKS $\bar{\mathcal{K}}(\mathcal{R},k)_{\mathcal{P}}$ for the bakery algorithm.

We are also interested in the liveness property "*process 0 is eventually served*," under the fairness assumption "*if process 0 can eventually wake up forever, it must wake up infinitely often*," expressed as the LTLR formula

$$(\Diamond\Box \mathit{enabled}(\mathit{wake}(0)) \to \Box\Diamond \mathit{wake}(0)) \to \Diamond \mathit{in.crit}(0),$$

where the spatial action pattern $\mathit{wake}(0)$ holds if the *wake* rule is applied for process 0 (i.e., the variable `I` in the *wake* rule is matched to `0`), and the state proposition $\mathit{in.crit}(0)$ holds in a state where process 0 is being served. The state proposition $\mathit{in.crit}(0)$ is defined as follows, where the variable `MW` has sort `ModeWait` for modes *idle* and $\mathit{wait}(n)$, and the variable `NZPS` of sort NzProcSet denotes a set of processes with non-zero identifiers:

```
eq N ; M ; [0,crit(K)] NZPS |= in.crit(0) = true .
eq N ; M ; [0,MW] NZPS |= in.crit(0) = false .
```

Similarly, the sort NzProcSet can be defined in an order-sorted signature as follows, where sort `NzNat` denotes non-zero natural numbers:

```
sorts NzProc NzProcSet .
subsorts NzProc < Proc NzProcSet < ProcSet .
op [_,_] : NzNat Mode -> NzProc .
op __ : NzProcSet NzProcSet -> NzProcSet [assoc comm id: none] .
```

For the set of state propositions $AP = \{\mathit{in.crit}(0), \mathit{ex?}, \mathit{enabled}(\mathit{wake}(0))\}$ and the set of spatial action patterns $ACT = \{\mathit{wake}(0)\}$, we can construct the corresponding LKS $\bar{\mathcal{K}}(\mathcal{R},k)_{\mathcal{P}}$ for the bakery algorithm specification $\mathcal{R}$. For example, given the initial state `0 ; 0 ; [0,idle]`, we obtain the infinite path in Figure 5.1 within $\bar{\mathcal{K}}(\mathcal{R},k)_{\mathcal{P}}$ that contains an infinite number of different states. Notice that this system is infinite-state in two ways: (i) the counters n and m are unbounded; and (ii) the system is parameterized by (an unbounded number of) customer processes.

```
    repeat                                      l4:  for j ≠ i do
l1:  flag[i] := 1                                      if flag[j] = 2 then goto l1
l2:  while turn ≠ i do                          crt: /* critical region */
       if flag[turn] = 0 then turn := i         l5:  flag[i] := 0
l3:  flag[i] := 2                                    forever
```

Figure 5.2: Dijkstra's Mutual Exclusion Algorithm (for a process i) [129]

5.2.2 Dijkstra's Mutual Exclusion Algorithm

We present a topmost rewrite theory specifying Dijkstra's mutual exclusion algorithm for an arbitrary number of processes. There are n processes with $n \geq 2$, and two shared variables: (i) $\mathit{flag}[1 \ldots n]$ is an array of values $\{0, 1, 2\}$ for each process $1 \leq i \leq n$, and (ii) *turn* is an integer between 1 and n. The behavior is summarized by the pseudo code from [129] in Figure 5.2.

For narrowing-based symbolic model checking, we represent each state as a multiset of triples $\langle \{f_1, p_1, t_1\} \cdots \{f_k, p_k, t_k\} \rangle$. Each triple $\{f_i, p_i, t_i\}$ represents a process with f_i a value of $\mathit{flag}[i]$, p_i a program counter, and t_i a turn specifier that can be either `on` (i.e., $\mathit{turn} = i$) or `off` (i.e., $\mathit{turn} \neq i$). Only one process can be turned on at a time. The behavior is then specified by the following topmost rewrite rules, where the variable `WAITPS` of sort WaitProcSet denotes a set of processes whose flag is either 0 or 1:

```
rl [l1] : < {F,l1,T} PS >              => < {1,l2,T} PS > .
rl [l2] : < {F,l2,off} {0,S,on} PS > => < {F,l2,on} {0,S,off} PS > .
rl [l2']: < {F,l2,on} PS >             => < {F,l3,on} PS > .
rl [l3] : < {F,l3,T} PS >              => < {2,l4,T} PS > .
rl [l4] : < {F,l4,T} {2,S,T'} PS >    => < {F,l1,T} {2,S,T'} PS > .
rl [l4']: < {F,l4,T} WAITPS >          => < {F,crt,T} WAITPS > .
rl [l5] : < {F,crt,T} PS >             => < {0,l1,T} PS > .
```

Mutual exclusion can be expressed by the formula $\Box ex?$, and the state proposition $ex?$ is defined as follows in a similar way to the Bakery example, where the variable `NCPS` of sort NCrtProcSet denotes a set of processes that are *not* in the critical section (i.e., their program counters are not `crt`):

```
eq < NCPS > |= ex? = true .
eq < {F, crt, T} NCPS > |= ex? = true .
eq < {F, crt, T} {F', crt, T'} PS > |= ex? = false .
```

This system has a finite number of reachable states for a "fixed" initial state; but the system is parameterized by an unbounded number of processes, and therefore is actually an infinite-state system.

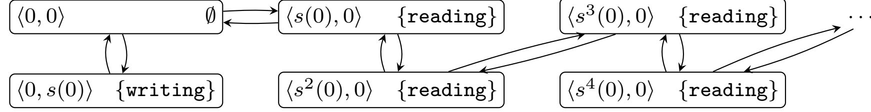


Figure 5.3: Infinite Kripke structure for the readers-writers problem.

5.2.3 Readers-Writers Problem

In the readers-writers problem, there exist a number of reader and writer processes that try to access the same critical resource. Several readers can access the resource at one time, but no other processes can access it if another process holds the resource for writing. This section presents two rewriting logic specifications of the readers-writers problem: one uses only unconditional rewrite rules, and the other also uses conditional rules.

First, we present an unconditional order-sorted topmost rewrite theory specifying a simplified version of the readers-writers problem (adapted from [61]). Each state of the system is modeled as a pair $\langle R, W\rangle \in \mathbb{N}^2$ in which R is the number of readers and W is the number of writers, given by the operator $\langle _, _\rangle : \mathsf{Nat}\ \mathsf{Nat} \to \mathsf{State}$. Natural numbers are expressed in Peano notation using the successor function $s : \mathsf{Nat} \to \mathsf{Nat}$ and the zero constant `0` of sort Nat. The behavior is defined by the following rewrite rules:

```
rl [ew]: < 0, 0 > => < 0, s(0) > .   rl [lw]: < R, s(W) > => < R, W > .
rl [er]: < R, 0 > => < s(R), 0 > .   rl [lr]: < s(R), W > => < R, W > .
```

Mutual exclusion is expressed by the LTL formula $\Box\neg(\mathit{reading} \wedge \mathit{writing})$, meaning that *a reader and a write cannot enter the critical section at the same time*, where the state propositions are defined as follows:

```
eq < s(R), W > |= reading = true .   eq < 0, W > |= reading = false .
eq < R, s(W) > |= writing = true .   eq < R, 0 > |= writing = false .
```

This system has an infinite number of (reachable) states, because the number of readers R is unbounded. For example, given the set of state propositions $AP = \{\mathit{reading}, \mathit{writing}\}$, we obtain the infinite path from the state $\langle 0, 0\rangle$ in Figure 5.3 within the Kripke structure $\mathcal{K}(\mathcal{R}, \mathsf{State})_{\mathcal{P}}$.

Next, we present a conditional topmost rewrite theory specifying a model of the readers-writers problem with explicit shared variables and processes, adapted from [4]. Each state has the form $\langle n, b \mid p_1; \cdots; p_n\rangle$, given by the operator $\langle _, _|_\rangle : \mathsf{Nat}\ \mathsf{Bool}\ \mathsf{ProcSet} \to \mathsf{State}$, where n denotes the number of readers, b is a Boolean flag to denote *no* readers and *no* writers, and $p_1; \cdots; p_n$ is a multiset of processes, each in a status $p_i \in \{\mathit{idle}, \mathit{read}, \mathit{write}\}$. The behavior of the system is specified by the following rewrite rules:

```
 rl [ew]:  < N, true  | idle ; PS >  => < N, false | write ; PS > .
 rl [lw]:  < N, false | write ; PS > => < N, true  | idle ; PS > .
crl [er]:  < N, B     | idle ; PS >
      =>   < s(N), B' | read ; PS >  if c(N,B,B') .
crl [lr]:  < s(N), B  | read ; PS >
      =>   < N, B'    | idle ; PS >  if c(N,B',B) .
```

where the function $c(n, b, b')$ returns *true* iff $b = \mathit{true}$ and $b' = \mathit{false}$ when $n = 0$, or $b = b'$ when $n > 0$, defined by the following equations:

```
eq c(0, true, false)  = true .     eq c(s(N), B, B)      = true .
eq c(0, false, B)     = false .    eq c(s(N), true, false) = false .
eq c(0, B, true)      = false .    eq c(s(N), false, true) = false .
```

Mutual exclusion is again expressed by the formula $\Box\neg(\mathit{reading} \wedge \mathit{writing})$, and the state propositions are defined by the following equations, where the variable `RS` denotes a set of processes in status *idle* or *read*, and `WS` denotes a set of processes in status *idle* or *write*:

```
eq < N, B | read ; PS >  |= reading = true .
eq < N, B | WS >         |= reading = false .
eq < N, B | write ; PS > |= writing = true .
eq < N, B | RS >         |= writing = false .
```

This system is finite-state for a *fixed* set of processes; but the system is actually infinite-state, since the number of processes is unbounded.

5.3 Equational Abstraction

When a system specified as a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ has an infinite number of reachable states, one of the simplest ways to collapse the reachable state space into a finite one is *equational abstraction* [139]. A set of equations G can be added to $\mathcal{R}$ to obtain the equivalence relation $\equiv_G$ on states $\mathcal{T}_{\Sigma/E,k}$ for a certain state kind $k \in \Sigma$, namely, $[t]_E \equiv_G [t']_E \iff t =_{E \cup G} t'$. For the Kripke structure $\mathcal{K}(\mathcal{R}, k)_{\mathcal{P}} = (S, AP, \mathcal{L}, \longrightarrow_{\mathcal{K}})$ associated to the rewrite theory $\mathcal{R} = (\Sigma, E, R)$, this equivalence relation $\equiv_G$ defines the *quotient abstraction* $\mathcal{K}/{\equiv_G} = (S/{\equiv_G}, AP, \mathcal{L}, \longrightarrow_{\mathcal{K}/\equiv_G})$ such that

$$[s_1]_{\equiv_G} \longrightarrow_{\mathcal{K}/\equiv_G} [s_2]_{\equiv_G} \iff (\exists s_1' \in [s_1]_{\equiv_G}, s_2' \in [s_2]_{\equiv_G})\ s_1' \longrightarrow_{\mathcal{K}} s_2',$$

and $\mathcal{R}/G = (\Sigma, E \cup G, R)$ is the abstract specification associated to $\mathcal{K}/{\equiv_G}$. This can be considered as a special form of an existential abstraction [56, 60] with the abstraction function $\alpha : [t]_E \mapsto [t]_{E \cup G}$.

This section shows how equational abstraction can be applied for LTLR properties to reduce an infinite state space of a rewire theory $\mathcal{R}$. That is, for an LTLR formula φ, if the equations G preserve the meaning of the state propositions and the spatial action patterns in φ, then:

$$\bar{\mathcal{K}}(\mathcal{R}/G, k)_{\mathcal{P}}, [t]_{E \cup G} \models \varphi \implies \bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}, [t]_E \models \varphi$$

for an initial state $[t]_E$, where $\bar{\mathcal{K}}(\mathcal{R}/G, k)_{\mathcal{P}}$ is the labeled Kripke structure associated to $\mathcal{R}/G$. However, a counterexample in $\mathcal{R}/G$ can be a *spurious* counterexample that has no counterpart in $\mathcal{R}$. Therefore, this section also presents simple criteria for equational abstractions to be bisimilar so that there exist no spurious counterexamples.

5.3.1 Simulation Relations for LKSs

For model checking techniques, an abstraction $\widehat{\mathcal{S}}$ of a concurrent system $\mathcal{S}$ typically preserves every behavior of the original system $\mathcal{S}$, in terms of a *simulation relation* between $\mathcal{S}$ and $\widehat{\mathcal{S}}$. In the following definition we extend the notion of simulation relations for Kripke structures [60] to one for LKSs, which also takes into account spatial action patterns.

Definition 5.2. *Given two LKS $\bar{\mathcal{K}}_i = (S_i, AP, \mathcal{L}_i, ACT, \longrightarrow_{\bar{\mathcal{K}}_i})$, $i = 1, 2$, a binary relation $H \subseteq S_1 \times S_2$ is a* simulation *from $\bar{\mathcal{K}}_1$ to $\bar{\mathcal{K}}_2$ iff:*

- *if $s_1 H s_2$, then $\mathcal{L}_1(s_1) = \mathcal{L}_2(s_2)$, and*
- *if $s_1 H s_2$ and $s_1 \xrightarrow{\Lambda}_{\bar{\mathcal{K}}_1} s_1'$ in $\bar{\mathcal{K}}_1$, then there exists a state $s_2' \in S_2$ such that $s_1' H s_2'$ and $s_2 \xrightarrow{\Lambda}_{\bar{\mathcal{K}}_2} s_2'$ in $\bar{\mathcal{K}}_2$.*

A simulation H is a bisimulation *iff H^{-1} is also a simulation, and is* total *iff for any $s_1 \in S_1$ there exists $s_2 \in S_2$ such that $s_1 H s_2$.*

As expected, if an LKS $\bar{\mathcal{K}}_2$ *simulates* $\bar{\mathcal{K}}_1$, then each infinite path in $\bar{\mathcal{K}}_1$ has a corresponding path in $\bar{\mathcal{K}}_2$, as shown in the following lemma.

Lemma 5.1. *Given a simulation H from an LKS $\bar{\mathcal{K}}_1$ to $\bar{\mathcal{K}}_2$, if $s_1 H s_2$, then for each path (π_1, α) of $\bar{\mathcal{K}}_1$ beginning at s_1, there exists a corresponding path (π_2, α) beginning at s_2 such that $\pi_1(i)\, H\, \pi_2(i)$ for each $i \in \mathbb{N}$.*

Proof. We construct π_2 by induction. Let $\pi_2(0) = s_2$. Clearly, $\pi_1(0)\, H\, \pi_2(0)$. Next, suppose that $\pi_1(k)\, H\, \pi_2(k)$ for some $k \in \mathbb{N}$. Since $\pi_1(k)\, H\, \pi_2(k)$ and $\pi_1(k) \xrightarrow{\alpha(k)}_{\bar{\mathcal{K}}} \pi_1(k+1)$, there exists a state s_2' such that $\pi_1(k+1)\, H\, s_2'$ and $\pi_2(k) \xrightarrow{\alpha(k)}_{\bar{\mathcal{K}}} s_2'$. Then, we choose $\pi_2(k+1) = s_2'$. □

Suppose that $s_0^1 \, H \, s_0^2$ for a simulation H from $\bar{\mathcal{K}}_1$ to $\bar{\mathcal{K}}_2$. If there exists a counterexample (π_1, α_1) in $\bar{\mathcal{K}}_1$ starting from s_0^1, then by the above lemma, there exists a corresponding counterexample (π_2, α_2) in $\bar{\mathcal{K}}_2$ starting from s_0^2 such that $\mathcal{L}_1(\pi_1(i)) = \mathcal{L}_2(\pi_2(i))$ and $\alpha_1(i) = \alpha_2(i)$ for $i \in \mathbb{N}$. Therefore:

Lemma 5.2. *Given an LTLR formula and a simulation H from an LKS $\bar{\mathcal{K}}_1$ to $\bar{\mathcal{K}}_2$, if $s_0^1 \, H \, s_0^2$, then $\bar{\mathcal{K}}_2, s_0^2 \models \varphi \implies \bar{\mathcal{K}}_1, s_0^1 \models \varphi$. In particular, if H is a bisimulation, then $\bar{\mathcal{K}}_2, s_0^2 \models \varphi \iff \bar{\mathcal{K}}_1, s_0^1 \models \varphi$.*

Using this expanded notion of simulation relations, in a similar way to the case of Kripke structures, we can also define quotient abstractions of labeled Kripke structures for LTLR model checking as follows:

Definition 5.3. *For an LKS $\bar{\mathcal{K}} = (S, AP, \mathcal{L}, ACT, \longrightarrow_{\bar{\mathcal{K}}})$ and an equivalence relation $\equiv \subseteq S^2$ such that $s_1 \equiv s_2$ implies $\mathcal{L}(s_1) = \mathcal{L}(s_2)$, the* quotient abstraction *is the LKS $\bar{\mathcal{K}}/{\equiv} = (S/{\equiv}, AP, \mathcal{L}_{\equiv}, ACT, \longrightarrow_{\bar{\mathcal{K}}/\equiv})$ such that*

- *$\mathcal{L}_{\equiv}([s]_{\equiv}) = \mathcal{L}(s)$; and*
- *$[s_1]_{\equiv} \xrightarrow{\Lambda}_{\bar{\mathcal{K}}/\equiv} [s_2]_{\equiv} \iff (\exists s_1' \in [s_1]_{\equiv}, \, s_2' \in [s_2]_{\equiv}) \; s_1' \xrightarrow{\Lambda}_{\bar{\mathcal{K}}} s_2'$.*

For any concrete transition $s_1 \xrightarrow{\Lambda}_{\bar{\mathcal{K}}} s_2$ in $\bar{\mathcal{K}}$, we can easily see that there exists an abstract transition $[s_1]_{\equiv} \xrightarrow{\Lambda}_{\bar{\mathcal{K}}/\equiv} [s_2]_{\equiv}$ in the quotient abstraction $\bar{\mathcal{K}}/{\equiv}$, where $s_1 \in [s_1]_{\equiv}$ and $s_2 \in [s_2]_{\equiv}$. Therefore, the membership relation $\in \subseteq S \times S/{\equiv}$ is a total simulation from $\bar{\mathcal{K}}$ to $\bar{\mathcal{K}}/{\equiv}$, and by Lemma 5.2, for any LTLR formula φ and an initial state $s_0 \in S$, we have:

$$\bar{\mathcal{K}}/{\equiv}, [s_0]_{\equiv} \models \varphi \implies \bar{\mathcal{K}}, s_0 \models \varphi.$$

5.3.2 Equational Abstractions for LTLR

Consider a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ and its associated equational theory $\mathcal{P} = (\Pi, D)$ that defines a set of state propositions AP and a set of spatial action patterns ACT.[2] A set G of equations always defines the equivalence relation $\equiv_G$ such that $[t]_E \equiv_G [t']_E \iff t =_{E \cup G} t'$. Therefore, in principle, the quotient abstraction $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}/{\equiv_G}$ of an LKS $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}$ associated to the original rewrite theory $\mathcal{R}$ always exists, according to Definition 5.3, provided that $t \equiv_G t'$ implies $\mathcal{L}_{\mathcal{P}}([t]_E) = \mathcal{L}_{\mathcal{P}}([t]'_E)$. However, it is not straightforward to conclude that the equational abstraction $\mathcal{R}/G = (\Sigma, E \cup G, R)$ indeed corresponds to the quotient abstraction $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}/{\equiv_G}$.

[2]Recall that the extended equational theory $(\Sigma \cup \Pi, E \cup D)$ should protect (Σ, E), that is, $\mathcal{T}_{\Sigma \cup \Pi / E \cup D, s} \simeq \mathcal{T}_{\Sigma/E, s}$ for each sort $s \in \Sigma$.

In fact, by the definition of the LKS $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}$ (explained in Section 3.2.3), if $\mathcal{R}$ has a deadlock state from which *no* one-step rewrites exists, then $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}/{\equiv_G}$ is *not* equivalent to $\bar{\mathcal{K}}(\mathcal{R}/G, k)_{\mathcal{P}}$. The reason is that for each deadlock state $[t]_E$, the LKS $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}$ adds the self loop

$$[t]_E \xrightarrow{\{\texttt{deadlock}\}}_{\bar{\mathcal{K}}} [t]_E$$

labeled by event `deadlock`. Consider a deadlock state $[t_1]_E$ and a one-step rewrite $\lambda : [t_2]_E \longrightarrow^1_{\mathcal{R}} [t_2']_E$ from a state $[t_2]_E$ in $\mathcal{R}$. For a set G of equations, if $t_1 =_{E \cup G} t_2$, then there exists a one-step rewrite $[t_1]_{E \cup G} \longrightarrow^1_{\mathcal{R}/G} [t_2']_{E \cup G}$ in the abstract theory $\mathcal{R}/G$ by definition. Therefore, in the LKS $\bar{\mathcal{K}}(\mathcal{R}/G, k)_{\mathcal{P}}$, there can be *no* transition from $[t_1]_{E \cup G}$ labeled by $\{\texttt{deadlock}\}$, whereas $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}/{\equiv_G}$ contains such a `deadlock` transition from $[t_1]_{E \cup G}$.

Example 5.1. *Consider the simple client-server communication model in Example 4.2. Then, the state* $[[a]\ [b, a, 1, f(a, b, 1)]]_E$ *with one server* a *and one client* b *is a deadlock state in the original model, since the client* b *has already received the answer* $f(a, b, 1)$*. Therefore, the LKS* $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}$ *contains the deadlock transition*

$$[[a]\ [b, a, 1, f(a, b, 1)]]_E \xrightarrow{\{deadlock\}}_{\bar{\mathcal{K}}} [[a]\ [b, a, 1, f(a, b, 1)]]_E.$$

However, this state is no longer deadlock state in the abstract model $\mathcal{R}/G$*, if we add the following extra equation* G*:*

```
eq [C,S,N,M] [S] S <-{C,N} = [C,S,N,M] [S] .
```

Because $[[a]\,[b, a, 1, f(a, b, 1)]]_{E \cup G} = [[a]\,[b, a, 1, f(a, b, 1)]\ a\,\texttt{<-\{}b, 1\texttt{\}}]_{E \cup G}$*, the rewrite rule* $loss : I\texttt{<-\{}J, M\texttt{\}} \longrightarrow \texttt{null}$*, which removes any message in the configuration, can be applied to* $[[a]\ [b, a, 1, f(a, b, 1)]]_{E \cup G}$*, and thus there is no deadlock transition from* $[[a]\,[b, a, 1, f(a, b, 1)]]_{E \cup G}$ *in* $\bar{\mathcal{K}}(\mathcal{R}/G, k)_{\mathcal{P}}$*.*

Consequently, to apply equational abstraction, we require that a rewrite theory $\mathcal{R}$ is k*-deadlock free* (i.e., there exist no deadlock terms of kind k). Note that as explained in [61, 139], a rewrite theory $\mathcal{R}$ can be transformed into a semantically equivalent deadlock-free theory, provided that its rules have no rewrites in the conditions, by simply adding the rule

```
crl [deadlock]: {X} => {X} if not enabled(X) .
```

where X is a variable of kind k, the function `enabled(`t`)` returns `true` iff a rewrite rule can be applied to the term t, and `{_}` $: k \to k'$ is an *encapsulation* operator with a new state kind k'.

To define an equational abstraction of $\mathcal{R}$ using a set G of equations, we also need that $t \equiv_G t' \implies \mathcal{L}_{\mathcal{P}}([t]_E) = \mathcal{L}_{\mathcal{P}}([t]'_E)$. This condition can be ensured by showing that $true \neq_{E\cup G} false$ [139]: if $t =_{E\cup G} t'$ but $p \in \mathcal{L}_{\mathcal{P}}([t]_E)$ and $p \notin \mathcal{L}_{\mathcal{P}}([t]'_E)$, then $true =_E (t \models p) =_{E\cup G} (t' \models p) =_E false$. Similarly, if $true \neq_{E\cup G} false$, then any two one-step proof terms with $\lambda =_{E\cup G} \lambda'$ satisfy the same set of spatial action patterns. Consequently:

Theorem 5.1. *Given a k-deadlock free rewrite theory $\mathcal{R} = (\Sigma, E, R)$, a set of equations G, and an associated equational theory $\mathcal{P} = (\Pi, D)$ defining AP and ACT, assuming that true $\neq_{E\cup D\cup G}$ false, for an LTLR formula φ and $[t]_E \in \mathcal{T}_{\Sigma/E,k}$: $\bar{\mathcal{K}}(\mathcal{R}/G, k)_{\mathcal{P}}, [t]_{E\cup G} \models \varphi \implies \bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}, [t]_E \models \varphi$.*

Proof. Let $H_G = \{([t]_E, [t]_{E\cup G}) \mid t \in \mathcal{T}_{\Sigma/E,k}\}$. It suffices to show that H_G is a total simulation from $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}$ to $\bar{\mathcal{K}}(\mathcal{R}/G, k)_{\mathcal{P}}, [t]_{E\cup G}$. Suppose that $[t]_E \xrightarrow{\Lambda}_{\bar{\mathcal{K}}(\mathcal{R})} [t']_E$. Since $\mathcal{R}$ is k-deadlock free, by definition, there exists a one-step rewrite $\lambda : [t]_E \longrightarrow^1_{\mathcal{R}} [t']_E$ and $\Lambda = \{\delta \in ACT \mid \lambda \models \delta =_{E\cup D} true\}$. Since $[t]_E \subseteq [t]_{E\cup G}$ and $[t']_E \subseteq [t']_{E\cup G}$, there exists a one-step rewrite $\lambda : [t]_{E\cup G} \longrightarrow^1_{\mathcal{R}/G} [t']_{E\cup G}$, and thus $[t]_{E\cup G} \xrightarrow{\Lambda}_{\bar{\mathcal{K}}(\mathcal{R}/G)} [t']_{E\cup G}$. ☐

In addition to all the above-mentioned conditions that must be checked to *verify* the *correctness* of an equational abstraction, the quotient rewrite theory $\mathcal{R}/G = (\Sigma, E\cup G, R)$ must satisfy the *executability conditions* defined in Chapter 2.2.4, which can be checked using the formal tools in the Maude environment [61]. This is further discussed in Section 5.3.4.

Example 5.2. *For the Bakery algorithm example in Section 5.2.1, consider the set G of extra equations (that also remove process identifiers):*

```
eq [I,MD] = [MD] .
eq s N ; s M ; [idle] [idle] = N ; M ; [idle] [idle] .
eq s N ; s M ; [wait(s K)] [idle] = N ; M ; [wait(K)] [idle] .
eq s N ; s M ; [crit(s K)] [idle] = N ; M ; [crit(K)] [idle] .
eq s N ; s M ; [wait(s K)] [wait(s L)] = N ; M ; [wait(K)] [wait(L)] .
eq s N ; s M ; [wait(s K)] [crit(s L)] = N ; M ; [wait(K)] [crit(L)] .
eq s N ; s M ; [crit(s K)] [crit(s L)] = N ; M ; [crit(K)] [crit(L)] .
```

For $AP = \{ex?, enabled(\{'wake\})\}$ and $ACT = \{\{'wake\}\}$,[3] *we obtain the abstract LKS $\bar{\mathcal{K}}(\mathcal{R}/G, k)_{\mathcal{P}}$ in Figure 5.4 having a finite number of reachable states from the abstract initial state $[$`0 ; 0 ; [idle] [idle]`$]_{E\cup G}$ with two processes. Since the formula $\Box ex?$ holds in $\bar{\mathcal{K}}(\mathcal{R}/G, k)_{\mathcal{P}}$, by Theorem 5.1, $\Box ex?$ also holds in $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}$ from the corresponding initial state.*

[3]Recall that $\{'wake\}$ is a spatial action pattern in $SP(\mathcal{R})$, defined in Section 3.2.1, meaning that a rule with label *wake* has been applied.

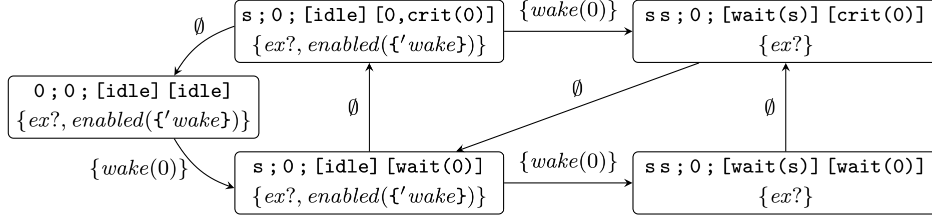


Figure 5.4: The reachable states in the abstract LKS $\bar{\mathcal{K}}(\mathcal{R}/G, k)_{\mathcal{P}}$.

5.3.3 Bisimilar Equational Abstractions

As usual for abstraction techniques, equational abstractions may generate spurious counterexamples, since they only define simulations, and in general are *not* bisimilar to their original system.

Example 5.3. *For the Bakery example in Section 5.2.1, now consider the set $G = \{s\,N = 0\}$ of abstraction equations. Then, there exists the following* spurious *counterexample of $\Box ex?$ in the abstract LKS $\bar{\mathcal{K}}(\mathcal{R}/G, k)_{\mathcal{P}}$:*

```
       0 ; 0 ; [0,idle] [0,idle]          ⟶ 0 ; 0 ; [0,idle] [0,wait(0)]
    ⟶ 0 ; 0 ; [0,wait(0)] [0,wait(0)] ⟶ 0 ; 0 ; [0,crit(0)] [0,wait(0)]
    ⟶ 0 ; 0 ; [0,crit(0)] [0,crit(0)].
```

Therefore, we introduce *bisimilar equational abstractions*, which ensure a bisimulation from $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}$ to its quotient abstraction $\bar{\mathcal{K}}(\mathcal{R}/G, k)_{\mathcal{P}}$.

Lemma 5.3. *Given a k-deadlock free rewrite theory $\mathcal{R} = (\Sigma, E, R)$, an associated equational theory $\mathcal{P} = (\Pi, D)$ defining AP and ACT, and a set of equations G with true $\neq_{E \cup D \cup G}$ false, there is a total bisimulation from an LKS $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}$ to $\bar{\mathcal{K}}(\mathcal{R}/G, k)_{\mathcal{P}}$, if the following holds for $t_1, t_1', t_2 \in \mathcal{T}_{\Sigma/E,k}$:*

$$\begin{aligned} & \lambda_1 : [t_1]_E \longrightarrow^1_{\mathcal{R}} [t_1']_E \ \wedge\ t_1 =_{E \cup G} t_2 \\ \Longrightarrow \quad (\exists t_2' \in \mathcal{T}_{\Sigma/E,k})\ & \lambda_2 : [t_2]_E \longrightarrow^1_{\mathcal{R}} [t_2']_E \ \wedge\ t_1' =_{E \cup G} t_2' \ \wedge \\ & (\lambda_1 \models \delta) =_{E \cup D} (\lambda_2 \models \delta) \ \textit{ for each } \ \delta \in ACT. \end{aligned}$$

Proof. We only need to prove that $H_G = \{([t]_E, [t]_{E \cup G}) \mid t \in \mathcal{T}_{\Sigma/E,k}\}$ is a simulation from $\bar{\mathcal{K}}(\mathcal{R}/G, k)_{\mathcal{P}}$ to $\bar{\mathcal{K}}(\mathcal{R}, k)_{\mathcal{P}}$. First, notice that $=_{E \cup G}$ is a bisimulation for $\mathcal{R}$ with respect to $\longrightarrow^1_{\mathcal{R}}$, since $=_{E \cup G}$ is symmetric. Suppose that $\lambda_1 : [u]_{E \cup G} \longrightarrow^1_{\mathcal{R}/G} [u']_{E \cup G}$ and $[t]_E \subseteq [u]_{E \cup G}$ (i.e., $[t]_E\, H_G\, [u]_{E \cup G}$). By definition of one-step rewrites, for *some* $v \in [u]_{E \cup G}$ and $v' \in [u']_{E \cup G}$, there exists $\lambda_1 : [v]_E \longrightarrow^1_{\mathcal{R}} [v']_E$. Since $v =_{E \cup G} t$, by the assumption, there exists a one-step rewrite $\lambda_2 : [t]_E \longrightarrow^1_{\mathcal{R}} [t']_E$ such that $v' =_{E \cup G} t'$, and λ_1 and λ_2 satisfies the same set of spatial action patterns. $\square$

A rewrite theory $\mathcal{R}/G$ is called a *bisimilar equational abstraction* of $\mathcal{R}$ if the set G of equations satisfies the conditions in Lemma 5.3. For a *topmost unconditional* rewrite theory $\mathcal{R}$, a bisimilar equational abstraction with a set G of *topmost equations*—of the form $t = t'$ with $t, t' \in \mathcal{T}_\Sigma(\mathcal{X})_{\mathsf{State}}$—can be easily identified by checking that the application of an equation in G does not interfere with the application of a rewrite rule.

Theorem 5.2 (Necessary/Sufficient Conditions for Bisimilarity)**.** *Given a* topmost unconditional *k-deadlock free rewrite theory $\mathcal{R} = (\Sigma, E, R)$ and an associated equational theory $\mathcal{P} = (\Pi, D)$ defining AP and ACT, for a set G of* topmost *equations, $\mathcal{R}/G$ is a bisimilar equational abstraction iff for each rule $l : q \longrightarrow r$ and each equation $u = v \in G$ or $v = u \in G$, the following condition holds for a substitution $\sigma : \mathcal{X} \to \mathcal{T}_\Sigma(\mathcal{X})$:*

$$\begin{aligned} &\sigma(u) =_E \sigma(q) \ \wedge \ (l(\sigma) \models \delta) =_{E \cup D} b \ \textit{ for some } b \in \{\textit{true}, \textit{false}\} \\ \Longrightarrow \ &(\exists \theta : \mathcal{X} \to \mathcal{T}_\Sigma(\mathcal{X}))\ \sigma(v) =_E \theta(q) \ \wedge \ \sigma(r) =_{E \cup G} \theta(r) \ \wedge \\ &\qquad (l(\sigma) \models \delta) =_{E \cup D} (l(\theta) \models \delta) \ \textit{ for each } \ \delta \in \textit{ACT}. \end{aligned}$$

Proof. The above condition can be expressed by the following diagram, where the truth value of δ for $l(\sigma)$ is decided into either *true* or *false*:

$$\begin{array}{lccccc} \forall \sigma : \mathcal{X} \to \mathcal{T}_\Sigma(\mathcal{X}). & \sigma(u) & =_E & \sigma(q) & \longrightarrow & \sigma(r) \\ & \|_G & & & & \|_{E \cup G} \\ \exists \theta : \mathcal{X} \to \mathcal{T}_\Sigma(\mathcal{X}). & \sigma(v) & =_E & \theta(q) & \longrightarrow & \theta(r) \end{array}$$

($\Leftarrow$) Suppose that $\lambda_1 : [t_1]_E \longrightarrow^1_{\mathcal{R}} [t_1']_E$ and $t_1 =_{E \cup G} t_2$. If $=^k_{G/E}$ denotes k applications of equations in G modulo E, then $t_1 =^n_{G/E} t_2$ for some $n \in \mathbb{N}$. We prove by induction on $n \in \mathbb{N}$ that $(\exists t_2')\ \lambda_2 : [t_2]_E \longrightarrow^1_{\mathcal{R}} [t_2']_E$, $t_1' =_{E \cup G} t_2'$, and $(\lambda_1 \models \delta) =_{E \cup D} (\lambda_2 \models \delta)$ for each $\delta \in \textit{ACT}$. When $n = 0$, it is immediate since $t_1 =_E t_2$ and then $\lambda_1 : [t_2]_E \longrightarrow^1_{\mathcal{R}} [t_1']_E$. For $n > 0$, assume that if $t_1 =^n_{G/E} w$ for $w \in \mathcal{T}_{\Sigma/E,\mathsf{State}}$, then $(\exists w')\ \lambda_2' : [w]_E \longrightarrow^1_{\mathcal{R}} [w']_E$, $t_1' =_{E \cup G} w'$, and $(\lambda_1 \models \delta) =_{E \cup D} (\lambda_2' \models \delta)$ for each $\delta \in \textit{ACT}$. If $t_1 =^n_{G/E} w =^1_{G/E} t_2$, then $\lambda_2' : [w]_E \longrightarrow^1_{\mathcal{R}} [w']_E$ and by using the condition in the statement, $(\exists t_2')\ \lambda_2 : [t_2]_E \longrightarrow^1_{\mathcal{R}} [t_2']_E$, $w' =_{E \cup G} t_2'$, and $(\lambda_2' \models \delta) =_{E \cup D} (\lambda_2 \models \delta)$ for each $\delta \in \textit{ACT}$. Finally, we have that $t_1' =_{E \cup G} w' =_{E \cup G} t_2'$, and for $\delta \in \textit{ACT}$, $(\lambda_1 \models \delta) =_{E \cup D} (\lambda_2' \models \delta) =_{E \cup D} (\lambda_2 \models \delta)$, and the conclusion follows.
($\Rightarrow$) If $\mathcal{R}/G$ is a bisimilar equational abstraction and $\lambda_1 : [t_1]_E \longrightarrow^1_{\mathcal{R}} [t_1']_E$, then for $t_2 \in \mathcal{T}_{\Sigma/E,\mathsf{State}}$ obtained by one application of G (i.e., $t_1 =^1_{G/E} t_2$), $(\exists t_2' \in \mathcal{T}_{\Sigma/E,\mathsf{State}})\ \lambda_2 : [t_2]_E \longrightarrow^1_{\mathcal{R}} [t_2']_E$, $t_1' =_{E \cup G} t_2'$, and for $\delta \in \textit{ACT}$, $(\lambda_1 \models \delta) =_{E \cup D} (\lambda_2 \models \delta)$, which implies the condition in the statement. $\square$

The reason why only a set G of topmost equations is allowed for bisimilar equational abstractions is to avoid problems caused by repeated variables in rewrite rules. For instance, consider $R = \{f(X, X) \longrightarrow h(X)\}$, $E = \emptyset$, and $G = \{a = b\}$. This topmost rewrite theory satisfies the condition of the previous theorem except G being topmost. Then, given the term $f(a, a)$, $f(b, a) =_G f(a, a)$ but now $f(b, a)$ cannot be rewritten with R. This process to check the conditions in Theorem 5.2 can easily be automated in a way similar to that used by the Maude coherence checker [75], which checks similar conditions between equations and rules. We show an example of bisimilar equational abstractions in Section 5.5 below.

5.3.4 Executability Conditions

For an equational abstraction $\mathcal{R}/G$ to be a practical abstraction, it needs to be computable; that is, as explained in Section 2.2.4, for an equational abstraction $\mathcal{R}/G = (\Sigma, E \cup B \cup G, R)$ with B a set of structural axioms, $E \cup G$ is sort-decreasing, and ground terminating, confluent, and coherent modulo B, and R is ground coherent with $E \cup G$ modulo B. Moreover, we also need that a one-step rewrite and its *canonical* one-step rewrite satisfy the same set of of spatial action patterns (i.e., $E \cup B \cup G$ *preserves ACT*).

Definition 5.4. *Given $\mathcal{R} = (\Sigma, E \cup B, R)$ and an associated equational theory $\mathcal{P} = (\Pi, D)$ defining AP and ACT, R is ground $\mathcal{P}$-*coherent *with E modulo B iff for $R/B = (\Sigma, B, R)$ and $t, t' \in \mathcal{T}_\Sigma$, if $\lambda : [t]_B \longrightarrow^1_{R/B} [t']_B$, then there is $\lambda' : [can_{E/B}(t)]_B \longrightarrow^1_{R/B} [t'']_B$ with $can_{E/B}(t') =_B can_{E/B}(t'')$ and $(\forall \delta \in ACT)\ (\lambda \models \delta) =_{E \cup B \cup D} (\lambda' \models \delta)$.*

Even if $\mathcal{R}$ is computable, $\mathcal{R}/G$ may *not* be computable, since the rules in R can be *not* ground $\mathcal{P}$-coherent with $E \cup G$ modulo B. In this case, we can try to "complete" the rewrite rules to have a semantically equivalent theory $\mathcal{R}'/G = (\Sigma, E \cup B \cup G, R')$ such that $[t]_{E \cup B \cup G} \rightarrow^1_{\mathcal{R}/G} [t']_{E \cup B \cup G}$ iff $[t]_{E \cup B \cup G} \rightarrow^1_{\mathcal{R}'/G} [t']_{E \cup B \cup G}$, manually or using some tools [61, 75].

Example 5.4. *For the client-server communication model in Example 4.2, the extra abstraction equation $G = \{I$ `<-{`C, N`}` I `<-{`C, N`}` $= I$ `<-{`C, N`}`$\}$ causes a lack of coherence for the reply, rec, and loss rules, because they can consume only a single message but it is equivalent to an arbitrary number of identical messages by G. For example, $t = a$ `<-{`$b, 1$`}` a `<-{`$b, 1$`}` $[a]$ is rewritten to $t' = a$ `<-{`$b, 1$`}` $[a]$ b `<-{`$a, f(a, c, 1)$`}` by the reply rule, whereas $[can_{E \cup G/B}(t)]_B = a$ `<-{`$b, 1$`}` $[a]$ is rewritten to $t'' = [a]$ b `<-{`$a, f(a, c, 1)$`}` by the same rule but $[can_{E \cup G/B}(t')]_B \neq [can_{E \cup G/B}(t'')]_B$.*

We can add extra versions R' of these rules to eliminate the cases of a lack of coherence (see Appendix A.1.1). For example, by the extra reply rule

```
rl [reply]: S <-{C,N} [S]  =>   S <-{C,N} [S]  C <-{S,f(S,C,N)} .
```

the term $[can_{E\cup G/B}(t)]_B = a$ <-{$b, 1$} $[a]$ *can now be rewritten to the term* $t''' = a$ <-{$b, 1$} $[a]$ b <-{$a, f(a, c, 1)$} *such that*

$$[can_{E\cup G/B}(t')]_B = [can_{E\cup G/B}(t''')]_B.$$

Furthermore, each spatial action pattern in $SP(\mathcal{R})$ is preserved by both G and R', since G does not *"completely" remove any message, and since each extra rule in R' has the same set of variables with the correlated rewrite rule in R. Consequently, $R \cup R'$ is ground $\mathcal{P}$-coherent with $E \cup G$ modulo B.*

It is worth noting that all the extra rules in R' can be simulated by the original equational abstraction $\mathcal{R}/G = (\Sigma, E \cup B \cup G, R)$, and therefore the extended equational abstraction $\mathcal{R}'/G = (\Sigma, E \cup B \cup G, R \cup R')$ is semantically equivalent to $\mathcal{R}/G$. For example, a one-step rewrite

$$[S \texttt{<-\{}C, N\texttt{\}}\ [S]]_{E\cup B\cup G} \rightarrow^1_{\mathcal{R}/G} [S \texttt{<-\{}C, N\texttt{\}}\ [S]\ C \texttt{<-\{}S, f(S, C, N)\texttt{\}}]_{E\cup B\cup G}$$

holds in the original equational abstraction $\mathcal{R}/G$ for the extra reply rule, since $[S \texttt{<-\{}C, N\texttt{\}}\ [S]]_{E\cup A\cup G} = [S \texttt{<-\{}C, N\texttt{\}}\ S \texttt{<-\{}C, N\texttt{\}}\ [S]]_{E\cup A\cup G}$.

5.4 Folding Abstraction

We can reduce a (labeled) Kripke structure to a simpler (labeled) Kripke structure using a *folding preorder* $\preccurlyeq$ between states.

Definition 5.5. *Given a transition system $\mathcal{A} = (A, \longrightarrow_{\mathcal{A}})$ with a set of states A and a transition relation $\longrightarrow_{\mathcal{A}} \subseteq A^2$, a* folding preorder *$\preccurlyeq \subseteq A^2$ is a reflexive and transitive relation that defines a simulation from $\mathcal{A}$ to $\mathcal{A}$.*

Each state $a \in A$ can be collapsed into a *previously seen* state $b \in A$, while traversing $\mathcal{A}$ from a set of initial states $I \subseteq A$, whenever b is *more general* than a according to the folding preorder $a \preccurlyeq b$. Such a *folded transition system* has in general much fewer states than the original system, and can collapse an infinite-state space to a finite-state space. This section explains how a *folding abstraction* of an LKS $\bar{\mathcal{K}}$ can be used to verify LTLR properties of $\bar{\mathcal{K}}$, while generating *no spurious counterexamples for safety properties.*

5.4.1 Folded Transition Systems

We can iteratively construct a folding abstraction of a transition system $\mathcal{A} = (A, \longrightarrow_{\mathcal{A}})$ from a set of initial states $I \subseteq A$ using a folding preorder $\preccurlyeq \subseteq A^2$ as shown in Definition 5.6 below. For a set of states $B \subseteq A$, let $Post_{\mathcal{A}}(B) = \{a \in A \mid \exists b \in B.\ b \longrightarrow_{\mathcal{A}} a\}$ (i.e., the *successors* of B) and $Post^*_{\mathcal{A}}(B) = \bigcup_{i\in\mathbb{N}}(Post_{\mathcal{A}})^i(B)$ (i.e., the *reachable states* from B).

Definition 5.6. *Given a transition system $\mathcal{A} = (A, \longrightarrow_{\mathcal{A}})$ and a folding preorder $\preccurlyeq$ on A, the* folding abstraction *from $I \subseteq A$ is the transition system*

$$\mathcal{R}each^{\preccurlyeq}_{\mathcal{A}}(I) = (Post^*_{\mathcal{A}\preccurlyeq}(I), \longrightarrow_{\mathcal{R}each^{\preccurlyeq}_{\mathcal{A}}(I)}),$$

where $Post^*_{\mathcal{A}\preccurlyeq}(I) = \bigcup_{i\in\mathbb{N}} Post^i_{\mathcal{A}\preccurlyeq}(I)$ such that $Post^{n+1}_{\mathcal{A}\preccurlyeq}(I)$ is the successor set of $Post^n_{\mathcal{A}\preccurlyeq}(I)$ not *subsumed by previously seen states:*

$$\begin{aligned} Post^0_{\mathcal{A}\preccurlyeq}(I) &= I, \\ Post^{n+1}_{\mathcal{A}\preccurlyeq}(I) &= \{a \in Post_{\mathcal{A}}(Post^n_{\mathcal{A}\preccurlyeq}(I)) \mid \forall l \leq n\ \forall b \in Post^l_{\mathcal{A}\preccurlyeq}(I).\ a \not\preccurlyeq b\}, \end{aligned}$$

and $\longrightarrow_{\mathcal{R}each^{\preccurlyeq}_{\mathcal{A}}(I)} = \bigcup_{i\in\mathbb{N}} \longrightarrow^{\preccurlyeq}_{\mathcal{A},i}$ such that $\longrightarrow^{\preccurlyeq}_{\mathcal{A},n+1}$ defines the transitions from $s \in Post^n_{\mathcal{A}\preccurlyeq}(I)$ to $t \in Post^l_{\mathcal{A}\preccurlyeq}(I)$ for $0 \leq l \leq n+1$, up to $n+1$ steps:

$$\begin{aligned} \longrightarrow^{\preccurlyeq}_{\mathcal{A},0} &= \emptyset, \\ \longrightarrow^{\preccurlyeq}_{\mathcal{A},n+1} &= \{(a, a') \in Post^n_{\mathcal{A}\preccurlyeq}(I) \times \bigcup_{0\leq i\leq n+1} Post^i_{\mathcal{A}\preccurlyeq}(I) \mid \exists b \in Post_{\mathcal{A}}(a).\ b \preccurlyeq a'\}. \end{aligned}$$

Example 5.5. *Consider the simplified specification of the readers-writers problem in Section 5.2.3, and a folding preorder $\preccurlyeq_{rw}$ between states defined by comparing the number of readers as follows:*

$$\langle R, W\rangle \preccurlyeq_{rw} \langle R, W\rangle \quad \wedge \quad \langle s^k(R), 0\rangle \preccurlyeq_{rw} \langle s(R), 0\rangle \ \ \textit{for } k \geq 1.$$

Then, we obtain the finite folding abstraction $\mathcal{R}each^{\preccurlyeq_{rw}}_{\mathcal{A}}(\{\langle 0,0\rangle\})$ from the initial state $\langle 0, 0\rangle$ in the right-hand side of Figure 5.5.

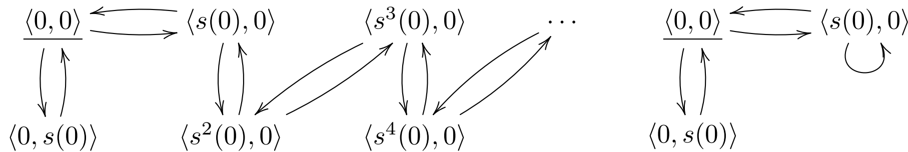


Figure 5.5: An infinite transition system from the initial state $\langle 0, 0\rangle$ (left), and its folded transition system by the folding preorder $\preccurlyeq_{rw}$ (right).

Each reachable state $a \in Post^*_{\mathcal{A}}(I)$ in $\mathcal{A}$ has a corresponding abstract state $\hat{a} \in Post^*_{\mathcal{A}\preccurlyeq}(I)$ in the folding abstraction $\mathcal{R}each^{\preccurlyeq}_{\mathcal{A}}(I)$ as follows.

Lemma 5.4. *Given a transition system $\mathcal{A} = (A, \longrightarrow_{\mathcal{A}})$, a folding preorder $\preccurlyeq \subseteq A^2$, and a set of states $I \subseteq A$, for each reachable state $s \in Post^*_{\mathcal{A}}(I)$, there exists a corresponding state $\hat{s} \in Post^*_{\mathcal{A}\preccurlyeq}(I)$ such that $s \preccurlyeq \hat{s}$.*

Proof. For a reachable state $a \in Post^*_{\mathcal{A}}(I)$ of $\mathcal{A}$, there exists a finite path $\pi_a : [n] \to A$ with length $n \in \mathbb{N}$, where $[n] = \{0, 1, \dots, n\}$, beginning in I and ending at a (i.e., $\pi_a(0) \in I$ and $\pi_a(n-1) = a$). We show this lemma by induction on the length of π_a. First, if $|\pi_a| = 0$, then

$$a \in I = Post^0_{\mathcal{A}\preccurlyeq}(I) \subseteq Post^*_{\mathcal{A}\preccurlyeq}(I), \quad \text{and} \quad a \preccurlyeq a.$$

Next, suppose that for any path π with length n beginning in the set of initial states I, there exists an abstract state $b_{n-1} \in Post^*_{\mathcal{A}\preccurlyeq}(I)$ such that $\pi(n-1) \preccurlyeq b_{n-1}$. Consider a path π_a with length $n+1$ such that $\pi_a(0) \in I$ and $\pi_a(n) = a$. By induction hypothesis, there exists $b_{n-1} \in Post^*_{\mathcal{A}\preccurlyeq}(I)$ such that $\pi_a(n-1) \preccurlyeq b_{n-1}$. Notice that $b_{n-1} \in Post^k_{\mathcal{A}\preccurlyeq}(I)$ for some $k \in \mathbb{N}$ by definition. Since $\preccurlyeq$ is a simulation from $\mathcal{A}$ to $\mathcal{A}$:

$$\begin{array}{ccccccc}
\pi_a(n-1) & \in & Post^*_{\mathcal{A}}(I) & \longrightarrow_{\mathcal{A}} & \pi_a(n) & \in & Post^*_{\mathcal{A}}(I) \\
\preccurlyeq & & & & \preccurlyeq & & \\
b_{n-1} & \in & Post^k_{\mathcal{A}\preccurlyeq}(I) & \longrightarrow_{\mathcal{A}} & \exists\, b_n & \in & Post_{\mathcal{A}}(Post^k_{\mathcal{A}\preccurlyeq}(I))
\end{array}$$

There are two possibilities. If $b_n \in Post^{k+1}_{\mathcal{A}\preccurlyeq}(I)$, we found $b_n \in Post^*_{\mathcal{A}\preccurlyeq}(I)$ such that $a = \pi_a(n) \preccurlyeq b_n$. Otherwise, there exist $l \leq k$ and $u \in Post^l_{\mathcal{A}\preccurlyeq}(I)$ such that $b_n \preccurlyeq u$, since by definition

$$Post^{k+1}_{\mathcal{A}\preccurlyeq}(I) = \{a \in Post_{\mathcal{A}}(Post^k_{\mathcal{A}\preccurlyeq}(I)) \mid \forall l \leq k\ \forall u \in Post^l_{\mathcal{A}\preccurlyeq}(I).\, a \not\preccurlyeq u\}.$$

That is, we found $u \in Post^*_{\mathcal{A}\preccurlyeq}(I)$ such that $a = \pi_a(n) \preccurlyeq b_n \preccurlyeq u$. Therefore, for each reachable state $a \in Post^*_{\mathcal{A}}(I)$ of $\mathcal{A}$, there exists a corresponding abstract state $\hat{a} \in Post^*_{\mathcal{A}\preccurlyeq}(I)$ such that $a \preccurlyeq \hat{a}$. □

Let $\mathcal{R}each_{\mathcal{A}}(I) = (Post^*_{\mathcal{A}}(I), \longrightarrow_{\mathcal{A}} \cap\, Post^*_{\mathcal{A}}(I)^2)$ be a *reachable subsystem* of a transition system $\mathcal{A} = (A, \longrightarrow_{\mathcal{A}})$ that only contains reachable states of $\mathcal{A}$ from a set of initial states I. Then, a folding abstraction $\mathcal{R}each^{\preccurlyeq}_{\mathcal{A}}(I)$ of $\mathcal{A}$ from I by any folding preorder $\preccurlyeq \subseteq A^2$ simulates the entire reachable subsystem $\mathcal{R}each_{\mathcal{A}}(I)$ *without folding*.

Lemma 5.5. *Given a transition system $\mathcal{A} = (A, \longrightarrow_{\mathcal{A}})$, a folding preorder $\preccurlyeq \subseteq A^2$, and a set of initial states $I \subseteq A$, the folding preorder $\preccurlyeq$ is a total simulation from $\mathcal{R}each_{\mathcal{A}}(I)$ to $\mathcal{R}each^{\preccurlyeq}_{\mathcal{A}}(I)$.*

Proof. Suppose $a \preccurlyeq b$ and $a \longrightarrow_{\mathcal{A}} a'$ for $a, a' \in Post^*_{\mathcal{A}}(I)$ and $b \in Post^*_{\mathcal{A}_{\preccurlyeq}}(I)$. By definition, $b \in Post^k_{\mathcal{A}_{\preccurlyeq}}(I)$ for some $k \in \mathbb{N}$. Since $\preccurlyeq$ is a simulation from $\mathcal{A}$ to $\mathcal{A}$, there exists $b' \in Post_{\mathcal{A}}(Post^k_{\mathcal{A}_{\preccurlyeq}}(I))$ such that $b \longrightarrow_{\mathcal{A}} b'$ and $a' \preccurlyeq b'$:

$$
\begin{array}{lllclll}
a & \in & Post^*_{\mathcal{A}}(I) & \longrightarrow_{\mathcal{A}} & a' & \in & Post^*_{\mathcal{A}}(I) \\
\preccurlyeq & & & & \preccurlyeq & & \\
b & \in & Post^k_{\mathcal{A}_{\preccurlyeq}}(I) & \longrightarrow_{\mathcal{A}} & \exists b' & \in & Post_{\mathcal{A}}(Post^k_{\mathcal{A}_{\preccurlyeq}}(I))
\end{array}
$$

There are also two possibilities. If $b' \in Post^{k+1}_{\mathcal{A}_{\preccurlyeq}}(I)$, we have $b \longrightarrow^{\preccurlyeq}_{\mathcal{A},k+1} b'$ such that $a' \preccurlyeq b'$. Otherwise, by definition of $Post^{k+1}_{\mathcal{A}_{\preccurlyeq}}(I)$, there exist $l \leq k$ and $b'' \in Post^l_{\mathcal{A}_{\preccurlyeq}}(I)$ such that $b' \preccurlyeq b''$, in a similar way to the proof of Lemma 5.4. By definition of $\longrightarrow^{\preccurlyeq}_{\mathcal{A},k+1}$, we then have $b \longrightarrow^{\preccurlyeq}_{\mathcal{A},k+1} b''$ again, where $a' \preccurlyeq b' \preccurlyeq b''$. Therefore, the folding preorder $\preccurlyeq$ is a simulation from $\mathcal{R}each_{\mathcal{A}}(I)$ to $\mathcal{R}each^{\preccurlyeq}_{\mathcal{A}}(I)$. Also, $\preccurlyeq$ is total by Lemma 5.4. □

Now consider an LKS $\bar{\mathcal{K}} = (S, AP, \mathcal{L}, ACT, \longrightarrow_{\bar{\mathcal{K}}})$ and a folding preorder $\preccurlyeq \subseteq S^2$. Since $\bar{\mathcal{K}}$ is an instance of the transition system $(S, \longrightarrow_{\bar{\mathcal{K}}})$, the folding preorder $\preccurlyeq$ defines a total simulation between the *transition systems* $\mathcal{R}each_{\bar{\mathcal{K}}}(I)$ and $\mathcal{R}each^{\preccurlyeq}_{\bar{\mathcal{K}}}(I)$ by Lemma 5.5. However, if $\preccurlyeq$ is also a simulation relation in terms of labeled Kripke structures, then by Definition 5.2, any two corresponding transitions by $\preccurlyeq$ satisfy the same set of state propositions and spatial action patterns. Therefore, the folding relation $\preccurlyeq$ also defines a total simulation between the *labeled Kripke structures* $\mathcal{R}each_{\bar{\mathcal{K}}}(I)$ and $\mathcal{R}each^{\preccurlyeq}_{\bar{\mathcal{K}}}(I)$. For model checking of an LTLR φ, only reachable states can contribute to the satisfaction of φ. Consequently, by Lemma 5.2:

Theorem 5.3. *Given an LKS $\bar{\mathcal{K}} = (S, AP, \mathcal{L}, ACT, \longrightarrow_{\bar{\mathcal{K}}})$, a set of initial states $S_0 \subseteq S$, and a folding preorder $\preccurlyeq \subseteq S^2$ that also defines a simulation from $\bar{\mathcal{K}}$ to $\bar{\mathcal{K}}$, for an LTLR formula φ and $s_0 \in S_0$:*

$$\mathcal{R}each^{\preccurlyeq}_{\bar{\mathcal{K}}}(S_0), s_0 \models \varphi \implies \bar{\mathcal{K}}, s_0 \models \varphi$$

Example 5.6. *Consider the folding preorder $\preccurlyeq_{rw}$ in Example 5.5 for the simplified specification of the readers-writers problem in Section 5.2.3. Since $\preccurlyeq_{rw}$ preserves the state propositions reading and writing, while $ACT = \emptyset$, $\preccurlyeq_{rw}$ also defines a simulation preorder on the corresponding LKS. Since the formula $\Box\neg(reading \wedge writing)$ holds in the folding abstraction, it also holds in the original (infinite-state) system by Theorem 5.3.*

5.4.2 Faithfulness for Safety Properties

A folding abstraction $\mathcal{R}each^{\preccurlyeq}_{\bar{\mathcal{K}}}(I)$ is in general an over-approximation of $\bar{\mathcal{K}}$. If an LTLR formula φ is *not* satisfied in $\mathcal{R}each^{\preccurlyeq}_{\bar{\mathcal{K}}}(I)$, it can generate a spurious counterexample for φ. Nonetheless, if a folding preorder $\preccurlyeq$ is *symmetric*, then $\preccurlyeq$ becomes a total bisimulation by Lemma 5.5, so that both satisfy exactly the same set of LTLR formulas.

Corollary 5.1. *Given an LKS $\bar{\mathcal{K}} = (S, AP, \mathcal{L}, ACT, \longrightarrow_{\bar{\mathcal{K}}})$, a* symmetric *folding preorder $\preccurlyeq \subseteq S^2$, and a set of initial states $S_0 \subseteq S$, for any LTLR formula φ and $s_0 \in S_0$, $\bar{\mathcal{K}}, s_0 \models \varphi$ iff $\mathcal{R}each^{\preccurlyeq}_{\bar{\mathcal{K}}}(S_0), s_0 \models \varphi$.*

Furthermore, a folding abstraction is *faithful* for invariants; that is, if there is a counterexample for any invariant $\Box\Phi$ in a folding abstraction $\mathcal{R}each^{\preccurlyeq}_{\bar{\mathcal{K}}}(I)$, where Φ is a Boolean formula over AP and ACT containing no temporal operators, there exists a *real* counterexample in the concrete LKS $\bar{\mathcal{K}}$. This faithfulness follows from the fact that each state in $\mathcal{R}each^{\preccurlyeq}_{\bar{\mathcal{K}}}(I)$ is still *reachable* from I in the original LKS $\bar{\mathcal{K}}$ by construction,

Lemma 5.6. *Given a transition system $\mathcal{A} = (A, \longrightarrow_{\mathcal{A}})$, a folding preorder $\preccurlyeq \subseteq A^2$, and a set of initial states $I \subseteq A$, we have $Post^*_{\mathcal{A}_{\preccurlyeq}}(I) \subseteq Post^*_{\mathcal{A}}(I)$.*

Proof. Recall that $Post^*_{\mathcal{A}_{\preccurlyeq}}(I) = \bigcup_{i \in \mathbb{N}} Post^i_{\mathcal{A}_{\preccurlyeq}}(I)$. Therefore, it suffices to show that $Post^k_{\mathcal{A}_{\preccurlyeq}}(I) \subseteq Post^*_{\mathcal{A}}(I)$ for $k \in \mathbb{N}$, by induction on k. First, by definition, $Post^0_{\mathcal{A}_{\preccurlyeq}}(I) = I \subseteq Post^*_{\mathcal{A}}(I)$. Suppose that $Post^n_{\mathcal{A}_{\preccurlyeq}}(I) \subseteq Post^*_{\mathcal{A}}(I)$ for some $n \in \mathbb{N}$. Since $Post^{n+1}_{\mathcal{A}_{\preccurlyeq}}(I) \subseteq Post_{\mathcal{A}}(Post^n_{\mathcal{A}_{\preccurlyeq}}(I))$, for $a' \in Post^{n+1}_{\mathcal{A}_{\preccurlyeq}}(I)$, there exists $a \in Post^n_{\mathcal{A}_{\preccurlyeq}}(I)$ such that $a \longrightarrow_{\mathcal{A}} s'$. By induction hypothesis, $a \in Post^*_{\mathcal{A}}(I)$, and thus $a' \in Post^*_{\mathcal{A}}(I)$. Hence, $Post^{n+1}_{\mathcal{A}_{\preccurlyeq}}(I) \subseteq Post^*_{\mathcal{A}}(I)$. Therefore, $Post^k_{\mathcal{A}_{\preccurlyeq}}(I) \subseteq Post^*_{\mathcal{A}}(I)$ for each $k \in \mathbb{N}$. $\Box$

If a folding abstraction $\mathcal{R}each^{\preccurlyeq}_{\bar{\mathcal{K}}}(I)$ does *not* satisfy an invariant, then there exists an *error* state $s \in Post^*_{\bar{\mathcal{K}}_{\preccurlyeq}}(I)$ in $\mathcal{R}each^{\preccurlyeq}_{\bar{\mathcal{K}}}(I)$ that violates the invariant. Because the error state s is again reachable from I in the original Kripke structure $\bar{\mathcal{K}}$ by Lemma 5.6, we can construct a *concrete counterexample* in $\bar{\mathcal{K}}$ by a backward search from s to I. Consequently:

Theorem 5.4 (Faithfulness for Invariants)**.** *Given an LKS $\bar{\mathcal{K}}$, a set of initial states $S_0 \subseteq S$, a folding preorder $\preccurlyeq \subseteq S^2$, for any invariant $\Box\Phi$ and $s_0 \in S_0$:*

$$\mathcal{R}each^{\preccurlyeq}_{\bar{\mathcal{K}}}(S_0), s_0 \models \Box\Phi \iff \bar{\mathcal{K}}, s_0 \models \Box\Phi.$$

For example, for the folding abstraction in Example 5.5, any counterexample found (if any) for invariants is *not* spurious.

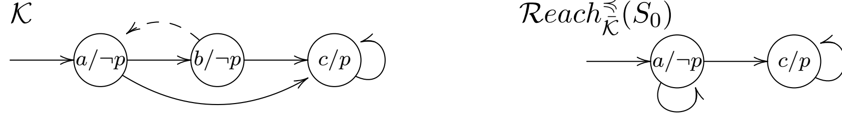


Figure 5.6: An LKS $\bar{\mathcal{K}}$ with $S = \{a, b, c\}$, $S_0 = \{a\}$, and $\mathcal{L}(c) = \{p\}$, and its folded abstraction $\mathcal{R}each^{\preccurlyeq}_{\bar{\mathcal{K}}}(S_0)$, where the dashed arrow denotes $b \preccurlyeq a$.

A folding abstraction $\mathcal{R}each^{\preccurlyeq}_{\bar{\mathcal{K}}}(S_0)$ is *not* a faithful abstraction for general LTLR formulas. For example, consider the simple LKS $\bar{\mathcal{K}}$ with only one state proposition p in Figure 5.6. Even though $\preccurlyeq$ is a folding preorder, for the formula $\bigcirc \bigcirc p$, $\bar{\mathcal{K}}, a \models \bigcirc \bigcirc p$, but $\mathcal{R}each^{\preccurlyeq}_{\bar{\mathcal{K}}}(S_0), a \not\models \bigcirc \bigcirc p$. As shown in Figure 5.6, there exists a spurious counterexample $(\neg p, \neg p, \neg p)$ in $\mathcal{R}each^{\preccurlyeq}_{\bar{\mathcal{K}}}(S_0)$. However, folding abstractions provide *faithful model checking procedures* for *safety* LTLR formulas. For a safety LTL formula φ, there is a *finite automaton* $\mathscr{F}_{\neg\varphi}$ that recognizes counterexamples for φ [32, 118]. Hence, as explained in Chapter 3, for a safety LTLR formula φ, there also exists a finite automaton $\mathscr{F}_{\neg\varphi}$ that recognizes its counterexamples.

Definition 5.7. *A finite automaton is a* 5*-tuple* $\mathscr{F} = (Q, Q_0, 2^{AP}, \delta, F)$ *with* Q *a finite set of states,* $Q_0 \subseteq Q$ *a set of initial states,* 2^{AP} *an alphabet of transition labels,* $\delta \subseteq Q \times 2^{AP} \times Q$ *a transition relation, and* $F \subseteq Q$ *a set of final states. The* language *accepted by* $\mathscr{F}$ *is the set* $L(\mathscr{F})$ *of finite runs of* $\mathscr{F}$ *starting in* Q_0 *and ending in* F*.*

Therefore, given an LKS $\bar{\mathcal{K}}$ and a set of initial states $S_0 \subseteq S$, the model checking problem of a *safety* LTLR formula φ can be characterized by using a finite automaton $\mathscr{F}_{\neg\varphi}$ associated to the negated formula $\neg\varphi$. Similarly, the *state/event product* of $\bar{\mathcal{K}}$ and $\mathscr{F}$ is the finite automaton

$$\bar{\mathcal{K}}[S_0] \otimes \mathscr{F} = (S \times Q, S_0 \times Q_0, 2^{AP \cup ACT}, \delta_{\bar{\mathcal{K}}}, S \times F)$$

such that $(s, b) \xrightarrow{\mathcal{L}(s) \cup \Lambda} (s', b') \in \delta_{\bar{\mathcal{K}}}$ iff $s \xrightarrow{\Lambda}_{\bar{\mathcal{K}}} s'$ and $b \xrightarrow{\mathcal{L}(s) \cup \Lambda} b' \in \delta$. Then, by the exactly same argument as Chapter 3, for a safety LTLR formula φ, we have $\bar{\mathcal{K}}, S_0 \models \varphi \iff L(\bar{\mathcal{K}}[S_0] \otimes \mathscr{F}_{\neg\varphi}) = \emptyset$.

Since the emptiness checking of the finite automaton $\bar{\mathcal{K}}[S_0] \otimes \mathscr{F}_{\neg\varphi}$ can be characterized by the reachability analysis of its final states, we can apply our previous result (Theorem 5.4) to *faithfully* abstract the synchronous product $\bar{\mathcal{K}}[S_0] \otimes \mathscr{F}_{\neg\varphi}$. For a folding preorder $\preccurlyeq$ of $\bar{\mathcal{K}}$, let the *product preorder* $\preccurlyeq_{\mathscr{F}} \subseteq (S \times Q)^2$ be defined by the equivalence:

$$(s, b) \preccurlyeq_{\mathscr{F}} (s', b') \iff s \preccurlyeq s' \wedge b = b'.$$

Lemma 5.7. *Given a finite automaton $\mathscr{F}$, and a folding preorder $\preccurlyeq$ and a set of initial state $S_0 \subseteq S$ for an LKS $\bar{\mathcal{K}}$, the product preorder $\preccurlyeq_{\mathscr{F}}$ is a folding preorder for the synchronous product $\bar{\mathcal{K}}[S_0] \otimes \mathscr{F}$.*

Proof. Suppose that $(s_1, b_1) \xrightarrow{\mathcal{L}(s_1)\cup\Lambda} (s_1', b_1') \in \delta_{\bar{\mathcal{K}}}$ and $(s_1, b_1) \preccurlyeq_{\mathscr{F}} (s_2, b_2)$. By definition, $s_1 \xrightarrow{\Lambda}_{\bar{\mathcal{K}}} s_1'$, $s_1 \preccurlyeq s_2$, and $b_1 = b_2$. Since $\preccurlyeq$ is a simulation from $\bar{\mathcal{K}}$ to $\bar{\mathcal{K}}$, there exists $s_2' \in S$ such that $s_2 \xrightarrow{\Lambda}_{\bar{\mathcal{K}}} s_2'$ and $s_1' \preccurlyeq s_2'$. Therefore, $(s_2, b_2) \xrightarrow{\mathcal{L}(s_2)\cup\Lambda} (s_2', b_1') \in \delta_{\bar{\mathcal{K}}}$, and $(s_1', b_1') \preccurlyeq_{\mathscr{F}} (s_2', b_1')$. Since $s_1 \preccurlyeq s_2$ implies $\mathcal{L}(s_1) = \mathcal{L}(s_2)$, $(s_2, b_2) \xrightarrow{\mathcal{L}(s_1)\cup\Lambda} (s_2', b_1') \in \delta_{\bar{\mathcal{K}}}$. Therefore, $\preccurlyeq_{\mathscr{F}}$ is a simulation to $\bar{\mathcal{K}}[S_0] \otimes \mathscr{F}$ and $\bar{\mathcal{K}}[S_0] \otimes \mathscr{F}$.[4] ☐

As a result, together with Theorem 5.4, for a safety LTLR formula φ, $L(\mathcal{R}each^{\preccurlyeq_{\mathscr{F}_{\neg\varphi}}}_{\bar{\mathcal{K}}[S_0]\otimes\mathscr{F}_{\neg\varphi}}(S_0 \times Q_0)) = \emptyset$ iff $L(\bar{\mathcal{K}}[S_0] \otimes \mathscr{F}_{\neg\varphi}) = \emptyset$. Consequently:

Theorem 5.5 (Faithfulness for Safety Properties)**.** *Given a labeled Kripke structure $\bar{\mathcal{K}} = (S, AP, \mathcal{L}, ACT, \longrightarrow_{\bar{\mathcal{K}}})$, a folding preorder $\preccurlyeq \subseteq S^2$, and a set of initial states $S_0 \subseteq S$, for a* safety *LTLR formula φ, there exists a* finite *automaton $\mathscr{F}_{\neg\varphi}$ with Q_0 a set of initial states such that:*

$$L(\mathcal{R}each^{\preccurlyeq_{\mathscr{F}_{\neg\varphi}}}_{\bar{\mathcal{K}}[S_0]\otimes\mathscr{F}_{\neg\varphi}}(S_0 \times Q_0)) = \emptyset \iff \bar{\mathcal{K}}, S_0 \models \varphi.$$

In order words, the product finite automaton $\mathcal{R}each^{\preccurlyeq_{\mathscr{F}_{\neg\varphi}}}_{\bar{\mathcal{K}}[S_0]\otimes\mathscr{F}_{\neg\varphi}}(S_0 \times Q_0)$ can be used to faithfully model check the safety LTLR formula φ.

The product automaton $\mathcal{R}each^{\preccurlyeq_{\mathscr{F}_{\neg\varphi}}}_{\bar{\mathcal{K}}[S_0]\otimes\mathscr{F}_{\neg\varphi}}(S_0 \times Q_0)$ can also be used to construct a faithful abstraction of $\bar{\mathcal{K}}$ for a safety LTLR formula φ, when $\mathscr{F}_{\neg\varphi}$ is *deterministic*. For $\mathcal{R}each^{\preccurlyeq_{\mathscr{F}_{\neg\varphi}}}_{\bar{\mathcal{K}}[S_0]\otimes\mathscr{F}_{\neg\varphi}}(S_0 \times Q_0) = (\widehat{Q}, \widehat{Q}_0, 2^{AP\cup ACT}, \widehat{\delta}, \widehat{F})$, we can construct the LKS $\bar{\mathcal{K}}[\mathscr{F}_{\neg\varphi}, S_0] = (\widehat{Q}, AP, \widehat{\mathcal{L}}, ACT, \rightarrow_{\bar{\mathcal{K}}[\mathscr{F}_{\neg\varphi}]})$, where:

- $\widehat{\mathcal{L}}((s, b)) = \mathcal{L}(s)$ for each $(s, b) \in \widehat{Q}$, and
- $(s, b) \xrightarrow{\Lambda}_{\bar{\mathcal{K}}[\mathscr{F}_{\neg\varphi}]} (s', b')$ iff $(s, b) \xrightarrow{\mathcal{L}(s)\cup\Lambda} (s', b') \in \widehat{\delta}$.

Lemma 5.8. *There exists a total simulation from $\mathcal{R}each_{\bar{\mathcal{K}}}(S_0)$ to the LKS $\bar{\mathcal{K}}[\mathscr{F}_{\neg\varphi}, S_0]$ induced by $\mathcal{R}each^{\preccurlyeq_{\mathscr{F}_{\neg\varphi}}}_{\bar{\mathcal{K}}[S_0]\otimes\mathscr{F}_{\neg\varphi}}(S_0 \times Q_0)$.*

Proof. Let $H = \{(s, (t, b)) \in S \times \widehat{Q} \mid s \preccurlyeq t\}$. Suppose that $s \xrightarrow{\Lambda}_{\bar{\mathcal{K}}} s'$ and $s\,H\,(t, b)$. For a state $b \in Q$ of $\mathscr{F}$, there is $b' \in Q$ with $(s, b) \xrightarrow{\mathcal{L}(s)\cup\Lambda} (s', b')$ in $\bar{\mathcal{K}}[S_0] \otimes \mathscr{F}$. Since $\preccurlyeq_{\mathscr{F}}$ is a total simulation from $\mathcal{R}each_{\bar{\mathcal{K}}}(S_0)[S_0] \otimes \mathscr{F}$ to $\mathcal{R}each^{\preccurlyeq_{\mathscr{F}}}_{\bar{\mathcal{K}}[S_0]\otimes\mathscr{F}}(S_0 \times Q_0)$ by Lemma 5.5 and Lemma 5.7, for some $(t', b') \in \widehat{Q}$, $(t, b) \xrightarrow{\mathcal{L}(t)\cup\Lambda} (t', b')$ in $\mathcal{R}each^{\preccurlyeq_{\mathscr{F}}}_{\bar{\mathcal{K}}[S_0]\otimes\mathscr{F}}(S_0 \times Q_0)$, and $(s', b') \preccurlyeq_{\mathscr{F}} (t', b')$. By definition, $s' \preccurlyeq t'$, and therefore $s' H (t', b')$. ☐

[4] Recall that for two finite automata $\mathscr{F}_i = (Q_i, Q_0^i, P, \delta_i, F_i)$, $i = 1, 2$, $H \subseteq Q_1 \times Q_2$ is a simulation iff for each $q_1 H q_2$, $q_1 \xrightarrow{a} q_1'$ implies $\exists q_2' \in B.\ q_2 \xrightarrow{a} q_2'$ and $q_1' H q_2'$.

Suppose that $\mathscr{F}$ is deterministic and $\mathscr{F}'$ is any (possibly nondeterministic) finite automaton such that $L(\mathscr{F}) = L(\mathscr{F}')$. If $L(\bar{\mathcal{K}}[\mathscr{F}, S_0] \otimes \mathscr{F}') \neq \emptyset$, then there exists a reachable accepting state $((s, b), b')$ of $\bar{\mathcal{K}}[\mathscr{F}, S_0] \otimes \mathscr{F}'$, and clearly, (s, b) is also reachable in $\bar{\mathcal{K}}[S_0] \otimes \mathscr{F}$. Because $L(\mathscr{F}) = L(\mathscr{F}')$ and $\mathscr{F}$ is deterministic, the state b is also an accepting state of $\mathscr{F}$. That is, $L(\bar{\mathcal{K}}[S_0] \otimes \mathscr{F}) \neq \emptyset$, and $L(\bar{\mathcal{K}}[S_0] \otimes \mathscr{F}') \neq \emptyset$ by the equivalence. Therefore:

Theorem 5.6 (Safety-Faithful Abstraction)**.** *Given an LKS $\bar{\mathcal{K}}$, a folding preorder $\preccurlyeq\ \subseteq S^2$, a* safety *LTLR formula φ, and a set of initial states $S_0 \subseteq S$, there exists an abstract LKS $\bar{\mathcal{K}}[\mathscr{F}_{\neg\varphi}, S_0]$ for a deterministic finite automaton $\mathscr{F}_{\neg\varphi}$ that* never *generates spurious counterexamples for φ.*

5.4.3 Safety Model Checking Procedure

A faithful folding abstraction for a safety LTLR property can be constructed on-the-fly by iteratively increasing its depth. Such *depth-bounded model checking* is useful for dealing with folding abstractions which may be infinite, since, as usual for abstractions of infinite-state systems, we cannot ensure a priori whether the folding abstraction will be finite. Therefore, we construct a *k-step folding abstraction* of $\bar{\mathcal{K}}$ whose states are reachable in k-steps from a set of initial states $I \subseteq S$. Such a depth k is iteratively incremented until a certain bound or until reaching a fixed-point if it exists.

Definition 5.8. *Given a transition system $\mathcal{A} = (A, \longrightarrow_{\mathcal{A}})$, a set of initial states $I \subseteq A$, and a folding preorder $\preccurlyeq$ on A, the k-*step folding abstraction *from I is the transition system $\mathcal{R}each_{\mathcal{A}}^{\preccurlyeq,k}(I) = (Post_{\mathcal{A}_{\preccurlyeq}}^{\leq k}(I), \longrightarrow_{\mathcal{R}each_{\mathcal{A}}^{\preccurlyeq,k}(I)})$, where $Post_{\mathcal{A}_{\preccurlyeq}}^{\leq k}(I) = \bigcup_{0 \leq i \leq k} Post_{\mathcal{A}_{\preccurlyeq}}^{i}(I)$ and $\longrightarrow_{\mathcal{R}each_{\mathcal{A}}^{\preccurlyeq,k}(I)} = \bigcup_{0 \leq i \leq k} \longrightarrow_{\mathcal{A},i}^{\preccurlyeq}$.*

For a (∞-step) folding abstraction $\mathcal{R}each_{\mathcal{A}}^{\preccurlyeq}(I)$, we can easily see that if its state set $Post_{\mathcal{A}_{\preccurlyeq}}^{*}(I)$ is finite, there exists a bound $n \in \mathbb{N}$ such that $\mathcal{R}each_{\mathcal{A}}^{\preccurlyeq,j}(I) = \mathcal{R}each_{\mathcal{A}}^{\preccurlyeq}(I)$ for any $j \geq n$. Therefore, unlike typical bounded model checking methods (e.g., [38]), our folding-based method can easily detect if $\mathcal{R}each_{\mathcal{A}}^{\preccurlyeq,n}(I)$ is *complete* or not.

The safety model checking procedure of a safety LTLR property φ for an LKS $\bar{\mathcal{K}}$ with a set of initial states S_0 and a folding preorder $\preccurlyeq$ consists in checking $\mathscr{S}_k = \mathcal{R}each_{\bar{\mathcal{K}}[S_0] \otimes \mathscr{F}_{\neg\varphi}}^{\preccurlyeq_{\mathscr{F}_{\neg\varphi}},k}(S_0 \times Q_0)$ for each $k \in \mathbb{N}$, where $\mathscr{F}_{\neg\varphi}$ is a finite automaton that recognizes the finite counterexamples of φ, iteratively from 0 until one of the following termination conditions holds: (i) $\mathscr{S}_k$ is complete (a fixpoint is found), (ii) a counterexample is found in $\mathscr{S}_k$, or (iii) k is greater than a given maximum bound n. Our depth-bounded model checking method can be briefly described as follows:

1. Apply a standard explicit-state model checking algorithm to check whether $\mathscr{S}_k$ contains an accepting state.

2. If $\mathscr{S}_k$ contains an accepting state, then there exists a counterexample of φ in $\bar{\mathcal{K}}$ by Theorem 5.5. Stop and return a counterexample that can be constructed by a backwards search in $\bar{\mathcal{K}}$ from the accepting state.

3. Suppose that there is no counterexample of φ in $\mathscr{S}_k$.

   (a) If k is greater than the maximal bound n, stop and report that $\bar{\mathcal{K}}$ does not violate φ until the current bound n.

   (b) Otherwise, compute $\mathscr{S}_{k+1}$: (i) if $\mathscr{S}_{k+1} = \mathscr{S}_k$, then stop and return *true* (the safety LTLR formula φ holds in $\bar{\mathcal{K}}$); (ii) if not, increment the depth-bound k by 1 and go to the first step.

Example. We illustrate how Dijkstra's algorithm in Section 5.2.2 can be verified by using our folding abstraction method. To collapse an *unbounded number* of processes to a bounded number we define two folding preorders by comparing a number of processes with specific patterns. If P^k denotes k P processes, the preorders are defined by:

$$\langle f, l_1, \texttt{off}\rangle^{k+1}\ PS \ \preccurlyeq_{l_1}\ \langle f, l_1, \texttt{off}\rangle\ PS$$
$$\langle f, l_2, \texttt{off}\rangle^{k+1}\ PS \ \preccurlyeq_{l_2}\ \langle f, l_2, \texttt{off}\rangle\ PS$$

Note that the infinite set of initial states $\{\langle 0, l_1, \texttt{off}\rangle^{k+1} \langle 0, l_1, \texttt{on}\rangle \mid k \in \mathbb{N}\}$ is collapsed into the finite set $\{\langle 0, l_1, \texttt{off}\rangle \langle 0, l_1, \texttt{on}\rangle\}$ by the preorder $\preccurlyeq_{l_1}$.

These preorder relations are *not* simulations in the original system. To make the folding preorders simulations between an infinite state space and a finite state space we define a variant of the system that is *stuttering bisimilar* [32] to the original system[5] by adding the following rules:

```
rl [l1'] : < {F,l1,off} PS >
       =>  < {F,l1,off} {1,l2,off} PS > .
rl [l2'] : < {F,l2,off} {0,S,on} PS >
       =>  < {F,l2,off} {F,l2,on} {0,S,off} PS > .
```

Both $\preccurlyeq_{l_1}$ and $\preccurlyeq_{l_2}$ clearly preserve the state proposition *ex*?. Also, the preorder $\preccurlyeq_{l_1}$ preserves transitions, since the newly added rules preserve `{F,l1,off}` processes, and similarly so does $\preccurlyeq_{l_2}$.[6]

[5] This extension is necessary to have a simulation between an infinite state space and a finite state space. However, it can be avoided by using a *symbolic representation* in which a single pattern can describe an infinite number of related states (see Section 5.5).

[6] This is similar to coherence completion for equational abstractions in Section 5.3.4.

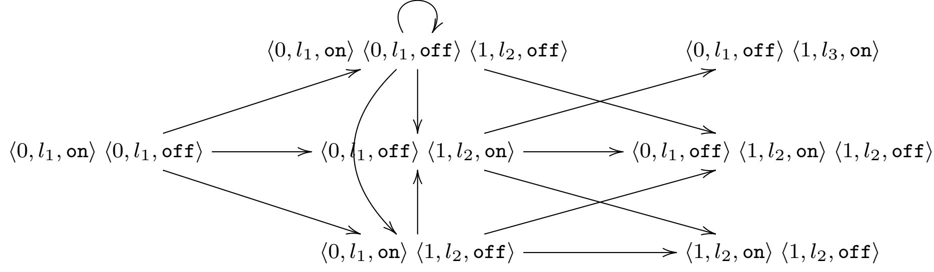


Figure 5.7: The finite 2-step folding abstraction by the folding preorder $\preccurlyeq_{l_1};\preccurlyeq_{l_2}$ for the Dijkstra's mutex algorithm from the set of initial states $\{\langle 0, l_1, \texttt{off}\rangle^{k+1}\,\langle 0, l_1, \texttt{on}\rangle \mid k \in \mathbb{N}\}$ with an unbounded number of processes.

By the composed folding preorder $\preccurlyeq_{l_1};\preccurlyeq_{l_2}$, the 2-step folding abstraction is shown in Figure 5.7 from the initial state $\{\langle 0, l_1, \texttt{off}\rangle\,\langle 0, l_1, \texttt{on}\rangle\}$, which represents the infinite set of initial states

$$\{\langle 0, l_1, \texttt{off}\rangle^{k+1}\,\langle 0, l_1, \texttt{on}\rangle \mid k \in \mathbb{N}\}$$

with an unbounded number of processes. Since there is no state containing two *crit* processes in the 2-step folding abstraction, we can ensure that there exists no counterexample of $\Box ex?$ of length less than or equal to 2. In fact, by using the composed folding preorder $\preccurlyeq_{l_1};\preccurlyeq_{l_2}$, we can obtain the folding abstraction with only 15 states from the abstract initial state $\{\langle 0, l_1, \texttt{off}\rangle\,\langle 0, l_1, \texttt{on}\rangle\}$ with *no* state containing two *crit* processes. Hence, by Theorem 5.3 and the stuttering bisimulation, the mutual exclusion $\Box ex?$ is also satisfied in the original system for an unbounded number of processes from the infinite set of initial states $\{\langle 0, l_1, \texttt{off}\rangle^{k+1}\,\langle 0, l_1, \texttt{on}\rangle \mid k \in \mathbb{N}\}$.

5.5 Narrowing-based Logical Abstraction

Narrowing [110, 112] generalizes term rewriting by allowing free variables in terms and by performing unification instead of matching. An *E-unifier* of an equation $t = t'$ is a substitution σ such that

$$\sigma t =_E \sigma t' \quad \text{and} \quad \mathit{dom}(\sigma) \subseteq \mathit{vars}(t) \cup \mathit{vars}(t'),$$

and $\mathit{CSU}_E(t = t')$ denotes a *complete set of E-unifiers* such that for any E-unifier ρ of $t = t'$, there is a more general substitution $\sigma \in \mathit{CSU}_E(t = t')$, i.e., $(\exists \eta)\ \rho =_E \eta \circ \sigma$. We assume that there exists a finitary E-unification algorithm to find a *finite* complete set $\mathit{CSU}_E(t = t')$ (e.g., there is a finitary E-unification algorithm if E has the *finite variant property* [64, 87]).

Definition 5.9. *For a* topmost unconditional *rewrite theory* $\mathcal{R} = (\Sigma, E, R)$*, each rule* $l : q \longrightarrow r \in R$ *specifies a topmost* narrowing step $t \rightsquigarrow_{l,\sigma,\mathcal{R}} t'$ *(or* $t \rightsquigarrow_{\mathcal{R}} t'$*) iff there exists an* E*-unifier* $\sigma \in CSU_E(t = q)$ *such that* $t' = \sigma(r)$*.*

Throughout this section we assume that a rewrite theory $\mathcal{R}$ is a topmost unconditional order-sorted State-deadlock free rewrite theory.

Such a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ also specifies a *logical* transition system $\mathcal{N}(\mathcal{R})$ [86]. The states of $\mathcal{N}(\mathcal{R})$ are elements of the free algebra $\mathcal{T}_{\Sigma/E}(\mathcal{X})_{\mathsf{State}}$, and its transitions are specified by topmost narrowing steps $\rightsquigarrow_{\mathcal{R}}$. A state of $\mathcal{N}(\mathcal{R})$ is not a *concrete state* (i.e., ground term), but a *state pattern* $t(x_1, \ldots, x_n)$ with *logical variables* $x_1, \ldots, x_n$, representing the set of all concrete states $[\theta t]_E$ that are its *ground instances*. Using such logical representation, this section presents narrowing-based LTLR model checking for infinite-state systems. A special form of *folding* abstractions can be *automatically* applied for narrowing-based model checking. Since the logical state space can still be infinite, we also show how *equational* abstraction can be combined with narrowing-based model checking.

5.5.1 Spatial Action Patterns for Narrowing

Spatial action patterns for rewriting define their matching one-step proof terms, representing the corresponding one-step rewrites (see Chapter 3). For a topmost rewrite theory $\mathcal{R} = (\Sigma, E, R)$, one-step proof terms have the form $l(\theta)$, indicating that a rule $l : q \longrightarrow r$ has been applied with a substitution θ (at the top position of the term), where $dom(\theta) \subseteq vars(q) \cup vars(r)$. To define *spatial action patterns for narrowing steps*, we also need an appropriate notion of one-step proof terms for narrowing. Consider a topmost narrowing step $t \rightsquigarrow_{l,\sigma,\mathcal{R}} t'$ using a rule $l : q \longrightarrow r$. Intuitively, the rule label l and the restriction of the substitution σ to the variables in the rule[7] give the one-step proof term for the narrowing step $t \rightsquigarrow_{l,\sigma,\mathcal{R}} t'$.

Definition 5.10. *Given a* topmost *rewrite theory* $\mathcal{R} = (\Sigma, E, R)$*, for a topmost narrowing step* $t \rightsquigarrow_{l,\sigma,\mathcal{R}} t'$ *using a rule* $l : q \longrightarrow r$*, its one-step proof term is given by* $l(\sigma|_{vars(q) \cup vars(r)})$*, often denoted by* $l(\sigma_l)$*.*

The following lemma implies that a one-step proof term $l(\sigma_l)$ for narrowing faithfully captures its corresponding one-step proof terms $l(\theta)$ for rewriting, in the sense that $\theta =_E \eta \circ \sigma_l$ for some substitution η. This lemma is adapted from the soundness and completeness results of topmost narrowing [142].

[7]Since one-step proof terms for rewriting only contain variables in rules, we restrict one-step proof terms for narrowing in the same way.

Lemma 5.9. *Given a topmost rewrite theory $\mathcal{R} = (\Sigma, E, R)$, a non-variable term u, and a substitution ρ, assuming no variable in u appears in the rules:*

$$\begin{aligned} & (\exists t',\, \theta)\;\; l(\theta) : \rho u \longrightarrow_{\mathcal{R}} t' \\ \iff \quad & (\exists u',\, \sigma,\, \eta)\;\; u \rightsquigarrow_{l,\sigma,\mathcal{R}} u' \;\;\wedge\;\; \rho|_{vars(u)} =_E (\eta \circ \sigma)|_{vars(u)}, \end{aligned}$$

where $\theta =_E (\eta \circ \sigma)|_{dom(\theta)}$ and $t' =_E \eta u'$.

Proof. ($\Rightarrow$) Suppose that $l(\theta) : \rho u \longrightarrow_{\mathcal{R}} t'$ for a topmost rule $l : q \longrightarrow r$, where $dom(\theta) \subseteq vars(q) \cup vars(r)$. Then, $\theta q =_E \rho u$ and $t' = \theta r$. Since *no* variable in u appears in $l : q \longrightarrow r$, we have $dom(\theta) \cap vars(u) = \emptyset$. Thus, we can define the substitution $\theta \cup \rho|_{vars(u)}$ with domain $dom(\theta) \cup vars(u)$ such that $(\theta \cup \rho|_{vars(u)})|_{dom(\theta)} = \theta$ and $(\theta \cup \rho|_{vars(u)})|_{vars(u)} = \rho|_{vars(u)}$. Because $\theta \cup \rho|_{vars(u)}$ is an E-unifier of $q = u$, there exist substitutions $\sigma \in CSU_E(u = q)$ and η' satisfying $(\theta \cup \rho|_{vars(u)})|_{vars(q) \cup vars(u)} =_E \eta' \circ \sigma$ with domain $vars(q) \cup vars(u)$. Therefore, $u \rightsquigarrow_{l,\sigma,\mathcal{R}} u'$ for $u' = \sigma r$. Next, let η be the extended substitution such that $\eta x = \eta' x$ if $x \in vars(q) \cup vars(u)$, and $\eta x = \theta x$ otherwise. Then, we have:

$$\rho|_{vars(u)} =_E (\eta \circ \sigma)|_{vars(u)} \quad \text{and} \quad \theta =_E (\eta \circ \sigma)|_{dom(\theta)},$$

since $dom(\theta) \cap vars(u) = \emptyset$ and $dom(\theta) \subseteq vars(q) \cup vars(r)$. Furthermore, $t' = \theta r =_E (\eta \circ \sigma) r = \eta u'$.
($\Leftarrow$) Suppose that $u \rightsquigarrow_{l,\sigma,\mathcal{R}} u'$ and $\rho|_{vars(u)} =_E (\eta \circ \sigma)|_{vars(u)}$. Then, for a topmost rule $l : q \longrightarrow r$, $\sigma \in CSU_E(u = q)$ and $u' = \sigma r$. Since $\sigma u =_E \sigma q$ and $(vars(q) \cup vars(r)) \cap vars(u) = \emptyset$, $l(\sigma|_{vars(q) \cup vars(r)}) : \sigma u \longrightarrow_{\mathcal{R}} u'$. Therefore, $l(\eta \circ \sigma|_{vars(q) \cup vars(r)}) : (\eta \circ \sigma) u \longrightarrow_{\mathcal{R}} \eta u'$, where $(\eta \circ \sigma) u =_E \rho u$, since rewrites are stable under substitutions. □

The semantics of a spatial action pattern can be defined by means of equations using the auxiliary operator _⊨_ : ProofTerm Action → Bool. By definition, $\delta \in \mathcal{T}_{\Sigma/E,Action}$ is matched to a one-step proof term γ iff $(\gamma \models \delta) =_E \mathit{true}$. For a topmost rewrite theory $\mathcal{R}$, a one-step proof term $l(\theta)$ can be represented as a term {$'l$: $'x_1 \backslash \theta x_1$; ... ; $'x_m \backslash \theta x_m$} of sort ProofTerm, as explained in Section 3.2.1.

Example 5.7. *Consider the bakery example in Section 5.2.1. A topmost narrowing step from the term* `N ; N ; [0,idle]` *by the wake rule gives the following one-step proof term:*

```
{'wake : 'N \ N ; 'M \ N ; 'I \ 0 ; 'PS \ none}.
```

For narrowing-based model checking we further require that there exists a finitary E-unification procedure. If a spatial action pattern δ is identified by a one-step proof term *pattern* u_δ (that is, $(\gamma \models \delta) =_E$ *true* iff γ is a substitution instance of u_δ),[8] and if u_δ *has complement patterns* $u_1, \dots, u_k$ (i.e., any ground one-step proof term is an instance of exactly one term in $\{u_\delta, u_1, \dots, u_k\}$), then δ can be defined by the equations:

$$u_\delta \models \delta = \mathit{true}, \quad u_1 \models \delta = \mathit{false}, \quad \dots, \quad u_k \models \delta = \mathit{false}.$$

Since the right-hand sides are all constants, these equations have the finite variant property, and thus they provide a finitary E-unification algorithm [64, 87]. Of course, this method can also be applied for "pattern-like" state propositions, as already illustrated in Section 5.2.

Several effective methods have been developed to check when a term t has complements and to compute such complement patterns, not only in the free case [121], but also modulo the associative and commutative (AC) axioms and modulo permutative theories [92, 93]. Hence, for unconditional rewrite theories with axioms B such as those used in [92, 93, 121], we can determine under fairly general conditions if a one-step proof term pattern u_δ of δ has complements, compute such complement patterns, and define pattern satisfaction of δ by equations.

Example 5.8. *Consider the spatial action pattern wake(0) in the bakery example of Section 5.2.1. The positive case can be defined by the equation:*

```
eq {'wake : 'I \ 0; SUBST} |= wake(0) = true .
```

For the negative cases, wake(0) does not *hold when the rule label is* not *`'wake` or the value of `'I` is* not *`0`. Therefore, they can be defined by the complement patterns of* 0 *and `'wake` as follows.*

```
eq {'wake : 'I \ s J ; SUBST} |= wake(0) = false .
eq {'crit : SUBST} |= wake(0) = false .
eq {'exit : SUBST} |= wake(0) = false .
```

The use of order-sorted signatures can greatly facilitate the existence of complement patterns that may not exist in an unsorted setting, as also illustrated in Section 5.2. For example, the unsorted term $y + 0 + 0$ for a signature with a constant 0, a unary s, and an AC symbol $+$ is shown not to have complements in [92], but can be easily shown to have complements when the signature is refined to an order-sorted signature.

[8] The spatial action patterns in $SP(\mathcal{R})$ are identified in this way, and their negative cases can be automatically defined by `owise` equations (see Chapter 3); however, the underlying E-unification procedure [87] does not support such `owise` equations.

5.5.2 Narrowing-based Labeled Kripke Structures

For a set of state propositions AP and a set of spatial action patterns ACT defined by an associated equational theory $\mathcal{P}$, we can define for a topmost rewrite theory $\mathcal{R} = (\Sigma, E, R)$ a corresponding *narrowing-based logical LKS* $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$. Each state of $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$ is a term in which the truth of every state proposition is decided into either *true* or *false*. A transition of $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$ is specified by using a topmost narrowing step $\rightsquigarrow_{\mathcal{R}}$, but further instantiated into possibly several transitions, so that the truth b_i of each state proposition p_i, where $1 \leq i \leq n$, and the truth b_{n+j} of each spatial action pattern δ_j, where $1 \leq j \leq m$, are decided into *true* or *false*.

Definition 5.11. *Given a topmost rewrite theory* $\mathcal{R} = (\Sigma, E, R)$ *and* finite *sets* $AP = \{p_1, \ldots, p_n\}$ *and* $ACT = \{\delta_1, \ldots, \delta_m\}$ *defined by an associated equational theory* $\mathcal{P} = (\Pi, D)$*, the narrowing-based* logical labeled Kripke structure *is* $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}} = (N(\mathcal{R})_{AP}, AP, \mathcal{L}_{\mathcal{P}}, ACT, \longrightarrow_{\bar{\mathcal{N}}(\mathcal{R})})$*, where:*

- $[t]_E \in N(\mathcal{R})_{AP}$ *iff* $[t]_E \in \mathcal{T}_{\Sigma/E}(\mathcal{X})_{\mathsf{State}} - \mathcal{X}$*, and for every* $p \in AP$*, either* $(t \models p) =_{E \cup D}$ *true or* $(t \models p) =_{E \cup D}$ *false;*
- $\mathcal{L}_{\mathcal{P}}([t]_E) = \{p \in AP \mid (t \models p) =_{E \cup D} \mathit{true}\}$*; and*
- $[t]_E \xrightarrow{\Lambda}_{\bar{\mathcal{N}}(\mathcal{R})} [t']_E$ *iff there exist a term* u*, a substitution* ζ*, and Boolean values* $b_1, \ldots, b_{n+m} \in \{\mathit{true}, \mathit{false}\}$ *such that*

$$
\begin{aligned}
t \rightsquigarrow_{l,\sigma,\mathcal{R}} u \ \wedge \ t' = \zeta u \ \wedge \\
\Lambda = \{\delta \in ACT \mid (\zeta(l(\sigma_l)) \models \delta) =_{E \cup D} \mathit{true}\} \ \wedge \\
\zeta \in CSU_{E \cup D}\Big(\bigwedge_{1 \leq i \leq n} (u \models p_i) = b_i \ \wedge \bigwedge_{1 \leq j \leq m} (l(\sigma_l) \models \delta_j) = b_{n+j}\Big).
\end{aligned}
$$

Example 5.9. *Consider the bakery example in Section 5.2.1. For the* logical *initial state* `N ; N ; [0,idle]`*, we obtain within the logical LKS* $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$ *the infinite path in Figure 5.8, which captures an infinite number of concrete paths in* $\bar{\mathcal{K}}(\mathcal{R}, \mathsf{State})_{\mathcal{P}}$ *from each ground instance of* `N ; N ; [0,idle]`*.*

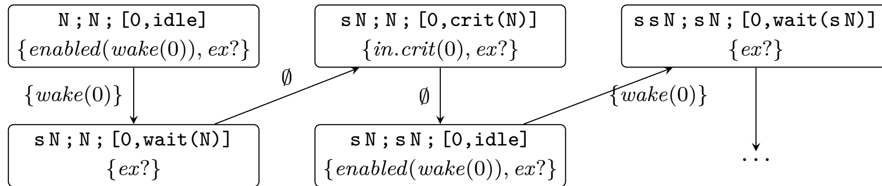


Figure 5.8: A path from `N ; N ; [0,idle]` in the logical LKS $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$.

For a narrowing-based LKS $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$, each logical state is related to a concrete state in $\bar{\mathcal{K}}(\mathcal{R}, \mathsf{State})_{\mathcal{P}}$ in terms of the *E-subsumption* relation. The E-subsumption $t \preccurlyeq_E t'$ holds iff there exists a substitution σ with $t =_E \sigma t'$, meaning that t' is *more general* than t modulo E.

Lemma 5.10. *Given a topmost rewrite theory $\mathcal{R} = (\Sigma, E, R)$ and finite sets AP and ACT defined by $\mathcal{P} = (\Pi, D)$, the E-subsumption $\preccurlyeq_E$ is a total simulation from the concrete LKS $\bar{\mathcal{K}}(\mathcal{R}, \mathsf{State})_{\mathcal{P}}$ to $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$.*

Proof. Suppose that $[t]_E \xrightarrow{\Lambda}_{\bar{\mathcal{K}}(\mathcal{R})} [t']_E$ and $t \preccurlyeq_E u$ for $u \in N(\mathcal{R})_{AP}$. Given $AP = \{p_1, \ldots, p_n\}$ and $ACT = \{\delta_1, \ldots, \delta_m\}$, fix $b_1, \ldots, b_{n+m} \in \{true, false\}$ such that for $1 \leq i \leq n$ and $1 \leq j \leq m$:

$$b_i =_{E \cup D} (t' \models p_i) \quad \text{and} \quad b_{n+j} =_{E \cup D} (l(\theta) \models \delta_j)$$

By definition, there is an one-step rewrite $l(\theta) : t \longrightarrow_{\mathcal{R}} t'$. Therefore, by Lemma 5.9, there exists a narrowing step $u \rightsquigarrow_{l,\sigma,\mathcal{R}} u'$ such that $t' =_E \eta u'$ and $\theta =_E (\eta \circ \sigma)|_{dom(\theta)}$. Hence, there exists

$$\zeta \in CSU_E(\textstyle\bigwedge_{1 \leq i \leq n}(u' \models p_i) = b_i \;\wedge\; \bigwedge_{1 \leq j \leq m}(l(\sigma_l) \models \delta_j) = b_{n+j}).$$

By definition, $[u]_E \xrightarrow{\Lambda}_{\bar{\mathcal{N}}(\mathcal{R})} [\zeta u']_E$. Since $\bigwedge_{1 \leq i \leq n} \eta((u' \models p_i) =_{E \cup D} b_i)$ and $\bigwedge_{1 \leq j \leq m} \eta((l(\sigma_l) \models \delta_j) =_{E \cup D} b_{n+j})$, $\eta \preccurlyeq_E \zeta$, and $t' =_E \eta u \preccurlyeq_E \zeta u'$. $\square$

This lemma implies that, by Lemma 5.2, any LTLR formula φ satisfied in a narrowing-based LKS $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$ from a logical state t is also satisfied in the concrete LKS $\bar{\mathcal{K}}(\mathcal{R}, \mathsf{State})_{\mathcal{P}}$ from each ground instance of t. However, $\preccurlyeq_E$ is *not* a bisimulation between $\bar{\mathcal{K}}(\mathcal{R}, \mathsf{State})_{\mathcal{P}}$ and $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$ in general.

Example 5.10. *Consider the bakery example in Section 5.2.1. Although* `0 ; 0 ; [I,wait(0)]` $\preccurlyeq_E$ `N ; M ;` PS_1 *holds, there exists a transition*

$$\texttt{N ; M ; } PS_1 \xrightarrow{\{wake(0)\}}_{\bar{\mathcal{N}}(\mathcal{R})} \texttt{s N ; M ; } PS_2 \texttt{ [0,wait(N)]},$$

in $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$ with the substitution $PS_1 \backslash PS_2$ `[0,idle]`, *but* no corresponding transition *exists from* `0 ; 0 ; [I,wait(0)]` *in $\bar{\mathcal{K}}(\mathcal{R}, \mathsf{State})_{\mathcal{P}}$. However, any* finite *path in $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$ can be* instantiated *to a corresponding concrete path in $\bar{\mathcal{K}}(\mathcal{R}, \mathsf{State})_{\mathcal{P}}$; e.g., the above transition can be instantiated as the transition*

$$\texttt{0 ; 0 ; [0,idle]} \xrightarrow{\{wake(0)\}}_{\bar{\mathcal{K}}(\mathcal{R})} \texttt{s ; 0 ; [0,wait(0)]}$$

As hinted at in the above example, for any finite logical path in $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$, there exists a corresponding concrete path in $\bar{\mathcal{K}}(\mathcal{R}, \mathsf{State})_{\mathcal{P}}$.

Lemma 5.11. *For a finite logical path* $u_1 \xrightarrow{\Lambda_1}_{\bar{\mathcal{N}}(\mathcal{R})} \cdots \xrightarrow{\Lambda_{n-1}}_{\bar{\mathcal{N}}(\mathcal{R})} u_n$ *of* $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$*, there exists a concrete path* $t_1 \xrightarrow{\Lambda_1}_{\bar{\mathcal{K}}(\mathcal{R})} \cdots \xrightarrow{\Lambda_{n-1}}_{\bar{\mathcal{K}}(\mathcal{R})} t_n$ *in the concrete LKS* $\bar{\mathcal{K}}(\mathcal{R}, \mathsf{State})_{\mathcal{P}}$ *such that* $t_i \preccurlyeq_E u_i$ *for each* $1 \leq i \leq n$.

Proof. Since $u_1 \xrightarrow{\Lambda_1}_{\bar{\mathcal{N}}(\mathcal{R})} u_2$, by definition, there are substitutions σ_1 and ζ_1 such that $u_1 \rightsquigarrow_{l_1,\sigma_1,\mathcal{R}} u_2'$ by a rule $l_1 : q_1 \to r_1 \in R$ and $u_2 = \zeta_1 u_2'$. Since $\sigma u_1 =_E \sigma q_1$ and $u_2 = \zeta_1 u_2' = (\zeta_1 \circ \sigma_1) r_1$, $(\zeta_1 \circ \sigma_1) u_1 \longrightarrow_{\mathcal{R}} u_2$. Similarly, $(\zeta_2 \circ \sigma_2) u_2 \longrightarrow_{\mathcal{R}} u_3$, $(\zeta_3 \circ \sigma_3) u_3 \longrightarrow_{\mathcal{R}} u_4$, etc. By composing them, we have:

$$(\zeta_{n-1} \circ \sigma_{n-1} \circ \cdots \circ \zeta_2 \circ \sigma_2 \circ \zeta_1 \circ \sigma_1) u_1 \longrightarrow_{\mathcal{R}} \cdots \longrightarrow_{\mathcal{R}} (\zeta_{n-1} \circ \sigma_{n-1}) u_{n-1} \longrightarrow_{\mathcal{R}} u_n.$$

Let ρ be a ground substitution instantiating every variable in the path, which exists since we assume that $\mathcal{T}_{\Sigma,s} \neq \emptyset$ for each sort s. Then, the path $(\rho \circ \zeta_{n-1} \circ \sigma_{n-1} \circ \cdots \circ \zeta_2 \circ \sigma_1) u_1 \longrightarrow_{\mathcal{R}} \cdots \longrightarrow_{\mathcal{R}} (\rho \circ \zeta_{n-1} \circ \sigma_{n-1}) u_{n-1} \longrightarrow_{\mathcal{R}} \rho u_n$ gives a desired concrete path in $\bar{\mathcal{K}}(\mathcal{R}, \mathsf{State})_{\mathcal{P}}$. ☐

Recall that counterexamples of *safety properties* are characterized by finite sequences [32, 60]. Therefore, the above lemma guarantees that $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$ does *not* generate spurious counterexamples for safety properties, since any finite counterexample in $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$ has a corresponding *real* counterexample in $\bar{\mathcal{K}}(\mathcal{R}, \mathsf{State})_{\mathcal{P}}$. Together with Lemma 5.2 and Lemma 5.10, we have:

Theorem 5.7. *Given a topmost rewrite theory* $\mathcal{R} = (\Sigma, E, R)$ *and finite sets AP and ACT defined by an associated equational theory* $\mathcal{P} = (\Pi, D)$*, for a safety LTLR formula* φ *and a pattern* $t \in N(\mathcal{R})_{AP}$*:*

$$\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}, [t]_E \models \varphi \iff (\forall \theta : \mathcal{X} \to \mathcal{T}_{\Sigma}) \ \bar{\mathcal{K}}(\mathcal{R}, \mathsf{State})_{\mathcal{P}}, [\theta t]_E \models \varphi.$$

5.5.3 Abstract Narrowing-based Model Checking

A narrowing-based logical LKS $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$ can have an infinite number of *logical* reachable states (e.g., see Figure 5.8). Hence, to reduce such an infinite-state narrowing-based LKS $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$ to a finite logical LKS, this section presents folding abstractions and equational abstractions of $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$, as special cases of those in Sections 5.3 and 5.4. Both folding abstraction and equational abstraction can be seamlessly combined to verify non-trivial infinite-state systems by narrowing-based model checking. Furthermore, since equational abstractions can be *bisimilar* and folding abstractions are *faithful* for safety properties, a combination of these abstractions can give a *faithful* abstraction that generates *no* spurious counterexamples for safety properties.

Folding Abstractions. A folding abstraction of a transition system $\mathcal{A}$ collapses each state a into a more general state b according to a folding preorder $a \preccurlyeq b$. To construct a folding abstraction of a narrowing-based LKS $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$ using the E-subsumption preorder $\preccurlyeq_E$, we need to show that $\preccurlyeq_E$ is a total simulation from $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$ to $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$.

Lemma 5.12. *Given a topmost rewrite theory $\mathcal{R} = (\Sigma, E, R)$ and a support theory $\mathcal{P} = (\Pi, D)$, $\preccurlyeq_E$ is a total simulation from $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$ to $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$.*

Proof. Suppose that $[t]_E \xrightarrow{\Lambda}_{\bar{\mathcal{N}}(\mathcal{R})} [t']_E$ and $t = \rho u$ for a substitution ρ (i.e., $t \preccurlyeq_E u$). Given $AP = \{p_1, \dots, p_n\}$ and $ACT = \{\delta_1, \dots, \delta_m\}$, by definition, there are substitutions σ and ζ such that $t \rightsquigarrow_{l,\sigma,\mathcal{R}} t''$ by $l : q \longrightarrow r \in R$ and $t' = \zeta t''$, where $\sigma \in CSU_E(t = q)$ and for some $b_1, \dots, b_{n+m} \in \{true, false\}$:

$$\zeta \in CSU_{E \cup D}\Big(\textstyle\bigwedge_{1 \le i \le n}(t'' \models p_i) = b_i \ \wedge\ \bigwedge_{1 \le j \le m}(l(\sigma_l) \models \delta_j) = b_{n+j}\Big) \quad (5.1)$$

Since $t'' = \sigma r$ and $\sigma t =_E \sigma q$, we have $t' = \zeta t'' = (\zeta \circ \sigma)r$, and therefore $l(\zeta \circ \sigma) : (\zeta \circ \sigma)t \longrightarrow_{\mathcal{R}} t'$. Since $t = \rho u$, $l(\zeta \circ \sigma) : (\zeta \circ \sigma \circ \rho)u \longrightarrow_{\mathcal{R}} t'$. By Lemma 5.9, there exists a narrowing step $u \rightsquigarrow_{l,\varsigma,\mathcal{R}} u'$ such that $t' =_E \eta u'$ and $(\zeta \circ \sigma)|_{vars(q) \cup vars(r)} =_E (\eta \circ \varsigma)|_{vars(q) \cup vars(r)}$. This implies:

$$\zeta t'' =_E \eta u' \quad \text{and} \quad \zeta(l(\sigma_l)) =_E \eta(l(\varsigma_l)). \quad (5.2)$$

Hence, $\nu \in CSU_{E \cup D}(\bigwedge_{1 \le i \le n}(u' \models p_i) = b_i \ \wedge\ \bigwedge_{1 \le j \le m}(l(\varsigma_l) \models \delta_j) = b_{n+j})$ exists. By definition, $[u]_E \xrightarrow{\Lambda}_{\bar{\mathcal{N}}(\mathcal{R})} [\nu u']_E$. Notice that by (5.1) and (5.2), $\bigwedge_{1 \le i \le n}\eta((u' \models p_i) =_{E \cup D} b_i)$ and $\bigwedge_{1 \le j \le m}\eta((l(\varsigma_l) \models \delta_j) =_{E \cup D} b_{n+j})$ hold. Therefore, $\eta \preccurlyeq_E \nu$, and $t' =_E \eta u \preccurlyeq_E \nu u'$. □

Therefore, by Lemma 5.5, given a set of initial states $S_0 \subseteq S$, $\preccurlyeq_E$ defines a total simulation from $\mathcal{R}each_{\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}}(S_0)$ to $\mathcal{R}each^{\preccurlyeq_E}_{\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}}(S_0)$. Consequently, by Theorem 5.3, for an LTLR formula φ and a pattern $t \in N(\mathcal{R})_{AP}$:

$$\mathcal{R}each^{\preccurlyeq_E}_{\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}}(\{[t]_E\}), [t]_E \models \varphi \implies \bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}, [t]_E \models \varphi$$

Example 5.11. *For the bakery example in Section 5.2.1, both formulas $\Box ex?$ and $(\Diamond\Box enabled(wake(0)) \to \Box\Diamond wake(0)) \to \Diamond in.crit(0)$ hold in the folding abstraction $\mathcal{R}each^{\preccurlyeq_E}_{\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}}(\{$`N ; N ; [0,idle][s,idle]`$\})$ of Figure 5.9. Specifically, ex? holds in every state, and any infinite path continuously staying in the first row violates the fairness assumption. Therefore, both properties are also satisfied in any corresponding concrete system.*

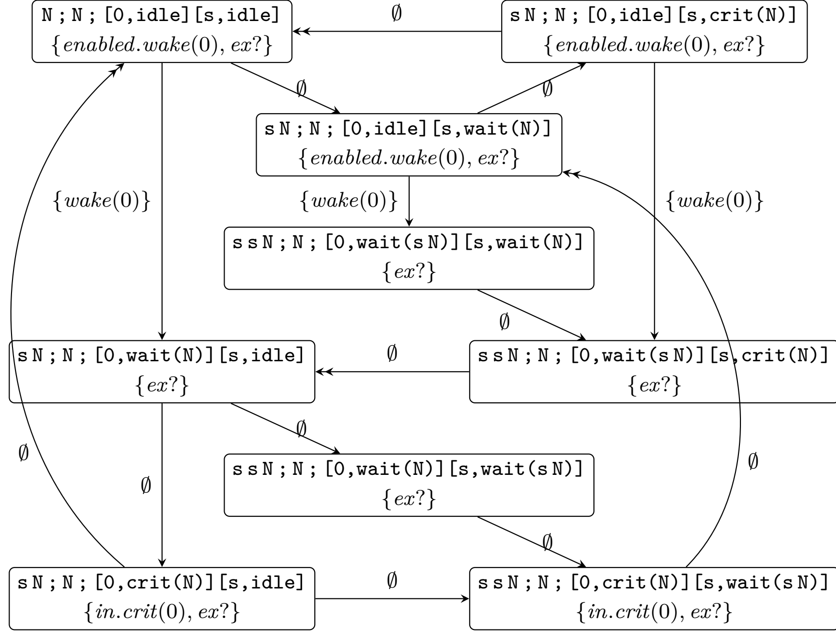


Figure 5.9: A folding abstraction for the bakery algorithm using the folding relation $\preccurlyeq_E$, where a double-headed arrow denotes a "folded" transition.

Equational Abstractions. A folding abstraction of a narrowing-based LKS $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$ may *not* be finite in general. For the bakery example, there is an infinite path within the folding abstraction of Figure 5.10 from the initial state `N ; N ; [0,idle] IS` for an unbounded number of processes, where the variable `IS` denotes a set of *idle* processes. To further reduce an infinite-state narrowing-based LKS, we can apply equational abstraction. An equational abstraction $\bar{\mathcal{N}}(\mathcal{R}/G)_{\mathcal{P}}$ simulates the original narrowing-based LKS $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$, in a similar way to the ground cases in Section 5.3.

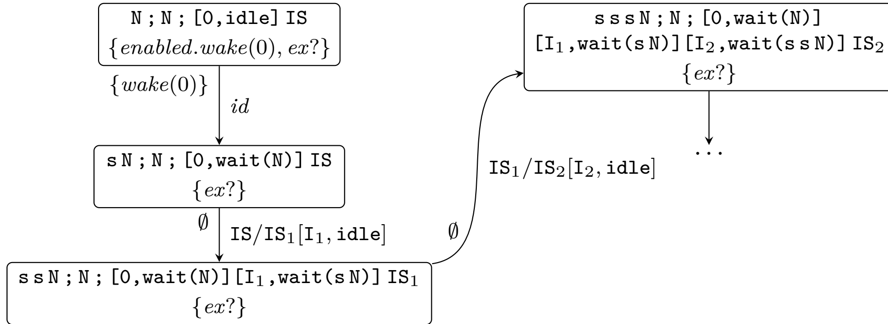


Figure 5.10: An infinite path in the folding abstraction from the initial state `N ; N ; [0,idle] IS` for the bakery algorithm with an unbounded number of processes, where `IS` stands for a set of *idle* processes.

Lemma 5.13. *Given a topmost rewrite theory $\mathcal{R} = (\Sigma, E, R)$, finite sets AP and ACT defined by $\mathcal{P} = (\Pi, D)$, and a set G of equations, there exists a total simulation from $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$ to $\bar{\mathcal{N}}(\mathcal{R}/G)_{\mathcal{P}}$, provided $true \neq_{E\cup D\cup G} false$.*

Proof. Let $H_G = \{([t]_E, [t]_{E\cup G}) \mid t \in N(\mathcal{R})_{AP}\}$. Suppose that $t =_{E\cup G} u$ and $[t]_E \xrightarrow{\Lambda}_{\bar{\mathcal{N}}(\mathcal{R})} [t']_E$. Given $AP = \{p_1, \ldots, p_n\}$ and $ACT = \{\delta_1, \ldots, \delta_m\}$, by definition, there exist substitutions σ and ζ such that $t \rightsquigarrow_{l,\sigma,\mathcal{R}} t''$ by a rule $l : q \longrightarrow r \in R$ and $t' = \zeta t''$, where $\sigma \in CSU_E(t = q)$, $t'' = \sigma r$, and for some Boolean values $b_1, \ldots, b_{n+m} \in \{true, false\}$:

$$\zeta \in CSU_{E\cup D}\Big(\textstyle\bigwedge_{1\leq i\leq n}(t'' \models p_i) = b_i \ \wedge\ \bigwedge_{1\leq j\leq m}(l(\sigma_l) \models \delta_j) = b_{n+j}\Big).$$

Because $\sigma \in CSU_E(t = q)$ and $t =_{E\cup G} u$ hold, there exists a substitution $\sigma' \in CSU_{E\cup G}(u = q)$ such that $\sigma =_{E\cup G} \sigma'$. Then, $u \rightsquigarrow_{l,\sigma',\mathcal{R}/G} u'$ holds using the same rule $l : q \longrightarrow r$, where $u' = \sigma' r =_{E\cup G} \sigma r = t''$. Notice that $(t'' \models p_i) =_{E\cup D\cup G} (u' \models p_i)$ and $(l(\sigma_l) \models \delta_j) =_{E\cup D\cup G} (l(\sigma'_l) \models \delta_j)$. Hence,

$$\exists \zeta' \in CSU_{E\cup D\cup G}\Big(\textstyle\bigwedge_{1\leq i\leq n}(u' \models p_i) = b_i \ \wedge\ \bigwedge_{1\leq j\leq m}(l(\sigma'_l) \models \delta_j) = b_{n+j}\Big)$$

with $\zeta =_{E\cup D\cup G} \zeta'$. Therefore, $[u]_{E\cup G} \xrightarrow{\Lambda}_{\bar{\mathcal{N}}(\mathcal{R}/G)} [\zeta' u']_{E\cup G}$. Also, since $(\Sigma \cup \Pi, E \cup G \cup D)$ protects $(\Sigma, E \cup G)$, $\zeta' u' =_{E\cup G} \zeta t'' = t'$. Further, since $true \neq_{E\cup D\cup G} false$, $[t']_E$ and $[\zeta' u']_{E\cup G}$ satisfy the same state propositions. Therefore, H_G is a total simulation from $\bar{\mathcal{N}}(\mathcal{R})_{\mathcal{P}}$ to $\bar{\mathcal{N}}(\mathcal{R}/G)_{\mathcal{P}}$. □

Example 5.12. *For the bakery example in Section 5.2.1, in order to verify the liveness property $(\Diamond\Box enabled(wake(0)) \to \Box\Diamond wake(0)) \to \Diamond in.crit(0)$, consider the set of state propositions $AP = \{in.crit(0), enabled(wake(0))\}$ and the set of spatial action patterns $ACT = \{wake(0), wake\}$, with the extra spatial action pattern wake that holds if the wake rule is applied.*

We can obtain the finite-state folded abstract logical LKS in Figure 5.11 from the initial state `N ; N ; IS` for an unbounded number of processes (where `IS` denotes a set of idle processes), by adding the following equations,[9] *which, intuitively, collapses extra waiting processes with non-zero identifiers that do* not *introduce new behaviors:*

```
eq [NZ,D] = [D] .                       --- remove non-zero identifiers
eq s s s N M ; M ; PS [wait(s N M)] [wait(s s N M)]
 =    s s N M ; M ; PS [wait(s N M)] .
```

[9] These equations G do *not* satisfy the finite variant property (see the conditions on [87]). However, all the *reachable* logical states from the given initial state `N ; N ; IS` have a finite set of most general G-variants, which is enough to have a finitary G-unification procedure for the *reachable* logical state space.

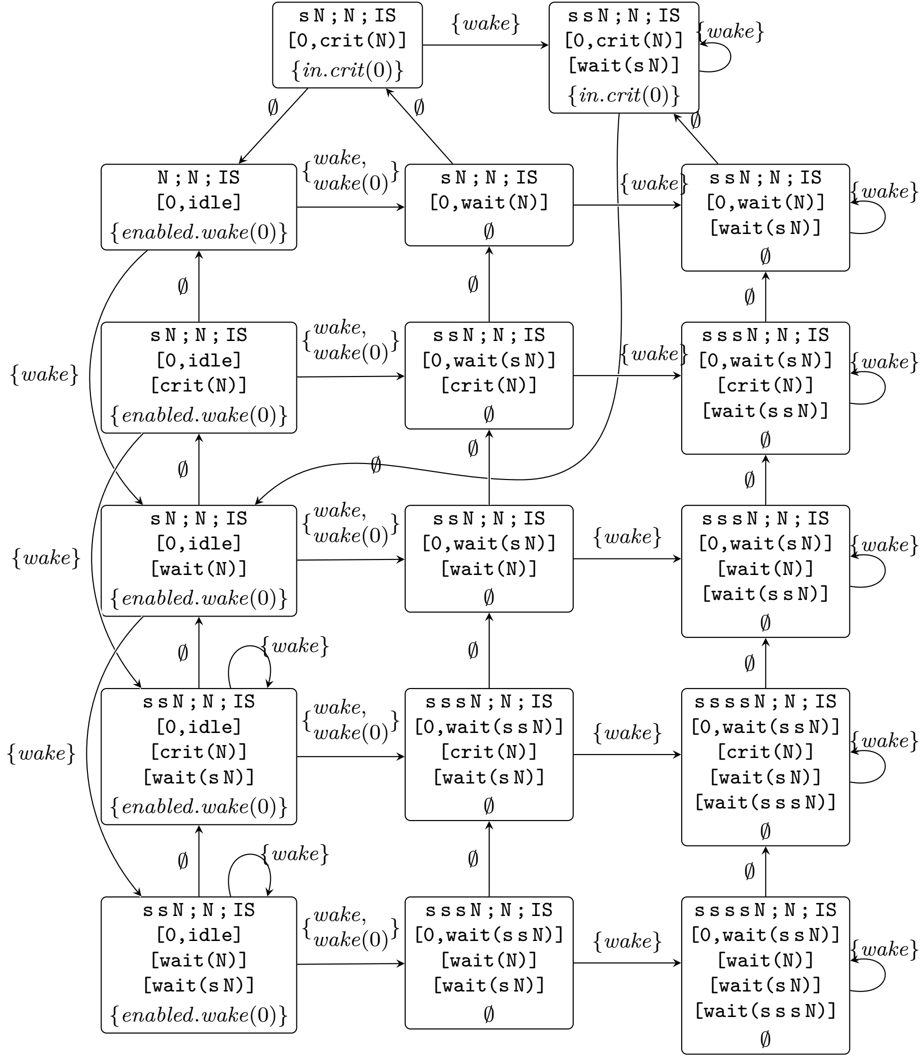


Figure 5.11: A folded equational abstraction for the bakery algorithm.

We can easily see that there is a counterexample of the property $\Diamond in.crit(0)$ under the fairness condition $\Diamond\Box enabled.wake(0) \to \Box\Diamond wake(0)$ in which the wake rule is continuously applied forever. Because there exists only a finite number of processes, the wake rule cannot be continuously applied forever. Therefore, we need an extra assumption to avoid this unrealistic behavior.

If we assume the extra fairness condition $\Box\Diamond\neg wake$, then the property $\Diamond in.crit(0)$ is now satisfied, because any infinite path staying in the first column forever violates $\Diamond\Box enabled.wake(0) \to \Box\Diamond wake(0)$, and any path staying in a self loop forever violates $\Box\Diamond\neg wake$. Consequently, under the fairness assumptions $\Diamond\Box enabled.wake(0) \to \Box\Diamond wake(0)$ and $\Box\Diamond\neg wake$, the formula $\Diamond in.crit(0)$ is satisfied for an unbounded number of processes.

Example 5.13. *Consider a variant of the bakery algorithm model of Section 5.2.1 in which process identifiers have been removed. The rewrite rules and the equations for the state proposition ex? are now given as follows:*

```
rl [wake]: N ; M ; [idle] PS    =>  s N ; M ; [wait(N)] PS .
rl [crit]: N ; M ; [wait(M)] PS =>    N ; M ; [crit(M)] PS .
rl [exit]: N ; M ; [crit(M)] PS =>    N ; s M ; [idle] PS .

eq N ; M ; WS |= ex? = true .
eq N ; M ; [crit(K)] WS |= ex? = true .
eq N ; M ; [crit(K)] [crit(L)] PS |= ex? = false .
```

Then, the following abstraction equation used in Example 5.12 actually defines a bisimilar equational abstraction, because this equation satisfies the bisimilarity conditions in Theorem 5.2:

```
eq s s s L M ; M ; PS0 [wait(s L M)] [wait(s s L M)]
 =   s s L M ; M ; PS0 [wait(s L M)] .
```

When $ACT = \emptyset$, the bisimilarity conditions can be simplified as follows: for each rewrite rule $l : q \longrightarrow r$ and each equation $u = v \in G$ or $v = u \in G$:

$$\begin{array}{rcccccc} \forall \sigma \in CSU_E(l = u). & \sigma(u) & =_E & \sigma(q) & \longrightarrow & \sigma(r) \\ & \|_G & & & & \|_{E \cup G} \\ \exists \theta : \mathcal{X} \to \mathcal{T}_\Sigma(\mathcal{X}). & \sigma(v) & =_E & \theta(q) & \longrightarrow & \theta(r) \end{array}$$

If we consider the wake rule, then $CSU_E(l = u)$ has the single E-unifier $\sigma = \{\texttt{N} \mapsto \texttt{sssML}, \texttt{PS} \mapsto \texttt{PS}_1\texttt{[wait(sML)][wait(ssML)]}, \texttt{PS}_0 \mapsto \texttt{PS}_1\texttt{[idle]}\}$, where E denotes the equational axioms. Then, $\sigma v =_E \theta l$ and $\sigma r =_{E \cup G} \theta r$ hold for the substitution $\theta = \{\texttt{N} \mapsto \texttt{ssML}, \texttt{PS} \mapsto \texttt{PS}_1\texttt{[wait(sML)]}\}$. For the other direction of the equation, $CSU_E(l = v)$ also has the single E-unifier $\sigma' = \{\texttt{PS} \mapsto \texttt{PS}_2\texttt{[wait(sML)]}, \texttt{PS}_0 \mapsto \texttt{PS}_2\texttt{[idle]}, \texttt{N} \mapsto \texttt{ssML}\}$ and for the substitution $\theta' = \{\texttt{N} \mapsto \texttt{sssML}, \texttt{PS} \mapsto \texttt{PS}_2\texttt{[wait(sML)] [wait(ssML)]}\}$, we have $\sigma' u =_E \theta' l$ and $\sigma' r =_{E \cup G} \theta' r$. The cases for the other rules are similar.

*To verify the invariant $\Box$**ex?** for an unbounded number of processes, we can then construct the* finite *abstract folded logical transition system from the initial state N ; N ; IS in Figure 5.12, where $AP = \{ex?\}$ and $ACT = \emptyset$. By Theorem 5.4 and Lemma 5.3, it is faithful for the property $\Box$**ex?**. Since $\Box$**ex?** holds in the abstract system, it is also satisfied for any corresponding concrete system with an unbounded number of processes.*

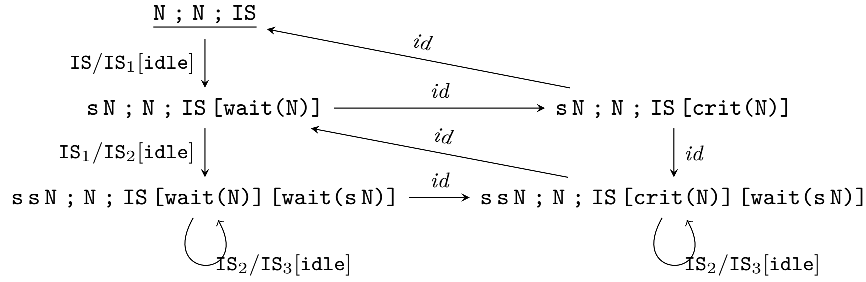


Figure 5.12: A folded bisimilar equational abstraction when $AP = \{ex?\}$.

5.5.4 The Maude LTL Logical Model Checker

This section illustrates the Maude LTL[10] logical model checker with two examples. This tool uses the existing narrowing framework in Full Maude to compute *narrowing* $\rightsquigarrow_{\sigma,R,E}$ [72]. However, the core algorithms for the folding graph construction and the LTL model checking are implemented at the C++ level within the Maude system. Our tool uses a depth-bounded model checking procedure, which applies an on-the-fly technique to reuse the previously generated states for the next step, in a way similar to the model checking procedure presented in Section 5.4.3. The tool is available at `http://maude.cs.illinois.edu/tools/lmc`.

Our tool provides the following two commands for logical model checking an LTL formula φ from an initial state t with the maximum bound $n \in \mathbb{N}$:

`(lmc [`n`]` t `|=` φ `.)` and `(lfmc [`n`]` t `|=` φ `.)`

This bound n is used to limit the depth of the k-step folding abstraction $\mathcal{R}each^{\preccurlyeq,k}_{\mathcal{N}(\mathcal{R})_{\mathcal{P}}}(\{[t]_E\})$. If a bound n is not specified in the command, then infinity is considered as the bound. Each command uses a different folding relation: the *E-renaming equivalence* $\approx_E$ for the `lmc` command, and the *E-subsumption* $\preccurlyeq_E$ for the `lfmc` command. The E-renaming equivalence $t \approx_E t'$ holds iff t and t' are equivalent up to variable renaming, implying that $t \preccurlyeq_E t'$ and $t' \preccurlyeq_E t$. Since $\approx_E$ is a symmetric folding relation, folding abstractions by $\approx_E$ are bisimilar to the original system.

If the tool returns a counterexample, there are three possibilities according to the underlying folding preorder $\preccurlyeq$. If $\preccurlyeq$ is the E-renaming equivalence $\approx_E$ or φ is an invariant, it is a real counterexample. If $\preccurlyeq$ is the E-subsumption $\preccurlyeq_E$ and φ is a general LTL formula, it may be a spurious counterexample. Of course, if an equation abstraction has been applied, then it is a real counterexample only for a bisimilar equational abstraction.

[10]Currently, our logical model checker only supports LTL formulas, and does not yet support LTLR formulas nor parameterized fairness.

Lamport's Bakery Algorithm. For the bakery algorithm specification in Section 5.2.1, the following model checking command verifies that, using the folding preorder $\preccurlyeq_E$, the mutual execution $\Box$`ex?` is satisfied from the initial state `N ; N ; [0,idle] [s,idle] [s s,idle]` for three processes:

```
Maude> (lfmc N:Nat ; N:Nat ; [0,idle] [s,idle] [s s,idle] |= [] ex? .)
logical model check in BAKERY-SAFETY-SATISFACTION :
  N:Nat ; N:Nat ; [0,idle] [s,idle] [s s,idle] |= [] ex?
result: true
```

For an *unbounded number* of processes, the following command partially verifies using $\approx_E$ that the mutual execution $\Box$`ex?` is satisfied from any initial state with the pattern `N ; N ; IS` within the bound 10:

```
Maude> (lmc [10] N ; N ; IS:ProcIdleSet |= [] ex? .)
logical model check in BAKERY-SATISFACTION :
  N:Nat ; N:Nat ; IS:ProcIdleSet |= [] ex?
result:
  no counterexample found within bound 10
```

This command does not terminate if the bound is not specified, since $\approx_E$ is not strong enough to collapse the reachable transition system to a finite system. The bound should be specified to ensure the termination even with $\preccurlyeq_E$, since, as already shown in Figure 5.10, for such a logical initial state the folding logical approximation is infinite:

```
Maude> (lfmc [50] N:Nat ; N:Nat ; IS:ProcIdleSet |= [] ex? .)
logical folding model check in BAKERY-SATISFACTION :
  N:Nat ; N:Nat ; IS:ProcIdleSet |= [] ex?
result:
  no counterexample found within bound 50
```

When the subsumption $\preccurlyeq_E$ is applied, with the equational abstraction shown in Example 5.12, the mutual exclusion $\Box$`ex?` can be verified from the initial pattern `N ; N ; IS:ProcIdleSet` as follows,[11] where, as shown in Figure 5.12, five logical states are generated in less than one second on an Intel Core i5 2.4 GHz with 4GB RAM:

```
Maude> (lfmc N:Nat ; N:Nat ; IS:ProcIdleSet |= [] ex? .)
logical folding model check in BAKERY-SATISFACTION-ABS :
  N:Nat ; N:Nat ; IS:ProcIdleSet |= [] ex?
result: true
```

[11] Note that the module `BAKERY-SATISFACTION-ABS` extends the previous module `BAKERY-SATISFACTION` with the abstraction equation.

Dijkstra's Algorithm. Consider the topmost rewrite theory specifying Dijkstra's mutual exclusion algorithm in Section 5.2.2. Recall that this system is infinite-state since the number of processes is unbounded. Indeed, given the initial state pattern `< IS:InitProcSet >` to denote an unbounded number of processes, where the variable `IS` stands for a set of processes with flag 0 and program counter `l1`, the reachable logical state space is infinite even with the folding relation $\preccurlyeq_E$.

However, we can obtain a *finite-state* folded abstract logical LKS from the initial state `< IS:InitProcSet >` by adding the following equation, which also satisfies the bisimilarity conditions in Theorem 5.2:

```
eq < {F,l2,off} {F,l2,off} PS >
 = < {F,l2,off} PS > .
```

The mutual exclusion $\Box$`ex?` can then be verified for an unbounded number of process by the following command, where 15 logical spaces are generated in less than two seconds on the same machine:

```
Maude> (lfmc < IS:InitProcSet > |= [] ex? .)
logical folding model check in DIJKSTRA-MUTEX-SATISFACTION-ABS:
  < IS:InitProcSet > |= [] ex?
result: true
```

5.6 Predicate Abstraction

For a set of *state predicates* AP for a system $\mathcal{S}$, the predicate abstraction $\mathcal{S}/AP$ has set of states 2^{AP}, and an abstract transition $s \to s'$ between states $s, s' \in 2^{AP}$ exists iff there exists a concrete transition $u \to v$ in the system $\mathcal{S}$ such that u (resp. v) satisfies exactly the predicates in s (resp. in s'). Since it may not always be possible to prove the existence of such a concrete transition, an *over-approximation* $\alpha(\mathcal{S}/AP)$, which adds extra transitions when in doubt, may instead be used.

This section shows how a predicate abstraction of a *topmost conditional* rewrite theory $\mathcal{R} = (\Sigma, E, R)$ can be constructed for *LTLR properties* by solving E-equality constraints using E-unification. Because E-unification problems are generally only semi-decidable, this chapter also presents a sound, terminating, but incomplete, procedure to check unsatisfiability of E-equality constraints. Finally, based on these procedures, this chapter presents a predicate abstraction algorithm to automatically construct a predicate abstraction of a rewrite theory, which may generate an over-approximation if such procedures fail to give an answer.

5.6.1 $\mathcal{P}$-Abstractions of Rewrite Theories

Consider a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, and a set of state propositions $AP = \{p_1, \ldots, p_n\}$ and a set of spatial action patterns $ACT = \{\delta_1, \ldots, \delta_m\}$, defined by an equational theory $\mathcal{P} = (\Pi, D)$. In predicate abstraction, abstract states are subsets of $AP = \{p_1, \ldots, p_n\}$, and an abstract transition $s \xrightarrow{\Lambda} s' \in 2^{AP} \times 2^{ACT} \times 2^{AP}$ is defined if there exists a *concrete* one-step rewrite $\lambda : t \longrightarrow_{\mathcal{R}} t'$ in $\mathcal{R}$ such that:

$$\begin{aligned} s &= \{p \in AP \mid (t \models p) =_{E \cup D} \mathit{true}\} \\ s' &= \{p \in AP \mid (t' \models p) =_{E \cup D} \mathit{true}\} \\ \Lambda &= \{\delta \in ACT \mid (\lambda \models \delta) =_{E \cup D} \mathit{true}\} \end{aligned} \tag{5.3}$$

Our approach is motivated by the following observation. For a topmost rewrite theory $\mathcal{R} = (\Sigma, E, R)$, a *concrete* one-step rewrite $l(\sigma) : t \longrightarrow_{\mathcal{R}} t'$ exists iff for a rule $(l : q \longrightarrow r \textbf{ if } C) \in R$ and a *ground* substitution σ:

$$t =_E \sigma q \quad \wedge \quad t' =_E \sigma r \quad \wedge \quad (\forall u = v \in C)\ \sigma u =_E \sigma v \tag{5.4}$$

For $\{x_1, \ldots, x_m\} = \mathit{vars}(q) \cup \mathit{vars}(r) \cup \mathit{vars}(C)$, a one-step proof term $l(\sigma)$ is represented as the term $\texttt{\{}'l : 'x_1 \backslash \sigma x_1; \ldots; 'x_m \backslash \sigma x_m\texttt{\}}$, that is,

$$l(\sigma) \ = \ \sigma(\texttt{\{}'l : 'x_1 \backslash x_1; \ldots; 'x_m \backslash x_m\texttt{\}}). \tag{5.5}$$

The abstraction $s \xrightarrow{\Lambda} s'$ of $t \longrightarrow_{\mathcal{R}} t'$ is given by Condition (5.3). Since $t =_E \sigma q$ and $t' =_E \sigma r$ by Condition (5.4), we can replace t and t' in (5.3) by σl and σr, respectively. Therefore, we obtain:

- $s = \{p \in AP \mid (\sigma q \models p) =_{E \cup D} \mathit{true}\}$,
- $s' = \{p \in AP \mid (\sigma r \models p) =_{E \cup D} \mathit{true}\}$,
- $(\forall u = v \in C)\ \sigma u =_E \sigma v$, and
- $\Lambda = \{\delta \in ACT \mid (\sigma(\texttt{\{}'l : 'x_1 \backslash x_1; \ldots; 'x_m \backslash x_m\texttt{\}}) \models \delta) =_{E \cup D} \mathit{true}\}$.

That is, $s \xrightarrow{\Lambda} s'$ holds if there exist a rewrite rule $(l : q \longrightarrow r \textbf{ if } C) \in R$ and a ground substitution σ that satisfy these E-equality constraints. Let $p \in _ : 2^{AP} \rightarrow \{\mathit{true}, \mathit{false}\}$ and $\delta \in _ : 2^{ACT} \rightarrow \{\mathit{true}, \mathit{false}\}$ be the truth functions defined as follows:

$$(p \in s) = \begin{cases} \mathit{true} & \text{if } p \in s \\ \mathit{false} & \text{if } p \notin s, \end{cases} \qquad (\delta \in \Lambda) = \begin{cases} \mathit{true} & \text{if } \delta \in \Lambda \\ \mathit{false} & \text{if } \delta \notin \Lambda. \end{cases}$$

Definition 5.12. *For a topmost rewrite theory* $\mathcal{R} = (\Sigma, E, R)$ *and finite sets of state propositions* AP *and spatial action patterns* ACT *defined by an associated equational theory* $\mathcal{P} = (\Pi, D)$*, the* $\mathcal{P}$-abstract labeled Kripke structure *is the LKS* $\bar{\mathcal{K}}(\mathcal{R}/\mathcal{P}) = (2^{AP}, AP, id_{2^{AP}}, ACT, \longrightarrow_{\mathcal{R}/\mathcal{P}})$*, where:*

- $id_{2^{AP}} : 2^{AP} \to 2^{AP}$ *is the identity labeling function, and*
- $s \xrightarrow{\Lambda}_{\mathcal{R}/\mathcal{P}} s'$ *iff there exists a rewrite rule* $(l : q \longrightarrow r \textbf{ if } C) \in R$ *such that the following constraints are* $E \cup D$*-satisfiable:*

$$\begin{aligned} &\bigwedge_{p\in AP}(q \models p) = (p \in s) \;\;\wedge\; \bigwedge_{p\in AP}(r \models p) = (p \in s') \;\;\wedge\; \bigwedge_{u=v\in C} u = v \;\;\wedge \\ &\bigwedge_{\delta\in ACT}(\{'l : 'x_1\backslash x_1; \ldots ; 'x_m\backslash x_m\} \models \delta) = (\delta \in \Lambda) \end{aligned} \qquad (\dagger)$$

If there exists a finitary $E \cup D$-unification algorithm (e.g., $E \cup D$ has the finite variant property), the satisfiability of (†) can be decided by checking for the emptiness of the *finite* complete set of the corresponding E-unifiers. For a set of equations $E \cup B$ with B a set of structural axioms, if $E \cup B$ has the *finite variant property*, there is a finitary $E \cup B$-unification algorithm to find *finite* $CSU_{E\cup B}(t = t')$ [64, 87]. As explained in [64], $E \cup B$ has the finite variant property iff for every term t there exists a bound n such that the canonical form of θt for a normalized substitution θ is reachable from t by applying E modulo B less than n times.

Checking satisfiability of the constraints (†) by E-unification is in general undecidable [12]. Therefore, $\bar{\mathcal{K}}(\mathcal{R}/\mathcal{P})$ may *not* have an effective procedure to *precisely* decide its transitions. In practice, there are three cases:

1. For *some* rule in R, we can prove the *satisfiability* of (†) for (s, s', Λ), in which case we know that $s \xrightarrow{\Lambda}_{\mathcal{R}/\mathcal{P}} s'$ holds.
2. For *every* rule in R, we can prove the *unsatisfiability* of (†) for (s, s', Λ), in which case we know that $s \xrightarrow{\Lambda}_{\mathcal{R}/\mathcal{P}} s'$ does *not* hold.
3. Otherwise, we cannot decide whether $s \xrightarrow{\Lambda}_{\mathcal{R}/\mathcal{P}} s'$ holds or not. In this case we can *add* $s \xrightarrow{\Lambda}_{\alpha(\mathcal{R}/\mathcal{P})} s'$ to an LKS $\alpha(\bar{\mathcal{K}}(\mathcal{R}/\mathcal{P}))$ *approximating* (and therefore simulating) $\mathcal{R}/AP$ with possibly more transitions.

By definition, an LKS $\alpha(\bar{\mathcal{K}}) = (S, AP, \mathcal{L}, ACT, \longrightarrow_{\alpha(\bar{\mathcal{K}})})$ is an *approximation* of $\bar{\mathcal{K}} = (S, AP, \mathcal{L}, ACT, \longrightarrow_{\bar{\mathcal{K}}})$ iff $\longrightarrow_{\bar{\mathcal{K}}} \subseteq \longrightarrow_{\alpha(\bar{\mathcal{K}})}$. Notice that the identity function $id_S : S \to S$ is a simulation from $\bar{\mathcal{K}}$ to $\alpha(\bar{\mathcal{K}})$. In Section 5.6.2 we propose some procedures for checking unsatisfiability of (†).

Theorem 5.8. *Given a topmost rewrite theory $\mathcal{R} = (\Sigma, E, R)$ and finite sets of AP and ACT defined by $\mathcal{P} = (\Pi, D)$, for an LTLR formula φ and an initial state $t \in \mathcal{T}_{\Sigma,\mathsf{State}}$, if $s = \mathcal{L}_{AP}([t]_E)$, then*

$$\alpha(\bar{\mathcal{K}}(\mathcal{R}/\mathcal{P})), s \models \varphi \implies \bar{\mathcal{K}}(\mathcal{R}, \mathsf{State})_{\mathcal{P}}, [t]_E \models \varphi.$$

Proof. It suffices to show that the state-labeling function $\mathcal{L}_{\mathcal{P}}$ of the LKS $\bar{\mathcal{K}}(\mathcal{R}, \mathsf{State})_{\mathcal{P}}$, where $\mathcal{L}_{\mathcal{P}}([u]_E) = \{p \in AP \mid (u \models p) =_{E \cup D} true\}$, is a total simulation from $\bar{\mathcal{K}}(\mathcal{R}, \mathsf{State})_{\mathcal{P}}$ to $\bar{\mathcal{K}}(\mathcal{R}/\mathcal{P})$. Then, $(id_{2^{AP}} \circ \mathcal{L}_{\mathcal{P}}) = \mathcal{L}_{\mathcal{P}}$ becomes a simulation from $\bar{\mathcal{K}}(\mathcal{R}, \mathsf{State})_{\mathcal{P}}$ to $\alpha(\bar{\mathcal{K}}(\mathcal{R}/\mathcal{P}))$, since $id_{2^{AP}}$ is a simulation from $\bar{\mathcal{K}}(\mathcal{R}/\mathcal{P})$ to $\alpha(\bar{\mathcal{K}}(\mathcal{R}/\mathcal{P}))$.

Suppose that a transition $t \xrightarrow{\Lambda}_{\mathcal{R}} t'$ exists in $\bar{\mathcal{K}}(\mathcal{R}, \mathsf{State})_{\mathcal{P}}$. By definition, for a rewrite rule $(l \longrightarrow r \textbf{ if } C) \in R$, there is a ground substitution σ such that $t =_E \sigma l$, $t' =_E \sigma r$, $\Lambda = \{\delta \in ACT \mid (l(\sigma) \models \delta) =_{E \cup D} true\}$ and for each condition $u = v \in C$, $\sigma u =_E \sigma v$, where $l(\sigma)$ is represented as a term $\sigma(\{'l : 'x_1 \backslash x_1; \ldots; 'x_m \backslash x_m\})$. Let $s = \mathcal{L}_{\mathcal{P}}([t]_E)$ and $s' = \mathcal{L}_{\mathcal{P}}([t']_E)$. Since $t =_E \sigma l$ and $t' =_E \sigma r$, we have $s = \{p \in AP \mid (\sigma l \models p) =_{E \cup D} true\}$ and $s' = \{p \in AP \mid (\sigma r \models p) =_{E \cup D} true\}$. Hence, $\bigwedge_{p \in AP} (\sigma l \models p) =_{E \cup D} (p \in s)$ and $\bigwedge_{p \in AP} (\sigma r \models p) =_{E \cup D} (p \in s')$ hold. That is, σ is a solution of (†), so that $s \xrightarrow{\Lambda}_{\mathcal{R}/\mathcal{P}} s'$. Therefore, $\mathcal{L}_{\mathcal{P}}$ is a desired total simulation. $\square$

However, since in general $\mathcal{L}_{\mathcal{P}}$ is *not* a bisimulation from $\bar{\mathcal{K}}(\mathcal{R}, \mathsf{State})_{\mathcal{P}}$ to $\alpha(\bar{\mathcal{K}}(\mathcal{R}/\mathcal{P}))$, there may exist *spurious* counterexamples in $\alpha(\bar{\mathcal{K}}(\mathcal{R}/\mathcal{P}))$. As usual for predicate abstraction methods, we can *refine* the $\mathcal{P}$-abstraction by adding extra state propositions to further specialize $\alpha(\bar{\mathcal{K}}(\mathcal{R}/\mathcal{P}))$.

Example 5.14. *We illustrate our ideas with the simplified version of the readers-writers problem in Section 5.2.3. This system is infinite-state, since the number of readers R is unbounded. We are interested in verifying the mutual exclusion $\square\neg(reading \wedge writing)$. The equations in Section 5.2.3 defining the state propositions satisfies the finite variant property since their right-hand sides are constants. For the sets $AP = \{reading, writing\}$ and $ACT = \emptyset$, we obtain the* finite *$\mathcal{P}$-abstract LKS $\bar{\mathcal{K}}(\mathcal{R}/\mathcal{P})$ in Figure 5.13, where its abstract transitions are decided by using E-unification.*

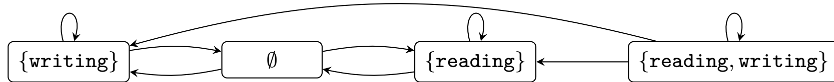


Figure 5.13: The $\mathcal{P}$-abstract LKS with the initial state $\emptyset$ for the simplified model of the readers-writers problem.

Trans \ Rule	$\langle 0,0\rangle \to \langle 0,s(0)\rangle$	$\langle r,0\rangle \to \langle s(r),0\rangle$	$\langle r,s(w)\rangle \to \langle r,w\rangle$	$\langle s(r),w\rangle \to \langle r,w\rangle$
$\emptyset \to \{\mathtt{w}\}$	$\langle 0,0\rangle \to \langle 0,s(0)\rangle$			
$\emptyset \to \{\mathtt{r}\}$		$\langle 0,0\rangle \to \langle s(0),0\rangle$		
$\{\mathtt{w}\} \to \emptyset$			$\langle 0,s(0)\rangle \to \langle 0,0\rangle$	
$\{\mathtt{w}\} \to \{\mathtt{w}\}$			$\langle 0,s(s(w))\rangle \to \langle 0,s(w)\rangle$	
$\{\mathtt{r}\} \to \emptyset$				$\langle s(0),0\rangle \to \langle 0,0\rangle$
$\{\mathtt{r}\} \to \{\mathtt{r}\}$		$\langle s(r),0\rangle \to \langle s(s(r)),0\rangle$		$\langle s(s(r)),0\rangle \to \langle s(r),0\rangle$
$\{\mathtt{r},\mathtt{w}\} \to \{\mathtt{w}\}$				$\langle s(0),s(w)\rangle \to \langle 0,s(w)\rangle$
$\{\mathtt{r},\mathtt{w}\} \to \{\mathtt{r}\}$			$\langle s(r),s(0)\rangle \to \langle s(r),0\rangle$	
$\{\mathtt{r},\mathtt{w}\} \to \{\mathtt{r},\mathtt{w}\}$			$\langle s(r),s(s(w))\rangle \to \langle s(r),s(w)\rangle$	$\langle s(s(r)),s(w)\rangle \to \langle s(r),s(w)\rangle$

Figure 5.14: The rule instances for each abstract transition $s \to s' \in 2^{AP^2}$.

Figure 5.14 shows the rule instances $\zeta l \to \zeta r$ for each rule $l \longrightarrow r \in R$, transition $s \longrightarrow s' \in 2^{AP^2}$, and E-unifier

$$\zeta \in CSU_E(\bigwedge_{p \in AP} (l \models p) = (p \in s) \wedge \bigwedge_{p \in AP} (r \models p) = (p \in s'))$$

that represent the transitions of $\bar{\mathcal{K}}(\mathcal{R}/\mathcal{P})$ in Figure 5.13. The property $\Box\neg(reading \wedge writing)$ holds from the initial state $\emptyset$ in $\bar{\mathcal{K}}(\mathcal{R}/\mathcal{P})$, since the state $\{reading, writing\}$ is not *reachable. Therefore, $\Box\neg(reading \wedge writing)$ also holds from $\langle 0,0\rangle$ in $\bar{\mathcal{K}}(\mathcal{R}, \mathsf{State})_{\mathcal{P}}$, thanks to Theorem 5.8.*

If we consider another LTL formula $\Box\Diamond\neg writing$ (i.e., infinitely often not writing), there exists the spurious counterexample

$$\emptyset \to \{writing\} \to \{writing\} \to \cdots .$$

We can refine *the $\mathcal{P}$-abstraction by adding the state proposition 1w to further specialize the abstract state space, meaning that there exists only one writer, defined by the following three equations:*

```
eq < R, s(0) > |= 1w = true .      eq < R, 0       > |= 1w = false .
                                   eq < R, s(s(W)) > |= 1w = false .
```

We obtain then the refined AP-abstract LKS in Figure 5.15 in which the formula $\Box\Diamond\neg writing$ holds from the initial state $\emptyset$. Again, by Theorem 5.8, $\Box\Diamond\neg writing$ also holds from $\langle 0,0\rangle$ in the concrete LKS of Figure 5.3.

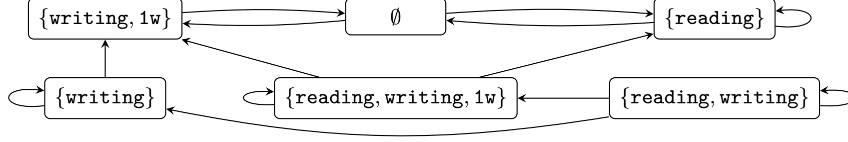


Figure 5.15: Refined $\mathcal{P}$-abstract LKS for the readers-writers problem.

5.6.2 Effective Procedures for Equality Constraints

This section presents an effective procedure to check the *unsatisfiability* of E-equality constraints for predicate abstractions. Since the problem is in general only semi-decidable [12], we are interested in a *sound*, but possibly *incomplete*, terminating procedures that can be easily automated, so that an over-approximation $\alpha(\bar{\mathcal{K}}(\mathcal{R}/\mathcal{P}))$ can be built when the procedure fails to give an answer. A semi-decidable E-unification algorithm does not give such a procedure because it may *not* terminate when no solution exists.

Our method relies on the fact that a state proposition p (or a spatial action pattern δ) is defined using only a subset of equations $E_p \subseteq E$, so that solving constraints for p may only involve E_p, *not* all of E. Therefore, if a finitary E_p-unification algorithm is available, then we can discharge the constraints for p using E_p-unification. After resolving all such *solvable* constrains, we apply a sound procedure based on E-reduction to test if the remaining constraints are inconsistent.

Decomposition of Constraints. In practice, the equational semantics of a state proposition p or a spatial action pattern δ can be restricted to a certain subset of the equations E. As assumed, E decomposes as $E = E_o \cup B$ with B a set of equational axioms and E_o convergent modulo B (that is, sort-decreasing, terminating, confluent, and coherent modulo B).

Definition 5.13. *Let $\Omega \subseteq \Sigma$ be a set of free constructors for E_o modulo B.*[12] *Given a set of patterns $U = \{u_1, \dots, u_n\} \subseteq \mathcal{T}_\Sigma(\mathcal{X})$, we define:*

$$\llbracket U \rrbracket = \{t \in \mathcal{T}_\Sigma(\mathcal{X}) \mid (\exists \sigma : \mathcal{X} \to \mathcal{T}_\Omega(\mathcal{X}), \exists u \in U)\ t =_B \sigma u\},$$
$$E_U = \{u = v \in E_o \mid u, v \in \llbracket U \rrbracket\}.$$

We call E_o syntactically independent *iff E_U is convergent modulo B and: (i) any proper subterm v of $u \in U$ is strongly E_o, B-irreducible (i.e., γv is a normal form for any normalized substitution γ; see Definition 4.9), and (ii) $CSU_B(t = u) = \emptyset$ for each equation $(t = t') \in E_o - E_U$ and $u \in U$.*

[12] That is, $\mathcal{T}_{\Sigma/E\cup B}|_\Omega \simeq \mathcal{T}_{\Omega/B}$.

Example 5.15. *For the simplified readers-writers problem in Section 5.2.3, every state proposition p has its syntactically independent patterns U_p; e.g., for the state proposition reading,* $U_{re} = \{(\texttt{S:State} \models \mathit{reading}), \mathit{true}, \mathit{false}\}$ *and* $E_{U_{re}} = \{\langle s(r), w\rangle \models \mathit{reading} = \mathit{true},\ \langle 0, w\rangle \models \mathit{reading} = \mathit{false}\}$.

We are interested in finding a subset of equations $G \subseteq E$ with a finitary unification algorithm that can make E-solvability of a constraint $u = v$ in (†) decidable by using G-unification. An E-equality constraint $u = v$ is *G-solvable* for $G \subseteq E$ iff $\sigma u =_E \sigma v$ implies $\sigma u =_G \sigma v$ for any substitution σ. Since $G \subseteq E$, if $u = v$ is G-solvable, then $\sigma u =_E \sigma v \iff \sigma u =_G \sigma v$.

Proposition 5.1. *If E_o is syntactically independent with respect to U, then for any $u, v \in [\![U]\!]$, an E-equality constraint $u = v$ is $E_U \cup B$-solvable.*

Proof. Since we assume that $\mathcal{T}_\Sigma(\mathcal{X})_{\mathsf{s}} \neq \emptyset$ for each sort s, a constraint $u = v$ is $E_o \cup B$-solvable iff for some normalized ground substitution $\sigma : \mathcal{X} \to \mathcal{T}_\Omega$, $\mathit{can}_{E_o/B}(\sigma u) =_B \mathit{can}_{E_o/B}(\sigma v)$, where $\mathit{can}_{E_o/B}(t)$ denotes an E_o/B-canonical form of the term t. Because $\sigma u, \sigma v \in [\![U]\!]$, by Ω-terms being free modulo B, convergence of E_U modulo B, and the conditions (i)–(ii) in Definition 5.13, $\mathit{can}_{E_o/B}(\sigma u) =_B \mathit{can}_{E_U/B}(\sigma u)$ and $\mathit{can}_{E_o/B}(\sigma v) =_B \mathit{can}_{E_U/B}(\sigma v)$. The lifting lemma for narrowing modulo B (e.g., see [142]) then forces $u = v$ to be $E_U \cup B$-solvable by narrowing with E_U modulo B. □

Even when $E_U \cup B$ does *not* have the finite variant property, there may still exist a subset $\tilde{E}_U \subseteq E_U$ where $\tilde{E}_U \cup B$ has the finite variant property and a constraint $u = v$ is $\tilde{E}_U \cup B$-solvable (see Section 5.6.3 for an example).

For a set of E-equality constraints D, if $G \subseteq E$ has a finitary G-unification algorithm and a constraint $u = v \in D$ is G-solvable, then $\mathit{CSU}_G(u = v)$ is finite, and $u = v$ is E-satisfiable iff $\mathit{CSU}_G(u = v) \neq \emptyset$. Therefore, we can decompose the problem D into finding $\zeta \in \mathit{CSU}_G(u = v)$ and solving one of the remaining constraints $\{\zeta u = \zeta v \mid u = v \in D - \{u = v\}\}$.

Lemma 5.14. *Given a set of E-equality constraints D, if $u = v \in D$ is G-solvable, then D is E-satisfiable iff there exists $\zeta \in \mathit{CSU}_G(u = v)$ such that $\{\zeta u = \zeta v \mid u = v \in D - \{u = v\}\}$ is E-satisfiable*

Proof. Suppose that there is a substitution ρ such that $\bigwedge_{u=v\in D} \rho u =_E \rho v$. Since $u = v$ is G-solvable, $\rho u =_G \rho v$ also holds. Therefore, there exists a substitution $\zeta \in \mathit{CSU}_G(u = v)$ such that $(\exists \eta)\ \rho =_G \eta \circ \zeta$. Since $G \subseteq E$, $\rho =_E \eta \circ \zeta$. Therefore, $\bigwedge_{u=v\in D-\{u=v\}} \eta(\zeta u) =_E \eta(\zeta v)$ holds. □

We can repeatedly apply this procedure to solve each G-solvable constraint in (†) to determine $s \xrightarrow{\Lambda}_{\mathcal{R}/\mathcal{P}} s'$, provided that G has a finitary unification algorithm. If there exists *no* solution, then $s \xrightarrow{\Lambda}_{\mathcal{R}/\mathcal{P}} s'$ does *not* hold.

Unfeasibility of Constraints. Applying Lemma 5.14, a set of E-equality constraints D can be transformed into an equivalent set F that contains *no* G-solvable constraints. We now present a sound but incomplete procedure to test for *E-unsatisfiability* of F. We have assumed that E decomposes as a disjoint union $E = E_o \cup B$, where a finitary B-unification algorithm exists and E_o is convergent, and $\mathcal{T}_{\Sigma,s} \neq \emptyset$ for each sort $\mathsf{s} \in \Sigma$. Therefore, we can use the *canonical term algebra* $Can_{\Sigma,E_o/B}$ (see Section 2.2.4), whose elements are B-equivalence classes of Σ-terms in E_o/B-canonical form, and which is an initial $E_o \cup B$-algebra. That is, F is E-satisfiable iff there exists a normalized ground substitution θ such that $(Can_{\Sigma,E_o/B}, q_B \circ \theta) \models F$, where $q_B : \mathcal{T}_\Sigma \to \mathcal{T}_{\Sigma/B}$ is the quotient map $t \mapsto [t]_B$ for B.

Given the set X of the variables in F, let $\bar{X} = \{\bar{x} \mid x \in X\}$ be the set in which each variable $x \in X$ of sort s is turned into the constant $\bar{x}$ of the same sort s, where $\bar{X} \cap \Sigma = \emptyset$, and let $\bar{F} = \{\bar{u} = \bar{v} \mid u = v \in F\}$ be the set of the *ground* constraints obtained from F by replacing each $x \in X$ by $\bar{x} \in \bar{X}$. Recall that a $\Sigma \cup \bar{X}$-algebra is exactly a pair (A, a) with a valuation $a : \bar{X} \to A$. Therefore, if $(Can_{\Sigma,E_o/B}, q_B \circ \theta) \models F$, then the valuation $q_B \circ \hat{\theta}$, with $\hat{\theta}(\bar{x}) = \theta(x)$ for $x \in X$, gives us a $\Sigma \cup \bar{X}$-algebra $(Can_{\Sigma,E_o/B}, q_B \circ \hat{\theta})$ satisfying *both* E and $\bar{F}$. Soundness of equational logic then ensures that whenever $(\Sigma \cup \bar{X}, E \cup \bar{F}) \vdash (\forall\emptyset)\ \bar{u} = \bar{v}$, where $vars(u) \cup vars(v) \subseteq X$, we must have $(Can_{\Sigma,E_o/B}, q_B \circ \hat{\theta}) \models (\forall\emptyset)\ \bar{u} = \bar{v}$.

Our sound procedure for testing unsatisfiability of F is based on the idea of obtaining a proof of the form $(\Sigma \cup \bar{X}, E \cup \bar{F}) \vdash (\forall\emptyset)\ \bar{u} = \bar{v}$, where u and v are *strongly E_o, B-irreducible* Σ-terms and $CSU_B(u = v) = \emptyset$. This gives a contradiction to the assumption that F is satisfiable; if F is satisfied by a normalized substitution θ, then $(Can_{\Sigma,E_o/B}, q_B \circ \hat{\theta}) \models \bar{u} = \bar{v}$ holds, that is, $[\theta u]_B = [\theta v]_B$, since u and v are strongly E_o, B-irreducible, but $CSU_B(u = v) = \emptyset$ implies $[\theta u]_B \neq [\theta v]_B$.

A practical way of obtaining such a proof $(\Sigma \cup \bar{X}, E \cup \bar{F}) \vdash (\forall\emptyset)\ \bar{u} = \bar{v}$ is by rewriting modulo B. We can use the set of rewrite rules $\vec{E_o} \cup \vec{\bar{F}}$ with $\vec{E_o}$ the oriented equations and for a B-compatible order $\succ$ [74] on ground terms, and $\vec{\bar{F}} = \{can_{E_o/B}(\bar{w}) \to can_{E_o/B}(\bar{w}') \mid\ w = w' \in F \text{ or } w' = w \in F, can_{E_o/B}(\bar{w}) \succ can_{E_o/B}(\bar{w}')\}$. If we obtain $\bar{u} \longleftrightarrow^*_{\vec{E_o} \cup \vec{\bar{F}}, B} \bar{v}$ by rewriting modulo B, then we have *a fortiori* derived $(\Sigma \cup \bar{X}, E \cup \bar{F}) \vdash (\forall\emptyset)\ \bar{u} = \bar{v}$ by equational reasoning. In summary:

Theorem 5.9. *For a set of E-equality constraints F, if there exist strongly E_o, B-irreducible Σ-terms u and v such that $vars(u) \cup vars(v) \subseteq vars(F)$, $CSU_B(u = v) = \emptyset$, and $\bar{u} \longleftrightarrow^*_{\vec{E_o} \cup \vec{\bar{F}}, B} \bar{v}$, then F is E-unsatisfiable.*

Example 5.16. *We consider the model of the readers-writers problem with explicit shared variables and processes in Section 5.2.3. In order to verify* $\Box\neg(\mathit{reading} \wedge \mathit{writing})$ *for an unbounded number of processes, we need to have two additional state propositions free and good:*

```
eq < N, B | PS > |= free = B .
eq < N, B | PS > |= good = good(N, PS) .
```

where $good(n, PS)$ *returns true iff* n *is equal to the number of readers in* PS*:*

```
eq good(s(N), read ; PS) = good(N, PS) .
eq good(N, write ; PS) = good(N, PS) .
eq good(N, idle ; PS) = good(N, PS) . eq good(0, WS) = true .
eq good(s(N), WS) = false .          eq good(0, read ; PS) = false .
```

Notice that every state proposition has syntactically independent equations. Hence, we can easily see that every constraint for c, reading, writing, and free is solvable by a set of equations satisfying the finite variant property. However, the equations defining good do not *have the finite variant property.*

We can obtain a finite $\bar{\mathcal{K}}(\mathcal{R}/\mathcal{P})$ *in which only two states* $\{\mathit{reading}, \mathit{good}\}$ *and* $\{\mathit{writing}, \mathit{good}\}$ *are reachable from the initial state* $\{\mathit{free}, \mathit{good}\}$*. For example, from* $\{\mathit{free}, \mathit{good}\}$*, after resolving each* $E_p \cup B$*-solvable constraint, we have the following sets of the remaining constraints:*

$$\langle 0, \mathit{true} \mid \mathit{idle} \,; \mathit{IS}\rangle \models \mathit{good} = \mathit{true} \;\wedge\; \langle s(0), \mathit{false} \mid \mathit{read} \,; \mathit{IS}\rangle \models \mathit{good} = b_1,$$
$$\langle s(n), \mathit{true} \mid \mathit{idle} \,; \mathit{IS}\rangle \models \mathit{good} = \mathit{true} \;\wedge\; \langle s(s(n)), \mathit{true} \mid \mathit{read} \,; \mathit{IS}\rangle \models \mathit{good} = b_2,$$
$$\langle n, \mathit{true} \mid \mathit{idle} \,; \mathit{IS}\rangle \models \mathit{good} = \mathit{true} \;\wedge\; \langle n, \mathit{false} \mid \mathit{write} \,; \mathit{IS}\rangle \models \mathit{good} = b_3,$$

where IS is a variable of sort ProcIdleSet *and* $b_1, b_2, b_3 \in \{\mathit{true}, \mathit{false}\}$*. By normalizing each constraint after replacing the variables into the constants, we have the following oriented constraint sets*

$$\{\mathit{true} \to \mathit{true},\; \mathit{true} \to b_1\}$$
$$\{\mathit{true} \to \mathit{false},\; b_2 \to \mathit{false}\}$$
$$\{\mathit{good}(\overline{n}, \overline{\mathit{IS}}) \to \mathit{true},\; \mathit{good}(\overline{n}, \overline{\mathit{IS}}) \to b_3\}$$

Notice that the cases of $b_1 = \mathit{false}$*,* $b_2 \in \{\mathit{true}, \mathit{false}\}$*, and* $b_3 = \mathit{false}$ *are unsatisfiable. That is,* $\{\mathit{free}, \mathit{good}\}$ *has only two next states* $\{\mathit{reading}, \mathit{good}\}$ *and* $\{\mathit{writing}, \mathit{good}\}$*. Similarly, the state* $\{\mathit{reading}, \mathit{good}\}$ *has the next states* $\{\mathit{free}, \mathit{good}\}$ *and* $\{\mathit{reading}, \mathit{good}\}$*, and* $\{\mathit{writing}, \mathit{good}\}$ *has the next states* $\{\mathit{free}, \mathit{good}\}$ *and* $\{\mathit{writing}, \mathit{good}\}$*. Therefore,* $\Box\neg(\mathit{reading} \wedge \mathit{writing})$ *holds in* $\bar{\mathcal{K}}(\mathcal{R}/\mathcal{P})$ *from* $\{\mathit{free}, \mathit{good}\}$*. Thanks to Theorem 5.8, the formula also holds in* $\bar{\mathcal{K}}(\mathcal{R}, \mathsf{State})_{\mathcal{P}}$ *for an unbounded number of processes.*

```
findNextStates(s ∈ 2^AP):
  NextStates := ∅;
  for each s′ ∈ 2^AP do
    for each (l ⟶ r if C) ∈ R do
      CTR := getConstraints(s, s′, l, r, C);    // the set of constraints (†)
      (D,F,G) := findSolvable(CTR);             // G-solvable equations
      if isSatisfiable(D,F,G) then add s′ to NextStates;
  return NextStates;

isSatisfiable(D, F, G):
  if D ≠ ∅ then
    choose u = v from D;
    for each ζ ∈ CSU_u(G) = v) do
      if isSatisfiable(ζ(D − {u = v}), ζF, G) then return true;
    return false;
  else
    return (if F = ∅ then true else ¬ testUnsatisfiable(F));
```

Figure 5.16: Predicate Abstraction Algorithm for $\mathcal{R} = (\Sigma, E, R)$.

Predicate Abstraction Algorithm. Figure 5.16 shows an algorithm to generate a predicate abstraction $\alpha(\bar{\mathcal{K}}(\mathcal{R}/\mathcal{P}))$ of a topmost rewrite theory $\mathcal{R} = (\Sigma, E, R)$. There exists a transition $s \xrightarrow{\Lambda}_{\mathcal{R}/\mathcal{P}} s'$ in $\alpha(\bar{\mathcal{K}}(\mathcal{R}/\mathcal{P}))$ iff $s' \in \mathit{findNextStates}(s)$, where the function *findNextStates*(s) returns a set of next *abstract* states from $s \in 2^{AP}$. The function *findNextStates* uses a number of subroutines that correspond to the methods in Sections 5.6.1 and 5.6.2. Given a set of E-equality constraints CTR, using Proposition 5.1, the function *findSolvable*(CTR) returns a triple (D, F, G) such that D is a set of G-solvable constraints, G has the finite variant property, and $CTR = D \cup F$. Then, using Lemma 5.14, the function *isSatisfiable*(D, F, G) returns *false* if the set of constraints $D \cup F$ is unsatisfiable, and *testUnsatisfiable*(F) returns *true* if F is shown to be unsatisfiable using Theorem 5.9.

To compute *findSolvable*(CTR) using Definition 5.13, we need a set of patterns $U \subseteq \mathcal{T}_\Sigma(\mathcal{X})$ such that any proper subterm of $u \in U$ is strongly E_o, B-irreducible, $CSU_B(t = u) = \emptyset$ for each $t = t' \in E - E_U$ and $u \in U$, and $G = \{u = v \in E_o \mid u, v \in [\![U]\!]\}$ has the finite variant property. A term appeared in the equations corresponding to the set of constraints CTR can be used to obtain a pattern in U. We can check by performing narrowing with E_o modulo B—available in Full Maude [72]—*one step* to check if a term t is strongly E_o, B-irreducible. Also, there exists a semi-decision procedure to check if a set of equations G has the finite variant property [53]; therefore, we apply this procedure *with a time limit* for checking if a set of equations G has the finite variant property.

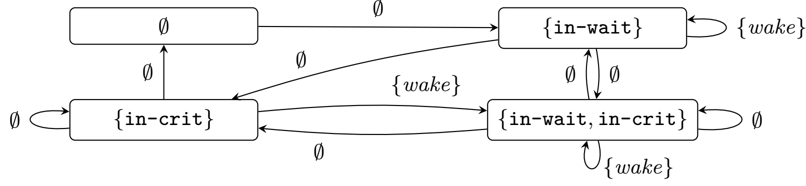


Figure 5.17: $\mathcal{P}$-abstract LKS for the livelock freedom property.

5.6.3 Case Study

We illustrate our predicate abstraction method with the bakery algorithm example in Section 5.2.1. We are first interested in verifying the livelock freedom "*if some process is waiting, then some (possibly different) process eventually enters the critical section*" under the fairness assumption "the *wake* rule is *not* taken infinitely many times," expressed as the LTLR formula

$$\Box\Diamond\neg \mathit{wake} \to \Box(\mathit{in\text{-}wait} \to \Diamond \mathit{in\text{-}crit}).$$

The state propositions are defined by the following equations that satisfy the finite variant property, where `WS` is a variable of sort ProcWaitSet to denote a set of processes with status *idle* or $\mathit{wait}(n)$, and `CS` is a variable of sort ProcCritSet to denote a set of processes with status *idle* or $\mathit{crit}(n)$:

```
eq N ; M ; [I,wait(K)] PS |= in-wait = true .
eq N ; M ; CS             |= in-wait = false .
eq N ; M ; [I,crit(K)] PS |= in-crit = true .
eq N ; M ; WS             |= in-crit = false .
```

We obtain the $\mathcal{P}$-abstract LKS in Figure 5.17 using E-unification, because a finitary E-unification algorithm is available. Since *in-crit* holds in every state in the second row, we can easily see that any infinite path *not* satisfying the formula $\Diamond \mathit{in\text{-}crit}$ has the suffix $\{\mathit{in\text{-}wait}, \mathit{wake}\} \to \{\mathit{in\text{-}wait}, \mathit{wake}\} \to \{\mathit{in\text{-}wait}, \mathit{wake}\} \to \cdots$, which violates the fairness assumption $\Box\Diamond\neg \mathit{wake}$. Indeed, when a system has a *finite* (but unbounded) number of processes, there is *no* infinite path with that suffix. Therefore, the livelock freedom formula $\Box\Diamond\neg \mathit{wake} \to \Box(\mathit{in\text{-}wait} \to \Diamond \mathit{in\text{-}crit})$ holds in the $\mathcal{P}$-abstract LKS from the initial state $\emptyset$. By Theorem 5.8, the formula also holds in the concrete LKS $\bar{\mathcal{K}}(\mathcal{R}, \mathsf{State})_{\mathcal{P}}$ from any initial state $[t]_E$ with $\mathcal{L}_{\mathcal{P}}([t]_E) = \emptyset$ for an unbounded number of processes.

We now consider the mutual exclusion $\Box \mathit{ex?}$. We need three extra state propositions to define a predicate abstraction: *mcrit*, *bound*, and *uniq*. First, the state proposition *mcrit* holds in a state $n\,;\,m\,;\mathit{PS}$ if at most one process in *PS* enters the critical section with number m:

```
eq N ; M ; WS |= mcrit = true .
eq N ; M ; [I,crit(M)] WS |= mcrit = true .
eq N ; M ; [I,crit(s K M)] WS |= mcrit = false .
eq N ; s K M ; [I,crit(K)] WS |= mcrit = false .
eq N ; M ; [I,crit(K)] [I',crit(K')] PS |= mcrit = false .
```

Next, the state proposition *bound* holds in a state $n\,;m\,;PS$ if any ticket number of a process in PS is less than n, defined by the equations:

```
eq N ; M ; PS |= bound = bd(N, PS) .               --- E_b
eq bd(N, [I,wait(N K)] PS) = false .               --- E_b
eq bd(N, [I,crit(N K)] PS) = false .               --- E_b
eq bd(s N K, [I,wait(K)] PS) = bd(s N K, PS) .
eq bd(s N K, [I,crit(K)] PS) = bd(s N K, PS) . eq bd(N, IS) = true .
```

Finally, the state proposition *uniq* holds in a state $n\,;m\,;PS$ if no duplicate ticket numbers of processes exist in PS, defined by the equations:

```
eq N ; M ; PS |= uniq = uq(PS) .                   --- E_q
eq uq([I,wait(K)] [I',wait(K)] PS) = false .       --- E_q
eq uq([I,wait(K)] [I',crit(K)] PS) = false .       --- E_q
eq uq([I,crit(K)] [I',crit(K)] PS) = false .       --- E_q
eq uq([I,wait(K)] [I',wait(s M K)] PS) = uq([I,wait(K)] PS) .
eq uq([I,wait(K)] [I',crit(s M K)] PS) = uq([I,wait(K)] PS) .
eq uq([I,crit(K)] [I',wait(s M K)] PS) = uq([I,crit(K)] PS) .
eq uq([I,crit(K)] [I',crit(s M K)] PS) = uq([I,crit(K)] PS) .
eq uq([I,idle] PS) = uq(PS) .       eq uq([I,wait(K)]) = true .
eq uq([I,crit(K)]) = true .         eq uq(none) = true .
```

Every state proposition clearly has syntactically independent equations. In particular, the equations defining *ex?* and *mcrit* have the finite variant property, because their right-hand sides are all constants. The equations defining *bound* and *uniq* do *not* have the finite variant property, but the equations $E_b \cup B$ and $E_q \cup B$ that define the *negative* cases of *bound* and *uniq* do have the finite variant property. Furthermore:

Lemma 5.15. *For a term $u \in \mathcal{T}_\Sigma(\mathcal{X})_{\mathsf{State}}$, a constraint $(u \models bound = false)$ is $E_b \cup B$-solvable, and $(u \models uniq = false)$ is $E_q \cup B$-solvable.*

Proof. A state term u has the form $n\,;m\,;t_{procs}$. For some substitution σ, if $\sigma u \models bound =_E false$, then σt_{procs} contains a process with ticket number greater than or equal to n. Since $E_b \cup B$ reduces such a negative case to *false* in 2 steps, $\sigma u \models bound =_{E_b \cup B} false$. Similarly, if $\sigma u \models uniq =_E false$, then σt_{procs} contains two processes with the same ticket number, and thus $\sigma u \models uniq =_{E_q \cup B} false$ holds in 2 steps. $\square$

We obtain $\bar{\mathcal{K}}(\mathcal{R}/\mathcal{P})$ having a *single* reachable state from the given initial state $\{ex?, mcrit, bound, uniq\}$. After resolving each constraint for $ex?$ and $mcrit$, from $\{ex?, mcrit, bound, uniq\}$, the remaining sets of constraints are $\{l_k \models bound = true,\ l_k \models uniq = true,\ r_k \models bound = b_k,\ r_k \models uniq = b'_k\}$ for $1 \leq j \leq 5$, where $b_k, b'_k \in \{true, false\}$ and each $l_k \to_k r_k$ is given by:

$$
\begin{aligned}
n\,;\, m\,;\, [i, idle]\ WS &\to_1\ s\,n\,;\, m\,;\, [i, wait(n)]\ WS \\
n\,;\, m\,;\, [i, idle]\,[j, crit(m)]\ WS &\to_2\ s\,n\,;\, m\,;\, [i, wait(n)]\,[j, crit(m)]\ WS \\
n\,;\, m\,;\, [i, wait(m)]\ WS &\to_3\ n\,;\, m\,;\, [i, crit(m)]\ WS \\
n\,;\, m\,;\, [i, crit(m)]\ WS &\to_4\ n\,;\, s\,m\,;\, [i, idle]\ WS \\
n\,;\, m\,;\, [i, wait(m)]\,[j, crit(m)]\ WS &\to_5\ n\,;\, m\,;\, [i, crit(m)]\,[j, crit(m)]\ WS
\end{aligned}
$$

The case of $k = 5$ is unsatisfiable for any values of $b_5, b'_5 \in \{true, false\}$, since $l_5 \models uniq =_E false$, conflicting with the constraint $l_5 \models uniq = true$. For the cases of $1 \leq k \leq 4$, if $b_k = false$ for $bound$, then its solution $\zeta \in CSU_{E_b \cup B}(r_k \models bound = false)$ makes $\zeta l_k \models bound =_E false$. That is, the cases of $b_k = false$ are unsatisfiable. Similarly, if $b'_k = false$ for $uniq$, then $\zeta \in CSU_{E_q \cup B}(r_k \models uniq = false)$ makes either $\zeta l_k \models bound =_E false$ or $\zeta l_k \models uniq =_E false$, i.e., unsatisfiable. Therefore, $\{ex?, mcrit, bound, uniq\}$ has the one next state, itself. Clearly, $\Box ex?$ holds in $\bar{\mathcal{K}}(\mathcal{R}/\mathcal{P})$, and thus $\Box ex?$ also holds in $\bar{\mathcal{K}}(\mathcal{R}, \mathsf{State})_{\mathcal{P}}$ for an unbounded number of processes.

5.7 Concluding Remarks

This chapter has presented various infinite-state model checking techniques for verifying LTLR properties of rewrite theories: (i) *equational abstractions* define quotients of the system by using equations, and can in some case define bisimulations; (ii) *folding abstractions* collapse the system's state space by folding preorders; (iii) *narrowing-based* symbolic model checking methods represent the infinite state space using logical terms, and are also naturally amenable to folding abstractions; and (iv) *predicate abstractions* generate finite-state abstractions of the system using state propositions. This work can also be understood as a contribution that increases the expressive power of these infinite-state model checking techniques for LTLR properties. These methods can be faithful, automated, and used in combination to effectively verify nontrivial infinite-state systems.

In future work we plan to implement a new LTLR model checker in Maude, extending the fair LTLR model checker and the LTL logical model checker, that supports predicate abstraction and narrowing-based model checking.

CHAPTER 6

MULTIRATE PALS

Distributed cyber-physical systems (DCPS), such as aeronautics and ground transportation systems, are very hard to design and verify, due to network delays, asynchronous communication, and clock skews. Therefore, the PALS ("physically asynchronous, logically synchronous") methodology has been proposed to reduce the effort and cost involved in design and verification for a *single-rate* virtually synchronous DCPS. This chapter[1] presents *Multirate PALS*, a multirate extension of PALS, which can reduce the design and verification of a *multirate* virtually synchronous DCPS to the much simpler task of designing and verifying its synchronous version. We illustrate the ideas with a multirate DCPS for an airplane maneuvered by a pilot, who turns the airplane to a specified angle by a distributed control system.

6.1 Introduction

Many cyber-physical systems such as cars, airplanes, robots, and networked medical devices are distributed real-time systems in which many components interact *asynchronously* through a network, yet must obey hard real-time *synchronization* constraints which are essential to their correctness; that is, they must be *virtually synchronous*. Such DCPS design and verification is quite challenging, since to the usual complexity of a non-distributed CPS one has to add the additional complexities of asynchronous communication, network delays, and clock skews. In particular, any hopes of applying model checking techniques in a direct manner to a DCPS look rather dim, due to the huge state space explosion caused by the system's concurrency.

[1]This chapter is based on [15, 16, 24, 25], joint work with Peter Ölveczky, José Meseguer, and Joshua Krisiloff who provided the aerodynamics model for the case study.

The PALS (physically asynchronous, logically synchronous) pattern has been developed for this reason [138, 143]. It can drastically reduce the system complexity of a *single-rate* virtually synchronous DCPS. The key idea of PALS is that if the underlying infrastructure provides some performance bounds Γ on the computation times, networks delays, and imprecisions of the local clocks, then the task of designing and verifying a DCPS can be reduced to the much simpler task[2] of designing and verifying the idealized synchronous system that should be realized in a distributed way. This is achieved by a *model transformation* $\mathcal{E} \mapsto \mathcal{A}(\mathcal{E}, \Gamma)$ that maps a synchronous design $\mathcal{E}$ and performance bounds Γ to a distributed implementation $\mathcal{A}(\mathcal{E}, \Gamma)$, which is *correct-by construction* as shown in [138].

However, the problem still remains that PALS assumes a *single period* for the virtually synchronous system. This excludes many DCPSs, in fact the majority, which do not operate at a single rate but are *multirate*. In practice, different sensors and effectors need to operate at different rates; and this necessitates using slower rates in the distributed control hierarchies that orchestrate and synchronize their actions.

This chapter presents *Multirate PALS* as a formal mathematical model providing a "formal pattern" that can drastically reduce the complexity of designing, verifying, and implementing multirate DCPSs, where their main architecture is one of *hierarchical distributed control*. Systems of this nature are very common in avionics, motor vehicles, and robotics. Although these systems are distributed, they must achieve virtual synchrony *in real time*, since actual deadlines must be met in physical time for physical reasons. This also poses strong requirements on the network infrastructures they can use, since these must ensure message delivery and clock synchronization within precise and tight enough bounds.

This chapter defines a Multirate PALS transformation $\mathcal{E} \mapsto \mathcal{MA}(\mathcal{E}, T, \Gamma)$, with underlying performance bounds Γ and global period T, that generalizes the original single-rate PALS transformation to multirate systems. Further, a DCPS often controls *physical entities* and is in fact a *distributed hybrid system*: a collection of digital components that communicate asynchronously with each other and that interact with their environment, whose continuous dynamics is typically governed by differential equations. Therefore, this chapter also investigates the suitability of Multirate PALS to design and verify nontrivial virtually synchronous distributed *hybrid* systems.

[2] For a simple avionics case study in [138], the number of system states for their simplest possible distributed version with perfect clocks and no network delays was 3,047,832, but the PALS pattern reduced the number of states to a mere 185.

6.1.1 Main Contributions

First, this chapter presents a *modular* way to specify multirate synchronous systems using the composition of several formal patterns. That is, we define component transformations, such as: (i) a transformation $M \mapsto M^{\times_k}$ that makes a state machine k times slower; (ii) a transformation $M \mapsto M_\alpha$ that adapts the inputs of machine M according to a adaptor function α; and (iii) a transformation $\mathcal{E} \mapsto \mathrm{MRSC}(\mathcal{E})$ that maps a *multirate ensemble* $\mathcal{E}$ to a single state machine equivalent to its synchronous composition, where $\mathcal{E}$ is a mathematical model of a collection of interconnected state machines running at different rates, yet synchronously in terms of their hyperperiod.

Second, based on these modular transformations, this chapter presents a Multirate PALS model transformation $(\mathcal{E}, T, \Gamma) \mapsto \mathcal{MA}(\mathcal{E}, T, \Gamma)$, which maps a multirate ensemble $\mathcal{E}$, together with period T and performance parameters Γ, to a semantically equivalent model of distributed real-time components $\mathcal{MA}(\mathcal{E}, T, \Gamma)$. In particular, we prove that the DCPS design $\mathcal{MA}(\mathcal{E}, T, \Gamma)$—a collection of multirate asynchronous components distributed in a network—is *bisimilar* to the enormously simpler *synchronous multirate ensemble* $\mathcal{E}$ of state machines. This bisimilarity provides a drastic reduction on the number of states, making model checking verification possible in many cases where it is unfeasible for the original DCPS.

Third, this chapter explains a general methodology to specify a DCPS using Multirate PALS, in particular, distributed hybrid systems. We define a modeling framework for formally specifying such Multirate PALS designs in the Real-Time Maude specification language [149]. Given a specification of a multirate ensemble $\mathcal{E}$ in Real-Time Maude, our framework defines an executable semantics for the synchronous composition of $\mathcal{E}$ that can be used to simulate and model check this multirate synchronous composition.

Finally, we use our methodology and the Real-Time Maude framework to formally specify in detail a multirate distributed hybrid system consisting of an airplane maneuvered by a pilot, who wants to turn the airplane in a desired direction through a distributed control system, with effectors located in the airplane's wings and rudder. Our formal analysis revealed that the original design did not achieve a smooth turn. This led to a redesign of the system with new control laws satisfying the desired correctness properties. This shows that the Multirate PALS methodology is not only effective for formal DCPS verification, but can also be used effectively in the DCPS *design* process, even before properties are verified. The Real-Time Maude specifications for the framework and the case study are available at [13].

6.1.2 Related Work

The most closely related work is the paper [6] that proposes a different multirate extension of PALS in terms of the AADL language. What is not attempted in [6] is to give *mathematical models* of either synchronous multirate systems or their multirate PALS transformation as distributed real-time systems, and to justify why the synchronous multirate system and its multirate PALS counterpart are equivalent. Moreover, our model of multirate PALS and theirs are considerably different; for example, their model lacks a systematic notion of *input adaptor*.

More generally, the PALS pattern can be seen as part of a broader body of work on *synchronizers*, which allow (single-rate) synchronous systems to be *simulated* by asynchronous ones. There are a number of synchronizers, such as [10, 101, 163, 153] (see [138] for an overview and comparison). To the best of our knowledge only Multirate PALS and the work in [6] propose synchronizers for multirate systems where tight time bounds must be met.

Single-rate PALS is closely related to the *time-triggered systems* [116, 155], where the goal is also to reduce an asynchronous real-time system to a synchronous one. One important difference between the work in [116, 155] and PALS comes from the different definitions of the synchronous models, which have significant repercussions in the behaviors of the corresponding asynchronous models. We refer to [157] for an in-depth comparison between time-triggered systems and PALS.

6.1.3 Structure of the Chapter

This chapter is organized as follows. Section 6.2 defines a number of formal patterns, leading to a formal definition of a hierarchical multirate machine ensemble $\mathfrak{E}$ and its synchronous composition MRSC($\mathfrak{E}$). Section 6.3 presents Multirate PALS as a model transformation from a multirate ensemble $\mathfrak{E}$ with period T and performance parameters Γ into a distributed real-time system $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$, and Section 6.3.3 proves the main bisimulation result between MRSC($\mathfrak{E}$) and $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$. Section 6.4 shows how multirate distributed hybrid systems can be modeled as multirate ensembles in Multirate PALS, and presents a modeling and execution framework for multirate ensembles in Real-Time Maude. Section 6.5 then illustrates the usefulness of Multirate PALS and the drastic state space reductions gained using it, by means of a case study that analyzes the turning maneuvers of an airplane. Finally, some concluding remarks are given in Section 6.6.

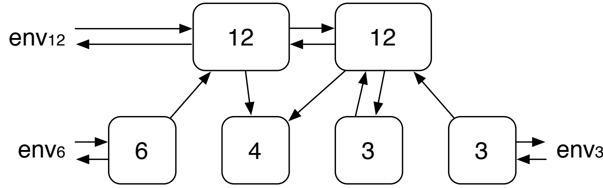


Figure 6.1: A simple multirate system, with each machine and each separate environment annotated by its period.

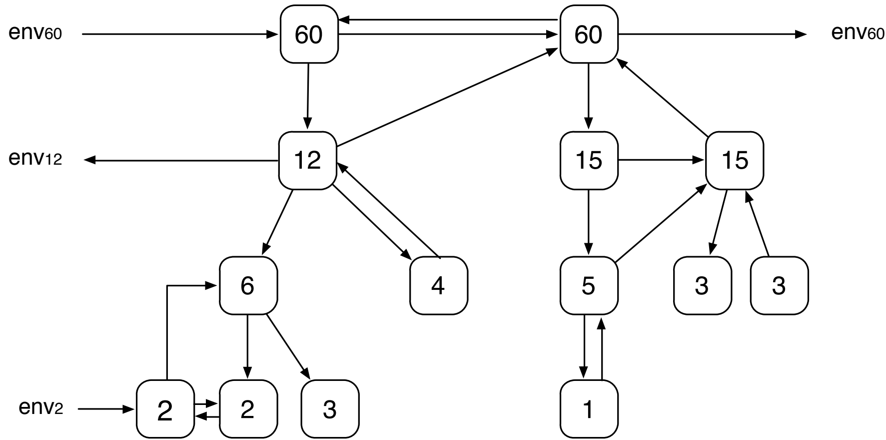


Figure 6.2: A hierarchical multirate system, with machines and separate environments annotated by their periods.

6.2 Multirate Synchronous Models

Virtually synchronized cyber-physical systems are commonly implemented as networked real-time systems consisting of distributed devices, controlled by a hierarchy of distributed controllers. The devices and controllers may operate at different rates, and in a perfectly synchronized distributed system, the synchronous changes of the local control applications can happen only at the *hyperperiod* boundary (i.e., at an interval equal to the least common multiple of the local control periods) [6]. We therefore consider multirate systems in which a set of components with the same rate may communicate with each other and faster components, so that the period of the higher-level components is a multiple of the period of each fast component, as illustrated in Figure 6.1. Such a multirate system can itself be a subcomponent of a larger system; in this way, we can capture complex *hierarchical* system configurations, e.g., the multirate system in Figure 6.2.

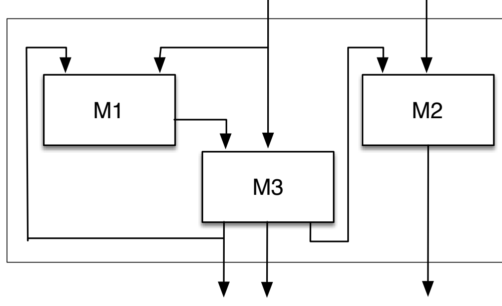


Figure 6.3: A machine ensemble.

6.2.1 Single-rate Ensembles

We first consider a *single-rate* synchronous model, since we define a multirate synchronous model as a generalization of a single-rate synchronous model. In this case, the synchronous model is defined by a synchronous composition of a collection of *nondeterministic typed machines*, an *environment*, and a *wiring diagram* that connects the machines [138].

Definition 6.1. *A* typed machine *is a tuple* $M = (D_i, S, D_o, \delta_M)$*, where* $D_i = D_{i_1} \times \cdots \times D_{i_n}$ *an* input set*,* S *a set of* states*,* $D_o = D_{o_1} \times \cdots \times D_{o_m}$ *an* output set*, and* $\delta_M \subseteq (D_i \times S) \times (S \times D_o)$ *a total* transition relation*.*

That is, a typed machine M is a state machine with n input ports and m output ports; an input to port k is an element of the set D_{i_k}, and an output from port j is an element of the set D_{o_j}.

Typed machines can be "wired together" into arbitrary sequential and parallel compositions by means of a "wiring diagram," as the one shown in Figure 6.3, where all the machines have the same *rate*.

Definition 6.2. *A single-rate* machine ensemble *is defined by a* 4 *tuple* $\mathcal{E} = (J \cup \{e\}, \{M_j\}_{j \in J}, E, src)$*, where:*

- $J \neq \emptyset$ *is a finite set of* indices*, and* $e \notin J$ *is the* environment index*.*
- $\{M_j\}_{j \in J}$ *is a* J*-indexed family of typed machines.*
- $E = (D_i^e, D_o^e)$ *is an* environment*, where* D_i^e *is the environment's* input set *and* D_o^e *is the environment's* output set*.*
- *src is a surjective function that assigns to each input port* (j, n) *(input port* n *of machine* j*) the "source" output port* $src(j, n)$*, where: (i) an output domain is a subset of the corresponding input domain; i.e., if* $src(j, q) = (k, l)$*, then* $D_{o_l}^k \subseteq D_{i_q}^j$*; and (ii) there are no self-loops from the environment to itself, i.e., if* $src(e, q) = (k, l)$*, then* $k \in J$*.*

A single-rate ensemble $\mathcal{E}$ has a *lock-step synchronous semantics*, in the sense that the transitions of all the machines are performed simultaneously, and whenever a machine has a feedback wire to itself and/or to any other machine, then the output becomes an input at the *next* instant. This means that any single-rate ensemble $\mathcal{E}$ is semantically equivalent to a *single machine* $M_\mathcal{E}$, called the *synchronous composition* of ensemble $\mathcal{E}$. For example, in Figure 6.3, the synchronous composition of the typed machines M_1, M_2, and M_3 can be seen as the single machine enclosed by the outer box.

Definition 6.3. *For a single-rate ensemble* $\mathcal{E} = (J \cup \{e\}, \{M_j\}_{j \in J}, E, src)$, *its* synchronous composition *is* $M_\mathcal{E} = (D_i^\mathcal{E}, S^\mathcal{E}, D_o^\mathcal{E}, \delta_\mathcal{E})$*, where:*

- $D_i^\mathcal{E} = D_o^e$ *and* $D_o^\mathcal{E} = D_i^e$.
- $S^\mathcal{E} = (\Pi_{j \in J} S_j) \times (\Pi_{j \in J} D_{OF}^j)$*, where* D_{OF}^j *stores the "feedback outputs" of machine* M_j *that will be used as input in the next iteration.*

  Formally, if $D_o^j = D_{o_1}^j \times \cdots \times D_{o_{m_j}}^j$ *is the output set of* M_j*, then* $D_{OF}^j = D_{OF_1}^j \times \cdots \times D_{OF_{m_j}}^j$ *such that for each* $1 \leq m \leq m_j$*, if* $(j, m) = src(l, q)$ *for some* $l \in J$*, then* $D_{OF_m} = D_{o_m}^j$*, and otherwise,* $D_{OF_m} = \{*\}$*, where* $\{*\}$ *is a singleton set to denote "no information."*
- $\delta_\mathcal{E} \subseteq (D_i^\mathcal{E} \times S^\mathcal{E}) \times (S^\mathcal{E} \times D_o^\mathcal{E})$ *"combines" the transitions of the single machines* $\{M_j\}_{j \in J}$ *into a synchronous step.*

  Formally, let: (i) $fo_j(\vec{d_j'}) \in D_{OF}^j$ *be the feedback output for machine* M_j *generated from* M_j*'s output* $\vec{d_j'} \in D_o^j$*;*[3] *(ii)* $in_l(\vec{d}, \{\vec{d_j}\}_{j \in J}) \in D_i^k$ *be the input for machine* M_l *generated from its "connected" outputs using an environment output* $\vec{d} \in D_i^\mathcal{E}$ *and a collection of feedback outputs* $\{\vec{d_j}\}_{j \in J} \in \Pi_{j \in J} D_{OF}^j$*;*[4] *and (iii)* $in_e(\{\vec{d_j'}\}_{j \in J}) \in D_o^\mathcal{E}$ *be the environment input generated from its "connected" outputs using machine outputs* $\{\vec{d_j}\}_{j \in J} \in \Pi_{j \in J} D_o^j$.[5] *Then, for* $\vec{d} \in D_i^\mathcal{E}$ *and* $(\{s_j\}_{j \in J}, \{\vec{d_j}\}_{j \in J}) \in S^\mathcal{E}$*:*

  $$((\vec{d}, (\{s_j\}_{j \in J}, \{\vec{d_j}\}_{j \in J})), ((\{s_j'\}_{j \in J}, \{fo_j(\vec{d_j'})\}_{j \in J}), in_e(\{\vec{d_j'}\}_{j \in J}))) \in \delta_\mathcal{E}$$

  iff $((in_l(\vec{d}, \{\vec{d_j}\}_{j \in J}), s_l), (s_l', \vec{d_l'})) \in \delta_{M_l}$ *for each machine index* $l \in J$. *Notice that* $\delta_\mathcal{E}$ *is a total relation, since each* δ_{M_l} *is a total relation.*

---

[3] If π_m denotes the m-th projection from the Cartesian product D_{OF}^j, then a feedback output function $\mathrm{fo}_j : D_o^j \to D_{OF}^j$ is defined as follows: if $\exists l \in J.\ (j, m) = src(l, q)$, then $\pi_m(\mathrm{fo}_j(d_1, \ldots, d_{m_j})) = d_m$, and otherwise, $\pi_m(\mathrm{fo}_j(d_1, \ldots, d_{m_j})) = *$, for $1 \leq m \leq m_j$.

[4] An input function $in_k : D_o^e \times (\prod_{j \in J} D_{OF}^j) \to D_i^k$ is formally defined as follows: for $1 \leq n \leq n_k$, if $\exists l \in J.\ src(k, n) = (l, q)$, then $\pi_n(in_k(\vec{d}, \{\vec{d_j}\}_{j \in J})) = \pi_q(\vec{d_l})$, and if $src(k, n) = (e, q)$, then $\pi_n(in_k(\vec{d}, \{\vec{d_j}\}_{j \in J})) = \pi_q(\vec{d})$.

[5] An environment input function $in_e : (\prod_{j \in J} D_o^j) \to D_i^e$ is formally defined as follows: for each $1 \leq n \leq n_e$, $\pi_n(in_e(\{\vec{d_j'}\}_{j \in J})) = \pi_r(\vec{d_l'})$ iff $src(e, n) = (l, r)$.

We can associate a transition system defining the behaviors of a machine ensemble that operates in a certain environment.

Definition 6.4. *For a single-rate ensemble* $\mathcal{E} = (J \cup \{e\}, \{M_j\}_{j \in J}, E, src)$*, the corresponding transition system is a pair*

$$ts(\mathcal{E}) = (S^{\mathcal{E}} \times D_i^{\mathcal{E}}, \longrightarrow_{\mathcal{E}}),$$

where $(\vec{s}, \vec{i}) \longrightarrow_{\mathcal{E}} (\vec{s'}, \vec{i'})$ *iff an ensemble in state* $\vec{s}$ *and with input* $\vec{i}$ *from the environment has a transition to state* $\vec{s'}$*:*[6]

$$(\vec{s}, \vec{i}) \longrightarrow_{\mathcal{E}} (\vec{s'}, \vec{i'}) \iff \exists \vec{o}.\ ((\vec{i}, \vec{s}), (\vec{s'}, \vec{o})) \in \delta_{\mathcal{E}}.$$

If $L : S^{\mathcal{E}} \times D_i^{\mathcal{E}} \to 2^{AP}$ is a *labeling function* that assigns to each state $(\vec{s}, \vec{i}) \in S^{\mathcal{E}} \times D_i^{\mathcal{E}}$ the set $L(\vec{s}, \vec{i})$ of atomic propositions that hold in $(\vec{s}, \vec{i})$, then the associated Kripke structure is $\mathcal{K}(\mathcal{E}) = (S^{\mathcal{E}} \times D_i^{\mathcal{E}}, AP, L, \longrightarrow_{\mathcal{E}})$.

6.2.2 Machine Transformations

There are in essence two ways of composing machines with different periods into a synchronous system in which all components operate in lock-step. On the one hand, one can "speed up" the slower components, so that all components operate at the fastest rate. On the other hand, one can "slow down" the faster components so that all components run at the slow rate. We follow the latter approach, since it is more natural to consider the system at the slower rate of the "higher-level" components for the following reasons:

1. the PALS transformation operates at the slowest rate and is the crucial one, so that slowing the system down provides the most adequate level of abstraction to explain how this transformation is used;

2. the synchronous interaction between a fast component and slow ones can be better understood by slowing down the fast component, since this makes explicit that the decelerated fast machine now has *tuples* of inputs and outputs that must be *adapted* in either direction to allow it to communicate with slow components; and

[6] An environment can be constrained by means of a *satisfiable* predicate $c_e : D_o^e \to Bool$ so that $c_e(d_1^e, \ldots, d_{o_{m_e}}^e)$ is *true* iff the environment can generate output $(d_1^e, \ldots, d_{o_{m_e}}^e)$ [138]. In this case, the constrained transition system $ts(\mathcal{E}, c_e) = (S^{\mathcal{E}} \times D_i^{\mathcal{E}}, \longrightarrow_{\mathcal{E}_{c_e}})$ is defined by: $(\vec{s}, \vec{i}) \longrightarrow_{\mathcal{E}_{c_e}} (\vec{s'}, \vec{i'}) \iff \exists \vec{o}.\ ((\vec{i}, \vec{s}), (\vec{s'}, \vec{o})) \in \delta_{\mathcal{E}} \wedge c_e(\vec{i'})$. However, we can easily see that $ts(\mathcal{E}, c_e)$ is exactly a subsystem of $ts(\mathcal{E})$, since $\longrightarrow_{\mathcal{E}_{c_e}} \subseteq \longrightarrow_{\mathcal{E}}$.

3. for model checking purposes, a potentially huge state space reduction accrues to slowing down all behaviors, since multiple "fast" transitions are combined into a single slower transition, so that all intermediate states can safely be ignored.

A fast machine that is slowed down, or *decelerated*, by a factor k *performs* k *internal transitions* during one (slow) period. Because the fast machine consumes an input and produces an output at each port in each of these internal steps, the decelerated machine consumes and produces k-tuples of inputs and outputs in each slow step. Hence, we define the k*-step machine pattern* by which we can decelerate a fast machine by a factor k, where the machine reads k inputs (in each port), performs a transition corresponding to k "internal transition steps," and outputs k-tuples of values:

Definition 6.5. *For* $k \in \mathbb{N}_+$*, the* k*-step* deceleration *of a typed machine* $M = (D_i, S, D_o, \delta_M)$*, with* $D_i = D_{i_1} \times \cdots \times D_{i_n}$ *and* $D_o = D_{o_1} \times \cdots \times D_{o_m}$*, is* $M^{\times k} = ((D_{i_1})^k \times \cdots \times (D_{i_n})^k,\ S,\ (D_{o_1})^k \times \cdots \times (D_{o_m})^k,\ \delta_{M^{\times k}})$*, where*

$$\begin{aligned}&((((d_{i_{1_1}}, \ldots, d_{i_{1_k}}), \ldots, (d_{i_{n_1}}, \ldots, d_{i_{n_k}})), s),\\ &\quad (s', ((d_{o_{1_1}}, \ldots, d_{o_{1_k}}), \ldots, (d_{o_{m_1}}, \ldots, d_{o_{m_k}})))) \in \delta_{M^{\times k}}\end{aligned}$$

iff there exist states $s_1, \ldots, s_{k-1} \in S$ *such that*

$$\begin{aligned}(((d_{i_{1_1}}, \ldots, d_{i_{n_1}}), s),\ (s_1, (d_{o_{1_1}}, \ldots, d_{o_{m_1}}))) &\in \delta_M\\ (((d_{i_{1_2}}, \ldots, d_{i_{n_2}}), s_1),\ (s_2, (d_{o_{1_2}}, \ldots, d_{o_{m_2}}))) &\in \delta_M\\ \vdots \qquad\qquad\qquad\qquad & \vdots\\ (((d_{i_{1_k}}, \ldots, d_{i_{n_k}}), s_{k-1}),\ (s', (d_{o_{1_k}}, \ldots, d_{o_{m_k}}))) &\in \delta_M.\end{aligned}$$

When composing a fast machine and a slow machine, a k-tuple output from the fast machine must be *adapted* so that it can be read by the slow component. That is, the k-tuple may need to be transformed to a single value (for example, the average of the k values, the first of the k values, or any other function of the k values). Likewise, since the "slowed-down" fast component expects a k-tuple of input values in each input port, the single output from a slow component must be transformed to a k-tuple of inputs to the fast machine; for example, transform an input value d to a k-tuple $(d, \bot, \ldots, \bot)$ for some "don't care" value $\bot$. Such a transformation is formalized by the *input adaptor pattern*, specifying how a typed machine with input adaptors can be transformed to an ordinary typed machine.

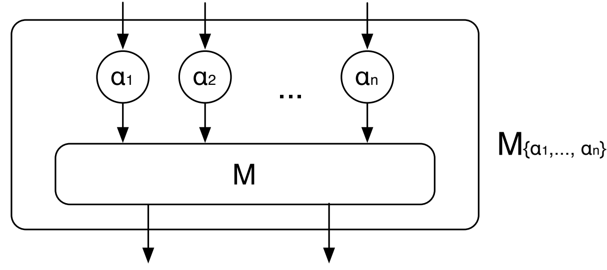


Figure 6.4: The adaptor closure $M_{\{\alpha_1,\ldots,\alpha_n\}}$ of the machine M.

Definition 6.6. *An* input adaptor *α for $M = (D_i, S, D_o, \delta_M)$ is a family of functions $\alpha = \{\alpha_k : D'_k \to D_{i_k}\}_{k\in\{1,\ldots,n\}}$, where $D_i = D_{i_1} \times \cdots \times D_{i_n}$. If $\vec{d} = (d_1, \ldots, d_n) \in D'_1 \times \cdots \times D'_n$, we define $\alpha(\vec{d}) = (\alpha_1(d_1), \ldots, \alpha_n(d_n))$.*

A typed machine with an input adaptor can be regarded as another typed machine, which corresponds to the "outer box" in Figure 6.4.

Definition 6.7. *The* adaptor closure *of $M = (D_i, S, D_o, \delta_M)$ with an input adaptor $\alpha = \{\alpha_k : D'_k \to D_{i_k}\}_{k\in\{1,\ldots,n\}}$ is $M_\alpha = ((D'_1 \times \cdots \times D'_n), S, D_o, \delta_{M_\alpha})$, where: $((\vec{d_i}, s), (s', \vec{d_o})) \in \delta_{M_\alpha} \iff ((\alpha(\vec{d_i}), s), (s', \vec{d_o})) \in \delta_M$.*

6.2.3 Multirate Ensembles

We define a *multirate machine ensemble* to be a network of typed machines with different rates and input adaptors, where a set of "slow" components with the same rate may communicate with each other and with a number of faster components, whose periods divide the period of the slow components. Since the "local" fast environments should be dealt with by the correlated fast machines, we assume that fast local environments are already integrated with their corresponding fast machines.[7] That is, the environment at the (slow) global level is only the environment of the high-level components. Finally, we make the definition more abstract by considering only the relative rates instead of the concrete periods. For example, the multirate "system" in Figure 6.1 corresponds to the multirate ensemble in Figure 6.5.

We formally define *multirate machine ensembles*, using input adaptors and k-step machines, and the associated *multirate synchronous composition pattern* by which it can be composed into a single typed machine.

[7]Note that an environment may satisfy certain input-output constraints [138] and can therefore be viewed as a nondeterministic typed machine. Therefore, a faster machine's environment and the fast machine itself form a simple 2-machine ensemble, whose *ensemble composition* has now only input wires from (resp. output wires to) slow machines.

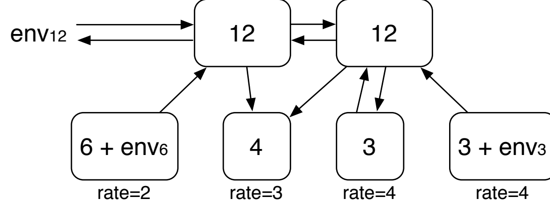


Figure 6.5: A flat multirate ensemble (input adaptors not shown).

Definition 6.8. *A flat* multirate *machine ensemble is defined by a tuple* $\mathfrak{E} = (J_S, J_F, e, \{M_l\}_{l \in J_S \cup J_F}, E, src, rate, adap)$*, where:*

- J_S *is a nonempty set of ("slow machine") indices and* J_F *is a set of ("fast machine") indices such that* $J_S \cap J_F = \emptyset$*, and* $e \notin J_S \cup J_F$ *is the environment index;*
- $\{M_l\}_{l \in J_S \cup J_F}$ *is a family of typed machines, and* $E = (D_i^e, D_o^e)$ *is the environment of the ensemble* $\mathfrak{E}$*;*
- *rate is a function* $rate : J_F \to \mathbb{N} - \{0, 1\}$*, assigning to each fast machine a value denoting how many times faster the machine runs compared to the slow machines; and*
- *src is a wiring diagram such that there exist no connections between fast machines, or between the environment and a fast machine; i.e., if* $src(l, q) = (k, p)$*, then* $l \in J_S$ *or* $k \in J_S$*; and*
- *adap is a function that assigns an input adaptor to each* $l \in J_F \cup J_S$*,*

such that $SR(\mathfrak{E})$ *is an ordinary (single-rate) typed machine ensemble:*

$$SR(\mathfrak{E}) = (J_S \cup J_F \cup \{e\}, \{(M_j)_{adap(j)}\}_{j \in J_S} \cup \{(M_j^{\times rate(j)})_{adap(j)}\}_{j \in J_F}, E, src).$$

As this definition indicates, we can reduce multirate ensembles to ordinary single-rate ensembles of Definition 6.2 by using the *k-machine* and the *input adaptor* patterns. The *multirate synchronous composition pattern* applied to $\mathfrak{E}$ is then the transformation $\mathfrak{E} \mapsto \mathrm{MRSC}(\mathfrak{E})$, assigning to a multirate ensemble $\mathfrak{E}$ the machine $\mathrm{MRSC}(\mathfrak{E}) = M_{SR(\mathfrak{E})}$, which is the synchronous composition of the corresponding single-rate ensemble $SR(\mathfrak{E})$. Notice that when the set J_F is empty and the input adaptors are identity functions, a multirate ensemble $\mathfrak{E}$ becomes an ordinary single-rate ensemble.

Such a flat multirate ensemble can be extended to the hierarchical case by allowing some fast components to be (hierarchical) multirate ensembles themselves, as precisely stated in the following definition:

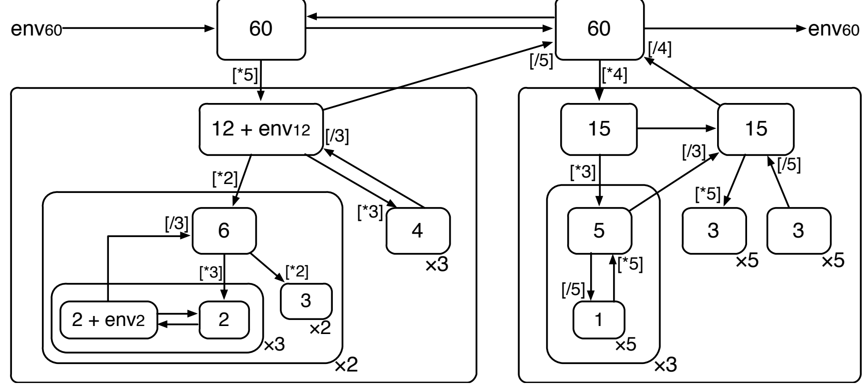


Figure 6.6: A hierarchical multirate ensemble, where $\times n$ denotes that the machine has been decelerated by a factor n, $[/n]$ denotes an input adaptor that transforms an n-tuple into a single value, and $[*n]$ denotes an input adaptor that transforms a single value to an n-tuple.

Definition 6.9. *A* hierarchical multirate ensemble of depth d *is defined by* $\mathfrak{E} = (J_S, J_F, K, e, \{M_j\}_{j \in J_S \cup (J_F - K)}, \{\mathfrak{E}_{se}\}_{se \in K}, E, src, rate, adap)$*, where:*

- *J_S, J_F, e, $\{M_j\}_{s \in J_S \cup (J_F - K)}$, E, src, rate, and adap are given as in Definition 6.8 of flat multirate ensembles, and must satisfy the same constraints as in Definition 6.8;*
- *$K \subseteq J_F$, where K is the subset of indices of the fast components that are themselves subensembles; and*
- *each $\mathfrak{E}_{se}$ for $se \in K$ is a* hierarchical multirate ensemble *of depth d_{se}, and* $\max(\{d_{se}\}_{se \in K}) + 1 = d$*, with* $\max(\emptyset) = 0$*,*

such that $SR(\mathfrak{E})$ is an ordinary (single-rate) machine ensemble:

$$SR(\mathfrak{E}) = (J_S \cup J_F \cup \{e\}, \{(M_s)_{adap(s)}\}_{s \in J_S} \cup \{(M_f^{\times rate(f)})_{adap(f)}\}_{f \in J_F}, E, src)$$

with $M_{SR(\mathfrak{E})}$ its synchronous composition, and where each M_{se} for $se \in K$ is itself the synchronous composition $M_{SR(\mathfrak{E}_{se})}$ of the subcomponent $\mathfrak{E}_{se}$.

In this definition, a flat multirate ensemble is just a hierarchical multirate ensemble of depth 1 with $K = \emptyset$. An ensemble $\mathfrak{E}_s$ is a *subensemble* of an ensemble $\mathfrak{E}$ defined above if either $\mathfrak{E} = \mathfrak{E}_s$ or $\mathfrak{E}_s$ is a subensemble of $\mathfrak{E}_{se}$ for some $se \in K$. An ensemble $\mathfrak{E}_s$ is a *proper subensemble* of $\mathfrak{E}$ iff $\mathfrak{E}_s \neq \mathfrak{E}$, and $\mathfrak{E}_s$ is an *immediate subensemble* of $\mathfrak{E}$ iff $\mathfrak{E}_s$ equals $\mathfrak{E}_{se}$ for some $se \in K$. For example, the hierarchical multirate system in Figure 6.2 can be seen as the hierarchical multirate ensemble in Figure 6.6.

6.3 Multirate PALS Transformation

This section presents *Multirate PALS* as a formal architectural pattern $(\mathfrak{E}, T, \Gamma) \mapsto \mathcal{MA}(\mathfrak{E}, T, \Gamma)$ that transforms a multirate ensemble $\mathfrak{E}$, together with its (global) period T and performance bounds Γ on network delays, clock skews, and execution times, into a formal specification $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ of a distributed real-time system where each machine performs at its own rate. Multirate PALS ensures that the synchronous composition and the distributed asynchronous real-time model satisfy the same properties.

6.3.1 System Assumptions

The PALS pattern is in general based on the facts that: (i) many network infrastructures for safety-critical cyber-physical systems (e.g., the networks in an airplane) can guarantee an upper bound on the network delays; and that (ii) clock synchronization is well understood, so that we can assume that the underlying infrastructure executes a clock synchronization algorithm that guarantees a certain bound on the imprecision of the local clocks [138]. Therefore, PALS is parametrized by the following *performance parameters* $\Gamma = (\epsilon, \alpha_{min}, \alpha_{max}, \mu_{\min}, \mu_{max})$ of the underlying network infrastructure:

- the difference between the time shown by a local clock and "real" time is always *strictly less* than ϵ;
- the time required for processing input, executing a transition, and generating output is always in the time interval $[\alpha_{min}, \alpha_{max}]$;
- the point-to-point message transmission time is within the interval $[\mu_{\min}, \mu_{max}]$, with $0 \leq \mu_{\min} \leq \mu_{max}$.

There exists an important difference between a single-rate system and a multirate system: a fast component (with rate k) in the multirate setting must perform k internal transitions during one (slow) period of the system. In the *synchronous* composition $MRSC(\mathfrak{E})$ the fast machine performs all k transitions "instantaneously," whereas in the *asynchronous* real-time system $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ the fast machine must operate according to its own fast period T/k and therefore performs *one* transition at the beginning of each fast period. It may happen that the fast component *cannot* finish all of its internal transitions *before* the messages must be sent to the slow component to ensure that they arrive before the beginning of the next round, even if T satisfies the constraints of the PALS period, as depicted in Figure 6.7.

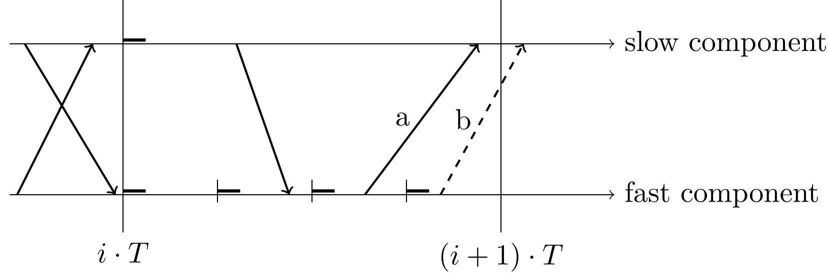


Figure 6.7: Timeline for multirate asynchronous PALS system with $k = 4$, where diagonal arrows denote message transmission and short horizontal lines denote the execution of a transition. The dashed diagonal arrow 'b' illustrates that the messages may arrive after the beginning of the next slow round if the fast component waits until all internal transitions are finished in a slow round before sending its messages; instead, the messages are sent after the third fast transition has taken place (solid diagonal arrow 'a').

In order to ensure that the sending of messages can be delayed until all fast transitions in a slow round have been performed, when $\alpha_{\max_f}$ is the maximal transition execution time for the fast component, the *fast* period T/k must satisfy the constraint

$$T/k \geq 2\epsilon + \mu_{\max} + \alpha_{\max_f}.$$

Using adaptors, this quite stringent constraint can be avoided as follows. If, *in the worst case*, there is not enough time for a fast machine M_f to execute all of its k transitions before the messages must be sent to the slow component, but can only send $k'_f < k$ inputs, then the slow component should only consider these k'_f values. The number of transitions that M_f can perform in a global round before its output must be sent is given by:

$$k'_f = 1 + \lfloor \frac{(T \;\texttt{monus}\;\; (2\epsilon + \mu_{max} + \alpha_{max_f})) \cdot k}{T} \rfloor,$$

where x $\texttt{monus}$ $y = \max(x - y, 0)$. That is, if the source of the ith input port of a slow machine M_j is a fast machine f whose k'_f is less than its rate k, then the adaptor function $adap(j)_i$ must satisfy

$$adap(j)_i(v_1, \ldots, v_{k'_f}, v_{k'_f+1}, \ldots, v_k) = adap(j)_i(v_1, \ldots, v_{k'_f}, v'_{k'_f+1}, \ldots, v'_k)$$

for all values v_l, $1 \leq l \leq k$, and v'_l, $k'_f + 1 \leq l \leq k$, of appropriate types. We call such input adaptor functions $(k'_f + 1)$-*oblivious*. We assume that a null value $\perp$ has been added to each type, so that the "don't care" values $v_{k'+1}, \ldots, v_k$ can always be chosen to be $\perp$.

6.3.2 Multirate Asynchronous Models

This section presents a high level summary of the asynchronous model $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$. The formal semantics of $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$—specified as a rewrite theory in Real-Time Maude—can be found in Appendix B.1. Basically, the asynchronous model $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ adds "wrappers" around each typed machine in a multirate ensemble $\mathfrak{E}$, where typed machines and wrappers in $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ perform *at their own rate*. These wrappers have an input buffer, an output buffer, and a local clock that deviates by less than ε from a global perfect clock, and some timers.

We first recall the asynchronous model $\mathcal{A}(\mathcal{E}, T, \Gamma)$ in single-rate PALS for a single-rate ensemble $\mathcal{E}$ [138]. The asynchronous model $\mathcal{A}(\mathcal{E}, T, \Gamma)$ adds a "PALS wrapper" around each machine in $\mathcal{E}$, where all components have the same period (the "PALS period" T). The behavior of such an asynchronous component can be summarized as follows:

1. Received messages are stored in the input buffer.

2. When a new round begins (according to the local clock), it reads all input from the input buffer, performs a "typed machine transition," which changes the "machine state" of the component and produces output which is put into the output buffer.

3. The output backoff timer is set to a value b to ensure that messages that are sent upon its expiration will not be received too early by components with slow local clocks.

4. When the output backoff timer expires or when the execution of the transition has finished (whichever comes last), the messages in the output buffer are sent into the network.

As shown in the paper [138], all messages are read in a "round-consistent" way as long as $b \geq 2\epsilon \texttt{ monus } \mu_{\min}$ and $T \geq \mu_{max} + 2\epsilon + \max(b, \alpha_{max})$.

In contrast, since Multirate PALS is based on a number of formal patterns, for modularity and readability purposes we use multiple wrappers to define the asynchronous model $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ of a multirate ensemble $\mathfrak{E}$, where each wrapper realizes the corresponding pattern in the distributed setting. For a *flat* multirate ensemble $\mathfrak{E}$, as illustrated in Figure 6.8, the outermost wrapper is the standard PALS wrapper, which encloses an input adaptor wrapper, which encloses either a (slow) typed machine or a k-decelerated machine wrapper, which in turn encloses an ordinary (fast) typed machine. The behavior of a fast component with rate k can be summarized as follows:

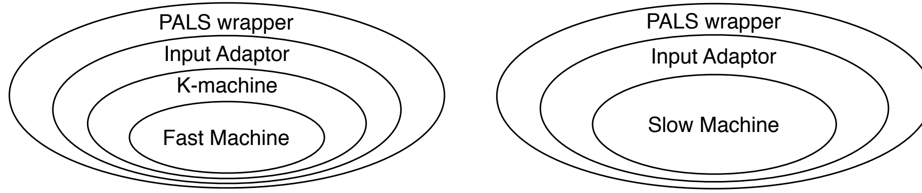


Figure 6.8: The wrapper hierarchies of a fast component (left) and a slow component (right) for a *flat* multirate ensemble $\mathfrak{E}$.

1. The PALS wrapper communicates with the other components in the network by sending and receiving messages. In particular, the PALS wrapper stores messages received during a round in its input buffer.

2. When a new (slow) round begins, according to the local clock of the PALS wrapper, the PALS wrapper puts the contents of its input buffer into the input buffer of the input adaptor object that it wraps around.

3. The input adaptor wrapper applies the input adaptor function to get k-tuples of inputs for each input value received, and then sends these k-tuples to the layer immediately below, a k-machine wrapper.

4. The k-machine wrapper extracts the first value from each k-tuple input and sends them to the layer immediately below (the typed machine). At the beginning of each fast period, it sends the "next" input values from the k-tuples to the layer below, until the end of the slow round.

5. The innermost layer is the typed machine itself. At the beginning of each (fast) period, it reads its input from the layer above, performs a transition that changes its state and generates some output. When the execution is finished, the machine sends out the generated messages.

6. These output messages are picked up by the k-machine wrapper. If it has received all k such sets of outputs before its timer expires, it sends out the resulting k-tuples of outputs to its outer layer. However, the k-machine wrapper may not be able to wait for all k rounds of outputs before having to send the outputs, in which case it sends whatever output sequences it has gotten when its timer expires, padded with don't care values $\bot$ to obtain k-tuples.

7. The outputs from the k-machine wrapper are immediately picked up by the input adaptor wrapper, which immediately propagates these outputs to its outer wrapper, the PALS wrapper.

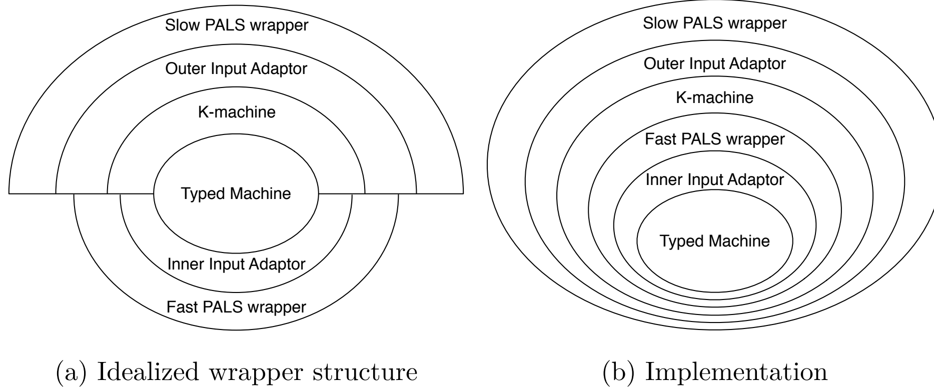
8. Whenever the PALS wrapper receives output from the layer below, it sends them out into the network, provided that its output backoff timer has expired. If the backoff timer has not expired, the outputs are stored in the PALS wrapper's output buffer, the contents of which is sent into the network when the backoff timer expires.

The behavior of a slow controller component is a simplified version of the behavior of a fast component. Since there is no k-machine wrapper around a slow component, it communicates with the other components at its own slow rate. The input adaptor wrappers apply adaptor functions that should map tuples of inputs from the fast components, as well as single values from the other slow controllers, to single input values.

For a *hierarchical* multirate ensemble $\mathfrak{E}$ of some depth d, we recursively construct $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ as follows. If $d = 1$, then we just construct the *flat* asynchronous model $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$. If $d > 1$, then for each immediate subensemble $\mathfrak{E}_{se}$ of $\mathfrak{E}$, we (recursively) construct $\mathcal{MA}(\mathfrak{E}_{se}, T/\mathit{rate}(se), \Gamma)$, and then "integrate" those submodels into $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$. For an immediate subensemble $\mathfrak{E}_{se}$ of a hierarchical ensemble $\mathfrak{E}$, the slow components in $\mathfrak{E}$ become the environment of $\mathfrak{E}_{se}$.[8] As a consequence, messages sent by the slow components in the fast subsystem $\mathcal{MA}(\mathfrak{E}_{se}, T/\mathit{rate}(se), \Gamma)$ to the "local" environment e_{se} of the subensemble $\mathfrak{E}_{se}$ must instead be addressed to the appropriate slow component(s) in $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$. Likewise, messages from slow components in $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ to the fast component se must be addressed to the appropriate component(s) inside $\mathcal{MA}(\mathfrak{E}_{se}, T/\mathit{rate}(se), \Gamma)$.

Moreover, the fast subsystem $\mathcal{MA}(\mathfrak{E}_{se}, T/\mathit{rate}(se), \Gamma)$ must communicate with the slow components in $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ as if it were a fast component communicating with a slow component. Consequently, such *relatively slow components* in $\mathcal{MA}(\mathfrak{E}_{se}, T/\mathit{rate}(se), \Gamma)$ no longer communicate at a single rate: they communicate at their own rate with the other components *inside* the fast subsystem $\mathcal{MA}(\mathfrak{E}_{se}, T/\mathit{rate}(se), \Gamma)$, but at a rate that is slower than their own by a factor $\mathit{rate}(se)$ with the slow components in $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$. We can conceptually represent these components with the "divided" wrapper structure in Figure 6.9a. In reality, such a (relatively) slow component in a fast subensemble has two sets of wrappers as shown in Figure 6.9b: one set of wrappers deals with inputs from, and outputs to, the components *inside* the subcomponents, and the other set of wrappers deals with communication with its environment (i.e., the slow components in $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$).

[8]Recall that any multirate ensemble $\mathfrak{E}$ has an *environment*, where (i) only the slowest components in $\mathfrak{E}$ communicate with the environment, and (ii) the ensemble communicates with its environment at its own rate (i.e., at the rate of its slowest components).



(a) Idealized wrapper structure (b) Implementation

Figure 6.9: The wrapper structures of a relatively slow component.

Finally, the global states of $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ have the form $\{C; t\}$, where C is the configuration consisting of hierarchical distributed components and messages traveling between the different components, and t is the global time. The environment E is also formalized as a typed machine surrounded only by the PALS wrapper in C, since we assume that E can generate any output and that E satisfies the same timing requirements as all other (slow) objects. A message has the form `(to` o `from` o' `(`p`,`d`))`, where o and o' are the components identifiers, p is the identifier of the receiving port, and d is the data element. A transition $\{C_1; t_1\} \longrightarrow \{C_2; t_2\}$ between global states is specified by means of rewrite rules (see Appendix B.1 for details).

6.3.3 Correctness of the Multirate PALS Transformation

This section formalizes the relationship between the multirate synchronous composition MRSC($\mathfrak{E}$) and the multirate asynchronous system $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$. Since each synchronous transition step in MRSC($\mathfrak{E}$) corresponds to several steps in $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$, we follow [138] and define "bigger" transition steps in $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$, so that each of these bigger transitions corresponds to a single step in MRSC($\mathfrak{E}$). In particular, for single-rate PALS, the *stable* states of $\mathcal{A}(\mathcal{E}, T, \Gamma)$ of a single-rate ensemble $\mathcal{E}$, where all components have *full input buffers* and *empty output buffers*, are those asynchronous states corresponding to states of the synchronous composition $M_{\mathcal{E}}$.

Definition 6.10. *Given two transition systems $(S_1, \longrightarrow_1)$ and $(S_2, \longrightarrow_2)$, a binary relation $R \subseteq S_1 \times S_2$ is a* bisimulation *iff for any $(s_1, s_2) \in R$: (i) if $s_1 \longrightarrow_1 s_1'$, then $s_2 \longrightarrow_2 s_2'$ and $(s_1', s_2') \in R$ for some $s_2' \in S_2$; and (ii) if $s_2 \longrightarrow_2 s_2'$, then $s_1 \longrightarrow_1 s_1'$ and $(s_1', s_2') \in R$ for some $s_1' \in S_1$.*

In single-rate PALS, the natural correspondence *sync* mapping stable asynchronous states to synchronous states defines a *bisimulation* between $ts(\mathcal{E}) = (S^{\mathcal{E}} \times D_i^{\mathcal{E}}, \longrightarrow_{\mathcal{E}})$ corresponding to the synchronous composition $M_{\mathcal{E}}$ and $(Stable(\mathcal{A}(\mathcal{E}, T, \Gamma)), \longrightarrow_{st})$ of stable asynchronous transitions. Recall that the states in $ts(\mathcal{E})$ consist of the value of the "state" of each single machine in $\mathcal{E}$, the values in all the "feedback" wires in $\mathcal{E}$, and the values of the inputs from the environment. The function *sync* maps stable states in $\mathcal{A}(\mathcal{E}, T, \Gamma)$ to states in $ts(\mathcal{E})$ in the obvious way: the values of the input buffers of the PALS wrappers of the stable state give the values in both the feedback wires and the input from the environment in the synchronous system, and the "local states" of the typed machine objects give the values of the states of the machines in $ts(\mathcal{E})$ [138].

For a multirate ensemble $\mathfrak{E}$, the synchronous composition MRSC($\mathfrak{E}$) is $M_{SR(\mathfrak{E})}$, defined by just an ordinary single-rate machine ensemble $SR(\mathfrak{E})$ (see Definitions 6.9). Therefore, by single-rate PALS:

Corollary 6.1. *Given a* multirate *ensemble* $\mathfrak{E}$*, the function sync defines a bisimulation between the transition systems* $(S^{SR(\mathfrak{E})} \times D_i^{SR(\mathfrak{E})}, \longrightarrow_{SR(\mathfrak{E})})$ *and* $(Stable(\mathcal{A}(SR(\mathfrak{E}), T, \Gamma)), \longrightarrow_{st})$*, where* $SR(\mathfrak{E})$ *is a single-rate ensemble.*

In order to obtain a bisimulation between $(S^{SR(\mathfrak{E})} \times D_i^{SR(\mathfrak{E})}, \longrightarrow_{SR(\mathfrak{E})})$ and a transition system corresponding to the multirate asynchronous system $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$, we define the transition system $(Stable(\mathcal{MA}(\mathfrak{E}, T, \Gamma)), \longrightarrow_{st})$ and prove that it is bisimilar to $(Stable(\mathcal{A}(SR(\mathfrak{E}), T, \Gamma)), \longrightarrow_{st})$.

Definition 6.11. *A state* $\{conf \ ; \ t\}$ *reachable from an admissible initial state (having* consistent *clocks and timers, formally defined in Appendix B.1) in the asynchronous system* $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ *is called* stable *iff:*

- *all the input buffers of the PALS wrapper objects in conf are full;*
- *all other input buffers and output buffers are empty;*
- *there are no (delayed or undelayed) messages in transit between the components in the networks in conf, and*
- *there are no messages waiting to be transmitted between the different layers of an object in conf.*

We call conf a stable configuration *if* $\{conf \ ; \ t\}$ *is a stable state. The set* $Stable(\mathcal{MA}(\mathfrak{E}, T, \Gamma))$ *denotes the set of all stable states in* $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$*.*

This definition is similar to that of the stable states $\mathit{Stable}(\mathcal{A}(\mathcal{E}, T, \Gamma))$ of an asynchronous single-rate system $\mathcal{A}(\mathcal{E}, T, \Gamma)$ in [138], except that those models are not layered. A stable state is a state of the asynchronous system $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ when a new "global" round of the system begins (by a "global" or "slow" round of the system we mean a round with period T, where T is the period of the slowest components in the ensemble). This definition also implies that a fast component with rate k will be in a state where it has finished all k "internal" fast rounds during a global period.

Intuitively, a transition in the synchronous composition MRSC($\mathfrak{E}$) of a multirate ensemble $\mathfrak{E}$ corresponds to a sequence of asynchronous transitions between two stable states in $\mathit{Stable}(\mathcal{MA}(\mathfrak{E}, T, \Gamma))$. However, due to time ticks, there could be a rewrite sequence from one stable state to another (similar) stable state that does not correspond to a synchronous transition in MRSC($\mathfrak{E}$). Stable transitions are therefore defined as follows:

Definition 6.12. *The "big step" transition system of a multirate ensemble* $\mathfrak{E}$ *is defined by:* $\mathit{ts}(\mathit{Stable}(\mathcal{MA}(\mathfrak{E}, T, \Gamma))) = (\mathit{Stable}(\mathcal{MA}(\mathfrak{E}, T, \Gamma)), \longrightarrow_{st})$*, where* $\{C \,;\, t\} \longrightarrow_{st} \{C' \,;\, t'\}$ *holds iff there exists a sequence of one-step rewrites* $\{C_1 \,;\, t_1\} \longrightarrow^1_{\mathcal{R}} \cdots \longrightarrow^1_{\mathcal{R}} \{C_k \,;\, t_k\}$ *such that:*

- $\{C \,;\, t\} = \{C_1 \,;\, t_1\}$ *and* $\{C' \,;\, t'\} = \{C_k \,;\, t_k\}$ *are stable states,*
- *the rewrite sequence contains at least one application of a rule for a "typed machine transition," and*
- *if* C_j *is* not *a stable state, for* $1 < j < k$*, then there exists* no *stable state* C_l *for* $j < l < k$ *(between* C_j *and* C_k *in the sequence).*

As mentioned in Section 6.3.1, the asynchronous systems $\mathcal{A}(SR(\mathfrak{E}), T, \Gamma)$ and $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ do not have the same behaviors. In particular, since a fast component f in $\mathcal{A}(SR(\mathfrak{E}), T, \Gamma)$[9] can finish all of its k fast transitions before messages are sent, whereas the same component in $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ is only guaranteed to finish $k'_f \leq k$ such transitions before the output tuple resulting from those transitions must be sent out. Since such outputs to the other (non-environment) components appear in (the feedback wire part of) the next state in $M_{SR(\mathfrak{E})}$ (and inputs in the input buffers in the next stable state of $\mathcal{A}(SR(\mathfrak{E}), T, \Gamma)$), the stable states in $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ and $\mathcal{A}(SR(\mathfrak{E}), T, \Gamma)$ therefore in general do not coincide.

[9] $\mathcal{A}(\mathcal{E}, T, \Gamma)$ is formally defined as (non-layered) single-rate asynchronous systems in [138]. However, we can also regard $\mathcal{A}(\mathcal{E}, T, \Gamma)$ as the multirate system $\mathcal{MA}(mr(\mathcal{E}), T, \Gamma)$, where $mr(\mathcal{E})$ is the obvious multirate system corresponding to $\mathcal{E}$ with no fast components.

However, since the $(k'+1)$th to the kth data in such k-tuples do not matter in the next round, due to the assumed $(k'+1)$-obliviousness of the input adaptors, we can relate stable states relative to $(k'+1)$-obliviousness:

Definition 6.13. *For any $f \in J_F$, k'_f the cutoff point of f defined above, and data values $d_1, \ldots, d_{rate(f)}, d'_{k'_f+1}, \ldots, d'_{rate(f)}$, two messages*

$$m_1 = \texttt{(to } s \texttt{ from } f \texttt{ (} p\texttt{,}(d_1, \ldots, d_{k'_f}, d_{k'_f+1}, \ldots, d_{rate(f)})\texttt{))}$$
$$m_2 = \texttt{(to } s \texttt{ from } f \texttt{ (} p\texttt{,}(d_1, \ldots, d_{k'_f}, d'_{k'_f+1}, \ldots, d'_{rate(f)})\texttt{))}$$

are equivalent up to $k'+1$-obliviousness, denoted by $m_1 \equiv_{obl} m_2$. We can also extend $\equiv_{obl}$ to sets *of messages in the obvious way: two sets of messages $msgs_1$ and $msgs_2$ are $\equiv_{obl}$-equivalent if $msgs_2$ can be obtained from $msgs_1$ by replacing each message m in $msgs_1$ with a $\equiv_{obl}$-equivalent message m'.*

Then, the following lemma follows immediately from the definition of $(k'+1)$-oblivious input adaptors:

Lemma 6.1. *If the adaptor $adap(j)$ for component j is $(k'+1)$-oblivious, then $adap(j)(\vec{d_1}, \ldots, \vec{d_{n_j}}) = adap(j)(\vec{d'_1}, \ldots, \vec{d'_{n_j}})$, for any two (complete sets of) $\equiv_{obl}$-equivalent sets of messages for component j.*

Two stable states can therefore be considered to be "equivalent" if they only differ in $\equiv_{obl}$-equivalent inputs:

Definition 6.14. *Let $\sim_{obl} \subseteq Stable(\mathcal{MA}(\mathfrak{E}, T, \Gamma)) \times Stable(\mathcal{A}(SR(\mathfrak{E}), T, \Gamma))$ relate two stable states by $s_{ma} \sim_{obl} s_a$ iff: (i) their corresponding input buffers are the same up to $\equiv_{obl}$-equivalence of sets messages, and (ii) for each component c in $\mathfrak{E}$, the value of the `state` attribute of c in s_{ma} is the same as the value of the `state` atribute of c in s_a.*

We now introduce the following notation to denote asynchronous objects in $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ used for brevity in this proof:

- $PALS_{T,\Gamma}(\alpha(M))$ denotes a layered object for a flat *slow* component (illustrated in Figure 6.8): i.e., a PALS wrapper object with period T, which encloses an input adaptor object for adaptor α, which encloses a typed machine object modeling the behaviors of machine M.
- $PALS_{T,\Gamma}(\alpha(k(M)))$ denotes a layered object for a flat *fast* component (illustrated in Figure 6.8): i.e., the PALS wrapper object with period T, which encloses an input adaptor object for input adaptor α, which again encloses a k-machine object with rate k, which finally encloses a typed machine object modeling the behaviors of M.

- $PALS_{T,\Gamma}(M)$ denotes the PALS wrapper object that encloses a single typed machine object modeling the behaviors of M. In particular, $PALS_{T,\Gamma}(E)$ models the behaviors of an environment E, when the environment is considered to be an ordinary typed machine.

Then, the asynchronous system $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ and its correlated single-rate asynchronous system $\mathcal{A}(SR(\mathfrak{E}), T, \Gamma)$ of a flat multirate machine ensemble $\mathfrak{E} = (J_S, J_F, e, \{M_l\}_{l \in J_S \cup J_F}, E, src, rate, adap)$ can be represented by:

$$\begin{aligned}
\mathcal{MA}(\mathfrak{E}, T, \Gamma) = \; & \{PALS_{T,\Gamma}(adap(j)(rate(j)(M_j))) \mid j \in J_F\} \cup \\
& \{PALS_{T,\Gamma}(adap(j)(M_j)) \mid j \in J_S\} \;\cup\; \{PALS_{T,\Gamma}(E)\}, \\
\mathcal{A}(SR(\mathfrak{E}), T, \Gamma) = \; & \{PALS_{T,\Gamma}((M_j^{\times rate(j)})_{adap(j)}) \mid j \in J_F\} \cup \\
& \{PALS_{T,\Gamma}((M_j)_{adap(j)}) \mid j \in J_S\} \;\cup\; \{PALS_{T,\Gamma}(E)\}.
\end{aligned}$$

Notice that we have the same asynchronous "components" in $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ and $\mathcal{A}(SR(\mathfrak{E}), T, \Gamma)$, but they are implemented differently.

In $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$, components can be considered in isolation during a single (slow) round. The next state of a component is only determined by its own current state and the inputs in the input buffer of its PALS wrapper at the start of the current (slow) round. The PALS wrapper also operates at the slow rate, and only transmits inputs to the inner layers once in each round (at the start of the slow round). Therefore, we can consider a *local asynchronous system* corresponding to only a single component in $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$.

Definition 6.15. *A* local asynchronous system *of $M = (D_i, S, D_o, \delta_M)$ is defined as follows, where $E_M = (D_o, D_i)$ is its* local environment*:*

- $\mathcal{MA}(\alpha(M), T, \Gamma) = \{PALS_{T,\Gamma}(\alpha(M)),\ PALS_{T,\Gamma}(E_M)\}$ *for a flat slow component $PALS_{T,\Gamma}(\alpha(M))$;*

- $\mathcal{MA}(\alpha(k(M)), T, \Gamma) = \{PALS_{T,\Gamma}(\alpha(k(M))),\ PALS_{T,\Gamma}(E_M)\}$ *for a flat fast component $PALS_{T,\Gamma}(\alpha(k(M)))$; and*

- $\mathcal{MA}(M, T, \Gamma) = \mathcal{A}(M, T, \Gamma) = \{PALS_{T,\Gamma}(M),\ PALS_{T,\Gamma}(E_M)\}$ *for a non-layered component $PALS_{T,\Gamma}(M)$.*

We can furthermore "locally" relate two asynchronous components with the same period T and performance bounds Γ as follows:

Definition 6.16. *Two components $PALS_{T,\Gamma}(c_1)$ and $PALS_{T,\Gamma}(c_1)$ with the same local environment E_M are* behaviorally $\sim_{obl}$-equivalent *iff $\sim_{obl}$ is a bisimulation between $ts(Stable(\mathcal{MA}(c_1, T, \Gamma)))$ and $ts(Stable(\mathcal{MA}(c_2, T, \Gamma)))$.*

The following important *decomposition lemma* allows us to analyze each component in isolation for showing that $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ and $\mathcal{A}(SR(\mathfrak{E}), T, \Gamma)$ are behaviorally $\sim_{obl}$-equivalent to each other.

Lemma 6.2. *If each component in $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ is $\sim_{obl}$-equivalent to the corresponding component in $\mathcal{A}(SR(\mathfrak{E}), T, \Gamma)$, then $\sim_{obl}$ is also a bisimulation between $ts(Stable(\mathcal{MA}(\mathfrak{E}, T, \Gamma)))$ and $ts(Stable(\mathcal{A}(SR(\mathfrak{E}), T, \Gamma)))$.*

Proof. Let $s_{ma} \in Stable(\mathcal{MA}(\mathfrak{E}, T, \Gamma))$ and $s_a \in Stable(\mathcal{A}(SR(\mathfrak{E}), T, \Gamma))$ be stable states of $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ and $\mathcal{A}(SR(\mathfrak{E}), T, \Gamma)$, respectively, such that $s_{ma} \sim_{obl} s_a$. Suppose that $s_{ma} \longrightarrow_{st} s'_{ma}$; i.e., for each component $c_{ma,j}$:

1. it performs a transition based on the input messages $msgs_{ma,j}$ in the input buffer of its PALS wrapper,
2. it sends output messages $msgs'_{ma,j}$ into the network through the PALS wrapper (that will arrive at the designated input buffers before the next stable state, according to results of single-rate PALS [138]), and
3. meanwhile, it receives new input messages $msgs''_{ma,j}$ that will be used in the next stable transition from s'_{ma}.

Let $O_{ma,j} \in s_{ma}$ and $O'_{ma,j} \in s'_{ma}$ be the PALS objects for each component $c_{ma,j}$ in the stable state s_{ma} and s'_{ma}, respectively. Now, consider a stable state $\{O_{ma,j}\ O_{E_j}\ ;\ t_j\} \in Stable(\mathcal{MA}(c_{ma,j}, T, \Gamma))$ of the local asynchronous system $\mathcal{MA}(c_{ma,j}, T, \Gamma)$. Since O_{E_j} can generate outputs $msgs''_{ma,j}$, there exists a local stable transition $\{O_{ma,j}\ O_{E_j}\ ;\ t_j\} \longrightarrow_{st} \{O'_{ma,j}\ O'_{E_j}\ ;\ t'_j\}$ in which O'_{E_j} contains the output messages $msgs'_{ma,j}$ in its input buffer of the PALS wrapper, generated from $c_{ma,j}$.

Let $O_{a,j} \in s_a$ be the PALS object for the corresponding component $c_{a,j}$ in $\mathcal{A}(SR(\mathfrak{E}), T, \Gamma)$ in the stable state s_a. Since $s_{ma} \sim_{obl} s_a$, by definition, $\{O_{ma,j}\ O_{E_j}\ ;\ t_j\} \sim_{obl} \{O_{a,j}\ O_{E_j}\ ;\ \tilde{t}_j\}$ for any stable state $\{O_{a,j}\ O_{E_j}\ ;\ \tilde{t}_j\}$ in $Stable(\mathcal{A}(c_{a,j}, T, \Gamma))$. Because $c_{ma,j}$ is behaviorally $\sim_{obl}$-equivalent to $c_{a,j}$, there exists a stable transition $\{O_{a,j}\ O_{E_j}\ ;\ \tilde{t}_j\} \longrightarrow_{st} \{O'_{a,j}\ \tilde{O}'_{E_j}\ ;\ \tilde{t}'_j\}$ such that $\{O'_{ma,j}\ O'_{E_j}\ ;\ t'_j\} \sim_{obl} \{O'_{a,j}\ \tilde{O}'_{E_j}\ ;\ \tilde{t}'_j\}$. Consider a stable configuration $C'_a = \{O'_{a,j} \mid c_{a,j}$ is a component in $\mathcal{A}(SR(\mathfrak{E}), T, \Gamma)\}$. We can easily see that there exists a stable transition[10] $s_a \longrightarrow_{st} \{C'_a\ ;\ t'\}$ in $\mathcal{A}(SR(\mathfrak{E}), T, \Gamma)$ such that $s'_{ma} \sim_{obl} \{C'_a ; t'\}$. Consequently, $s_{ma} \sim_{obl} s_a$ and $s_{ma} \longrightarrow_{st} s'_{ma}$ implies that $s_a \longrightarrow_{st} s'_a$ and $s'_{ma} \sim_{obl} s'_a$ for some $s'_a \in Stable(\mathcal{A}(SR(\mathfrak{E}), T, \Gamma))$. The "vice versa" direction is entirely similar. □

[10] Since $O'_{E_j} \sim_{obl} \tilde{O}'_{E_j}$ for each j, the outputs of each component in $\mathcal{A}(SR(\mathfrak{E}), T, \Gamma)$ are pairwise $\equiv_{obl}$-equivalent. Thus, in the next stable state, these outputs have reached the input buffers of the PALS wrappers, which will also be pairwise $\equiv_{obl}$-equivalent.

The following two lemmas state that for each component $j \in J_S \cup J_F$ of a flat multirate machine ensemble $\mathfrak{E}$, the single-rate object in $\mathcal{A}(SR(\mathfrak{E}), T, \Gamma)$ is behaviorally $\sim_{obl}$-equivalent to the multirate object in $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ (the proofs of the lemmas are given in Appendix B.2).

Lemma 6.3. *For a slow machine M_j in a flat multirate machine ensemble $\mathfrak{E}$, $j \in J_S$, the correlated local asynchronous systems $\mathcal{MA}(adap(j)(M_j), T, \Gamma)$ and $\mathcal{A}((M_j)_{adap(j)}, T, \Gamma)$ are behaviorally $\sim_{obl}$-equivalent.*

Lemma 6.4. *For a fast machine M_f in a flat multirate machine ensemble $\mathfrak{E}$, $f \in J_F$, the local asynchronous systems $\mathcal{MA}(adap(f)(rate(f)(M_f)), T, \Gamma)$ and $\mathcal{A}((M_f^{\times rate(f)})_{adap(f)}, T, \Gamma)$ are behaviorally $\sim_{obl}$-equivalent.*

Consequently, $\sim_{obl}$ is a bisimulation between $ts(Stable(\mathcal{MA}(\mathfrak{E}, T, \Gamma)))$ and $ts(Stable(\mathcal{A}(SR(\mathfrak{E}), T, \Gamma)))$ by Lemma 6.2. Because *sync* is a bisimulation between $ts(Stable(\mathcal{A}(SR(\mathfrak{E}), T, \Gamma)))$ and $ts(SR(\mathfrak{E}))$ (by Corollary 6.1), the following theorem follows from the fact that bisimulations compose:

Theorem 6.1. *For a flat multirate ensemble $\mathfrak{E}$, the relation $\sim_{obl} ; sync$ is a bisimulation between $ts(Stable(\mathcal{MA}(\mathfrak{E}, T, \Gamma)))$ and $ts(SR(\mathfrak{E}))$.*

Notice that by the above theorem, $\sim_{obl} ; sync$ is a bisimulation between $ts(Stable(\mathcal{MA}(\mathfrak{E}, T, \Gamma)))$ and $ts(Stable(\mathcal{A}(M_{SR(\mathfrak{E})}, T, \Gamma)))$ as well. Therefore, the bisimilarity for a hierarchical ensemble can be proved by induction on depth d, using the flat case as a base case (see Appendix B.2).

This correspondence can be lifted to temporal logic properties. For a set AP of atomic propositions, a labeling function $L : S^{SR(\mathfrak{E})} \times D_i^{SR(\mathfrak{E})} \to 2^{AP}$ defines which predicates hold in a state of $ts(SR(\mathfrak{E}))$, giving rise to a Kripke structure $\mathcal{K}(SR(\mathfrak{E})) = (S^{SR(\mathfrak{E})} \times D_i^{SR(\mathfrak{E})}, AP, L, \longrightarrow_{SR(\mathfrak{E})})$. This labeling function can be extended to $(sync; L) : Stable(\mathcal{MA}(\mathfrak{E}, T, \Gamma)) \to 2^{AP}$, yielding the corresponding Kripke structure

$$\mathcal{K}(Stable(\mathcal{MA}(\mathfrak{E}, T, \Gamma))) = (Stable(\mathcal{MA}(\mathfrak{E}, T, \Gamma)), AP, (sync\ ;\ L), \longrightarrow_{st}).$$

However, $ts(SR(\mathfrak{E}))$ and $\mathcal{K}(Stable(\mathcal{MA}(\mathfrak{E}, T, \Gamma)))$ may not satisfy the same temporal logic properties for *sync*-related initial states in general, since the input components in the feedback wires in the former may have "complete" k-tuples, whereas the latter may have some different "don't care" value $\bot$ in $(k'+1)$-oblivious inputs. As already mentioned, since the machines cannot see the difference between such $\equiv_{obl}$-equivalent inputs once their input adaptors have been applied to the inputs, we require that the atomic propositions cannot distinguish between two states that only differ in their $\equiv_{obl}$-equivalent feedback inputs. Formally:

Definition 6.17. *A labeling function* $L : S^{SR(\mathfrak{E})} \times D_i^{SR(\mathfrak{E})} \to 2^{AP}$ *cannot distinguish between* $\equiv_{obl}$*-equivalent inputs iff the condition*

$$\begin{aligned} & L\Big((\vec{s},\ \{\vec{d_j}\}_{j\in J_S} \cup \{(\mathbf{d}_{f_1},\dots,\mathbf{d}_{f_{m_f}})\}_{f\in J_F}),\ \ \vec{d_i}\Big) \\ = \quad & L\Big((\vec{s},\ \{\vec{d_j}\}_{j\in J_S} \cup \{(\mathbf{d}'_{f_1},\dots,\mathbf{d}'_{f_{m_f}})\}_{f\in J_F}),\ \ \vec{d_i}\Big), \end{aligned}$$

holds, where the feedback output $(\mathbf{d}_{f_1},\dots,\mathbf{d}_{f_{m_f}})$ *denotes the feedback output from the fast machine with index* f*, whenever each* $\mathbf{d}_{f_l}$*,* $1 \le l \le m_f$*, is equivalent to* $\mathbf{d}'_{f_l}$ *up to* (k'_f+1)*-obliviousness for* k'_f *its output cutoff number, that is, for some values* $d_{f_{l_1}},\dots,d_{f_{l_{rate(f)}}},d'_{f_{l_{k'_f+1}}},\dots,d'_{f_{l_{rate(f)}}}$*:*

$$\begin{aligned} \mathbf{d}_{f_l} &= (d_{f_{l_1}},\dots,d_{f_{l_{k'_f}}},d_{f_{l_{k'_f+1}}},\dots,d_{f_{l_{rate(f)}}}) \\ \mathbf{d}'_{f_l} &= (d_{f_{l_1}},\dots,d_{f_{l_{k'_f}}},d'_{f_{l_{k'_f+1}}},\dots,d'_{f_{l_{rate(f)}}}). \end{aligned}$$

Since this requirement is equivalent to requiring that $s \sim_{obl} s'$ implies $(sync;\, L)(s) = (sync;\, L)(s')$, it follows that bisimilar states satisfy the same atomic propositions. Since it is well-known that bisimilar Kripke structures satisfy the same CTL* formulas (see, e.g., [60]), we obtain the following:

Theorem 6.2. *If* L *cannot distinguish between* $\equiv_{obl}$*-equivalent inputs, then for any formula* $\varphi \in \mathrm{CTL}^*(AP)$ *and for any stable initial configuration* $\{C_0\ ;\ t_0\}$ *of the form described in Definition B.1, we have*

$$\begin{aligned} & \mathcal{K}(Stable(\mathcal{MA}(\mathfrak{E},T,\Gamma))),\{C_0\ ;\ t_0\} \models \varphi \\ \iff \quad & \mathcal{K}(SR(\mathfrak{E})), sync(\{C_0\ ;\ t_0\}) \models \varphi \end{aligned}$$

6.4 Multirate PALS Methodology

As mentioned in the introduction, many distributed cyber-physical systems, including the airplane control system in Section 6.5, interact with *physical* entities exhibiting continuous dynamics, and are therefore *distributed hybrid systems.* In Multirate PALS, any "local" environment of faster components is assumed to have been incorporated into the corresponding typed machine itself, as explained in Section 6.2.3. This section shows how such physical environments can be integrated into their controlling typed machines within Multirate PALS, and then presents a generic framework for both *specifying and executing* a multirate ensemble in Real-Time Maude.

6.4.1 Physical Environments

We can reasonably assume that a controller component is tightly integrated with its physical environment, and contains the sensors and actuators. Even if the sensors and actuators are *remote* (instead of *included* in the controller) and therefore there also exists a network delay between sensor/actuator and controller, the system can still be modeled by having another typed machine, containing the sensors and actuators and tightly integrated with its physical environment, that interacts with the controller like any other machine.

Our methodology can be summarized as follows. First, a (local) controller component M is represented by a *nondeterministic typed machine* that is parameterized by different behaviors of environments; that is, the behavior of M is defined for *any possible* environment behavior. Second, its physical environment E_M defines the continuous behaviors of a *certain* environment, and also defines how the environment E_M reacts to actuator commands from the controller component M. Then, the *environment restriction* of M by E_M defines an ordinary typed machine $M \upharpoonright E_M$ whose transitions are constrained by the behavior of the specified environment E_M.

The state of a physical environment E_M at a certain point can generally be represented as a tuple of the values $(v_1, v_2, \ldots, v_l) \in \mathbb{R}^l$ of the physical parameters $x_1, \ldots, x_l$ of interest. For example, the Newtonian dynamics of a single object involves the parameters $(x, y, z, t) \in \mathbb{R}^4$ for a position (x, y, z) of the object at time t. A physical environment E_M is called *l-dimensional* iff its state space is a subset of $\mathbb{R}^l$. As usual in physical dynamics, the behavior of a physical environment E_M can be expressed by differential equations and continuous functions involving its parameters $x_1, \ldots, x_l$, which indeed specify *trajectories* of the parameters $x_1, \ldots, x_l$.

Definition 6.18. *A* trajectory *of duration T is a function $\tau : [0, T] \to \mathbb{R}$. The set $\mathcal{T}_T$ denotes the set of all trajectories of duration $T \in \mathbb{R}$. For an l-tuple of trajectories $\vec{\tau} = (\tau_1, \ldots, \tau_l) \in \mathcal{T}_T^l$, let $\vec{\tau}(x) = (\tau_1(x), \ldots, \tau_l(x))$.*

That is, a trajectory $\tau : [0, T] \to \mathbb{R}$ defines the continuous behavior of a physical parameter in a period of duration T. Such a trajectory is normally a continuous function, but can also be more general. We refer to [128] for more details about general trajectories for hybrid systems.

A machine M collects the state $(v_1, \ldots, v_l)$ of its physical environment E_M using its sensors, and affects the physical environment E_M through its actuators. Since we consider "periodic" digital components with period T, E_M can be modeled as a *periodic dynamic system* that specifies any possible trajectories of its physical state during its period T.

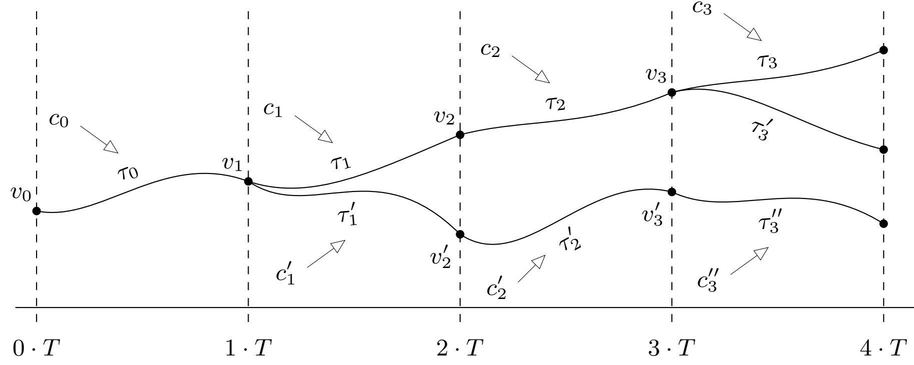


Figure 6.10: A periodic dynamic system in which its physical state follows the continuous trajectory τ_i from the value v_i to v_{i+1} according to the control command c_i from the controller during each period; e.g., $((c_0, v_0), \tau_0) \in \Lambda$, $((c_1, v_1), \tau_1) \in \Lambda$, and $((c_1', v_1), \tau_1') \in \Lambda$.

Definition 6.19. *An l-dimensional* periodic dynamic system *is a tuple $E_M = (C,\ P,\ T,\ \Lambda)$ where:*

- *C is a set of* control commands*, representing "actuator outputs" from the corresponding controller;*
- *$P \subseteq \mathbb{R}^l$ is a set of all possible values of the "physical parameters" $x_1, \ldots, x_l$ of E_M;*
- *$T \in \mathbb{R}_{>0}$ is the* period *of E_M;*
- *$\Lambda \subseteq (C \times P) \times \mathcal{T}_T^l$ is a total* physical transition relation *that defines possible trajectories $\vec{\tau} \in \mathcal{T}_T^l$ of duration T for each control command $c \in C$ beginning at each physical state $\vec{v} \in P$, satisfying:*

$$((c, \vec{v}), \vec{\tau}) \in \Lambda \implies \vec{\tau}(0) = \vec{v} \ \wedge \ (\forall t \in [0, T])\ \vec{\tau}(t) \in P$$

That is, if the current physical state of the environment E_M is $\vec{v}$ at the beginning of a period and its controller M gives an actuator output c to E_M, then the physical state of E_M follows the continuous trajectory $\vec{\tau}$ during period T, as illustrated in Figure 6.10.

Then, a generic discrete controller M for the periodic dynamic system $E_M = (C, P, T, \Lambda)$ is specified as just an ordinary *nondeterministic* typed machine $M = (D_i, S, D_o, \delta_M)$ with period T, where the physical parameters of E_M are also included in M's state, and the control commands to E_M (i.e., actuator outputs) depend (only) on the current state of M. Such a relationship between M and E_M is captured by two projection functions:

Definition 6.20. *Given a machine* $M = (D_i, S, D_o, \delta_M)$ *and a periodic dynamic system* $E_M = (C, P, T, \Lambda)$*:*

- *A* command function $\pi_C : S \to C$ *determines the control command to the physical environment* E_M*.*
- *A* environment state function $\pi_P : S \to P$ *represents the* observed *environment state.*

That is, the controller machine M is defined as a highly nondeterministic machine which takes *all possible* environment behaviors into account. Then, the environment restriction $M \upharpoonright E_M$ (formally defined below) restricts the behaviors of M to those that "fit" with the environment E_M.

The same discrete controller can be placed in many different physical environments that share the same physical parameters but show different behaviors. The behavior of $M = (D_i, S, D_o, \delta_M)$ is *nondeterministically* defined for any possible behaviors of the physical parameters as follows:

- At the start of each period, M is in state $s \in S$, which determines the control command $\pi_C(s)$ to its physical environment, and the physical state $\pi_P(s)$ at the beginning of the current round.
- The actuator output $\pi_C(s)$ takes effect on its environment from the beginning of the current round and lasts until the end of the round.
- δ_M *nondeterministically* decides the output $\vec{d_o} \in D_o$ and the next state $s' \in S$, based on the current state $s \in S$ and the input $\vec{d_i} \in D_i$.
- The next state s' also determines the physical state $\pi_P(s')$ and the control command $\pi_C(s')$ for the beginning of the *next* period. That is, δ_M also updates the physical parameters to their (expected) values at the end of the current period/beginning of the next period.
- $\pi_P(s')$ gives the values of the physical parameters at the beginning of the next round, while the physical values (continuously) change from the current physical state $\pi_P(s)$, according the control command $\pi_C(s)$ until the beginning of the next round.
- However, since the behavior of those physical parameters has not been specified *yet*, the transition relation δ_M assumes that the values of the physical parameters in the current state s can be changed to any real numbers in the next state s', and assigns such *nondeterministically chosen* values to the physical parameters in the next state s'.

When the behavior of those physical parameters in machine M is specified by a periodic dynamic system $E_M = (C, P, T, \Lambda)$, the transition relation $\delta_{M \upharpoonright E_M}$ of the restricted machine $M \upharpoonright E_M$ is a *subset* of δ_M that also follows the *physical constraints* Λ given by E_M. That is, $M \upharpoonright E_M$ captures each *observable* moment of the E_M's continuous behavior for M.

Definition 6.21. *Given a typed machine $M = (D_i, S, D_o, \delta_M)$, a physical environment $E_M = (C, P, T, \Lambda)$, a control command function $\pi_C : S \to C$, and an environment state function $\pi_P : S \to P$, the* environment restriction *is the typed machine $M \upharpoonright E_M = (D_i, S, D_o, \delta_{M \upharpoonright E_M})$ such that:*

$$((\vec{d_i}, s), (s', \vec{d_o})) \in \delta_{M \upharpoonright E_M}$$

iff $((\vec{d_i}, s), (s', \vec{d_o})) \in \delta_M$ holds and there exists a trajectory $\vec{\tau} \in \mathcal{T}_T^l$ such that

$$((\pi_C(s), \pi_P(s)), \vec{\tau}) \in \Lambda \ \wedge \ \vec{\tau}(0) = \pi_P(s) \ \wedge \ \vec{\tau}(T) = \pi_P(s').$$

That is, for the control command $\pi_C(s)$, the next physical state $\pi_P(s')$ must be reachable *from the current physical state $\pi_P(s)$ by a* valid trajectory *$\vec{\tau}$ of the physical environment E_M.*

Notice that M can be considered to be a typed machine *parameterized* by different behaviors of physical parameters, and its *environment restriction* $M \upharpoonright E_M$ defines the actual behavior of M when it is placed in the particular physical environment specified by the periodic dynamic system E_M.

We model a distributed cyber-physical system as a multirate machine ensemble $\mathfrak{E}$, where each machine is regarded as the environment restriction $M \upharpoonright E_M$ of a controller typed machine M by its physical environment E_M. Therefore, the Multirate PALS transformation can in principle generate a correct-by-construction distributed asynchronous model $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ from the synchronous model $\mathfrak{E}$. However, there are two "hybrid system" problems that Multirate PALS cannot handle at the moment:

- The physical environments of different distributed components in the system can be *physically correlated* to each other. For example, if we consider two adjacent rooms, the temperature of one room can immediately affect the temperature of the other room.

- In the asynchronous model $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ different components read their sensor values at slightly different times, due to the clock skews. The "continuous behavior" of $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ can therefore be slightly different from the synchronous model $\mathfrak{E}$.

Providing general solutions for those challenges remains a topic for future work. We therefore assume that any “physical correspondences” between any different physical environments are faithfully captured by the ensemble connections in $\mathfrak{E}$. Also, we assume that the system is *stable* in the sense that small differences in the “sensor/effector timing” caused by the clock skews do not affect the correctness of the system.

Example 6.1 (Digital Thermostat)**.** *Figure 6.11 shows a digital thermostat controller, operating at a certain frequency to control the temperature of a room, specified by the typed machine*

$$M = (\mathbb{R}^2,\ \{s_{on}, s_{off}\} \times \mathbb{R},\ \{*\},\ \delta_M)$$

where each state $(s, x) \in \{s_{on}, s_{off}\} \times \mathbb{R}$ *contains the “current” temperature* x *of the room (i.e., the temperature at the beginning or at the end of the current period),* $\{*\}$ *is a singleton set to denote that there is no output port, and for any* $x' \in \mathbb{R}$*, the transition relation* δ_M *is defined by:*

$$((t_{\max}, t_{\min}), (s_{on}, x)), ((\textbf{if } x \le t_{\max} \textbf{ then } s_{on} \textbf{ else } s_{off} \textbf{ fi},\ x'), *)) \in \delta_M$$
$$((t_{\max}, t_{\min}), (s_{off}, x)), ((\textbf{if } x < t_{\min} \textbf{ then } s_{on} \textbf{ else } s_{off} \textbf{ fi},\ x'), *))\ \in \delta_M.$$

At each step, M *receives two inputs* $(t_{\max}, t_{\min}) \in \mathbb{R}^2$ *from other typed machines, where* $t_{\max}$ *is the desired maximum temperature and* $t_{\min}$ *is the desired minimum temperature. Based on these inputs and its current state* (s, x)*, the machine makes a transition to the next state* (s', x')*, where* $x' \in \mathbb{R}$ *can be any value since the behavior of the physical parameter* x *has not been specified* yet*. During its period until the next step,* M *gives a command* `out` *to turn the heater* on/off*, according to its state* s*.*

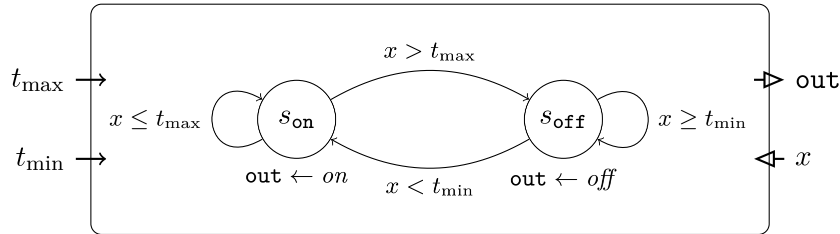


Figure 6.11: A digital thermostat controller modeled as a typed machine with two input ports (for the maximum temperature $t_{\max}$ and the minimum temperature $t_{\min}$) and no output ports. Its physical environment is modeled by one environment parameter (the current temperature x), and one control command (the switch of the heater `out`).

The physical environment E_M of the nondeterministic *controller M is specified by the periodic dynamic system $E_M = (\{on, off\},\ \mathbb{R},\ T,\ \Lambda)$ with period T, control commands $\{on, off\}$, $\mathbb{R}$ for the current temperature x, and the physical transition relation $\Lambda \subseteq (\{on, off\} \times \mathbb{R}) \times \mathcal{T}_T$ given by:*

$$
\begin{aligned}
&((\mathbf{out}, v), x)) \in \Lambda \\
\iff\ &(\exists x : [0,T] \to \mathbb{R})\ \ x(0) = v \ \wedge\ \dot{x} = \begin{cases} K(h-x) & \text{if } \mathbf{out} = on \\ -Kx & \text{if } \mathbf{out} = off, \end{cases}
\end{aligned}
$$

where $K, h \in \mathbb{R}$ are constants depending on the size of the room and the power of the heater, respectively. That is, if the heater is on, the temperature x rises according to the differential equation $\dot{x} = K(h-x)$, and if the heater is off, the temperature x falls according to the differential equation $\dot{x} = -Kx$.

The physical environment E_M conforms to the controller machine M with the environment projection functions $\pi_C : (\{s_{on}, s_{off}\} \times \mathbb{R}) \to \{on, off\}$ and $\pi_P : (\{s_{on}, s_{off}\} \times \mathbb{R}) \to \mathbb{R}$ where

$$\pi_C(s_{on}, x) = on, \qquad \pi_C(s_{off}, x) = off, \qquad \pi_P(s, x) = x.$$

The environment restriction is $M \upharpoonright E_M = (\mathbb{R}^2, \{s_{on}, s_{off}\} \times \mathbb{R}, \{*\}, \delta_{M \upharpoonright E_M})$, where $((t_{\max}, t_{\min}), (s, x)), ((s', x'), *)) \in \delta_{M \upharpoonright E_M}$ iff

$$
\begin{aligned}
&((t_{\max}, t_{\min}), (s, x)), ((s', x'), *)) \in \delta_M \\
\wedge\ &\exists \tau \in \mathcal{T}_T.\ ((\pi_C(s), x), \tau) \in \Lambda \ \wedge\ \tau(0) = x \ \wedge\ \tau(T) = x'.
\end{aligned}
$$

The combined thermostat behavior, specified by the environment restriction $M \upharpoonright E_M$, is illustrated in Figure 6.12.

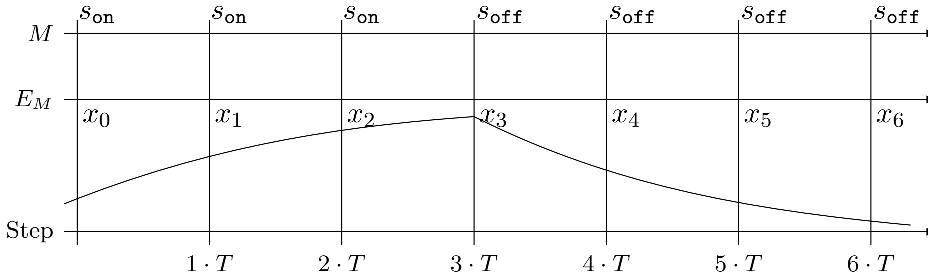


Figure 6.12: The behavior of the digital thermostat controller M integrated with its physical environment E_M, where x_{i+1} is the next temperature after period T elapsed from x_i according to the given differential equation.

6.4.2 The Real-Time Maude Framework

This section presents a generic framework for specifying and executing a *hierarchical* multirate ensemble in Real-Time Maude. Given a specification of typed machines, their periods, input adaptors, and a wiring diagram that defines the connections between output ports and input ports, the framework produces an executable model of the synchronous composition of the ensemble which can be used to simulate and formally verify this synchronous composition. If the system has "local" physical environments, then they should be integrated with the corresponding controller machines, as explained in the previous section.

Representing Multirate Machine Ensembles. A multirate machine ensemble can naturally be specified in an object-oriented style, where the machines and the (sub)ensembles are modeled as objects. That is, the global state has the form `{`*Ensemble*`}` where *Ensemble* is an object representing the entire multirate machine ensemble.

A typed machine is represented as an object instance of a subclass of the class `Component`, which has the attributes `period` and `ports`. The attribute `period` denotes the period of the typed machine, and `ports` contains the input and output "ports," represented as a multiset of `Port` objects.

```
class Component | period : Time,
                  ports : Configuration .
```

We assume that any (input and output) data are terms of a supersort `Data`, which also contains the constant `bot`, denoting the "don't care" value $\perp$ used in the input adaptor functions.

```
sort Data .
op bot : -> Data [ctor] .
```

A port is modeled by an object instance of the class `Port`, whose attribute `content` contains the data content as a list of values. The subclasses `InPort` and `OutPort` denote input and output ports, respectively.

```
class Port | content : List{Data} .
class InPort .
class OutPort .
subclass InPort OutPort < Port .
```

Component and port identifiers are terms of the subsorts ComponentId and PortId, respectively, of the sort Oid of *object identifiers*.

```
sorts ComponentId PortId .
subsorts ComponentId PortId < Oid .
```

To define the transition relation, the user must define the following built-in operator `delta`, for each single (atomic) machine, by means of equations (for deterministic transitions) or rewrite rules (for nondeterministic transitions):

```
op delta : Object ~> Object .
```

The operator `delta` is declared to be a *partial function* using the *kind* [Object]. That is, a term containing the `delta` operator will only have a kind, but *not* a sort. This is used to ensure that a transition equation/rule in a *composition* is only applied when the transitions have been performed in all subcomponents (see below).

A multirate machine ensemble is modeled as an object instance of the class `Ensemble`, where the `connections` attribute denotes the wiring diagram and the `subcomponents` attribute denotes the typed machines in the ensemble. We support the specification of *hierarchical* multirate ensembles by declaring `Ensemble` to be a subclass of `Component`, whose attribute `ports` denotes the ports to its external "environment."

```
class Ensemble | subcomponents : Configuration,
                 connections : Set{Connection} .
subclass Ensemble < Component .
```

The rate of each component in the `subcomponents` attribute is implicitly given by the period of the component. The period of the entire ensemble (in its `period` attribute) must be equal to the period of the slowest components in the ensemble. To define the input adaptor for each input port of a single component, the user must declare the following built-in function `adaptor` by equations, where sort NeList{Data} denotes a non-empty list of data:

```
op adaptor : ComponentId PortId NeList{Data} -> NeList{Data} .
```

A wiring diagram of an ensemble is modeled as a semicolon-separated set of connections. A connection is a term p_i `-->` p_o of sort Connection, where p_i **and** p_o are the source and target *port names*, respectively:

```
sorts Connection PortName .          subsort PortId < PortName .
op _._ : ComponentId PortId -> PortName [ctor] .
op _-->_ : PortName PortName -> Connection [ctor] .
```

A term C_1 `.` P_1 `-->` C_2 `.` P_2 represents a connection from an output port P_1 of a component C_1 to an input port P_2 of a component C_2. A connection between an environment port P and a port P_2 of a subcomponent C_2 is represented as a term P `-->` C_2`.`P_2 (for an environment input) or a term C_2`.`P_2 `->` P (for an environment output), where both sides are input ports or output ports. Section 6.5.3 illustrates how our airplane control system is represented as an ensemble object.

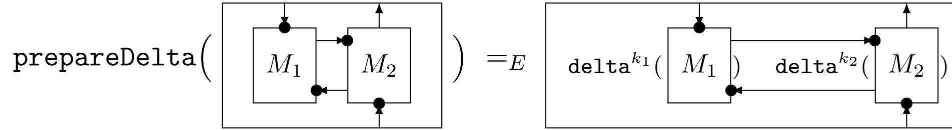


Figure 6.13: The `prepareDelta` function; the operator `delta` is distributed to each subcomponent $i \in \{1, 2\}$ as many times as its rate k_i.

Defining Synchronous Compositions. Because an ensemble object of class `Ensemble` is also an instance of class `Component`, the transitions of the synchronous composition of the ensemble can be defined using the same operator `delta`. The following rewrite rule defines the transition relation of the synchronous composition specified in Definition 6.3:

```
var C : ComponentId .   var OBJ : Object .   var KOBJ : [Object] .

ops transferInputs transferResults : Object -> Object .
op prepareDelta : Object ~> Object .

crl [sync]: delta(< C : Ensemble | >) => transferResults(OBJ)
  if KOBJ := prepareDelta(transferInputs(< C : Ensemble | >))
  /\ KOBJ => OBJ .
```

using the three auxiliary functions `transferInputs`, `prepareDelta`, and `transferResults`. The meaning of the rule can be summarized as follows:

1. Each input port in the ensemble `C` receives a value from its source output port (`transferInputs`).

2. Appropriate input adaptors are applied to each input port, and then the operator `delta` of each subcomponent is applied multiple times according to its rate as illustrated in Figure 6.13 (`prepareDelta`); the resulting term is assigned to the variable `KOBJ` of *kind* [Object].

3. Any term of sort Object resulting from rewriting the term given in `KOBJ` in zero or more steps can be *nondeterministically* assigned to the variable `OBJ`; since the operator `delta` does not yield terms of this sort, these objects are "quiescent" objects where all the operations in a round have been performed. That is, the variable `OBJ` will only capture a term containing *no* `delta` in which all the transition relations for the subcomponents in the ensemble `C` are completely evaluated.

4. The new outputs in the subcomponent objects are transferred to the environment ports (`transferResults`).

When the system is specified by one top-level component for a multirate ensemble of (nondeterministic) typed machines, the dynamics of the system is specified by the following *conditional* tick rewrite rule that models an iteration of the synchronous composition of the ensemble:

```
crl [step]: {< C : Ensemble | period : T >} => {OBJ'} in time T
  if OBJ := clearOutput(< C : Ensemble | >
  /\ delta(OBJ) => OBJ' .
```

For the top-level ensemble `C`, the outputs generated in the previous round is cleared by the `clearOutputs` function and the resulting term is assigned to the variable `OBJ` of sort Object. Then, in a similar way to the `sync` rule above, any possible term of sort Object resulting from rewriting `delta(OBJ)` in zero or more steps can be nondeterministically assigned to the variable `OBJ'` of sort Object where `delta` is completely evaluated by rewrite rules.

Executing Subcomponents using Partial Order Reductions. The transition of the synchronous composition is performed by distributing the operator `delta` to all the subcomponents, and then executing the transitions *concurrently* in the different typed machines. However, this straight-forward solution is computationally expensive, since the Maude engine computes all possible interleavings caused by applying the rewrite rules for `delta` in different orders. It is totally unnecessary to explore all these interleavings caused by executing the transitions concurrently in the different components during one period, since the PALS synchronous semantics ensures that the behaviors of the subcomponents are *independent* of each other during the execution of the transitions in a single round, as illustrated in Figure 6.14. Therefore, we can equally well "schedule" the execution of `delta` to avoid these interleavings, which amounts to *partial order reduction* [60].

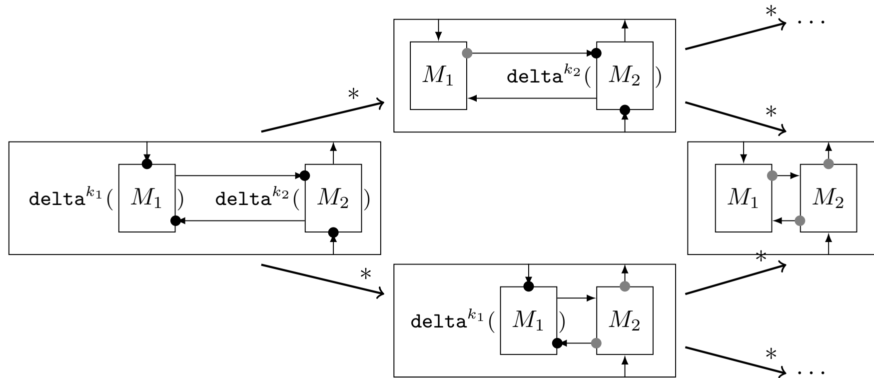


Figure 6.14: The two different execution orders that give the same result.

For a multirate ensemble, we define a scheduling queue that gives a certain execution order to compute the transitions of the subcomponents. It is basically a list of component objects in which only the first component can execute its (nondeterministic) `delta` rewrite rules. We can specify such evaluation strategies for component lists by using the `frozen` attribute of the list cons operator `_::_` as follows:

```
sort ObjectQueue .
op nil : -> ObjectQueue [ctor] .
op _::_ : Object ObjectQueue -> ObjectQueue [ctor frozen(2)] .
```

Since the second argument of the operator `_::_` is declared `frozen`, for any list constructed by the operator, it allows rule rewriting only inside its first item. The point is that we can put the objects in some list

$$object_1 \ :: \ (object_2 \ :: \ (\ldots \ :: \ object_n)\ldots),$$

which means that the rewrite rules are first applied to $object_1$, and only when no rewrite rule can be applied to $object_1$ do we start applying rewrite rules to $object_2$, and so on. If some component is deterministic so that its `delta` is defined using only equations, then those equations can still be applied even though the component object is in the middle of the list.

When the first item in the scheduling queue is fully evaluated, it is removed from the queue so that the next item can be rewritten by rewrite rules:

```
sort DetConfig .
subsort Configuration < DetConfig .
op _|_ : ObjectQueue Configuration -> DetConfig [ctor] .
var COMPS : Configuration .          var QUEUE: [ObjectQueue] .

ceq (OBJ :: QUEUE) | COMPS = QUEUE | (COMPS OBJ) .
eq  nil | COMPS = COMPS .
```

where DetConfig is a supersort of Configuration to denote a pair of *scheduling queues* and ordinary configurations for objects and messages.

Definition of the `prepareDelta` Function. The `prepareDelta` function used in the `sync` rule defines the transitions of the synchronous composition of an ensemble (the other functions `transferInput` and `transferResult` are explained in Appendix B.3). Rather than just distributing the `delta` operator over each subcomponent, the `prepareDelta` function constructs a scheduling queue of the subcomponents, where, for each subcomponent, its input adaptor function and the `delta` operator are applied:

```
var NDL : NeList{Data} .     var QUEUE: [ObjectQueue] .

op prepareDelta : Object ~> Object .
op prepareDelta : Time Configuration ObjectQueue ~> DetConfig .
op prepareDelta : Time Oid Configuration ObjectQueue ~> DetConfig .

eq prepareDelta(< C : Ensemble | period : T, machines : COMPS >)
 = < C : Ensemble | machines : prepareDelta(T, COMPS, nil) > .
eq prepareDelta(T, COMPS, QUEUE)
 = if COMPS == none
   then (QUEUE | none)
   else prepareDelta(T, minCid(COMPS), COMPS, QUEUE) fi .
eq prepareDelta(T, C, < C : Component | period : T' > COMPS, QUEUE)
 = prepareDelta(T, COMPS, k-delta(quo(T,T'),
                applyAdaptors(< C : Component | >)) :: QUEUE) .
```

The function `quo(T,T')` returns the integer quotient of T and T', and `minCid(COMPS)` returns the smallest component identifier within `COMPS` in alphabetical order, assuming that there are *no* duplicate identifiers in an ensemble at the same level. For n subcomponents there exist in general $n!$ different scheduling queues, which one we choose does not matter.

The `k-delta` function applies the operator `delta` for the component (in the second argument) as many times as its rate (in the first argument); e.g., `k-delta(3,OBJ)` is reduced to `delta(delta(delta(OBJ)))`:

```
op k-delta : Nat Object ~> Object .

eq k-delta(s N, < C : Component | >)
 = k-delta(N, delta(< C : Component | >)) .
eq k-delta( 0, < C : Component | >) = < C : Component | > .
```

The `applyAdaptors` function distributes the `adaptor` operator for input adaptors (taking a component identifier, a port identifier, and a non-empty list of data) over the input ports of the component:

```
op applyAdaptors : Object -> Object .
op applyAdaptors : ComponentId Configuration -> Configuration .

eq applyAdaptors(< C : Component | ports : PORTS >)
 = < C : Component | ports : applyAdaptors(C, PORTS) > .
eq applyAdaptors(C, < P : InPort | content : NDL > PORTS)
 = < P : InPort | content : adaptor(C, P, NDL) >
   applyAdaptors(C, PORTS) .
eq applyAdaptors(C, PORTS) = PORTS [owise] .
```

6.5 Case Study: an Airplane Turning Control System

To smoothly turn an airplane, the airplane's *ailerons* and its *rudder* need to move in a synchronized way. An aileron is a flap attached to the end of the left or the right wing, and a rudder is a flap attached to the vertical tail. However, the (distributed) controllers for the ailerons and the rudder typically operate at different frequencies. A *turning algorithm* should give instructions to those device controllers in order to achieve a smooth turn.

6.5.1 The Aerodynamics Model

When an aircraft makes a turn, it rolls towards the desired direction of the turn, so that the lift force caused by the two wings acts as the centripetal force and the aircraft moves in a circular motion. The turning rate $\mathrm{d}\psi$ can be given by a function of the aircraft's roll angle ϕ:

$$\mathrm{d}\psi = (g/v) * \tan\phi \tag{6.1}$$

where ψ is the direction of the aircraft, g is the gravity constant, and v is the velocity of the aircraft [63]. The ailerons are used to control the rolling angle ϕ of the aircraft by generating different amounts of lift force in the left and the right wings. Figure 6.15 describes such a banked turn using the ailerons (the aircraft figures in this section are borrowed from [63]).

However, the rolling of the aircraft causes a difference in drag on the left and the right wings, which produces a yawing moment in the opposite direction to the roll, called *adverse yaw*. This adverse yaw makes the aircraft sideslip in a wrong direction with the amount of the yaw angle β, as described in Figure 6.16. This is countered by the aircraft's rudder, which generates the side lift force on the vertical tail that opposes the adverse yaw.

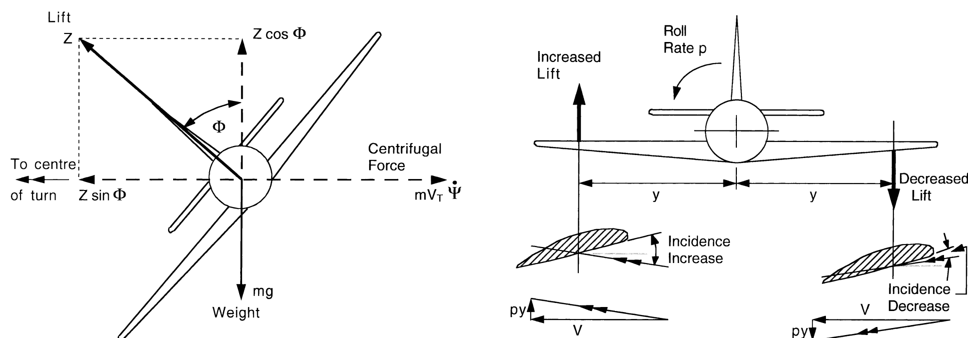


Figure 6.15: Forces acting in a turn of an aircraft.

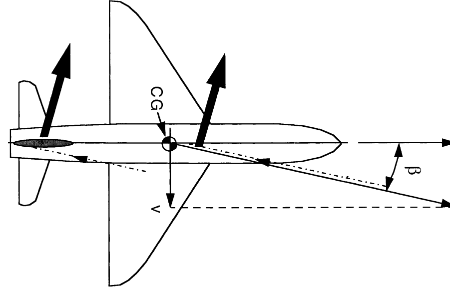


Figure 6.16: Adverse yaw.

The aerodynamics of turning an aircraft involves many factors, such as shape, pitch, speed, acceleration, temperature, air pressure, and so on. Therefore, we make the following assumptions to simplify the model:

- The wings have no dihedral angle (i.e., they are flat with respect to the horizontal axis of the aircraft).
- The altitude does not change, which can be separately controlled by the elevator (a flap attached to the horizontal tail of the aircraft).
- The aircraft maintains a constant speed by separately controlling the thrust power of the aircraft.
- There are no external influences, such as wind or turbulence.

The roll angle ϕ and the yaw angle β can then be modeled by the following differential equations (where a differential equation $\mathrm{d}y^2 = f(x)$ is interpreted by: $\mathrm{d}y = \sqrt{f(x)}$ if $f(x) \geq 0$ and $\mathrm{d}y = \sqrt{-f(x)}$ if $f(x) < 0$) [8]:

$$\mathrm{d}\phi^2 = (\mathit{Lift\ Right} - \mathit{Lift\ Left}) \,/\, (\mathit{Weight} * \mathit{Length\ of\ Wing}) \tag{6.2}$$

$$\begin{aligned}\mathrm{d}\beta^2 = {} & \mathit{Drag\ Ratio} * (\mathit{Lift\ Right} - \mathit{Lift\ Left}) \,/\, (\mathit{Weight} * \mathit{Length\ of\ Wing}) \\ & + \mathit{Lift\ Vertical} \,/\, (\mathit{Weight} * \mathit{Length\ of\ Aircraft}).\end{aligned} \tag{6.3}$$

The lift force from the left, the right, or the vertical tail wing is given by the following linear equation:

$$\mathit{Lift} = \mathit{Lift\ constant} * \mathit{Angle} \tag{6.4}$$

where, for *Lift Right* and *Lift Left*, *Angle* is the angle of the aileron, and for *Lift Vertical*, *Angle* is the angle of the rudder. The lift constant depends on the geometry of the corresponding wing, and the drag ratio is given by the size and the shape of the entire aircraft.

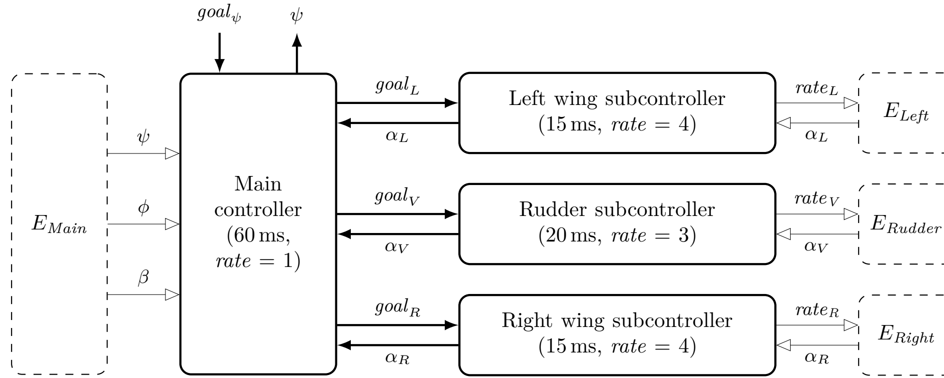


Figure 6.17: The distributed controllers for the airplane control system.

6.5.2 Architecture of the Distributed Controllers

We consider an airplane control system with the four distributed controllers operating at different frequencies: the main controller, and the left wing, the right wing, and the rudder subcontrollers. The main controller has a 60 ms period, the left and the right wing controllers have a 15 ms period (rate 4), and the rudder controller has a 20 ms period (rate 3). The distributed controllers, their physical environments, and the connections between the components are illustrated in Figure 6.17.

Each subcontroller moves the surface of the wing towards the goal angle specified by the main controller, and sends back the current angle of the wing, while the angle of the wing is modeled by its physical environment. For example, the left wing subcontroller receives the goal angle $goal_L$ from the main controller, sends the current angle α_L to the main controller, and determines the moving rate $rate_L$ of the wing for its physical environment E_{Left}. In the meantime, the angle of the left wing changes according to the differential equation $\dot{\alpha_L} = rate_L$ during its 15 ms period. The right wing and the rudder subcontrollers are similar.

Given a desired direction $goal_\psi$ specified by the pilot, the main controller should determine the angles of the (devices of the) subcontrollers needed to make a smooth turn. The physical environment E_{Main} maintains the position of the aircraft (ψ, ϕ, β), where ψ is the current direction, ϕ is the roll angle, and β is the yaw angle. The continuous dynamics of those angles in E_{Main} are specified by the aeronautics equations (6.1–6.3) above, which are parameterized by the current wing angles $(\alpha_L, \alpha_V, \alpha_R)$. For each step of the main controller (period 60 ms), it receives the goal direction $goal_\psi$, the wing angles $(\alpha_L, \alpha_V, \alpha_R)$, and the position (ψ, ϕ, β), and sends the new goal angles $(goal_L, goal_V, goal_R)$ to the subcontrollers.

The behavior of such a physical environment can be specified by a periodic dynamic system. For example, the physical environment E_{Left} of the left wing subcontroller can be modeled as the 1-dimensional periodic dynamic system $E_{Left} = (\mathbb{R},\ (-180°, 180°],\ 15\,\text{ms},\ \Lambda_{E_{Left}})$, where:

- the control command set is $\mathbb{R}$ for the moving rate $rate_L$;
- the state set is $(-180°, 180°]$ to maintain the angle of the left wing;
- the physical transition relation $\Lambda_{E_{Left}} \subseteq (\mathbb{R} \times (-180°, 180°]) \times \mathcal{T}_{15}$ is given by: $((rate_L, \alpha_{L_0}), \alpha_L) \in \Lambda_{E_{Left}}$ iff there exists $\alpha_L \in \mathcal{T}_{15}$ such that

$$\dot{\alpha_L} = rate_L \ \wedge\ \alpha_L(0) = \alpha_{L_0}.$$

  That is, if the current wing angle is α_{L_0} at the beginning of the round and the moving rate is $rate_L$, then the trajectory α_L of the wing angle changes according to the differential equation $\dot{\alpha_L} = rate_L$ during its 15 ms period, where the initial value $\alpha_L(0)$ is the current angle α_{L_0}.

For the main controller, E_{Main} is specified by the 6-dimensional periodic dynamic system $E_{Main} = (\{*\}, (-180°, 180°]^6, 60\,\text{ms}, \Lambda_{E_{Main}})$, where:

- the control command set is $\{*\}$, since there is no control command;
- the state set is $(-180°, 180°]^6$ to denote the direction angle ψ, the roll angle ϕ, the yaw angle β, the left wing angle α_L, the right wing angle α_R, and the rudder angle α_V;
- the physical transition relation $\Lambda_{E_{Main}} \subseteq (\{*\} \times (-180°, 180°]^6) \times \mathcal{T}_{60}^6$ is: $((*, (\psi_0, \phi_0, \beta_0, \alpha_{L0}, \alpha_{R0}, \alpha_{V0})), (\psi, \phi, \beta, \alpha_L, \alpha_R, \alpha_V)) \in \delta_{E_{Main}}$ iff there exist trajectories $\psi, \phi, \beta, \alpha_L, \alpha_R, \alpha_V \in \mathcal{T}_{60}$ such that:

$$\begin{aligned}
&\dot{\psi} = (g/v) * \tan\phi \\
&\wedge\ \dot{\phi}^2 = (C_l \cdot \alpha_R - C_l \cdot \alpha_L)/(W \cdot L_{Wing}) \\
&\wedge\ \dot{\beta}^2 = C_d \cdot (C_l \cdot \alpha_R - C_l \cdot \alpha_L)/(W \cdot L_{Wing}) + C_l \cdot \alpha_V/(W \cdot L_{Aircraft}) \\
&\wedge\ (\psi_0, \phi_0, \beta_0, \alpha_{L0}, \alpha_{R0}, \alpha_{V0}) = (\psi(0), \phi(0), \beta(0), \alpha_L(0), \alpha_R(0), \alpha_V(0)),
\end{aligned}$$

  where the first three lines denote the differential equations (6.1–6.3). That is, given the current physical values $\psi_0, \phi_0, \beta_0, \alpha_{L0}, \alpha_{R0}, \alpha_{V0}$ at the beginning of the round, the trajectories ψ, ϕ, and β of the position angles change according the aeronautics equations during its 60 ms period, *provided that* the trajectories α_L, α_R, and α_V of the wing angles are given, where the initial values of the trajectories are the current values at the beginning of the round.

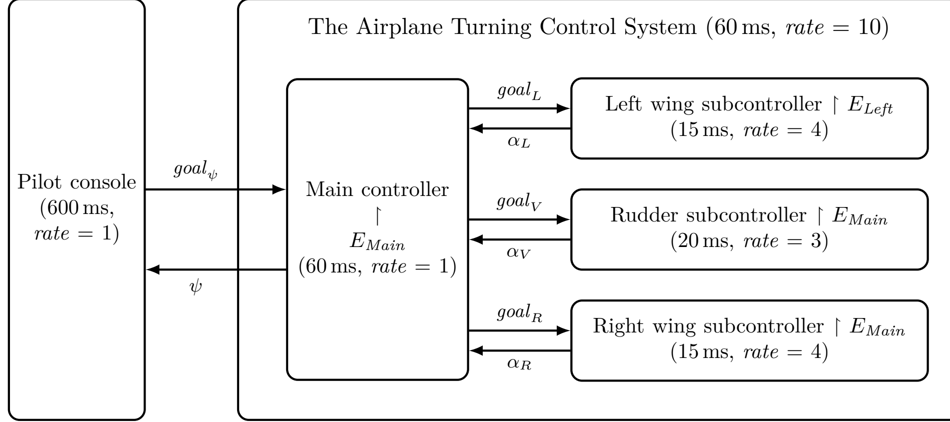


Figure 6.18: The architecture of the airplane turning control system.

We specify the airplane turning control system as a multirate ensemble $\mathfrak{E}$ with four typed machines, as illustrated in Figure 6.18. Each machine in the ensemble $\mathfrak{E}$ incorporates its physical environment, where the behaviors of the physical parameters are specified by the corresponding periodic dynamic systems. For example, the main controller component now also maintains the position angles (ψ, ϕ, β) of the aircraft, and updates the angles using the aeronautics equations for each 60 ms period, according to the received wing angles $(\alpha_L, \alpha_V, \alpha_R)$ in its input port. The "external" environment for the entire airplane turning control system is the pilot console, which is modeled by another typed machine. The pilot console component is connected to the main controller on the outside of the control system.

The synchronous model in Figure 6.18 is actually a *discrete approximation* of the original model in Figure 6.17. The physical environment E_{Main} of the main controller is *physically related* to the physical environments of the subcontrollers E_{Left}, E_{Rudder}, and E_{Right}, since the aeronautic equations depend on the continuous behavior of the wing angles. Such "continuous trajectories" of values cannot be captured by typical ensemble connections, since any communication through ensemble connections is one-step delayed but those continuous values should be *immediately* delivered. Therefore, in order to compute the airplane position (ψ, ϕ, β) in E_{Main}, we use the "fixed" angles $(\alpha_L, \alpha_V, \alpha_R)$ received in the input ports of the main controller, instead of using continuous trajectories of the wing angles during a 60 ms period. This also resolves the "nondeterministic" behavior problem of E_{Main} shown above, since the wing angles are provided by the main controller. It is important that the system is stable, so that the small differences caused by the approximation do not significantly affect the behavior of the system.

6.5.3 Modeling the Airplane Turning Control System

This section formally specifies the multirate ensemble $\mathfrak{E}$ defining the airplane control system, using the framework introduced in Section 6.4.2 to specify and execute multirate synchronous ensembles in Real-Time Maude. This specification is indeed *stable*, since it only uses continuous control functions. Section 6.5.4 exploits the equivalence $\mathfrak{E} \simeq \mathcal{MA}(\mathfrak{E}, T, \Gamma)$ to verify properties about the asynchronous realization $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$.

Parameters and Data Types. The following parameters are chosen to be representative of a small general aircraft. The speed of the aircraft is assumed to be $50\,m/s$ and the gravity constant is $g = 9.80555\,m/s^2$.

```
eq planeSize     = 4.0 .            eq weight        = 1000.0 .
eq wingSize      = 2.0 .            eq virtLiftConst = 0.6 .
eq horzLiftConst = 0.4 .            eq dragRatio     = 0.05 .
```

An angle of a wing is given by a floating-point number. The function `angle` returns the value between $-180°$ and $180°$:

```
subsort Float < Data .

op angle : Float -> Float .         var F : Float .
eq angle(F)
 = if F > 180.0
   then angle(F - 360.0)
   else angle(F + 360.0) fi .
```

Subcontrollers. The subcontrollers for the ailerons and the rudder are modeled as object instances of the following class `SubController`:

```
class SubController |  curr-angle : Float,
                       goal-angle : Float,
                       diff-angle : Float .
subclass SubController < Component .
```

The behavior of each subcontroller is straight-forward: in each iteration, it reads its input from the main controller, denoting the (updated) desired goal angle or $\bot$, moves the corresponding aileron/rudder towards the goal angle, and outputs its current angle to the main controller. In particular, a subcontroller increases or decreases the `curr-angle` toward the `goal-angle`, but the difference in a single (fast) round should be less than or equal to the maximal angle `diff-angle`. The transition function `delta` is therefore defined by the following equation:

```
vars CA GA DA CA' GA' : Float .        vars LI LO : List{Data} .
ops input output : -> PortId [ctor] .

ceq delta(
      < C : SubController |
          ports : < input : InPort | content : D LI >
                  < output : OutPort | content : LO >,
          curr-angle : CA,
          goal-angle : GA,
          diff-angle : DA >)
  =
        < C : SubController |
          ports : < input : InPort | content : LI >
                  < output : OutPort | content : LO CA' >,
          curr-angle : CA',
          goal-angle : GA' >
   if CA' := angle(moveAngle(CA, GA, DA))
   /\ GA' := angle(if D == bot then GA else D fi) .

op moveAngle : Float Float Float -> Float .
eq moveAngle(CA,GA,DA)
 = if abs(GA + (- CA)) > DA
   then CA + DA * sign(GA + (- CA)) else GA fi .
```

The function `delta` updates the goal angle according to the input `D` received from the main controller, and keeps the previous goal if it receives `bot` (that is, $\bot$). The function `moveAngle(CA,GA,DA)` gives the angle that is increased or decreased from the current angle `CA` to the goal angle `GA` up to the maximum angle difference `DA`. Finally, its (updated) current angle `CA'` is sent to the main controller through the output port `output`.

Main Controller. The main controller is modeled as an object instance of the class `MainController` below. The `velocity` attribute denotes the speed of the aircraft. The `curr-yaw`, `curr-roll`, and `curr-dir` attributes model the position sensors of the aircraft by indicating the current yaw angle β, roll angle ϕ, and direction ψ, respectively. The `goal-dir` attribute denotes the goal direction given by the pilot.

```
class MainController | velocity : Float,
                       goal-dir : Float,
                       curr-yaw : Float,
                       curr-rol : Float,
                       curr-dir : Float .
subclass MainController < Component .
```

In each iteration, the main controller reads its input (the reported angles from the device controllers and input from the pilot), computes and sends the new desired angles to the device controllers, and updates its state attributes `curr-yaw`, `curr-roll`, and `curr-dir` according to the aeronautics equations above. The `goal-dir` is also updated if a new goal direction arrives in the `input` port from the external environment (i.e., the pilot console). The transition function `delta` of the main controller is then defined as follows:

```
vars VEL LA RA TA CY CR CD GD RA' TA' CY' CR' CD' GD' : Float .
vars PI LI RI TI PO LO RO TO : List{Data} .      var IN OUT : Data .
ops input output inLW inRW inTW outLW outRW outTW : -> PortId [ctor] .

ceq delta(
     < C : MainController |
         ports : < input : InPort | content : IN PI >
                 < inLW : InPort | content : LA LI >
                 < inRW : InPort | content : RA RI >
                 < inTW : InPort | content : TA TI >
                 < output : OutPort | content : PO >
                 < outLW : OutPort | content : LO >
                 < outRW : OutPort | content : RO >
                 < outTW : OutPort | content : TO >,
         velocity : VEL, period : T,
         curr-yaw : CY, curr-rol : CR,
         curr-dir : CD, goal-dir : GD >)
  =
     < C : MainController |
         ports : < input : InPort | content : PI >
                 < inLW : InPort | content : LI >
                 < inRW : InPort | content : RI >
                 < inTW : InPort | content : TI >
                 < output : OutPort | content : PO OUT >
                 < outLW : OutPort | content : LO (- RA') >
                 < outRW : OutPort | content : RO RA' >
                 < outTW : OutPort | content : TO TA' >,
         curr-yaw : CY', curr-rol : CR',
         curr-dir : CD', goal-dir : GD' >
   if CY' := angle(CY + dBeta(LA,RA,TA) * float(T) )
   /\ CR' := angle(CR + dPhi(LA,RA) * float(T) )
   /\ CD' := angle(CD + evalPsi(CR, dPhi(LA,RA), VEL, float(T)))
   /\ GD' := angle(if IN == bot then GD else GD + IN fi )
   /\ RA' := angle(horizWingAngle(CR', goalRollAngle(CR',CD',GD')))
   /\ TA' := angle(tailWingAngle(CY') )
   /\ OUT := dir: CD' roll: CR' yaw: CY' goal: GD' .
```

The first four lines in the condition compute new values for `curr-yaw`, `curr-roll`, `curr-dir`, and `goal-dir`, based on values in the input ports. A non-$\bot$ value in the `input` port is added to `goal-dir`. The variables `RA'` and `TA'` denote new angles of the ailerons and the rudder, computed by the control functions explained below. Such new angles are queued in the corresponding output ports, and will be transferred to the subcontrollers at the next synchronous step, since they are feedback outputs.

The new values of the yaw angle β, the roll angle ϕ, and the direction angle ψ are defined by the aeronautical differential equations (6.1-6.3). Since we assume discrete movements of the ailerons and the rudder as explained in Section 6.5.2, their moving rate $\dot{\beta}$ and $\dot{\phi}$ are approximated as some *constant values* during each period of the main controller. Consequently, given the current yaw angle β_0 and the current roll angle ϕ_0, the angles are also approximated as the linear equations $\beta(t) = \beta_0 + \dot{\beta} \cdot t$ and $\phi(t) = \phi_0 + \dot{\phi} \cdot t$. Notice that these linear approximations are closely related to the Euler's method to numerically solve differential equations. In the `delta` equation of the main controller, the first two lines of the condition compute the new angles using the linear equations for $\beta(t)$ and $\phi(t)$.

Assuming that the moving rate $\dot{\phi}$ is constant, we can actually solve the differential equation for the direction angle ψ. For the current direction ψ_0, the direction angle ψ is given by the following function:

$$
\begin{aligned}
\psi(x) &= \psi_0 + \int_0^x \frac{g}{v} \tan(\phi_0 + \dot{\phi} \cdot t)\, \mathrm{d}t \\
&= \psi_0 + \frac{g \cdot \big(\log(\cos \phi_0) - \log(\cos(\dot{\phi} \cdot x + \phi_0))\big)}{\dot{\phi} \cdot v}
\end{aligned}
$$

The third line of the condition in the above `delta` equation of the main controller computes the new direction `GD'` using the function $\psi(t)$, where the `evalPsi` function evaluate the second term of the function.

Finally, the new angles of the ailerons and the rudder are computed by using three *control functions*. The function `horizWingAngle` computes the new angle for the aileron in the right wing, based on the current roll angle and the goal roll angle. The angle of the aileron in the left wing is always exactly opposite to the one of the right wing. The function `goalRollAngle` computes the desired roll angle ϕ to make a turn, based on the current roll angle and the difference between the goal direction and the current direction. The function `tailWingAngle` computes the new rudder angle based on the current yaw angle. We define those control functions by linear equations as follows, where `CR` is a current roll angle and `CY` is a current yaw angle:

```
eq goalRollAngle(CR, CD, GD)
 = sign(angle(GD - CD)) * min(abs(angle(GD - CD)) * 0.3, 20.0) .
eq horizWingAngle(CR, GR)
 = sign(angle(GR - CR)) * min(abs(angle(GR - CR)) * 0.3, 45.0) .
eq tailWingAngle(CY)
 = sign(angle(- CY)) * min(abs(angle(- CY)) * 0.8, 30.0) .
```

That is: (i) the goal roll angle is proportional to the difference `GD - CD` between the goal and current directions (with the maximum 20°), (ii) the aileron angles are proportional to the difference `GR - CR` between the goal and current roll angles (with the maximum 45°), and (iii) the rudder angle is proportional to the difference $-$ `CY` between the goal and current yaw angles (with the maximum 30°), where the goal yaw angle is always 0°.

Pilot Console. The pilot console, the environment for the aircraft turning control system, is modeled as an object instance of the class `PilotConsole`:

```
class PilotConsole | scenario : List{Data} .
subclass PilotConsole < Component .
```

The attribute `scenario` contains a list of goal angles to be transferred to the main controller. The pilot console keeps sending goal angles in the `scenario` to its output port until no more data remains, while a nondeterministically generated value can be *added* to the value in the `scenario` output. Note that the pilot console receives the position information from the main controller in its input port, but it does not affect the behavior of the pilot console.

In the conditional rule below, the new desired direction sent by the pilot is `angle(F + F')`, where `F` is the first angle in the pilot's `scenario` attribute, and `F'` is an angle that is nondeterministically assigned to `0.0`, `10.0`, `-10.0`, `60.0`, or `-60.0`. The equation below shows that if the `scenario` is empty, then the pilot console sends $\perp$ through its `output` port.

```
op pVar : -> Data .   rl pVar => 0.0 .      rl pVar => 10.0 .
rl pVar => -10.0 .    rl pVar => 60.0 .     rl pVar => -60.0 .

crl delta(< C : PilotConsole |
             ports : < input : InPort | content : IN LI >
                     < output : OutPort | content : LO >,
             scenario : F LI >)
  => < C : PilotConsole |
         ports : < input : InPort | content : LI >
                 < output : OutPort | content : LO OUT >,
         scenario : LI >
   if pVar => F' /\ OUT := angle(F + F') .
```

```
eq delta(< C : PilotConsole |
             ports : < input : InPort | content : IN LI >
                     < output : OutPort | content : LO >,
             scenario : nil >)
  = < C : PilotConsole |
         ports : < input : InPort | content : LI >
                 < output : OutPort | content : LO bot >,
         scenario : nil > .
```

The Airplane System. An initial state of the airplane turning control system is then represented as the ensemble object in Figure 6.19, where the top-level ensemble `airplane` includes the pilot console `pilot` and the ensemble `csystem` for the airplane turning control system. Notice that each feedback output port in the ensemble contains some default value, whereas other ports are empty (`nil`). For example, the output port `outLW` of the main controller `main` contains the initial value `bot`, which will be delivered to the left wing subcontroller `left` in the first synchronous step.

Input Adaptors. In the ensemble object `csystem`, the left and right wing subcontrollers (with 15 ms period) are 4 times faster than the main controller (with 60 ms period), and the rudder subcontroller (with 20 ms period) is 3 times faster than the main controller. Therefore, to transform a tuple of data into a single value for the main controller, we define input adaptors for the main controller that choose the last value from the input vector:

```
eq adaptor(main, P:PortId, LI D) = D .
```

The subcontrollers receive a single value from the main controller for a "slow" synchronous step, but the left and right wing subcontrollers require a 4-tuples of data, and the rudder subcontroller needs a 3-tuple of data in each step. Therefore, we define the input adaptors for the subcontrollers that generate a vector with extra $\perp$'s as follows:

```
eq adaptor(left, input, D)    = D bots(3) .
eq adaptor(rudder, input, D)  = D bots(2) .
eq adaptor(right, input, D)   = D bots(3) .
```

Similarly, in the top ensemble `airplane`, since the ensemble `csystem` is 10 times faster than the pilot console, the input adaptor for the ensemble `csystem` generates a vector with 9 extra $\perp$'s, and the input adaptor for the pilot console takes the last value from the received input vector:

```
eq adaptor(pilot, input, LI D)  = D .
eq adaptor(csystem, input, D)   = D bots(9) .
```

```
< airplane : Ensemble |
   period : 600,
   ports :  none,
   connections : (pilot . output --> csystem . input ;
                  csystem . output --> pilot . input),
   machines :
     (< pilot : PilotConsole |
         period : 600,
         ports : < input : InPort | content : nil >
                 < output : OutPort | content : bot >,
         scenario : nil >
      < csystem : Ensemble |
         period : 60,
         ports : < input : InPort | content : nil >
                 < output : OutPort | content : nil >,
         connections : (input --> main . input ;
                        main . output --> output ;
                        left . output --> main . inLW ;
                        main . outLW --> left . input ;
                        right . output --> main . inRW ;
                        main . outRW --> right . input ;
                        rudder . output --> main . inTW ;
                        main . outTW --> rudder . input),
         machines :
           (< main : MainController |
               period : 60,
               ports : < input : InPort | content : nil >
                       < inLW : InPort | content : nil >
                       < inRW : InPort | content : nil >
                       < inTW : InPort | content : nil >
                       < output : OutPort | content : nil >
                       < outLW : OutPort | content : bot >
                       < outRW : OutPort | content : bot >
                       < outTW : OutPort | content : bot >,
               velocity : 50.0,  goal-dir : 0.0,
               curr-yaw : 0.0,   curr-rol : 0.0,  curr-dir : 0.0 >
            < left : SubController |
               period : 15,
               ports : < input : InPort | content : nil >
                       < output : OutPort | content : 0.0 >,
               curr-angle : 0.0,  goal-angle : 0.0,  diff-angle : 1.0 >
            < right : SubController |
               period : 15,
               ports : < input : InPort | content : nil >
                       < output : OutPort | content : 0.0 >,
               curr-angle : 0.0,  goal-angle : 0.0,  diff-angle : 1.0 >
            < rudder : SubController |
               period : 20,
               ports : < input : InPort | content : nil >
                       < output : OutPort | content : 0.0 >,
               curr-angle : 0.0,  goal-angle : 0.0,  diff-angle : 0.5 >) >)
>
```

Figure 6.19: An initial state of the airplane turning control system.

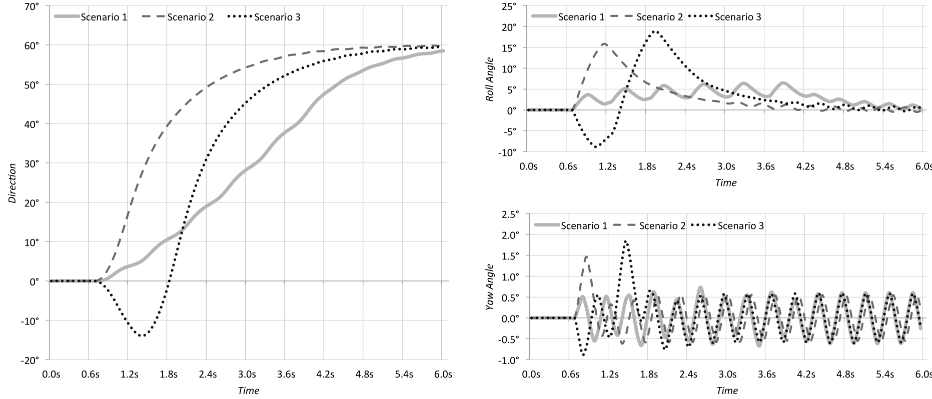


Figure 6.20: The simulation results for the three turning scenarios: the direction (left), the roll angle (top right), and the yaw angle (bottom right).

6.5.4 Formal Analysis

This section explains how we have formally analyzed the Real-Time Maude model of the multirate synchronous design of the airplane turning control system, and how the turning algorithm has been improved as a result of our analysis. The two main requirements that the system should satisfy are:

- *Safety*: during a turn, the yaw angle should always be close to 0.
- *Liveness*: the airplane should reach the goal within a reasonable time, and with both the roll angle and the yaw angle close to 0.

First Analysis Results. We first analyze deterministic behaviors where the airplane turns $+60°$ to the right.[11] In this case, the pilot (console) can send different sequences of commands to the control system to achieve this ultimate goal. We consider the following variations:

1. The pilot *gradually* increases the goal direction by $+10°$ six times.
2. The pilot sets the goal direction to $+60°$ immediately.
3. The goal direction is at first $-30°$, and then it is suddenly set to $+60°$.

Figure 6.20 shows the Real-Time Maude simulation results for these three scenarios up to 6 seconds (10 steps of the pilot). The graphs in Figure 6.20 show that the airplane reaches the desired 60° direction in a fairly short time in all three scenarios, but the yaw angle seems to be quite unstable.

[11] A turn of positive degrees is a right turn, and one of negative degrees a left turn.

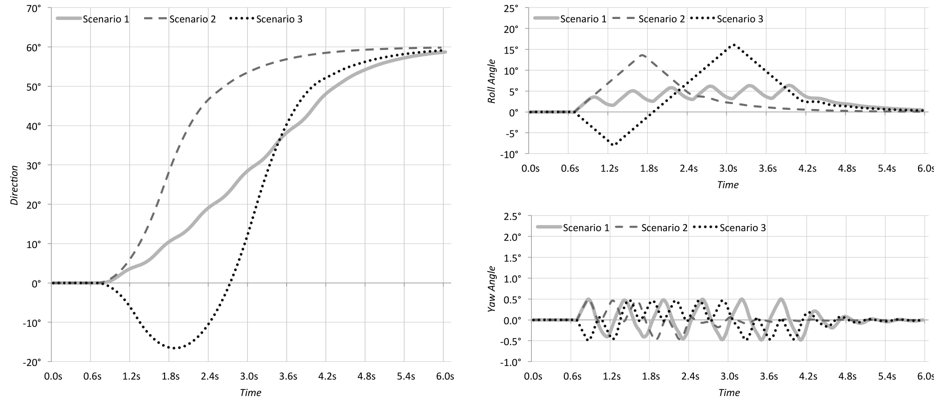


Figure 6.21: The simulation results of the redesigned model: the direction (left), the roll angle (top right), and the yaw angle (bottom right).

New Control Functions. There are two main reasons why the yaw angle is not sufficiently close to 0° during a turn. First, since the control functions are linear, the new angles for the ailerons and the rudder are not small enough when the yaw angle is near 0°. Second, the roll angle is sometimes changing too fast, so that the rudder cannot effectively counter the adverse yaw. Therefore, we modify the control functions as follows:

```
ceq horizWingAngle(CR, GR)
  = sign(FR) *  (if abs(FR) > 1.0 then min(abs(FR) * 0.3, 45.0)
                 else FR ^ 2.0 * 0.3 fi)   if FR := angle(GR - CR) .
ceq tailWingAngle(CY)
  = sign(FY) *  (if abs(FY) > 1.0 then min(abs(FY) * 0.8, 30.0)
                 else FY ^ 2.0 * 0.8 fi)   if FY := angle(- CY) .
ceq goalRollAngle(CR, CD, GD)
  =  if abs(FD * 0.32 - CR) > 1.5 then CR + sign(FD * 0.32 - CR) * 1.5
     else FD * 0.32 fi                     if FD := angle(GD - CD) .
```

When the difference between the goal and the current angles (`FR` or `FY`) is less than or equal to 1°, the functions `horizWingAngle` and `tailWingAngle` are now *not* proportional to the difference, but proportional to the *square* of the difference. This implies that if the difference is close to 0°, then the result becomes much smaller than before. Furthermore, the goal roll angle can be changed at most 1.5° at a time, so that there is no more abrupt rolling. Then, we obtain the Real-Time Maude simulation results of the redesigned model with the new control functions in Figure 6.21, where the yaw angle now shows a much stabler behavior than before.

Verifying the New Control Functions. To analyze whether the new control functions ensure a smooth turn, we define some auxiliary functions. The function `sysOut(`*Object*`)` returns the content of the output port of the airplane control system, and `safeYawAll(`*OutputDataList*`)` checks whether every output data in the given list has a yaw angle less than 1.0°:

```
eq sysOut(
   < airplane : Ensemble | machines :
         (< csystem : Ensemble |
            ports : < output : OutPort | content : LI > PORTS >
          COMPS) >) = LI .
eq safeYawAll(D LI) = abs(yaw(D)) < 1.0 and safeYawAll(LI) .
eq safeYawAll(nil) = true .
```

The following Real-Time Maude search command can verify that there is no dangerous yaw angle reachable within 27 seconds for Scenario 3, where `model(`*Scenario*`)` gives the initial state of the system with a list of directions *Scenario* for the pilot console (the number of states explored is 46):

```
Maude> (tsearch [1] {model(-30.0 90.0)} =>* {SYSTEM}
        such that not safeYawAll(sysOut(SYSTEM)) in time <= 27000 .)
No solution
```

Although each state of the transition system captures only the slow steps of the top ensemble (i.e., every 600 ms), `safeYawAll` also checks all fast steps of the main controller (every 60 ms), since it accesses the *history* of the main controller's status in the output port of the top ensemble.

The system requirement (i.e., the airplane reaches the desired direction with a stable status while keeping the yaw angle close to 0°) can be expressed as the LTL formula $\Box(\neg\texttt{stable} \rightarrow (\texttt{safeYaw}\ \mathbf{U}\ (\texttt{reach} \wedge \texttt{stable})))$. When the function `stableAll(`*OutputDataList*`)` returns `true` only if both the yaw angle and the roll angle are less than 0.5° for every output data in the given list, the atomic propositions in the formula can be defined as follows:

```
 eq {SYSTEM} |= safeYaw  = safeYawAll(sysOut(SYSTEM)) .
 eq {SYSTEM} |= stable   = stableAll(sysOut(SYSTEM)) .
ceq {SYSTEM} |= reach    = abs(angle(goal(D) - dir(D))) < 0.5
   if D := last(sysOut(SYSTEM)) .
```

We have verified that all three scenarios satisfy the above LTL property with the *new* control functions, using the time-bounded LTL model checking command of Real-Time Maude. For example, the following command shows the case for Scenario 3 (the number of states explored is 13):

```
Maude> (mc {model(-30.0 90.0)}  |=t
          [] (~stable -> (safeYaw U reach /\ stable))
             in time <= 7200 .)
Result Bool : true
```

Finally, we have verified nondeterministic behaviors in which the pilot sends one of the turning angles $-60.0°$, $-10.0°$, $0°$, $10°$, and $60.0°$ to the main controller for 6 seconds. The following model checking command then shows that the redesigned system satisfies the above LTL formula up to 18 seconds, where one of the five angles is nondeterministically chosen and added to the angle $0°$ at each step of the pilot console:

```
Maude> (mc {model(0.0 0.0 0.0 0.0 0.0 0.0)}  |=t
          [] (~ stable -> (safeYaw U (reach /\ stable)))
             in time <= 18000 .)
Result Bool : true
```

This model checking analysis took 75 minutes on Intel Core i5 2.4 GHz with 4 GB memory, and the number of states explored was $335,363$.[12] It is a huge state space reduction compared to the distributed asynchronous model, since: (i) asynchronous behaviors are eliminated thanks to Multirate PALS, and (ii) any intermediate fast steps for the sub-components are merged into a single-step of the system's top-level ensemble.

6.5.5 Model Checking the Asynchronous System

To demonstrate the performance benefits obtained by Multirate PALS, this section compares the execution times and the number of states explored for model checking the synchronous model and the distributed asynchronous model. Instead of directly using the asynchronous model in Section 6.3.2, this section uses a *highly simplified* model with the following assumptions: (i) the time domain is discrete and all clocks are perfectly synchronized; (ii) a machine takes zero time to perform a transition, including processing I/O; (iii) each message is *instantaneously* delivered to its recipient; and (iv) when a component sends output messages into the network, it sends all of them *at the same time*, instead of sending them one by one. This *simplified* model is also formally specified in Real-Time Maude (see Appendix B.4).

[12]Only the outermost "big-step" transitions contribute to the state space. However, computing each "big-step" transition involves computing many "small-step" transitions that do not contribute to the state space, while different small-step computation paths can lead to the same big-step state, because of nondeterministic behaviors. Therefore, the model checking involves computing significantly more than 335,363 behaviors.

Model	N	$T \leq$ 1,200 ms states	time	$T \leq$ 1,800 ms states	time	$T \leq$ 2,400 ms states	time	$T \leq$ 3,000 ms states	time
Sync	2	7	0.2	13	0.2	25	0.3	49	0.5
	3	13	0.2	28	0.3	73	0.6	202	1.6
	4	21	0.3	57	0.5	169	1.3	593	4.4
	5	31	0.3	116	0.9	471	3.3	2,111	15
Async	2	12,552	1.6	25,088	3.2	50,144	6.3	100,256	13
	3	25,088	3.1	56,512	7.5	150,576	22	420,288	84
	4	41,816	9	117,232	30	351,680	117	1,238,648	611
	5	62,752	35	240,784	168	983,960	998	4,415,784	8,679

Table 6.1: The number of reachable states and the execution times (in seconds) up to time bound T with N nondeterministic pilot choices.

We have compared the number of reachable states in the both models for various *nondeterministic* pilot choices. In this analysis, the goal direction was initially set to 60°, but the pilot can nondeterministically choose one of the five angles 0°, 10°, −10°, 60°, −60.0°, and add it to the goal direction every 600 ms. These experiments were conducted on an Intel Xeon 2.93 GHz with 24 GB memory. Table 6.1 summarizes the experiment results with different time bounds and different nondeterministic choices. For each N in the table we use the first N angles from 0°, 10°, −10°, 60°, −60.0° in order. Table 6.1 illustrates how quickly the state space explodes, even for the unrealistically simplified asynchronous model. The number of reachable states up to 3,000 ms in the synchronous model is 2,111 with 5 choices. However, even for only 2 nondeterministic choices, the asynchronous model generates 12,552 states within time bound 1,200 ms.

6.6 Concluding Remarks

This chapter has proposed Multirate PALS as an answer to the big challenge of designing, verifying, and implementing a virtual synchronous multirate DCPS. To achieve this in a rigorous way, we have defined the mathematical semantics of synchronous multirate ensembles and of several key component transformations, which together allow us to give a precise mathematical definition of a Multirate PALS transformation $\mathfrak{E} \mapsto \mathcal{MA}(\mathfrak{E}, T, \Gamma)$ and to prove the bisimilarity of the distributed implementation $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ with $\mathfrak{E}$. We have also shown that Multirate PALS can be well-supported by Real-Time Maude to formally specify and verify multirate distributed hybrid systems. We have illustrated its power by showing how it can be used to verify properties about the turning maneuvers of an airplane.

Part III

Applications to Modeling Languages

CHAPTER 7

MULTIRATE SYNCHRONOUS AADL

The Multirate PALS pattern, introduced in Chapter 6, reduces the problem of designing and verifying virtually synchronous distributed cyber-physical systems to the much simpler tasks of designing and verifying their underlying synchronous designs. In order to make this Multirate PALS methodology available within an industrial modeling environment, this chapter[1] presents the modeling language *Multirate Synchronous AADL*, which can be used to specify multirate synchronous designs using the AADL modeling standard. This chapter then defines the formal semantics of Multirate Synchronous AADL in Real-Time Maude, and integrates Real-Time Maude verification into the OSATE tool environment for AADL. We exemplify such formal verification with avionics and aeronautical systems.

7.1 Introduction

Modeling languages are widely used for designing and analyzing embedded systems. Since design errors are much more expensive than coding errors, it has been generally agreed that system verification should first be carried out at the level of designs by verifying the *model* representing a system design. However, modeling languages tend to be weak on the formal analysis side; if they can be endowed with formal analysis capabilities "under the hood" with minimal disruption to the established modeling processes, formal methods can be more easily adopted and many design errors can be detected early in the design phase, resulting in higher quality systems and in substantial savings in the development and verification processes.

[1]This chapter is based on [26, 28, 31], joint work with Peter Ölveczky, José Meseguer, and Abdullah. Al-Nayeem.

The *Architecture Analysis & Design Language* (AADL) [91] is a widely used industrial modeling standard for embedded real-time systems, such as avionics and automative systems. AADL models describe cyber-physical systems made up of *distributed components* that communicate with each other. To make *formal* verification of AADL models possible, there exist several approaches to give a formal semantics for a behavioral fragment of AADL, such as [36, 43, 52, 111, 147]. However, due to the combinatorial explosion caused by the distributed nature of cyber-physical systems, even for simple models, straightforward model checking of AADL models quickly becomes unfeasible. The point is that the inherent difficulties of verifying distributed cyber-physical systems *do not disappear* at the level of models: they are common to both models and code.

To tame the combinatorial explosion the PALS ("physically asynchronous, logically synchronous") pattern [7, 138] has been proposed to support the model checking of a wide class of distributed cyber-physical systems by drastically reducing the state space. The key idea behind PALS is that the intended behavior of many distributed cyber-physical systems is that they should be *virtually synchronous*, since they are controlled in a periodic way (conceptually there is a logical period during which all components perform a transition and send data to each other). However, PALS is limited by the unrealistic requirement that all components have the same period. This has led us to develop *Multirate PALS* that generalizes PALS to the multirate case. The Multirate PALS transformation (explained in Chapter 6) reduces the design and verification of a multirate distributed system of this kind to that of its much simpler synchronous counterpart.

Taking advantage of Multirate PALS for verifying distributed designs in AADL by model checking the corresponding *synchronous* design requires:

- defining appropriate extensions of AADL, where such synchronous models can be specified;
- giving a *formal synchronous semantics* to such language extensions, e.g., using rewriting logic; and
- building tools as OSATE plugins that *automate* the model checking verification of the synchronous models.

This can be very useful, because: (i) synchronous designs are much easier to understand by engineers, (ii) they are much easier to model check, and (iii) generation of their more complex distributed versions can be automated and made *correct by construction* using Multirate PALS.

7.1.1 Main Contributions

First, this chapter defines the *Multirate Synchronous AADL* language as a sublanguage of AADL, and presents a formal *synchronous semantics* for this subset in Real-Time Maude. Multirate Synchronous AADL generally identifies a set of AADL models that can be considered as synchronous; hence, Multirate Synchronous AADL makes it possible to specify multirate synchronous real-time systems in AADL. Then, using the Multirate PALS transformation, it is possible to transform such a synchronous model into a correct-by-construction asynchronous AADL model.

Second, this chapter presents the *MR-SynchAADL* tool as an OSATE[2] plugin. The MR-SynchAADL tool is a simulation and LTL model checking tool for Multirate Synchronous AADL. The tool automatically synthesizes a Real-Time Maude [149] model from a Multirate Synchronous AADL model, provides a requirements specification language to conveniently define LTL properties of the Multirate Synchronous AADL model, and performs the model checking *within* the OSATE modeling environment. This enables a model-engineering process for important classes of distributed real-time systems that combines the convenience of AADL modeling, the complexity reduction of Multirate PALS, and formal verification in Real-Time Maude.

The MR-SynchAADL tool, the entire Real-Time Maude semantics, and the Multirate Synchronous AADL specifications for the case studies are available at `http://maude.cs.illinois.edu/tools/synchaadl`.

7.1.2 Related Work

The paper [94] formalizes the AADL data port protocol in Event-B. Despite the title of the paper, it does not define a synchronous subset of AADL and therefore does not provide an executable formal semantics of any such subset. There exist a number of formalizations and verification tools for different subsets of AADL (see, e.g., [36, 43, 52, 147]). These approaches target ordinary (asynchronous) AADL models and do not define synchronous subsets of AADL. In [111], the behaviors of single AADL threads are given by synchronous Lustre programs. Since [111] also targets asynchronous AADL models, the authors show how asynchronous computation can be *encoded* in a synchronous language. This encoding does of course *not* reduce the state space of the asynchronous system.

[2]The OSATE modeling environment provides a set of Eclipse plug-ins for AADL.

7.1.3 Structure of this Chapter

This chapter is organized as follows. Section 7.2 briefly introduces AADL and defines *Multirate Synchronous AADL* as an annotated sublanguage of AADL. Section 7.3 presents the Real-Time Maude semantics of Multirate Synchronous AADL. Section 7.4 presents the *MR-SynchAADL* tool that supports both checking if a model is a legal Multirate Synchronous AADL model and verifying Multirate Synchronous AADL models within OSATE. Section 7.5 illustrates the effectiveness of the Multirate Synchronous AADL language and the MR-SynchAADL tool with case studies, and Section 7.6 presents some concluding remarks.

7.2 Multirate Synchronous AADL

The *Architecture Analysis & Design Language* (AADL) [91] is an industrial modeling standard used in avionics, aerospace, automotive, medical devices, and robotics to describe an embedded real-time system as an assembly of software components mapped onto an execution platform. The OSATE modeling environment provides a set of Eclipse plug-ins for AADL.

An AADL model describes a system as a set of hardware and software components. This chapter focuses on the software components of AADL, since we use AADL to specify *synchronous designs*. Software components include: (i) *thread* components that model the application software to be executed; (ii) *process* components defining protected memory that can be accessed by its thread and data subcomponents; and (iii) *data* components representing data types. The *dispatch protocol* of a thread determines when the thread is executed. For example, a *periodic* thread is activated at fixed time intervals, and an *aperiodic* thread is activated when it receives an event. *System* components are the top level components of AADL.

Multirate Synchronous AADL is defined as a subset of AADL, extended with a *property set* `MR_SynchAADL`, in such a way that: (i) for each "round" of a thread, the exception is independent of the other threads, and (ii) output of a thread generated in one round is available as input of the receiving thread at the beginning at the next round (i.e., hierarchical multirate ensembles). In AADL, such threads would be executed asynchronously; however, since the threads are independent of each other in each round, the "final" states in each round are the same in both asynchronous and synchronous executions. Therefore, the AADL constructs in the (common) subset have the same meaning in AADL and in Multirate Synchronous AADL.

7.2.1 Subset of AADL

Multirate Synchronous AADL is intended to model synchronous *designs*, and therefore it focuses on the behavioral and structural subset of AADL: hierarchical system, process, and thread components; ports and connections; and thread behaviors defined in the *behavior annex* sublanguage [97].

In AADL, a component *type* specifies the component's *interface* (e.g., ports) and *properties* (e.g., periods), and a component *implementation* then specifies the internal structure of the component as a set of *subcomponents* and a set of *connections* linking their ports. An AADL construct may have *properties* describing its parameters, declared in *property sets*.

Example 7.1. *The following type declarations define a system component `Controller` and a process component `CtrlProcess`. The system component has one data input port, one data output port, and* 12 ms *period:*

```
system Controller
  features
    input: in data port Base_Types::Float;
    output: out data port Base_Types::Float;
  properties
    Period => 12 ms;
end Controller;

process CtrlProcess
  features
    pin: in data port Base_Types::Float;
    pout: out data port Base_Types::Float;
end CtrlProcess;
```

The implementation `impl` of `Controller` is defined as follows. It contains two processes `in_proc` and `out_proc` of type `CtrlProcess`. The input port `input` and the output port `output` of `Controller`, respectively, are connected to the input port `pin` of `in_proc` and the output port `pout` of `out_proc`. The output port of `in_proc` and the input port of `out_proc` is also connected by the connection `c3`, where its `Timing` property is declared to be `Delayed`:

```
system implementation Controller.impl
  subcomponents
    in_proc: process CtrlProcess;   out_proc: process CtrlProcess;
  connections
    c1: port input -> in_proc.pin;
    c2: port out_proc.pout -> output;
    c3: port in_proc.pout -> out_proc.pin { Timing => Delayed; };
end Controller.impl;
```

The dispatch protocol is used to trigger an execution of a thread. In Multirate Synchronous AADL, each thread must have *periodic* dispatch. This means that, *in the absence of immediate connections*, the thread is dispatched at the beginning of each period of the thread. *Event-triggered* dispatch (such as *aperiodic*, *sporadic*, *timed*, and *hybrid* dispatch) is not suitable to define a system in which all threads must execute in lock-step, since the execution of one thread can trigger the execution of another thread. In AADL, they are declared by the component properties:

```
Dispatch_Protocol => Periodic;
Period => time;
```

There are three kinds of ports in AADL: *data* ports, *event* ports, and *event data* ports. Event and event data ports are used to dispatch event-triggered threads, but can also be used with periodic dispatch in version 2 of AADL. The main difference is that an event (or event data) port may contain a *buffer* of untreated received events, whereas a data port always contains (at most) one element. Multirate Synchronous AADL only allows *data* ports, since each component in multirate ensembles gets only one piece of data in each input port (the user should only specify single machines and the input adaptors, dealing with the k-tuples of inputs/outputs).

In Multirate Synchronous AADL, all outputs generated in one iteration must be available at the beginning of the next iteration, and *not before*, since output generated in one step becomes input of its destination component in the next step in a multirate machine ensemble. This can be achieved in AADL by having *delayed* connections, declared by the connection property `Timing => Delayed`, for which the value from the sender is transmitted at its deadline and is available to the receiver at its next dispatch.

Thread behavior is modeled as a guarded transition system using the *behavior annex* sublanguage [97]. Each transition has the form

$$s \; \texttt{-[}guard\texttt{]->} \; s' \; \{actions\};$$

where s and s' are states, and *guard* is a Boolean condition on the local variables and the input ports. The actions performed when a transition is applied may update local variables, call methods (subprograms), generate new outputs to ports, and/or suspend the thread. Actions are built from basic actions using sequencing, conditionals, and finite loops. When a thread is activated, any *enabled* transition can be (nondeterministically) applied; if the resulting state is not a *complete* state, another transition is applied, until a complete state is reached (or the thread suspends).

7.2.2 The MR_SynchAADL Property Set

The additional features in Multirate Synchronous AADL are defined in the following property set MR_SynchAADL:

```
property set MR_SynchAADL is
  Synchronous: inherit aadlboolean
               applies to (system, process, thread group, thread);
  Nondeterministic: aadlboolean applies to (thread);
  InputConstraints: list of aadlstring applies to (thread);
  InputAdaptor: aadlstring applies to (port);
end MR_SynchAADL;
```

The main system component in a Multirate Synchronous AADL model should declare the Boolean component property Synchronous to state that the system component can be executed synchronously:

```
MR_SynchAADL::Synchronous => true;
```

In Multirate Synchronous AADL, we assume that the observable behavior of an environment is defined by a *nondeterministic* machine, and that all other threads are *deterministic*. A nondeterministic environment component should add the Boolean component property Nondeterministic:

```
MR_SynchAADL::Nondeterministic => true;
```

The possible outputs in such a nondeterministic environment component can often be defined by using an *environment constraint* c_e so that $c_e(\vec{o})$ is *true* iff the environment can nondeterministically generate output $\vec{o}$ in any iteration. The component property InputConstraints defines an input constraint on a set of Boolean-valued outputs:

MR_SynchAADL::InputConstraints => ("*Boolean formula*");

The main feature needed to define a multirate ensemble is *input adaptors* (explained in Chapter 6). Multirate Synchronous AADL provides a number of *predefined input adaptors*. The 1-to-k input adaptors, mapping a single value to a k-vector of values,[3] include:

"repeat_input"	(maps v to $(v, v, \ldots, v)$)
"use in first iteration"	(maps v to $(v, \bot, \ldots, \bot)$)
"use in last iteration"	(maps v to $(\bot, \ldots, \bot, v)$)
"use in iteration i"	(maps v to $(\underbrace{\bot, \ldots, \bot}_{i-1}, v, \bot, \ldots, \bot)$).

[3]A fast component with rate k performs k internal transitions during one slow period. Since the fast component expects a k-tuple of inputs, a single-value output from a slow component is transformed to a k-tuple of inputs by a 1-to-k input adaptor.

The k-to-1 input adaptors, mapping k-vectors to single values,[4] include:

`"first"`	(maps $(v_1, \ldots, v_k)$ to v_1)
`"last"`	(maps $(v_1, \ldots, v_k)$ to v_k)
`"use element` i`"`	(maps $(v_1, \ldots, v_k)$ to v_i)
`"average"`	(maps $(v_1, \ldots, v_k)$ to $(v_1 + \cdots + v_k)/k$)
`"max"`	(maps $(v_1, \ldots, v_k)$ to $\max(v_1, \ldots, v_k)$)
`"min"`	(maps $(v_1, \ldots, v_k)$ to $\min(v_1, \ldots, v_k)$)
`"sum"`	(maps $(v_1, \ldots, v_k)$ to $v_1 + \cdots + v_k$),

where the first two adaptors are special cases of the third one, and the last four adaptors can be only applied for numerical inputs. In Multirate Synchronous AADL, such an input adaptor is assigned to an input port as a property `MR_SynchAADL::InputAdaptor =>` *input adaptor*, e.g.:

```
goal_angle: in data port Base_Types::Float
        {MR_SynchAADL::InputAdaptor => "use in first iteration";};
```

The `"use in ..."` 1-to-k adaptors generate some "don't care" values $\bot$. Instead of explicitly having to define such default values, the fact that a port p has an input "$\bot$" is manifested by p`'fresh` being `false`.

7.2.3 Case Study: Turning an Airplane

This section shows how the *design* of a virtually synchronous control system in Chapter 6.5 for smoothly turning an airplane can be specified in Multirate Synchronous AADL. In order to achieve a smooth turn of the airplane, the controller must synchronize the movements of the airplane's two *ailerons* and its *rudder*. As shown in Figure 7.1, our system consists of four periodic controllers with different periods. Note that Section 6.5 has also defined a model of the control algorithm *directly* in Real-Time Maude, and we refer to it for more details about the turning control algorithm.

The *environment* is the pilot console that allows the pilot to select a new desired direction every 600 ms. The *left wing controller* receives the desired angle $goal_L$ of the aileron from the main controller, and moves the aileron towards that angle. The *right wing* (resp., the *rudder*) *controller* operates in the same way for the right wing aileron (resp., the rudder). The *main controller* receives the desired direction (from the pilot console) and the current angle of each device (from the device controllers), computes the new desired device angles, and sends them to the device controllers.

[4] A k-tuple output from a fast component is transformed a single value by a k-to-1 input adaptor so that it can be read by the slow component.

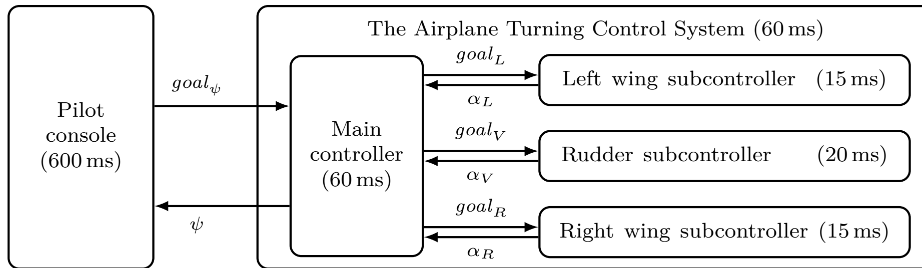


Figure 7.1: The architecture of our airplane turning control system.

The following AADL component declares the top-level "implementation" of the system in terms of connections and subcomponents:

```
system implementation Airplane.impl
  subcomponents
    pilotConsole: system PilotConsole.impl;
    turnCtrl: system TurningController.impl;
  connections
    C1: port pilotConsole.goal_dr -> turnCtrl.pilot_goal;
    C2: port turnCtrl.curr_dr  -> pilotConsole.curr_dr;
  properties
    MR_SynchAADL::Synchronous => true;      Period => 600 ms;
    Timing => Delayed applies to C1, C2;
    Data_Model::Initial_Value => ("0.0") applies to
      pilotConsole.goal_dr, turnCtrl.curr_dr;
end Airplane.impl;
```

The property declaration '`Timing => Delayed applies to C1, C2`' states that the inter-component connections `C1` and `C2` are *delayed* connections. The property `Data_Model::Initial_Value` assigns the initial value to the corresponding port. In Multirate Synchronous AADL, this property must be declared for each "feedback" output port, unless its initial value is $\perp$ (e.g., the output port `goal_dr` has initially the value `0.0`).

The pilot may in any round nondeterministically *add* 0°, 10°, or −10° to the current desired direction. The input port `curr_dr` receives the current direction ψ from the turning system, which operates 10 times faster than the pilot; we must therefore use an input adaptor to map the 10-tuple of directions into a single value, for which it is natural to use the *last* value:

```
system PilotConsole              -- "interface" of the pilot console
  features
    curr_dr: in data port Base_Types::Float
                          {MR_SynchAADL::InputAdaptor => "last";};
    goal_dr: out data port Base_Types::Float;
end PilotConsole;
```

The implementation of `PilotConsole` contains the following thread that defines the pilot behavior. When the thread dispatches, the transition from state `idle` to `select` is taken. Since `select` is not a *complete* state, the thread continues executing, by nondeterministically selecting one of the other transitions, which assigns the selected angle change to the output port `goal_dr`. Since the resulting state `idle` is a complete state, the execution of the thread in the current dispatch ends:

```
thread implementation PilotConsoleThread.impl
  properties
    MR_SynchAADL::Nondeterministic => true;
    Dispatch_Protocol => Periodic;
  annex behavior_specification {**
    states
      idle: initial complete state;        select: state;
    transitions
      idle -[on dispatch]-> select;
      select -[ ]-> idle {goal_dr := 0.0};
      select -[ ]-> idle {goal_dr := 10.0};
      select -[ ]-> idle {goal_dr := -10.0};
  **};
end PilotConsoleThread.impl;
```

The *turning controller* component consists of the main controller and the three device subcontrollers. The desired *change* in the direction is received from the pilot console in the input port `pilot_goal`. Since the turning controller is 10 times faster than the pilot console, the controller will execute 10 "internal" iterations in a global period; hence the single input in the input port `pilot_goal` from the pilot must be mapped into 10 values, and we choose to use the input in the first local iteration:

```
system TurningController
  features
    pilot_goal: in data port Base_Types::Float
        {MR_SynchAADL::InputAdaptor => "use in first iteration";};
    curr_dr: out data port Base_Types::Float;
end TurningController;
```

The following `TurningController` implementation defines the structure of the turning controller shown in Figure 7.1, where the subcontrollers are specified as instances of `Subcontroller.impl`. Since the periods of the device controllers are different, and since the rudder can move at most 0.5° in each 20 ms period, whereas the ailerons can move 1° in each 15 ms period, we must define these values in the turning controller:

```
system implementation TurningController.impl
 subcomponents
   mainCtrl: system Maincontroller.impl;
   leftCtrl: system Subcontroller.impl;
   rightCtrl: system Subcontroller.impl;
   rudderCtrl: system Subcontroller.impl;
 connections
   C1: port pilot_goal -> mainCtrl.goal_angle;
   C2: port mainCtrl.curr_dr -> curr_dr;
   C3: port leftCtrl.curr_angle -> mainCtrl.left_angle;
   C4: port rightCtrl.curr_angle -> mainCtrl.right_angle;
   C5: port rudderCtrl.curr_angle -> mainCtrl.rudder_angle;
   C6: port mainCtrl.left_goal -> leftCtrl.goal_angle;
   C7: port mainCtrl.right_goal -> rightCtrl.goal_angle;
   C8: port mainCtrl.rudder_goal -> rudderCtrl.goal_angle;
 properties
   Period => 60 ms;
   Timing => Delayed applies to C3, C4, C5, C6, C7, C8;

   -- ailerons can move 1° in 15 ms
   Period => 15 ms applies to leftCtrl, rightCtrl;
   Data_Model::Initial_Value => ("1.0") applies to
       leftCtrl.ctrlProc.ctrlThread.diffAngle,
       rightCtrl.ctrlProc.ctrlThread.diffAngle;

   -- rudder can move 0.5° in 20 ms
   Period => 20 ms applies to rudderCtrl;
   Data_Model::Initial_Value => ("0.5") applies to
       rudderCtrl.ctrlProc.ctrlThread.diffAngle;

    -- initial feedback output
   Data_Model::Initial_Value => ("0.0") applies to
       leftCtrl.curr_angle, rightCtrl.curr_angle,
       rudderCtrl.curr_angle, mainCtrl.left_goal,
       mainCtrl.right_goal, mainCtrl.rudder_goal;
end TurningController.impl;
```

The behavior of a subcontroller is straightforward: move the device toward the goal angle up to `diffAngle` (declared in `TurningController.impl`), update the goal angle if a new value has received in `goal_angle`, and report back the current angle through `curr_angle`. Since the main controller is slower than the device controller, the single input in `goal_angle` received from the main controller must be adapted to a k-tuple; in this case, we use the input in the first of the k internal iterations:

```
system Subcontroller
  features
    goal_angle: in data port Base_Types::Float
          {MR_SynchAADL::InputAdaptor => "use in first iteration";};
    curr_angle: out data port Base_Types::Float;
end Subcontroller;

thread implementation SubcontrollerThread.impl
  subcomponents
    currAngle: data Base_Types::Float;
    goalAngle: data Base_Types::Float;
    diffAngle: data Base_Types::Float;
  properties
    Data_Model::Initial_Value => ("0.0") applies to
       currAngle, goalAngle;
  annex behavior_specification {**
    states
      init: initial complete state;
      move, update: state;
    transitions
      init -[on dispatch]-> move;
      move -[abs(goalAngle - currAngle) > diffAngle]-> update {
        if (goalAngle - currAngle >= 0)
          currAngle := currAngle + diffAngle
        else
          currAngle := currAngle - diffAngle
        end if};
      move -[otherwise]-> update {currAngle := goal_angle};
      update -[ ]-> init {
        if (goal_angle'fresh) goalAngle := goal_angle end if;
        curr_angle := currAngle};
  **};
end SubcontrollerThread.impl;
```

The *main controller* is responsible for deciding the desired angles of the devices, and also for updating the current state of the aircraft. The main controller must adapt the tuples received from the device subcontrollers; the controller naturally chooses the *last* value in these tuples, which denote the most recent angle of the flap. The input from the pilot has already been adapted in the turning control system, which has the same period as the main controller. In the thread implementation below, the data components define the current state (ψ, ϕ, β) and the speed of the airplane, and the transitions compute the new values of the state variables and the output ports (a part is replaced by '`...`'):

```
system Maincontroller
 features
   goal_angle: in data port Base_Types::Float;
   left_angle: in data port Base_Types::Float;
   right_angle: in data port Base_Types::Float;
   rudder_angle: in data port Base_Types::Float;
   curr_dr: out data port Base_Types::Float;
   left_goal: out data port Base_Types::Float;
   right_goal: out data port Base_Types::Float;
   rudder_goal: out data port Base_Types::Float;
  properties
   MR_SynchAADL::InputAdaptor => "last" applies to
     left_angle, right_angle, rudder_angle;
end Maincontroller;

thread implementation MaincontrollerThread.impl
 subcomponents
  currDir: data Base_Types::Float; currRol: data Base_Types::Float;
  currYaw: data Base_Types::Float; goalDir: data Base_Types::Float;
 properties
  Data_Model::Initial_Value => ("0.0") applies to
     currDir, currRol, currYaw, goalDir;
  Dispatch_Protocol => Periodic;
 annex behavior_specification {**
  variables
    d, x, y, z, w : Base_Types::Float;
  states
    init : initial complete state;
    yaw, rollNdir, goal, aileron, rudder, output : state;
  transitions
    init -[on dispatch]-> yaw;
    yaw -[ ]-> rollNdir { ... };    -- computes currYaw
    rollNdir -[ ]-> goal { ... };   -- computes currRol and currDir
    goal -[ ]-> aileron {           -- updates goalDir
      if (goal_angle'fresh)
        goalDir := goalDir + goal_angle end if};
    aileron -[ ]-> rudder { ... };  -- computes aileron angles
    rudder -[ ]-> output {          -- compute desired rudder angle
      MathLib::angle!(- currYaw, x);
      if (abs(x) > 1.0) MathLib::min!(abs(x) * 0.8, 30.0, d)
      else d := x * x * 0.8 end if;
      if (x<0) d:=-d endif; MathLib::angle!(d, rudder_goal)};
    output -[ ]-> init {curr_dr := currDir}; **};
end MaincontrollerThread.impl;
```

7.3 Real-Time Maude Semantics

This section summarizes the formal semantics of Multirate Synchronous AADL in Real-Time Maude (more details can be found in Appendix C.1).

7.3.1 Real-Time Maude Representation

The Real-Time Maude semantics is defined in an object-oriented style, where a Multirate Synchronous AADL component instance is represented as an object instance of a subclass of the following class `Component`:

```
class Component | features : Configuration,
                  subcomponents : Configuration,
                  connections : Set{Connection},
                  properties : PropertyAssociation .
```

The attribute `features` represents the ports of a component as a multiset of `Port` objects; `subcomponents` denotes its subcomponents as a multiset of `Component` objects; `properties` denotes its *properties*; and `connections` denotes its connections. The hierarchical structure of AADL components is reflected in the nested structure of objects in which an attribute of an object contains its subcomponents as a multiset of objects.

A component whose behavior is given by its subcomponents, such as a *system* or a *process*, is represented as an instance of a subclass of `Ensemble`:

```
class Ensemble .
class System .
class Process .
subclass System Process < Ensemble < Component .
```

The `Thread` class contains the attributes for the thread's behavior. The attribute `variables` denotes the local *temporary* variables of the thread component, `transitions` denotes its transitions, `currState` denotes the current state, and `completeStates` denotes its *complete* states. A transition system is represented as a semicolon-separated set of transitions, each of which has the form s `-[`*guard*`]->` s' `{`*actions*`}` with s a source state, s' a destination state, *guard* a Boolean condition, and `{`*actions*`}` an action block:

```
class Thread | variables : Set{VarId},
               transitions : Set{Transition},
               currState : Location,
               completeStates : Set{Location} .
subclass Thread < Component .
```

The *data* subcomponents of a thread can specify the thread's local *state* variables, whose `value` attribute denotes its current value v, expressed as the term `[`v`]`, where the constant `bot` denotes the "don't care" value $\perp$:

```
class Data | value : DataContent .
subclass Data < Component .

sorts DataContent Value .
subsort Value < DataContent .
op bot : -> DataContent [ctor] .
op [_] : Bool -> Value [ctor] .
op [_] : Int -> Value [ctor] .
op [_] : Float -> Value [ctor] .
```

A *data port* is represented as an object instance of a subclass of the class `Port`, whose `content` attribute contains a *list* of data contents (either a value or $\perp$) and `properties` can denote its input adaptor. The subclasses `InPort` and `OutPort` denote input and output data ports, respectively. An input data port also contains the `cache` to keep the previously received "value"; if an input port p received $\perp$ in the latest dispatch, then the thread can use a value in the `cache`, while the expression p`'fresh` becomes *false*:

```
class Port |  content : List{DataContent},
              properties : PropertyAssociation .
class InPort | cache : DataContent .
class OutPort .
subclass InPort OutPort < Port .
```

A *connection set* of a component is a semicolon-separated set, each of which has the form *source* `-->` *target*. A connection from an output port p_1 in a subcomponent c_1 to an input port p_2 in c_2 is represented as a term c_1 `..` p_1 `-->` c_2 `..` p_2. Similarly, a connection c `..` p `-->` p' (resp., p' `-->` c `..` p) represents a level-up (resp., level-down) connection, linking a port p in a subcomponent c with the corresponding port p' in the "current" component (the double dots `..` is used to avoid parsing problems):

```
sort Connection FeatureRef .
subsort FeatureId < FeatureRef .
op _-->_ : FeatureRef FeatureRef -> Connection [ctor] .
op _.._ : ComponentRef FeatureId -> FeatureRef [ctor] .

sort ComponentRef .
subsort ComponentId < ComponentRef .
op _._ : ComponentRef ComponentRef -> ComponentRef [ctor assoc] .
```

Example 7.2. *An instance of the `TurningController.impl` component in our airplane controller example can be represented by the term:*

```
< turnCtrl : System |
   features : < pilot_goal : InPort | content : [0.0], cache : [0.0],
                 properties : InputAdaptor => {use in first iteration} >
             < curr_dr : OutPort | content : [0.0], properties : none >
   subcomponents : < mainCtrl : System | ... >
                   < leftCtrl : System | ... >
                   < rightCtrl : System | ... >
                   < rudderCtrl : System | ... >,
   connections : leftCtrl .. curr_angle --> mainCtrl .. left_angle ;
                 rightCtrl .. curr_angle --> mainCtrl .. right_angle ;
                 rudderCtrl .. curr_angle --> mainCtrl .. rudder_angle ;
                 mainCtrl .. left_goal --> leftCtrl .. goal_angle ;
                 mainCtrl .. right_goal --> rightCtrl .. goal_angle ;
                 mainCtrl .. rudder_goal --> rudderCtrl .. goal_angle ;
                 pilot_goal --> mainCtrl .. goal_angle ;
                 mainCtrl .. curr_dr --> curr_dr,
   properties : Period => {60} >
```

Similarly, an instance of the thread component `SubcontrollerThread.impl` can be represented by the following term, where some identifiers in behavior transitions are enclosed by `{...}` or `[...]` for parsing purposes:

```
< ctrlThread : Thread |
   features : < goal_angle : InPort | content : bot, cache : [0.0],
                                      properties : none >
             < curr_angle : OutPort | content : [10.0], properties : none >
   subcomponents : < currAngle : Data | value : [10.0], ... >
                   < goalAngle : Data | value : [0.0], ... >
                   < diffAngle : Data | value : [1.0], ... >,
   connections : none,
   properties : Period => {15},
   variables : empty,     currState : move,     completeStates : init,
   transitions :
      init -[on dispatch]-> move { skip } ;
      move -[abs([goalAngle] - [currAngle]) > [diffAngle]]-> update {
         if (([goalAngle] - [currAngle]) >= [0])
            {currAngle} := [currAngle] + [diffAngle]
         else
            {currAngle} := [currAngle] - [diffAngle]
         end if } ;
      move -[ otherwise ]-> update { {currAngle} := [goalAngle] } ;
      update -[ [true] ]-> init {
         if (fresh(goal_angle)) {goalAngle} := [goal_angle] end if ;
         {curr_angle} := [currAngle] } >
```

7.3.2 Thread Behavior

The behavior of a single component is specified using the *partial* function `executeStep`, by means of equations (for deterministic components) or rules (for nondeterministic components):

```
op executeStep : Object ~> Object .
```

Since a term containing `executeStep` will *not* have a sort, this is used to ensure that an equation or a rule is only applied to an object of sort Object in which the transitions have already been performed in all subcomponents. The following rule defines the behavior of *nondeterministic* threads (the *equation* for *deterministic* threads is quite similar to this rewrite rule):

```
vars PORTS PORTS' DATA DATA' : Configuration .
var VARS : Set{VarId} .            vars FMAP FMAP' : FeatureMap .
vars L L' : Location .             var LS : Set{Location} .
var PROPS : PropertyAssociation .  var TRS : Set{Transition} .

crl [execute]:
   executeStep(< C : Thread | features : PORTS,
                              subcomponents : DATA,
                              currState : L,
                              completeStates : LS,
                              transitions : TRS,
                              variables : VARS,
                              properties : PROPS >)
 =>
   < C : Thread | features : writeFeature(FMAP', PORTS', none),
                  subcomponents : DATA',
                  currState : L' >
if Nondeterministic => true in PROPS
/\ PORTS' | FMAP := readFeature(PORTS, none, empty)
/\ execTrans(L, LS, TRS, VARS, FMAP | DATA | PROPS)
                                          => L' | FMAP' | DATA' .
```

The function `readFeature` returns a map from each input port to its current value (i.e., the first value of the data content list), while removing the value from the input port and using the *cached* value if the value is $\bot$. Then, any possible computation result of the thread's transition system—based on the temporary variables `VARS`, the input port values `FMAP`, the state variables `DATA`, and the property values `PROPS`—is nondeterministically assigned to the pattern `L' | FMAP' | DATA'` using the operator `execTrans`, which defines the semantics of behavior transitions. Finally, the function `writeFeature` updates the content of each output port from the result.

The meaning of the operator `execTrans` is that transitions are repeatedly applied until a *complete* state is reached:

```
crl [trans]:
   execTrans(L, LS, TRS, VARS, FMAP | DATA | PROPS)
 =>
   if (L' in LS) then L' | FMAP' | DATA'
   else  execTrans(L', LS, TRS, VARS, FMAP' | DATA' | PROPS) fi
if (L -[GUARD]-> L' ACTION) ; TRS' :=
                    enabledTrans(L, TRS, FMAP | DATA | PROPS, empty)
/\ FMAP' | DATA' | PROPS :=
                    execAction(ACTION, VARS, FMAP | DATA | PROPS) .
```

The function `enabledTrans` finds all *enabled* transitions from the current state `L` whose `GUARD` evaluates to *true*, and *any* of these enabled transitions is nondeterministically assigned to the pattern `(L -[GUARD]-> L' ACTION)`. The function `execAction` executes the actions of the chosen transition and returns a new configuration. If the next state `L'` is *not* a *complete* state (`else` branch), then `execTrans` is applied again with the new configuration (see Appendix C.1 for more details and definitions).

7.3.3 Ensemble Behavior

For *ensemble* components such as processes and systems, their synchronous behavior is also defined by using `executeStep`:

```
crl [execute]: executeStep(< C : Ensemble | >)
           =>  transferResults(OBJ')
 if OBJ := applyAdaptors(transferInputs(< C : Ensemble | >))
 /\ prepareExec(OBJ) => OBJ' .
```

First, each input port of a subcomponent receives a value from its source port (`transferInputs`). Second, an input adaptor is applied to each input port (`applyAdaptors`). Third, the operator `executeStep` is applied multiple times to each subcomponent according to its period (`prepareExec`). Next, any term of sort Object resulting from rewriting `prepareExec(OBJ)` in zero or more steps is nondeterministically assigned to `OBJ'` of sort Object. Since `executeStep` does *not* yield terms of this sort, `OBJ'` will only capture an object where `executeStep` has been completely evaluated. Finally, the new outputs are transferred to the output ports of `C` (`transferResults`). This rule is isomorphic to the `sync` rule for the *multirate synchronous composition* in Section 6.4.2; we refer to it for more details about this rule.

A multirate synchronous step of the entire system is then formalized by the following conditional tick rewrite rule:

```
crl [step]:
    {< C : System | properties : Period => {T} ;
                                 Synchronous => {true} ; PROPS,
                  features : none >}
 => {SYSTEM} in time T
 if executeStep(< C : System | >) => SYSTEM .
```

Any term of sort Object, in which `executeStep` is completely evaluated, resulting from rewriting `executeStep(< C : System | >)` in zero or more steps can be nondeterministically assigned to the variable `SYSTEM`.

7.4 The MR-SynchAADL Tool

To support the modeling and verification of Multirate Synchronous AADL models within OSATE, this section presents the *MR-SynchAADL* plugin that: (i) checks whether a given model is a *valid* Multirate Synchronous AADL model; (ii) provides a simple language to specify *system requirements*; (iii) automatically synthesizes a Real-Time Maude model from a Multirate Synchronous AADL model; and (iv) uses Real-Time Maude to both simulate the execution of the Multirate Synchronous AADL model and model check whether the model satisfies the given system requirements.

The MR-SynchAADL tool provides a requirements specification language that allows the user to easily define system requirements, without having to understand Real-Time Maude. The requirements specification language defines a number of useful atomic propositions. The proposition

full component name `@` *location*

holds in a state when the thread identified by the full component name is in state *location*. A full component name is a component path in the AADL syntax, a period-separated path of identifiers. Similarly, the proposition

full component name `|` *Boolean expression*

holds in a state if *Boolean expression* evaluates to *true* in the component. Any Boolean expression can be used in the AADL behavior annex syntax involving data components, feedback output ports, and property values. The semantics of the requirements specification language is defined by using equations in Real-Time Maude (see Appendix C.1).

In MR-SynchAADL, we can easily declare *formulas* and *requirements* for Multirate Synchronous AADL models as LTL formulas, using the usual Boolean connectives and temporal logic operators. They can be declared using the following syntax, where LTL formulas can also contain references to other "formulas" defined by **formula** statements:

formula *name*: *proposition*; **formula** *name*: *LTL formula*;
requirement *name*: *LTL formula*;

For the airplane turning control system example, the declaration

```
formula safeYaw:
    turnCtrl.mainCtrl.ctrlProc.ctrlThread | abs(currYaw) < 1.0;
```

states that the proposition **safeYaw** holds when the current yaw angle in the main controller is less than 1°. The following *requirement* defines the safety requirement of the system: *the yaw angle should always be close to* 0°.

```
requirement safety: [] safeYaw;
```

Figure 7.2 shows the MR-SynchAADL window for the airplane example. In the editor part, two requirements are specified using the requirements specification language. The `Constraints Check`, the `Code Generation`, and the `Perform Verification` buttons are used to perform, respectively, the syntactic validation, the Real-Time Maude code generation, and the LTL model checking. The `Perform Verification` has been clicked and the results are shown in the "Maude Console."

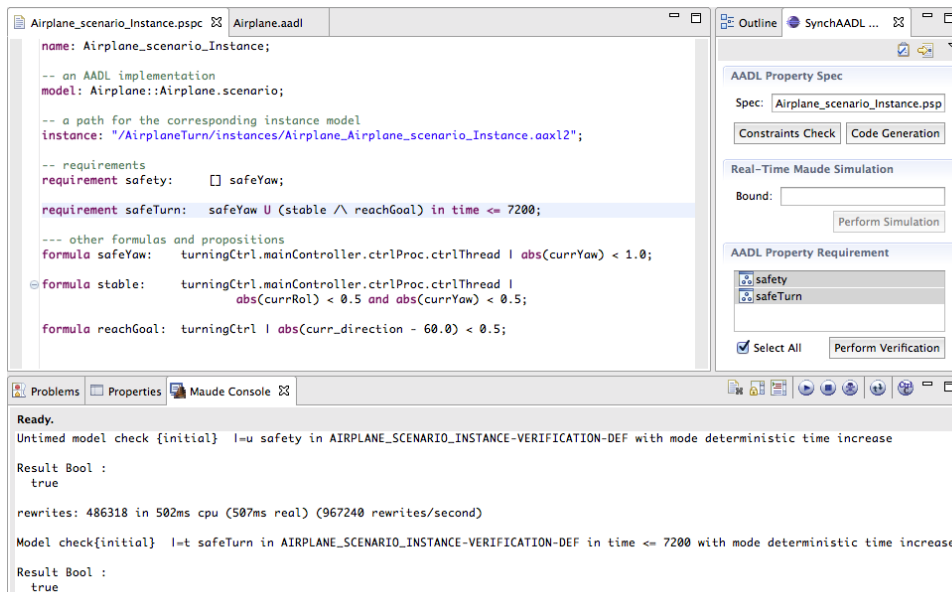


Figure 7.2: MR-SynchAADL window in OSATE.

7.5 Case Studies

This section presents two Multirate Synchronous AADL models and their verification in OSATE using the MR-SynchAADL plugin. Section 7.5.1 shows how the Multirate Synchronous AADL model of the airplane turing controller can be verified in MR-SynchAADL, and Section 7.5.2 presents a Multirate Synchronous AADL model of the *active standby* system in [143] (another avionics case study can be found in Appendix C.3).

7.5.1 The Airplane Turing Controller Revisited

As mentioned in Section 6.5.4, the airplane turing controller system must satisfy the system requirement: *the airplane must reach the desired direction with a stable status within reasonable time, while keeping the yaw angle close to* 0°. In order to verify whether the airplane can reach a *specific* goal direction, we first consider a *deterministic* pilot given by the following AADL implementation, where the pilot gradually turns the airplane 60° to the right by adding 10° to the goal direction 6 times, instead of using the nondeterministic pilot in Section 7.2.3:

```
thread implementation PilotConsoleThread.scenario
  subcomponents
    counter: data Base_Types::Integer;
  properties
    Data_Model::Initial_Value => ("0") applies to counter;
  annex behavior_specification {**
    states
      idle: initial complete state;
      select: state;
    transitions
      idle -[on dispatch]-> select;
      select -[counter >= 6]-> idle;
      select -[counter < 6]-> idle {
        goal_dr := 10.0; counter := counter + 1};
  **};
end PilotConsoleThread.scenario;
```

The desired requirement, with the additional constraint that the desired state must be reached within 7,200 ms, can be formalized as an LTL formula in MR-SynchAADL as follows, where `safeYaw` is defined in Section 7.4, `stable` holds if both roll and yaw angles are close to 0, and `reachGoal` holds if the current direction is close to 60°:

Bound (ms)	# States	Time (s)	Bound	# States	Time
$\leq$ 3,000	364	7	$\leq$ 4,800	9,841	189
$\leq$ 3,600	1,093	21	$\leq$ 5,400	29,524	600
$\leq$ 4,200	3,280	62	$\leq$ 6,000	88,573	2,323

Table 7.1: The model checking result for the nondeterministic pilot.

```
requirement safeTurn:
       safeYaw U (stable /\ reachGoal) in time <= 7200;

formula stable: turnCtrl.mainCtrl.ctrlProc.ctrlThread |
                     abs(currRol) < 0.5 and abs(currYaw) < 0.5;

formula reachGoal: turnCtrl | abs(curr_dr - 60.0) < 0.5;
```

Figure 7.2 shows the model checking results for the two requirements `safety` ($\Box$ `safeYaw`, declared in Section 7.4) and `safeTurn` in our tool. In the deterministic scenario, the airplane controller satisfies both properties as displayed in the Maude console. These model checking analyses took 1.6 and 0.5 seconds, respectively, on Intel Core i5 2.4 GHz with 4 GB memory and the numbers of states explored are 59 and 13.

We have verified the `safety` requirement for the nondeterministic pilot and have summarized the results in Table 7.1, which shows a huge state space reduction compared to the asynchronous model: for the same pilot behavior and time bound 3,000 ms, the number of reachable states in the *simplest possible* distributed asynchronous model—with perfect local clocks and no network delays—in Section 6.5.5 is 420,288, whereas there are only 364 reachable states in the Multirate Synchronous AADL model.

7.5.2 The Active Standby System

In *integrated modular avionics* (IMA), a cabinet is a chassis with a power supply, internal bus, general purpose computing, etc. Aircraft applications are implemented using the resources in the cabinets. There are always two or more physically separated cabinets so that physical damage does not take out the computer system. The *active standby* system considers the case of two cabinets and focuses on the logic for deciding which side is *active*. Each side can fail, and a failed side can recover after failure. In case one side fails, the non-failed side should be the active side. The pilot can also toggle the active status of the sides. The full functionality of each side depends on the two sides' perception of the availability of other system components.

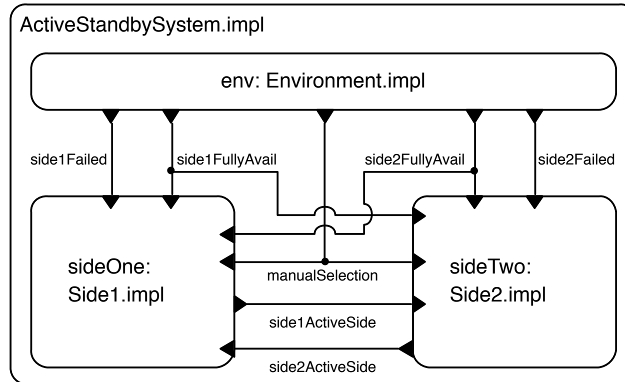


Figure 7.3: The architecture of the active standby system.

The architecture of the active standby system is in Figure 7.3, based on a specification by Steve Miller and Darren Cofer at Rockwell-Collins [143]. Each time `Environment` dispatches, it nondeterministically sends 5 Boolean values, one through each ports, so that two sides cannot fail at the same time: i.e., it can send any one of 24 different 5-tuples in each round.

The following top-level system implementation declares the architecture of the entire system in Figure 7.3, which contains the three subcomponents `sideOne`, `sideTwo`, and `env` (a part is replaced by '`...`'):

```
system implementation ActiveStandbySystem.impl
  subcomponents
    sideOne: system Side1.impl;      sideTwo: system Side2.impl;
    env: system Environment.impl;
  connections
    C1: port sideOne.side1ActiveSide -> sideTwo.side1ActiveSide;
    C2: port sideTwo.side2ActiveSide -> sideOne.side2ActiveSide;
    C3: port env.side1FullyAvail -> sideOne.side1FullyAvail;
    ...
    C9: port env.side1Failed -> sideOne.side1Failed;
    C10: port env.side2Failed -> sideTwo.side2Failed;
  properties
    MR_SynchAADL::Synchronous => true;
    Period => 2 ms;
    Timing => Delayed applies to C1,C2,C3,C4,C5,C6,C7,C8,C9,C10;
    Data_Model::Initial_Value => ("true") applies to
      env.side1FullyAvail, env.side2FullyAvail;
    Data_Model::Initial_Value => ("false") applies to
      env.side1Failed, env.side2Failed, env.manualSelection;
end ActiveStandbySystem.impl;
```

The environment thread has a single transition that sends the values of the local variables to the corresponding output ports. These variables can be assigned any values satisfying the input constraint `not (s1F and s2F)`, stating that both sides cannot fail at the same time:

```
system Environment
  features
    side1FullyAvail: out data port Base_Types::Boolean;
    side2FullyAvail: out data port Base_Types::Boolean;
    manualSelection: out data port Base_Types::Boolean;
    side1Failed: out data port Base_Types::Boolean;
    side2Failed: out data port Base_Types::Boolean;
end Environment;

thread implementation EnvironmentThread.impl
  properties
    MR_SynchAADL::InputConstraints => ("not (s1F and s2F)");
    MR_SynchAADL::Nondeterministic => true;
    Dispatch_Protocol => Periodic;
  annex behavior_specification {**
    states            s0 : initial complete state;
    variables         s1FA, s2FA, mS, s1F, s2F: Base_Types::Boolean;
    transitions
      s0 -[on dispatch]-> s0 {       side1FullyAvail := s1FA;
        side2FullyAvail := s2FA;     manualSelection := mS;
        side1Failed := s1F;          side2Failed := s2F; };  **};
end EnvironmentThread.impl;
```

The system `sideOne` contains an instance of the following thread that defines the behavior of side 1. We show only one transition in this thread, which takes the thread from state `side2_active_t` to `side1_active`. As a result, the output 1 is sent through the port `side1ActiveSide`, and the state variables `prevSide2` and `prevMS` are assigned the values in the ports `side2ActiveSide` and `manualSelection`, respectively:

```
thread Side1Thread
  features
    side1Failed: in data port Base_Types::Boolean;
    side1FullyAvail: in data port Base_Types::Boolean;
    side2FullyAvail: in data port Base_Types::Boolean;
    manualSelection: in data port Base_Types::Boolean;
    side2ActiveSide: in data port Base_Types::Integer;
    side1ActiveSide: out data port Base_Types::Integer;
end Side1Thread;
```

```
thread implementation Side1Thread.impl
  subcomponents
    prevSide2: Base_Types::Integer;    prevMS: Base_Types::Boolean;
  properties
    Dispatch_Protocol => Periodic;
    Data_Model::Initial_Value => ("0") applies to prevSide2;
    Data_Model::Initial_Value => ("false") applies to prevMS;
  annex behavior_specification {**
    states
      init: initial complete state;
      side1_failed, side2_failed, ..., side2_active: complete state;
      ...
    transitions
      ...
      side2_active_t -[side1Failed = false and side2ActiveSide != 0 and
          side1FullyAvail = true and (side2FullyAvail = false or
          (prevMS = false and manualSelection = true))]-> side1_active
      {side1ActiveSide := 1;
       prevSide2 := side2ActiveSide;  prevMS := manualSelection;};
      ...
  **};
end Side1Thread.impl;
```

The paper [143] describes five requirements of the active standby system. This section explains how we have verified one of these requirements (R_1): *Both sides should agree on which side is active, provided that the availability of a side has not changed, neither side has failed, and the pilot has not made a manual selection* (see Appendix C.2 for the other properties). As explained in [138], the requirement $R1$ is not satisfied in the active standby system; instead, we have verified the following weaker property:

```
requirement R1: O ([] (
    (noChangeAvail /\ O (neitherSideFailed /\ ~ manSelectPressed))
 -> O (agreeOnActiveSide \/
             O (neitherSideFailed -> agreeOnActiveSide)) ));
```

Side i thinks that side j is active if it sends the number j to its output port `side`i`ActiveSide`. The formula `agreeOnActiveSide` holds when both sides think that side 1 is active or side 2 is active:

```
formula agreeOnActiveSide:
    (sideOne.sideProcess.sideThread | side1ActiveSide = 1 /\
     sideTwo.sideProcess.sideThread | side2ActiveSide = 1)
\/  (sideOne.sideProcess.sideThread | side1ActiveSide = 2 /\
     sideTwo.sideProcess.sideThread | side2ActiveSide = 2);
```

Side i has failed if it has received the value `true` in its side*i*Failed port. We also define a property that the pilot has made a manual selection:

```
formula side1Failed:
   sideOne.sideProcess.sideThread | side1Failed = true;
formula side2Failed:
   sideTwo.sideProcess.sideThread | side2Failed = true;
formula neitherSideFailed:
   (~ side1Failed) /\ (~ side2Failed);
formula manSelectPressed:
   sideOne.sideProcess.sideThread | manualSelection = true;
```

Likewise, the proposition side*i*FullyAvailable holds if side i is fully available. There is no change in availability if both sides are equally available in the current state and in the next state:

```
formula side1FullyAvailable:
   sideOne.sideProcess.sideThread | side1FullyAvail = true;
formula side2FullyAvailable:
   sideTwo.sideProcess.sideThread | side2FullyAvail = true;
formula noChangeAvail:
   (side1FullyAvailable <-> O side1FullyAvailable)
 /\ (side2FullyAvailable <-> O side2FullyAvailable);
```

We have verified every requirement of the Multirate Synchronous AADL model. Each model checking analysis, generating 203 system states, took 0.6 seconds on an Intel Xeon 2.93 GHz with 24GB RAM. Note that it is unfeasible to model check the corresponding *asynchronous* design: as shown in [138], the *simplest possible* asynchronous model (no message delays, no execution times, etc.) has 3,047,832 reachable states. If the message delay can be 1 then no model checking terminates in reasonable time.

7.6 Concluding Remarks

This chapter has explained how Multirate PALS has been made available to AADL modelers by: (i) defining Multirate Synchronous AADL, which allows the modelers to specify synchronous designs in AADL; (ii) giving a formal semantics for Multirate Synchronous AADL, which allows simulation and model checking of Multirate Synchronous AADL models; (iii) providing an intuitive way of specifying temporal logic *requirements* that such models should satisfy; and (iv) integrating both modeling and automated model checking into the OSATE tool environment for AADL.

CHAPTER 8

PTOLEMY II DISCRETE-EVENT MODELS

This chapter presents[1] a rewriting logic semantics for a significant subset of Ptolemy II discrete-event (DE) models. This is a challenging task, since DE models combine a synchronous fixed-point semantics with hierarchical structure, explicit time, and a rich expression language. The code generation features of Ptolemy II have been leveraged to automatically synthesize a Real-Time Maude verification model from a Ptolemy II design model, and to integrate the formal verification of the synthesized model into Ptolemy II. This enables a model-engineering process that combines the convenience of Ptolemy II DE modeling and simulation with formal verification in Maude. We illustrate such formal verification of Ptolemy II models with case studies.

8.1 Introduction

Model-based design [113, 159, 160] emphasizes the construction of high-level models for system design. Useful models typically provide simulations of system functionality and performance during the design phases as a much less costly alternative to building prototypes and testing them. Model-based design generally raises the level of abstraction in system design; specifically, for embedded software, from low-level languages (such as C++ and Java) to high-level modeling formalisms where concepts like concurrency and time are first-class notions. This makes it feasible to design systems that would be hard to design using low-level methods. Ideally, models are translated (code generated) automatically to produce deployable software. AADL, introduced in Chapter 7, is one of such model-based design languages.

[1] This chapter is based on the papers [27, 29, 30], joint work with Peter Ölveczky, Thomas Huining Feng, Stavros Tripakis, and Edward A. Lee.

Ptolemy II [79] is a well established open-source modeling and simulation tool used in industry. A major reason for its popularity is Ptolemy II's powerful yet intuitive graphical modeling language that allows a user to build hierarchical models that combine different models of computations. In this chapter we focus on *discrete-event* (DE) models; such models are explicit about the timing behavior of systems, which is an essential feature for the high-level specification of embedded system applications [106, 107]. Discrete-event modeling is a widely used approach for system simulation [96], and it has been proposed as basis for synthesis of embedded real-time software [169]. The Ptolemy II DE models have a semantics rooted in the fixed-point semantics of synchronous languages [125].

As already mentioned, many *embedded* systems—in areas such as avionics, motor vehicles, robotics, and medical systems—are *safety-critical* systems, whose failures may cause great damage to persons and/or valuable assets. Models of such embedded systems should therefore be formally analyzed to verify safety requirements. Instead of requiring designers to develop models in some formal framework, a promising approach to formally verify design models is to add formal analysis capabilities to the intuitive, often graphical, *informal* modeling languages preferred by practitioners by:

- providing a formal semantics for the informal modeling language;
- leveraging the code generation features of the modeling framework to automatically translate an design model to a formal model; and
- verifying the synthesized formal model, e.g., using Real-Time Maude.

However, as usual for many graphical modeling languages, Ptolemy II DE models lack formal verification capabilities. Despite already having a synchronous semantics in contrast to AADL, Ptolemy II DE models seem to fall outside the class of languages which can be given an automata-based semantics, because: (i) the variables range over infinite domains such as the integers; (ii) certain Ptolemy II constructs use unbounded data structures; (iii) executing a synchronous step requires fixed-point computations; and (iv) Ptolemy II has a powerful expression language. Rewriting logic is in this case a suitable formalism, since its expressiveness allows us to give a formal semantics to languages with advanced functions and data types, unbounded data structures, variables over unbounded domains, etc.

8.1.1 Main Contributions

This chapter defines a Real-Time Maude semantics for a significant subset of *hierarchical* Ptolemy II DE models. A Ptolemy II DE model is a hierarchical composition of *actors* with connections between the actors' input ports and output ports. The supported subset includes: (i) *finite state machine* (FSM) actors, "guarded" transition systems with variables; (ii) *composite* actors, Ptolemy II models encapsulated as single actors; (iii) *modal model* actors, state machines where each state has a *refinement* actor, either a composite actor or an FSM actor; and (iv) *atomic actors*, such as clock actors, timer actors, delay actors, algebraic expression actors, etc.

This chapter shows how rewriting-based model checking of Ptolemy II DE models is integrated into Ptolemy II, so that Ptolemy II DE models can be formally analyzed from within Ptolemy II. We define a property specification language so that the users can easily define their temporal logic requirements without having to understand the underlying formal representation. Hence, this provides a model-engineering process that combines the convenience of Ptolemy II modeling with formal verification in Real-Time Maude. This Real-Time Maude integration is officially available in version 8.0.1 (or later) of Ptolemy II. The entire Real-Time Maude semantics, as well as the case studies in this chapter, can be found in the Ptolemy II source code, available at `http://ptolemy.eecs.berkeley.edu/ptolemyII`.

8.1.2 Related Work

The semantics of Ptolemy II is often given in terms of *abstract semantics* which consists of a set of functions such as "initialize","fire", "postfire", and so on [79, 125]. Denotational semantics of DE models based on metric spaces are given in [48, 123, 127]. A different type of denotational semantics, based on complete partial orders and domain theory, are given in [35, 126]. These semantics differ from ours, e.g., in that they are not executable and therefore cannot be used for formal model checking analyses.

A preliminary exploration of translations of *synchronous reactive* (that is, untimed) Ptolemy II models into Kripke structures, which can be analyzed by using the NuSMV model checker, and of DE models into communicating timed automata is given in [51]. However, they require *data abstraction* to map models into finite automata, in order to deal with integer variables in FSM actors, and they do not use the code generation framework. We refer to [29] for other model transformation approaches of embedded systems.

8.1.3 Structure of this Chapter

This chapter is organized as follows. Section 8.2 introduces Ptolemy II DE models and explains their operational semantics. Section 8.3 defines the formal semantics of Ptolemy II DE models in Real-Time Maude, and Section 8.4 explains how Real-Time Maude verification has been integrated into Ptolemy II, allowing users to easily check desired system requirements. Section 8.5 illustrates such formal verification of Ptolemy II models with case studies, and Section 8.6 gives some concluding remarks.

8.2 Ptolemy II and its DE Model of Computation

Ptolemy II is a modeling environment for embedded systems to support multiple modeling paradigms, called *models of computations* (MoCs), that govern the interaction between concurrent components. Such MoCs include FSM (finite state machine), dataflow, and DE (discrete events), and can be composed to create heterogeneous models. A Ptolemy II model consists of a set of interconnected *actors*. *Ports* represent points of communication for an actor, and *parameters* are used to configure the operation of an actor.

A composition of actors can be encapsulated as an actor in its own right, which may also have input and output ports. Such an actor, obtained by composition, is called a *composite actor*. An output port of an inner actor can be connected to an output port of its enclosing composite actor. An input port of a composite actor can be connected to input ports of the actors inside, which means that external inputs are transferred to those inner actors. An actor that is not composite is called an *atomic actor*.

8.2.1 Discrete-Event Models

This chapter focuses on the formalization of Ptolemy II *discrete-event* (DE) models. In DE, the data sent and received at actors' ports are *events*. Each event has two components: a *tag* and a *value*. According to the *tagged signal model* [124], a tag t is a pair $(\tau, n) \in \mathbb{R}_{\geq 0} \times \mathbb{N}$, where τ is the *timestamp* denoting the model time at which the event occurs, and n is the *microstep index*. Microstep indices are useful for modeling multiple events with the same timestamps happening in sequence, where earlier events may cause later ones. Tags are totally ordered using a lexicographical order: that is, $(\tau_1, n_1) \leq (\tau_2, n_2)$ iff $\tau_1 < \tau_2$, or $\tau_1 = \tau_2$ and $n_1 \leq n_2$. Two events are *simultaneous* if they have identical tags.

```
Q := empty; // initialize the global event queue to be empty.
for each actor A do
  A.init(); // initialize actor A, and generate initial events in Q.
end for;
while Q is non-empty do
  E := set of all simultaneous events at the head of Q;
  remove E from Q;
  initialize ports with values in E or "unknown";
  while port values changed do
    for each actor A do
      A.fire();  // may change port values .
    end for;
  end while;     // fixed-point reached for the current tag.
  for each actor A do
    A.postfire(); // updates state, and generates new events in Q.
  end for;
end while;
```

Figure 8.1: Pseudo-code of Ptolemy II DE semantics.

The operational semantics of DE in Ptolemy II can be explained with the pseudo-code in Figure 8.1. Events in the *event queue* are ordered by their tags. Initially, the event queue is empty. At the beginning of the execution, all actors are initialized, and some actors may post initial events to the event queue. Operation then proceeds by iterations. In each iteration, the events with the smallest tag are extracted from the event queue and presented to the actors receiving them. Those actors are *fired*, i.e., they are invoked to process their input events, and they may also output events through their output ports. Finally, when the fixed-points for the port values have been found, the actors that have received input or have been fed events are executed, in the sense that their states are updated and that they may generate future events that are inserted into the event queue (*postfire*).

The DE MoC in Ptolemy II is different from standard DE simulators, since the Ptolemy II DE MoC incorporates a synchronous-reactive semantics for processing simultaneous events [125]. When events are extracted from the event queue for the receiving actors to process, the semantics for that iteration is defined as the *least fixed-point* of the output values, in a way similar to a *synchronous* model [78]. The outputs are first set to "unknown," and the actors receiving events are fired in an arbitrary order, possibly repeatedly, until a fixed-point of all output values is reached. This semantics allows Ptolemy II models to have *feedback loops*. If the model contains *causality cycles*, the fixed-point may have ports with value *unknown*.

8.2.2 Ptolemy II Actors

This section explains a subset of the Ptolemy II actors whose semantics has been formalized in Real-Time Maude. Their semantics is defined in terms of the actions *init*, *fire*, and *postfire* (see also Appendix D.1 for more actors whose Real-Time Maude semantics has been defined).

Current Time. Ptolemy II's *current time* actor produces an output token on each firing with a value that is the current model time. That is, the *init* and *postfire* actions do nothing, and the *fire* action consumes an input event and outputs an event given by the input event.

Timed Delay. A *timed delay* actor propagates an incoming event after a given delay. If the *delay* parameter is 0.0, then there is a "microstep" delay in the generation of the output event. The *init* and *fire* actions do nothing, but *postfire* generates an event with a delay equal to the *delay* parameter.

Clock. Ptolemy II's *clock* actors have as parameters a *clock period*, and same-sized arrays *values* and *offsets*. In each period, a clock generates events with given values and offsets within the period. If the period is p, then, for each $n \geq 0$ and $i \leq \mathit{length}(\mathit{values})$, the clock generates an event with value $\mathit{values}(i)$ at time $n \cdot p + \mathit{offsets}(i)$. The *init* action posts an event to the event queue with timestamp 0 for itself, the *fire* action is triggered by that event and sends the value to the output port, and the *postfire* action posts the next event, with timestamp equal to the beginning of the next period.

Set Variable. A *set variable* actor contains a name of a parameter in its *container* actor. The *fire* action outputs the value of the parameter, and *postfire* updates the variable if a new value has been received.

Finite State Machine (FSM) Actor. A *finite state machine* actor is a transition system containing a finite set of states (or "locations"), a finite set of "variables," and a finite set of "guarded" transitions. A transition has a guard expression, and can contain a set of output actions. Output actions may assign values to the variables belonging to the FSM actor and/or may send values to the output ports of the actor. It is assumed in Ptolemy II that there is never more than one enabled transition when an FSM actor is fired. If there is exactly one enabled transition, then it is chosen and the actions contained by the transition are executed. Under the DE director, only one transition step is performed in each iteration.

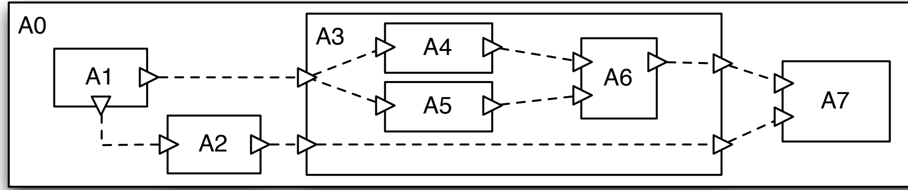


Figure 8.2: A hierarchical composition of actors, where $A0$–$A7$ are actors, $A0$ and $A3$ are composite actors, and triangles are ports.

Composite Actors. Ptolemy II *hierarchical* models contain components (or *actors*) that are themselves Ptolemy II models. Such a hierarchical model can be seen as a single *composite actor*. The ports of a composite actor can be connected to its inner actors so that the sub-model interacts with the outside. Figure 8.2 illustrates a hierarchical composition of actors.

Each composite actor can have its own model of computation, given by the *director* of the actor, to support heterogeneous modeling. If the director of a composite actor is the same as the director of the parent actor, then it is called a *transparent* actor. This chapter considers only transparent cases, since we verify DE models. In these cases, a DE composite actor is *fired* when a new value has arrived to its input port, and an inner actor of a DE composite actor is *fired* when that inner actor receives some events at its input ports or if it is fed an event from the (global) event queue.

Modal Models. *Modal models* are finite state machines where each state has a *refinement* actor, either a composite actor or an FSM actor. The input and output ports of the refinements are the same as those of the modal model. The output ports of a modal model are regarded as *both* input and output ports, so that the transitions of modal models may use the evaluation result of refinement actors in the *current* computation step.

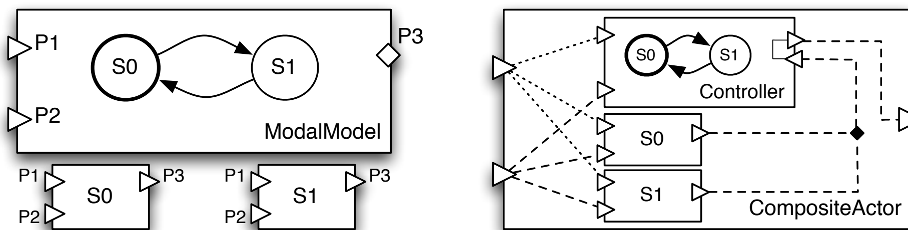


Figure 8.3: A modal model with 2 refinement states (left), and its equivalent representation as a composite actor (right), where $S0$ and $S1$ are refinement states and diamonds denote input/output ports.

A modal model can be seen as syntactic sugar for a composite actor with *frozen* inner actors, as illustrated in Figure 8.3. That is, a modal model A is semantically equivalent to a composite actor $\tilde{A}$ that has the controller FSM actor and the refinement actors as inner actors, so that: (i) the ports are connected as indicated in Figure 8.3; (ii) the controller FSM actor is fired *after* the refinement actors are fired; (iii) only the refinement inner actors corresponding to the current state of the controller are evaluated, whereas the other refinement actors are frozen, in the sense that their states do not evolve and the values of their outports are ignored; and (iv) if an output port of the controller actor has no value but its "coupled" input port has a value, then the output port will have the same value as the input port.

Example 8.1 (A Simple Traffic Light)**.** *Figure 8.4 shows a Ptolemy II DE model of a simple traffic light system. The traffic light system consists of one car light and one pedestrian light at a pedestrian crossing. Each light is represented as a set of* set variable *actors (`Pred` and `Pgrn` for the pedestrian light, and `Cred`, `Cyel`, and `Cgrn` for the car light). A light is* on *iff the corresponding variable has the value 1. The lights are controlled by two FSM actors, `CarLight` and `PedestrianLight`, that send values to set the variables; in addition, `CarLight` sends signals (that are* delayed *by one time unit) to `PedestrianLight` through its `Pgo` and `Pstop` output ports.*

Figure 8.5a shows the FSM actor `PedestrianLight`. This actor has three input ports (`Pstop`, `Pgo`, and `Sec`), two output ports (`Pgrn` and `Pred`), three internal states, and three transitions. This actor reacts to signals from the car light (via the delay actors) by turning the pedestrian lights on and off. For example, if the actor is in local state `Pred` and receives input through its `Pgo` port, then it goes to state `Pgreen`, outputs the value 0 through its `Pred` port, and outputs the value 1 through its `Pgrn` port.

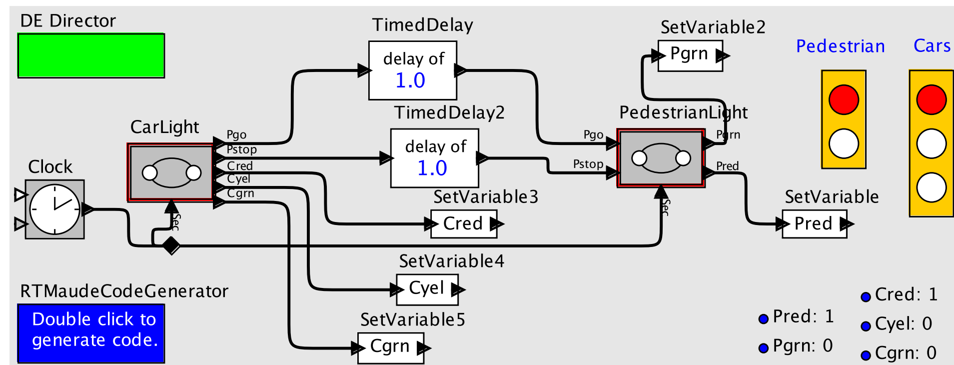
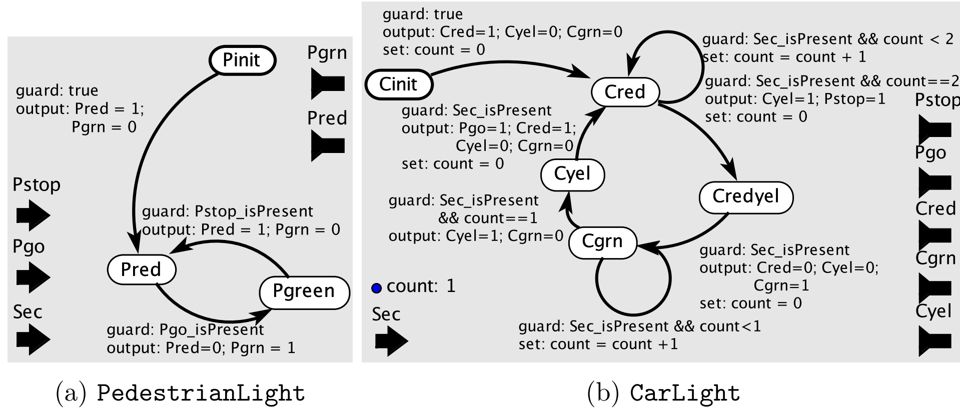


Figure 8.4: A simple traffic light model in Ptolemy II.



(a) `PedestrianLight` (b) `CarLight`

Figure 8.5: The FSM actors for pedestrian lights and car lights.

Figure 8.5b shows the FSM actor ***`CarLight`****. Assuming that the* clock *actor sends a signal every time unit, we notice, e.g., that one time unit after both the red and yellow car lights are on (*`Credyel`*), these are turned off and the green car light is turned on by sending the appropriate values to the variables (*`output: Cred = 0; Cyel = 0; Cgrn = 1`*). The car light then stays green for two time units before turning yellow.*

8.3 The Semantics of Ptolemy II DE Models

This section gives an overview of our Ptolemy II DE semantics in Real-Time Maude (see Appendix D.3 for more details). Section 8.3.1 explains how a generic Ptolemy II DE model is represented as a Real-Time Maude term, Section 8.3.2 presents a semantic framework for specifying the DE semantics, Section 8.3.3 uses this framework to define the semantics of each actor, and Section 8.3.4 shows the semantics of the Ptolemy II expression language.

8.3.1 Representing Ptolemy II DE Models

The Real-Time Maude semantics is defined in an object-oriented style, where the global state has the form of a *multiset*

{*actors* *connections* `< global : EventQueue | queue :` *event queue* `>`}

where: (i) *actors* are objects corresponding to the actor instances in the model; (ii) *connections* are the connections between the ports of the actors; and (iii) `< global : EventQueue | queue :` *event queue* `>` is an object with the `queue` attribute that denotes the global event queue.

Each Ptolemy II actor is represented in Real-Time Maude as an object instance of a subclass of the class `Actor`:

```
class Actor | ports : ObjectConfiguration,
              parameters : ObjectConfiguration,
              computation : Computation,
              status : ActorStatus .
```

The `ports` attribute denotes the *ports* of the actor, and the `parameters` attribute denotes the *parameters* of the actor. Both ports and parameters are modeled as objects. The `computation` attribute denotes a "processor" of the actor to compute *expressions*. The `status` attribute is used to denote *frozen* actors for modal models: the `status` of an actor is either `enabled` or `disabled`, depending on whether the actor is disabled as a result of being contained in a refinement of a "frozen" state in a modal model.

A *port* is represented as an object with a name (a quoted identifier of sort Qid), a `value` (a Ptolemy II expression of sort Exp), and a `status` (`unknown`, `present`, or `absent`, denoting the "current" knowledge about whether there is input or output in the current iteration for the *fire* action):

```
class Port | value : Exp,
             status : PortStatus,
class InPort .
class OutPort .
subclass InPort OutPort < Port .

sorts PortId PortStatus .
subsort Qid < PortId < Oid .        --- local port names
ops unknown present absent : -> PortStatus [ctor] .
```

A *parameter* is also represented as an object. The `exp` attribute denotes an expression, which may include a *variable* that refers to either a parameter or an input port, the `value` attribute denotes the value of such an expression to be used in the *current* iteration (based on the parameter values computed in the previous iteration), and the `next-value` attribute denotes the value that will be used in the *next* iteration (initially `noValue`):

```
class Parameter | exp : Exp,
                  value : Value,
                  next-value : Value? .

sorts ParamId Value Value? Exp .
subsorts Qid < ParamId < Oid .      --- parameter names
subsorts Value < Value? Exp.
op noValue : -> Value? [ctor] .
```

Composite actors are represented as object instances of `CompositeActor`, which extends its superclass `Actor` with one attribute, `innerActors`, to denote the inner actor objects and connections of the composite actor:

```
class CompositeActor | innerActors : ObjectConfiguration .
subclass CompositeActor < Actor .
```

To distinguish *atomic actors* from composite actors, each atomic actor is modeled as an object instance of a subclass of the class `AtomicActor`:

```
class AtomicActor .
subclass AtomicActor < Actor .
```

A connection is given by a term p_o `==>` p_{i_1} `; ... ;` p_{i_n} of sort Connection, where each p_j has the form a `!` p with a the *relative* name of an actor and p a local port name. Such a connection connects the output port p_o to all the input ports $p_{i_1}, \ldots, p_{i_n}$. Since connections appear in configurations, and are not messages, they are also declared to have sort ObjectConfiguration.

Some actors, such as current time actors, have an internal clock that measures "model time." Such actors are represented as object instances of subclasses of the class `TimeActor`, a subclass of `Actor`, where the attribute `currentTime` denotes the current model time:

```
class TimeActor | currentTime : Time .
subclass TimeActor < Actor .
```

Example 8.2 (Representing the Simple Traffic Light Model)**.** *Consider the flat traffic light system given in Example 8.1. The `TimedDelay2` delay actor can be represented as an object instance of the class `Delay`, a subclass of the class `Actor`, where a number N is expressed as the term `#` N:*

```
< 'TimedDelay2 : Delay |
    parameters : < 'delay : Parameter | exp : # 1.0, value : # 1.0,
                                        next-value : noValue >,
    ports : < 'input : InPort | value : # 0, status : absent >
           < 'output : OutPort | value : # 0, status : absent >,
    computation : noComputation,    status : enabled >
```

The connection from the output port `output` of the `Clock` actor to the input port `Sec` of the `PedestrianLight` actor and the input port `Sec` of the `CarLight` actor is represented by the term

```
('Clock ! 'output) ==> ('PedestrianLight ! 'Sec) ; ('CarLight ! 'Sec)
```

The entire state consists of two FSM actor objects, ten connections, two delay objects, five set variable objects, and the global event queue object.

The Global Event Queue. The global event queue is represented as an object `< global : EventQueue | queue :` *queue* `>`, where *queue* is a `::`-separated list, ordered according to time until firing, of terms of the form:

set of events ; *time to fire* ; *microstep*.

Each event in the *set of events* is characterized by the "global port name," *time to fire* is the time *until* the events are supposed to fire, and *microstep* is the additional "microstep" until the event fires.

Events communicated between an actor and the global event queue may cross hierarchical boundaries, since the actor can be located deep down in the actor hierarchy. This "traveling" of events is modeled as *message passing*: inserting an event into the output port p of an actor with actor identifier g corresponds to generating the message `active-evt(event(`g `!` p`,` v`))`, and an event generated by an actor is "sent" to the global event queue as a message of the form `schedule-evt(`*event*`,` *time*`,` *microstep*`)`:

```
msg active-evt : Event -> Msg .
msg schedule-evt : Event Time Nat -> Msg .

var PORTS OBJS : ObjectConfiguration .     var CF : Configuration .
var QUEUE : EventQueue .                   var EVENT : Event .
var O : Oid .          var P : PortId .    var AI : ActorID .
var T : Time .         var N : Nat .       var V : Value .

eq active-evt(event(O ! P, V))
   < O : Actor | ports : < P : Port | > PORTS >
 = < O : Actor | ports : < P : Port | value : V,
                                      status : present > PORTS > .
eq schedule-evt(EVENT, T, N)
   < global : EventQueue | queue : QUEUE >
 = < global : EventQueue | queue : add(EVENT, T, N, QUEUE) > .
```

where the function `add` inserts the new event (that should fire at time T and microstep N from the current time) in the correct place of the event queue.

Such a `schedule-evt` message is propagated towards the top of the actor hierarchy by the following equation, which moves the `schedule-evt` message inside a composite actor one level up (the propagation of a `active-evt` message from the global event queue to inner actors is explained below):

```
eq < O : CompositeActor |
       innerActors : CF  schedule-evt(event(AI ! P, V), T, N) >
 = < O : CompositeActor | innerActors : CF >
   schedule-evt(event((O . AI) ! P, V), T, N)} .
```

8.3.2 Specifying the Behavior of DE Models

The behavior of Ptolemy II DE models can be summarized as repeatedly performing the following actions from an initial state:

1. Advance time until the first event(s) in the event queue should fire.
2. The events supposed to fire are added to the corresponding output ports; the `status` of all other ports is set to `unknown`.
3. The *fixed point* of all ports is computed by gradually increasing the knowledge about the presence/absence of inputs to and output from ports until a fixed-point is reached (the *fire* action).
4. The actors with inputs or scheduled events are executed; states are changed and new events are generated and inserted into the global event queue (the *postfire* action).

Initial States. The initial state is given by the following term, where the `init` function yields the initial event messages to the global event queue:

```
{< global : EventQueue | queue : nil >  init(actors)  connections}
```

The `init` function distributes over actor objects, and propagates to the inner actors for composite actors. For each actor that generates initial events, `init` produces `schedule-evt` messages, declared by the equations of the form:

```
eq init(< O : C | ... >) =  schedule-evt(e, t, n) < O : C | ... > .
```

Advancing Time. The following tick rule advances time until the time when the first events in the event queue are scheduled; that is, until the time-to-fire of the first events in the event queue is `0`:

```
var NZT : NzTime .  vars CF CF' : Configuration .  var EVTS : Events .

rl [tick] :
   {OBJS < global : EventQueue | queue : (EVTS ; NZT ; N) :: QUEUE >}
 => {delta(OBJS, NZT)
     < global : EventQueue |
         queue : (EVTS ; 0 ; N) :: delta(QUEUE, NZT) >} in time NZT .
```

The first element in the event queue has *non-zero* delay `NZT`, and time is advanced by this amount `NZT`. As a consequence, the first component of the event timer goes to zero, and the function `delta` is applied to all the other objects (denoted by `OBJS` of sort ObjectConfiguration) in the system.

The function `delta` defines the effect of time elapse on the objects. Time only affects the state of `TimeActor` objects, that have an "clock" attribute `currentTime`, by increasing the value of `currentTime` according to the elapsed time. For a composite actor, `delta` just propagates to its inner actors. Time elapse does not affect other actors and connections:

```
eq delta(< O : TimeActor | currentTime : T > CF, T')
 = < O : TimeActor | currentTime : T + T' > delta(CF,T') .
eq delta(< O : CompositeActor | innerActors : CF > CF', T)
 = < O : CompositeActor | innerActors : delta(CF,T) > delta(CF',T) .
eq delta(CF, T) = CF [owise] .
```

This function is also applied to the other elements in the event queue, where it decreases the remaining time of each event set by the elapsed time `NZT`, where x `monus` y equals $\max(0, x - y)$:

```
eq delta((EVTS ; T ; N) :: QUEUE, T')
 = (EVTS ; T monus T' ; N) :: delta(QUEUE, T') .
eq delta(nil, T) = nil .
```

The next rule is a "microstep tick rule" that advances "time" with some microsteps if needed to enable the first events in the event queue:

```
crl [shortTick] :
    {OBJS < global : EventQueue | queue : (EVTS ; 0 ; NZ) :: QUEUE >}
 => {OBJS < global : EventQueue | queue : (EVTS ; 0 ; 0) :: QUEUE >} .
```

Executing Steps. When the remaining time and microsteps of the first events in the global event queue are both 0, the `executeStep` rule can be applied to perform an iteration of the system:

```
rl [executeStep] :
   {OBJS  < global : EventQueue | queue : (EVTS ; 0 ; 0) :: QUEUE >}
=> {update(< global : EventQueue | queue : QUEUE >
           postfire(
               portFixPoints(clearPorts(OBJS) releaseEvt(EVTS))))} .
```

The `clearPorts` function sets the status of evert port to *unknown*, and `releaseEvt` produces the `active-evt` message for each event scheduled to fire, so that those events are delivered into the output ports. The `portFixPoints` function finds the fixed points for all the ports (the *fire* action). The `postfire` function "executes" the steps by changing the states of the objects and generating new events, and the `update` function updates the `value` of each parameter of an actor for the next iteration. We first define the general cases of these *semantic functions* that apply to any `Actor` instances, and then define those for each actor in Section 8.3.3.

It is also important to notice that these functions are declared to be *partial* functions, and the `innerActors` attribute of a composite actor is declared to have sort ObjectConfiguration. Therefore, a term containing these function symbols, or a term of a composite actor containing messages in its `innerActors` attribute, will only have a *kind*, but not a *sort*:

```
op releaseEvt : Events ~> Configuration .
ops clearPorts portFixPoints : Configuration ~> Configuration .
ops postfire update : Configuration ~> Configuration .
```

Since the equations defining these functions only apply to terms of *sort* Configuration, this ensures that each function can be computed only *after* all messages are completely delivered and the "previous" function is computed (e.g., `portFixPoints` can be computed *after* `clearPorts` is computed).

Computing Expressions. As already mentioned, parameters or ports may include expressions that need to be computed before performing any other actions. When computing the value of an expression, the following values are used for the variables in the expression: (i) if the variable refers to an input port, then the "current" value of the input port is used *after* the status of the port has been determined to be either `present` or `absent`; and (ii) if the variable refers to a parameter, then the "previous" value of the parameter, computed at the end of the previous iteration, is used.

An expression is evaluated inside the `computation` attribute of an actor using a *computation configuration*, given by either a pair of an expression and a variable environment or a computation result:

```
sorts Computation ComputationID ComputationConfig ConfigItem .
op noComputation : -> Computation [ctor] .
op _/_ : ComputationID ComputationConfig -> Computation [ctor] .

op k : Exp -> ConfigItem [ctor] .
op env : EnvMap -> ConfigItem [ctor] .
op __ : ConfigItem ConfigItem -> ComputationConfig [ctor comm] .
op result : Value -> ComputationConfig [ctor] .
```

A variable environment is a semicolon-separated set $x_1 \leftarrow\!\!\dashv v_1; \cdots ; x_n \leftarrow\!\!\dashv v_n$ of variable assignments, constructed by the function `makeEnv` from the input ports and the parameters of an actor (see Appendix D.3 for its definition). A variable for an *unknown* input port is initially set to "unknown" (denoted by `NAME <-?`). For each unknown free variable in the variable environment, the corresponding value is transferred when it is available, i.e., when the status of an input port becomes `present`:

```
eq < O : Actor |
      ports : < P : InPort | status : present, value : V > PORTS,
      computation : CI / env(P <-? ; ENV) K:ConfigItem >
 =
   < O : Actor |
      computation : CI / env(P <-| V ; ENV) K:ConfigItem > .
```

Roughly, when an actor need to evaluate an expression E for a semantic function *func* (such as `portFixPoints` or `postfire`), the actor creates the computation configuration of E using equations of the form:

```
eq func(< O : C | computation : noComputation, ... > ...)
 = func(< O : C | computation : k(E) makeEnv(...), ... > ...) .
```

Our semantics of the Ptolemy II expression language (in Section 8.3.4) gives the rewrite sequence $k(E)\ \mathit{env}(x_1 \leftarrow\!\shortmid v_1; \cdots ; x_n \leftarrow\!\shortmid v_n) \longrightarrow_{\mathcal{R}} \mathit{result}(V)$, and *func* can be continued with the evaluated value V:

```
eq func(< O : C | computation : result(V), ... > ...)
 = func(< O : C | computation : noComputation, ... > ...) .
```

Postfire. The `postfire` function updates internal states and generates future events that are inserted into the event queue. The `postfire` function distributes over the actor objects in the configuration, and propagates to the *enabled* inner actors of a composite actor (disabled actors should not change their states or generate new events). Whenever `postfire` generates a new event with value v that should fire at time t and microstep n from the current time, it produces the `schedule-evt` message with the newly generated event, declared by the equations of the form:

```
eq postfire(< O : C |  status : enabled,
                       ports : < P : OutPort | > PORTS, ... >)
 = < O : C | ... >    schedule-evt(event(O ! P, v), t, n)
```

Finally, the following equation defines the *default* case when `postfire` does not change the state of an actor and does not generate a new event, *provided* that there exists no ongoing computation of an expression:

```
ceq postfire(OBJS) = OBJS if noComputation(OBJS) [owise] .

op noComputation : Configuration -> Bool .   var CC : Computation .
ceq noComputation(< O : Actor | computation : CC > CF)
  = false   if CC =/= noComputation .
ceq noComputation(< O : CompositeActor | innerActors : CF > CF')
  = false   if noComputation(CF) = false .
eq noComputation(CF) = true [owise] .
```

Computing the Fixed-Point for Ports. The idea behind the definition of the function `portFixPoints` is simple. Initially, the only port information are the events scheduled for this iteration. For each possible case when the status of an `unknown` port can be determined to be `present`, there is an equation of the following form (and similarly for deciding that a status of an `unknown` port will be `absent`):

```
eq portFixPoints(
     < O : ... | ports : < P : Port | status : unknown > PORTS, ... >
     connections and other objects)
 = portFixPoints(
     < O : ... | ports : < P : Port | value : ...,
                                       status : present > PORTS, ... >
     connections and other objects) .
```

The fixed-point is reached when no such equation can be applied. Then, the `portFixPoints` operator is removed by the `owise` construct, provided that no computation of an expression is going on:[2]

```
ceq portFixPoints(OBJS) = OBJS if noComputation(OBJS) [owise] .
```

The following equation propagates port status from a *known* output port to a connecting `unknown` input port. The present or absent `status` (and possibly the *fully evaluated* `value` of sort Value) of the output port `P` of *enabled* actor `O` is propagated to the input port `P'` of the actor `O'` through the corresponding connection `O ! P ==> (O' ! P') ; EPIS`:

```
vars PORTS PORTS' REST : ObjectConfiguration .  vars O O' : Oid .
var PS : PortStatus .  var EPIS : EPortIdSet .  vars P P' : PortId .

ceq portFixPoints(
     < O : Actor | status : enabled,
        ports : < P : OutPort | status : PS, value : V > PORTS >
     < O' : Actor |
        ports : < P' : InPort | status : unknown > PORTS' >
     (O ! P ==> (O' ! P') ; EPIS)   REST)
  = portFixPoints(
     < O : Actor | >
     < O' : Actor |
        ports : < P' : InPort | status : PS, value : V > PORTS' >
     (O ! P ==> (O' ! P') ; EPIS)   REST)
 if PS =/= unknown .
```

[2]The equations defining `portFixPoints` are terminating, because in each application of such an equation (except for the '`owise`' equation), the status of a port goes from `unknown` to either `present` or `absent`. Confluence of the equations follows from the fact that Ptolemy II DE models are assumed to be deterministic [125].

If some output port has status `present` but has a *non-value* expression PE of sort ProperExp (i.e., expressions that can be further evaluated, defined in Section 8.3.4), then the computation configuration is created to evaluate the expression, and the resulting value is plugged back into the output port:

```
eq portFixPoints(
    < O : Actor |
        ports : < P : OutPort | status : present, value : PE > PORTS,
        parameters : PARAMS, computation : noComputation > REST)
 = portFixPoints(
        < O : Actor | computation : #port(P) / makeEnv(PORTS PARAMS)
                                                k(PE)  > REST) .

eq portFixPoints(< O : Actor |
                       ports : < P : OutPort | > PORTS,
                       computation : #port(P) / result(V) > REST)
 = portFixPoints(< O : Actor |
                       ports : < P : OutPort | value : V > PORTS,
                       computation : noComputation > REST) .
```

We start the fixed-point computation of inner actors in `portFixPoints` of composite actors in the following cases: (i) some events from the event queue are passed to some inner actors; or (ii) an input port of a composite actor is connected to some inner actors and the status of the input port is decided (i.e., either received some value or became absent).

In Case (i), when released events are propagated to some inner actor of a composite actor, the `portFixPoints` computation of those inner actors begins. The following equations describe the propagation of `active-evts` from the event queue to inner actors. If there are *some* events toward an inner actor of a composite actor, then *all* such events are passed to the inner actors and `portFixPoints` of the inner actors is started:

```
ceq portFixPoints(
        active-evt(event((O . AI) ! P, V))
        < O : CompositeActor | innerActors : OBJS > CF)
  = portFixPoints(
        < O : CompositeActor |
            innerActors : portFixPoints(MSGS OBJS) > CF')
 if fr(MSGS, CF') := filterMsg(O, CF, active-evt(event(AI ! P, V))) .
```

where the function `filterMsg` separates the events toward inside from the others, and returns a constructor `fr(`*Events*`,`*Conf*`)` which is a pair of the desired events and the other configuration.

In Case (ii), when a composite actor passes a value (or the knowledge that input will be `absent`) to its inner actors, if the inner fixed-point computation has not started yet or already been finished, then `portFixPoints` must again be called to (re-) compute the fixed-point of the inner diagram (we use the special name '`parent`' to denote the containing actor of an actor):

```
ceq portFixPoints(
      < O : CompositeActor |
         ports : < P : InPort | status : PS, value : V > PORTS,
         innerActors :
            (parent ! P) ==> (O' ! P' ; EPIS)
            < O' : Actor | ports : < P' : InPort | status : unknown >
                                    PORTS2 > OBJS,
         status : enabled > REST)
  = portFixPoints(
      < O : CompositeActor |
         innerActors :
            portFixPoints(   *** (re-) start the inner fixed-point
               (parent ! P) ==> (O' ! P' ; EPIS)
               < O' : Actor | ports : < P' : InPort | status : PS,
                                          value : V > PORTS2 > OBJS) >
      REST) if PS =/= unknown .
```

Likewise, an inner actor can propagate the status of output ports to the containing actor when the inner fixed-point is already finished:

```
ceq portFixPoints(
      < O : CompositeActor |
         ports : < P : OutPort | status : unknown > PORTS,
         innerActors :
            (O' ! P') ==> (parent ! P ; EPIS)
            < O' : Actor |
               status : enabled,
               ports : < P' : OutPort | status : PS,
                                    value : V > PORTS2 > OBJS > REST)
  = portFixPoints(
      < O : CompositeActor |
         ports : < P : OutPort | status : PS, value : V > PORTS,
         innerActors :
            (O' ! P') ==> (parent ! P ; EPIS)
            < O' : Actor | > OBJS > REST) if PS =/= unknown .
```

For "inactive" cases (e.g., all input ports of an actor are absent or the status of an actor is disabled), every output port is set to `absent`, unless it has a scheduled event from the global event queue (see Appendix D.3.1).

8.3.3 DE Semantics of Actors

This section explains how the Real-Time Maude semantics of each actor can be defined in a modular way. The syntax of each actor is specified by declaring a suitable subclass of the class `Actor` in Real-Time Maude, and the semantics of the actor is specified by declaring three semantic functions for the actor: `init`, `portFixPoints`, and `postfire`. In this section we only show the semantics of three actors, and refer to Appendix D.3.2 for the semantics of other actors, including timed delays and modal models.

Current Time. A *current time* actor generates an output to denote the current model time. Because the superclass `TimeActor` already contains the current time, the `currentTime` subclass does not add any new attributes:

```
class CurrentTime .
subclass CurrentTime < AtomicActor TimeActor .
```

When the *current time* actor gets an input, it outputs the current model time, given by its `currentTime` attribute. Furthermore, when its lone input port is `absent`, its lone output port is also set to `absent`:

```
ceq portFixPoints(
      < O : CurrentTime | ports : < P' : OutPort | status : unknown >
                                  < P : InPort | status : PS > PORTS,
                          currentTime : T, status : enabled > REST)
  =
   portFixPoints(
     < O : CurrentTime |
         ports : < P' : OutPort | status : PS, value : # T >
                 < P : InPort | > PORTS > REST) if PS =/= unknown .
```

Since the *init* and *postfire* actions do nothing for current time actors, the equations are not declared (i.e., the default *owise* equations are used).

Clocks. The Ptolemy II parameters of a *clock* actor (*period*, *offsets*, and *values*) are represented in the `parameters` attribute. The only additional attribute is the attribute `index` to keep track of the "index" of the *offsets* and *values* arrays for the next event to be generated:

```
class Clock | index : Nat .
subclass Clock < AtomicActor .
```

A clock actor generates an initial event according to the parameters *offsets* and *values* (i.e., the event with value *values*(0) at time *offsets*(0)), where $A(\# n)$ denotes the nth element of an array A:

```
eq init(< O : Clock |
          parameters : < 'offsets : Parameter | value : V1 >
                       < 'value : Parameter | value : V2 > PARAMS >)
 =
   < O : Clock | >
  schedule-evt(event(O ! 'output, V2(#0)), toTime(V1(#0)), 0) .
```

A clock actor does not produce any output as a result of any input. Thus, if the `status` of its output port is `unknown` (that is, the clock actor did not schedule an event for this iteration), it should be set to `absent`:

```
eq portFixPoints(
     < O : Clock |
         ports : < P : OutPort | status : unknown > PORTS > REST)
 =
   portFixPoints(
     < O : Clock |
         ports : < P : OutPort | status : absent > PORTS > REST) .
```

If a clock actor produces an *output*, then the `postfire` function should schedule the next event, and update the `index` variable as follows, where A `.'length(())` denotes the length of an array A:

```
ceq postfire(
     < O : Clock |
         ports : < P : OutPort | status : present > PORTS,
         parameters : < 'offsets : Parameter | value : V1 >
                      < 'values : Parameter | value : V2 > PARAMS,
         index : N, status : enabled >)
  =
   < O : Clock | index : N + 1 >
   schedule-evt(event(O ! P, V2(#(N + 1))),
                TIME-TO-FIRE,
                if TIME-TO-FIRE == 0 then 1 else 0 fi)
 if ((# N + # 1) lessThan (V1 . 'length(()))) == # true
 /\ TIME-TO-FIRE := toTime((V1(#(N + 1))) - (V1(# N))) .
```

In the above equation, when the next index $N+1$ is less than the length of the *offsets* array, the clock actor generates an event with value $values(N+1)$ at time `TIME-TO-FIRE` $= offsets(N+1) - offsets(N)$. If `TIME-TO-FIRE` is `0`, then the microstep of the event is `1`, and otherwise, the microstep is `0`. A similar equation defines `postfire` when a new "cycle" is started, i.e., when the next index $N+1$ equals the length of the *offsets* array. In this case, the `index` becomes 0, the generated event has the value $values(0)$ and `TIME-TO-FIRE` $= period - offsets(N) + offsets(0)$.

Finite State Machine (FSM) Actors. A `FSM-Actor` is characterized by its *current state* (or locations), its transitions, and its local variables:

```
class FSM-Actor | currState : Location,
                  initState : Location,
                  transitions : TransitionSet .
subclass FSM-Actor < AtomicActor .
```

The attribute `transitions` denotes a semicolon-separated set of transitions, each has the form $id : s$ `->` s' {`guard:` g `output:` p_{i_1} `|->` $e_{i_1}; \ldots; p_{i_k}$ `|->` e_{i_k} `set:` v_{j_1} `|->` e_{j_1} ; ... ; v_{j_l} `|->` e_{j_l}} for a transition id, states/locations s and s', Boolean expression g, port names $p_{i_1}, \ldots, p_{i_k}$, variable names $v_{j_1}, \ldots, v_{j_l}$, and expressions $e_{i_1}, \ldots, e_{i_k}$ and $e_{j_1}, \ldots, e_{j_l}$.

The `portFixPoints` function must check whether at most one transition is enabled at any time by evaluating the guard expressions. Given a set of transitions $ti_1 : s_1 \to s'_1 \; \{guard : g_1 \ldots\}; \cdots ; ti_n : s_n \to s'_n \; \{guard : g_n \ldots\}$, the guard expressions $g_1, \ldots, g_n$ can be computed at the same time using the corresponding Ptolemy II *record* expression $\{ti_1$ `<-` $g_1, \ldots, ti_n$ `<-` $g_n\}$. The following equation sets all the guard expressions from the current state `STATE` to be computed in the `computation` attribute, provided that there is *some* input to the actor and some output port has status `unknown`:

```
vars STATE STATE' : Location .      var TRANSSET : TransitionSet .
var TG : Exp .      var VREC : ValueRow .     vars OL AL : ExpMap .

ceq portFixPoints(
     < O : FSM-Actor |
         status : enabled,     parameters : PARAMS,
         currState : STATE,    transitions : TRANSSET,
         ports : < P : InPort | status : present >
                 < P' : OutPort | status : unknown > PORTS,
         computation : noComputation > REST)
=
   portFixPoints(
     < O : FSM-Actor | computation : #guards / k(E) EV > REST)
 if E := makeGuardExp(STATE, TRANSSET)
 /\ EV := makeEnv(PARAMS PORTS < P : InPort | status : present >) .
```

In the following equation, only one transition from the current state `STATE` is enabled (i.e., its guard expression is evaluated to `true`), and the function `updateOutPorts` then updates the status and the values of the output ports according to the current state and input, where the function `noGuardTrue` returns *true* iff the rest of the guard expressions contains no record item of the form ti `<- true` (i.e., no enabled transition ti):

```
ceq portFixPoints(
     < O : FSM-Actor |
         computation : #guards / result({TI <- # true, VREC}),
         transitions : (TI : STATE --> STATE'
                         {guard: TG output: OL set: AL}) ; TRANSSET,
         ports : PORTS > REST)
=  portFixPoints(
     < O : FSM-Actor | computation : noComputation,
                       ports : updateOutPorts(OL, PORTS) > REST) .
 if noGuardExpTrue(VREC) .
```

On the other hand, if no transition is enabled, then every output port must be set to *absent* as follows, where the function `setUnknownOutPortsAbsent` sets the `status` of each output port with status `unknown` to `absent`:

```
ceq portFixPoints(
     < O : FSM-Actor | computation : #guards / result({VREC}),
                       ports : PORTS > REST)
= portFixPoints(
      < O : FSM-Actor | computation : noComputation,
                        ports : setUnknownOutPortsAbsent(PORTS) >
      REST)
   if noGuardExpTrue(VREC) .
```

For `postfire`, the guard expressions are first computed again in a similar way. An FSM actor does not generate future events, but `postfire` updates the internal state (location and parameters) of the actor if exactly one of its transitions was enabled, and does nothing if no transition is enabled:

```
ceq postfire(
     < O : FSM-Actor |
         computation : #guards / result({TI <- # true, VREC}),
         transitions : (TI : STATE --> STATE'
                         {guard: TG output: OL set: AL}) ; TRANSSET,
         parameters : PARAMS >)
 = < O : FSM-Actor | currState : STATE',
                     computation : noComputation,
                     parameters : updateParam(AL, PARAMS) >
 if noGuardExpTrue(VREC) .

ceq postfire(
       < O : FSM-Actor | computation : #guards / result({VREC}) >)
 = < O : FSM-Actor | computation : noComputation >
 if noGuardExpTrue(VREC) .
```

More details, including the definitions of the functions `updateOutPorts` and `updateParam`, can be found in Appendix D.3.2.

8.3.4 Expression Language Semantics

Ptolemy II provides a simple expression language to specify the values of parameters, guards/actions in FSM actors, etc. The expression language is similar to expression languages in widely used programming languages. Expressions consist of constants (e.g., numbers), algebraic operators (e.g., arithmetic), and variables that refer to parameters and input ports.

The Ptolemy II expression language supports composite data types such as arrays, records, and matrices. Arrays are lists of expressions in curly brackets, e.g., `{1, 2.0, "x"}`. Records are lists of fields where each field consists of a name and a value. For example, `{a = 1, b = "foo"}` is a record with two fields, named `a` and `b`, with values `1` and `"foo"`, respectively.

The Ptolemy II expression language also provides functional expressions. A functional expressions is either a method call $\mathit{obj.method}(\mathit{arg_1}, \ldots, \mathit{arg_n})$ on objects or a general function call $\mathit{function_name}(\mathit{arg_1}, \ldots, \mathit{arg_n})$. A new function can be defined by giving a definition of form `function(`$\mathit{arg}_1 : \mathit{Type}_1, \ldots, \mathit{arg}_n : \mathit{Type}_n$`)` *function_body_expression*. In addition, the expression language includes a set of built-in methods and functions, such as `sin()`, `cos()`, A`.length()` for an array A, etc.

Algebraic Semantics. In our representation, Ptolemy II expressions are terms of sort Exp. *Values* are expressions that cannot be further evaluated, and are represented as terms of sort Value, a subsort of Exp. Variables are terms of sort VarId in our semantics. Constants have sort Value, and are represented by values in Real-Time Maude, prefixed with the # symbol. Numerical constants are either rational numbers (including the integers) or fixed-point constants. For example:

```
ops -_ ~_ !_ : Exp -> Exp .                               --- unary
ops _+_ _-_ _*_ _/_ _%_ _^_ : Exp Exp -> Exp .   --- numerical
ops _&&_ _||_ : Exp Exp -> Exp .                          --- logical
...
op _?_:_ : Exp Exp Exp -> Exp [ctor prec 60] .   --- conditional
```

We also define sort ProperExp to denote *non-value* expressions; that is, all expressions that can be further evaluated are defined as ProperExp, e.g.:

```
sort ProperExp .
subsorts VarId < ProperExp < Exp .
ops -_ ~_ !_ : ProperExp -> ProperExp .
ops _+_ _-_ _*_ _/_ _%_ _^_ : ProperExp Exp -> ProperExp .
ops _+_ _-_ _*_ _/_ _%_ _^_ : Exp ProperExp -> ProperExp .
...
```

The algebraic semantics of each operator and data structure can be easily defined by equations as a usual way. For example, logical expressions and conditional expressions (*condition* ? exp_1 : exp_2) are defined as follows:

```
eq ! (# B) = #(not B) .                  --- negation
eq (# B) && (# B') = # (B and B') .      --- and
eq (# B) || (# B') = #( B or B') .       --- or
eq # true ? E : E' = E .             eq # false ? E : E' = E' .
```

K Semantics. The semantics of the Ptolemy II expression language is defined in an actor by a computation configuration—a pair of an expression E and a variable environment $x_1 \hookleftarrow v_1; \cdots ; x_n \hookleftarrow v_n$—and specifies the rewrite sequence $k(E)\ env(x_1 \hookleftarrow v_1; \cdots ; x_n \hookleftarrow v_n) \longrightarrow_{\mathcal{R}} result(V)$ to the value V of E based on the environment.

We define the entire semantics of the expression language by using the K rewriting-based semantics framework [154], since it provides a more clear way to describe interactions between expressions and actors. The semantics is defined by repeating the following three steps until a single value remains, given a computation configuration $k(E)\ env(ENV)$:

1. Systemically construct a data structure, called a *computation*, which is a $\curvearrowright$-separated lists $T_1 \curvearrowright T_2 \curvearrowright \cdots \curvearrowright T_n$ of computational tasks to "sequentially" evaluate the expression E;

2. Define an equation or a rule to evaluate the first and most elementary task T_1 in the list to a value V_1 using the environment ENV; and

3. Combine the completed task V_1 with the next task T_2, and obtain a new computation $T_2' \curvearrowright T_3 \curvearrowright \cdots \curvearrowright T_n$.

For example, given the environment $x \hookleftarrow 3$, the expression $x + 1$ becomes $x \curvearrowright (\square + 1)$ by Step 1, meaning that x will be evaluated first. By Step 2, it becomes $3 \curvearrowright (\square + 1)$, and then by Step 3, it becomes $3 + 1$. Finally, by the algebraic semantics of $+$, we have the value 4. The auxiliary operators to construct computations, such as □+_, are called *freezer* operators.

Computations are represented as terms of sort K, constructed by the list concatenation operator `->`. In particular, an expression is an item of a computation, and a computation configuration containing only a single value `V` is immediately reduced to the configuration `result(V)`:

```
sort K .                        subsort Exp < K .
op nil : -> K [ctor] .          op _->_ : K K -> K [ctor assoc id: nil] .
eq k(V) env(ENV) = result(V) .
```

Our K-based semantics defines the following three types of equations: (i) equations, called *heating* rules, to divide a non-value expressions E to a sequence $T_1 \curvearrowright T_2 \curvearrowright \cdots \curvearrowright T_n$ of simpler tasks (for Step 1); (ii) equations to fully evaluate "atomic" tasks, such as the equations for the algebraic semantics of operators; and (iii) equations, called *cooling* rules, to combine a completed task (i.e., a value) with the next task (for Step 3).

Variable Expressions. If the first item in a computation is a variable, then the corresponding value is obtained from the environment (in Step 2):

```
var I : VarId .        var ENV : EnvMap .    var K : K .
vars V V' : Value .    vars E E' : Exp .     vars PE PE' : ProperExp .

eq k(I -> K) env(I <-| V ; ENV) = k(V -> K) env(I <-| V ; ENV) .
```

Recall that each variable for an `unknown` input port is initially set to be unknown in a variable environment. Intuitively, when an actor evaluates an expression that contains a variable for an input port, the actor must wait until the status of the input port is determined (by the *fire* action). Such a "waiting" mechanism is naturally specified by the above equation.

Structural Equations. The heating and cooling equations for unary and binary operators are defined by identifying non-value subexpressions of sort ProperExp; e.g., for the negation operator `!` and the addition operator `+`:

```
eq k(      ! PE -> K) = k(PE -> ![] -> K) .  --- heating
eq k(V ->  ![] -> K) = k(      ! V -> K) .  --- cooling

eq k(     PE + E' -> K) = k(PE -> []+ E' -> K) . --- heating (left)
eq k(V ->  []+ E' -> K) = k(      V + E' -> K) . --- cooling (left)
eq k(     E + PE' -> K) = k(PE' -> E +[] -> K) . --- heating (right)
eq k(V' ->  E +[] -> K) = k(      E + V' -> K) . --- cooling (right)
```

For a conditional expression, only the first argument (i.e., the condition) needs to be evaluated before computing the other arguments:

```
eq k(     PE ? E : E' -> K) = k(PE -> []? E : E' -> K) . --- heating
eq k(V ->  []? E : E' -> K) = k(      V ? E : E' -> K) . --- cooling
```

Notice that declaring heating and cooling equations also requires defining the corresponding freezer operators.[3] We refer to Appendix D.3.3 for more details on our expression language semantics, including other constructs.

[3] As explained in [154], the heating and cooling rules (and the freezer operators) can be automatically generated from the syntax of the language.

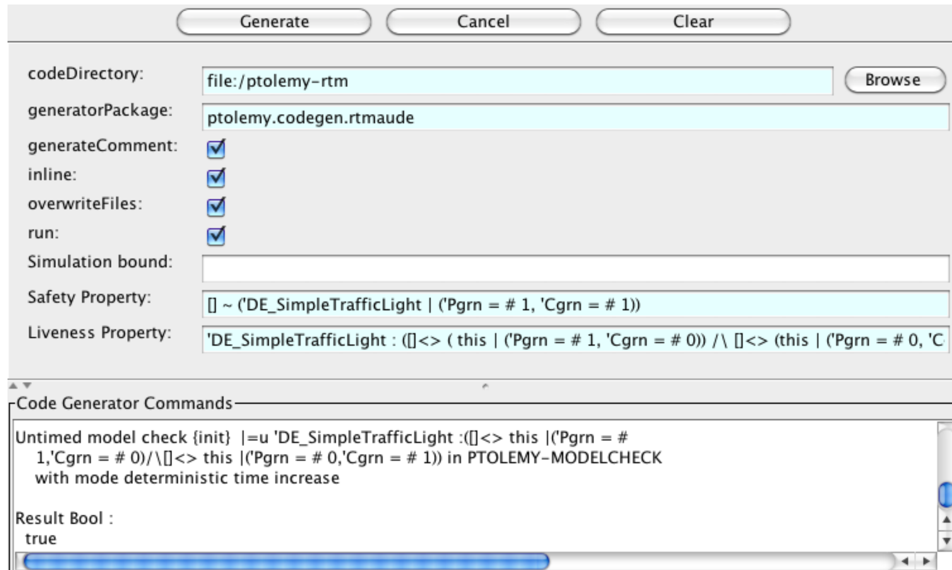


Figure 8.6: Dialog window for the Real Time Maude code generation

8.4 Formal Verification in Ptolemy II

Although simuations of Ptolemy II models are very useful for prototyping purposes, it is hard to use simulations to verify that a Ptolemy II model—even though it is assumed to be deterministic—satisfies more advanced safety and liveness properties. Indeed, the verification effort described in Section 8.5 made us aware of a design flaw in the Ptolemy II model of the fault-tolerant traffic light that had not been discovered during Ptolemy II simulations. This section explains how the Real-Time Maude verification of a Ptolemy II DE design model has been integrated into the Ptolemy II tool, and how users can easily verify their Ptolemy II models without needing to understand the Real-Time Maude representation.

Code Generation. Ptolemy II gives the user the possibility of adding a "code generation button." When the `RTMaudeCodeGenerator` button in a Ptolemy II DE model is double-clicked, Ptolemy II opens a dialog window which allows the user to start code generation and to formally analyze the generated code. After clicking the `Generate` button in the dialog window, the generated Real-Time Maude code and the analysis result are displayed. Figure 8.6 shows the dialog window for the simple traffic light system in Example 8.1. The two temporal logic properties discussed below have been entered into the window. The `Generate` button has already been clicked and the results of model checking those properties are displayed in the "Code Generator Commands" box (see Appendix D.2 for details).

Predefined State Propositions. An LTL formula is constructed from a set of *state propositions*. Having to define such state propositions makes the verification process nontrivial for the Ptolemy II user, since it requires some knowledge of the Real-Time Maude representation of the Ptolemy II model, as well as the ability to define functions in Real-Time Maude. To free the user from this burden, we have predefined several generic atomic propositions for Ptolemy II models. For example, the state proposition $\mathit{actorId} \;\mid\; \mathit{var_1} = \mathit{value_1}, \ldots, \mathit{var_n} = \mathit{value_n}$ holds in a state if the value of the parameter $\mathit{var_i}$ of an actor equals $\mathit{value_i}$ for each $1 \leq i \leq n$, where *actorId* is its *global actor identifier*. Similarly, the propositions

$$\mathit{actorId} \mid \texttt{port}\ p\ \texttt{is}\ \mathit{value} \qquad \mathit{actorId} \mid \texttt{port}\ p\ \texttt{is}\ \mathit{status}$$
$$\mathit{actorId}\ ?\ \mathit{boolean_expression}$$

hold if, respectively, the port p of actor *actorId* has the value *value*, the port p has status *status*, or the given *boolean_expression* is evaluated to `true`. For FSM actors and modal models, the proposition *actorId* `@` *location* is satisfied iff the actor with global name *actorId* is in local state *location*.

An LTL formula may contain multiple occurrences of atomic propositions. To avoid having to write long global actor names too many times, we can simplify a formula with *actor scope*. The formula *actorId* : *formula* denotes that *formula* should hold in the actor with the global identifier *actorId*. For example, the formula $o_1.\, o_2\ :\ \Box\, (\texttt{this}\ @\ l_1\ \wedge\ o_4\, .\, o_5\ @\ l_2)$ is equivalent to $\Box\, (o_1.\, o_2\ @\ l_1\ \wedge\ o_1.\, o_2.\, o_4\, .\, o_5\ @\ l_2)$, where `this` denotes the identity.

Consider the flat traffic light system given in Example 8.1, where each traffic light is represented by set of variables. The safety property we want to verify is that it is never the case that both the car light and the pedestrian light show green at the same time. If the name of the model is `'DE_SimpleTrafficLight`, then using the predefined state propositions, this safety property can be specified as the LTL formula:

```
[] ~ ('DE_SimpleTrafficLight | ('Pgrn = # 1, 'Cgrn = # 1))
```

We can also check the liveness property that both pedestrian and cars can cross infinitely often. That is, it is infinitely often the case that the pedestrian light is green when the car light is *not* green, and it is also infinitely often the case that the car light is green when the pedestrian light is not green, specified as the LTL formula:

```
'DE_SimpleTrafficLight : ([]<> (this | 'Pgrn = # 1, 'Cgrn = # 0) /\
                          []<> (this | 'Pgrn = # 0, 'Cgrn = # 1))
```

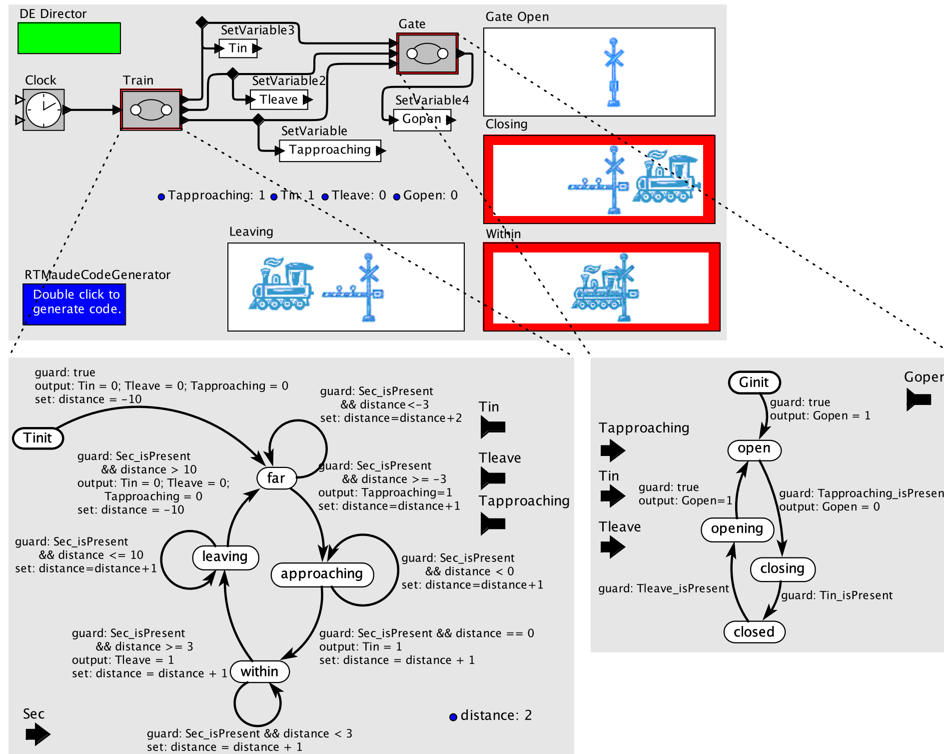


Figure 8.7: Ptolemy II DE model of the railroad crossing.

8.5 Case Studies

This section presents three Ptolemy II DE models and shows how they have been verified in Real-Time Maude from within Ptolemy II. Section 8.5.1 presents the benchmark railroad crossing example, Section 8.5.2 presents a hierarchical model of a fault-tolerant traffic light system, and Section 8.5.3 presents an assembly line due to Misra [144].

8.5.1 Railroad Crossing

A gate at the intersection of the train track and a road should be lowered when a train is in the intersection. Figure 8.7 shows a Ptolemy II DE model `RailroadSystem`. This model consists of two finite state machine (FSM) actors: a `Train` actor that models trains, and a `Gate` actor that controls the gate. In addition, the model has Boolean variables `Tin` (denoting if a train is in the intersection), `Tleave` (denoting if a train is leaving), `Tapproaching` (denoting 1 if a train is approaching), and `Gopen` (denoting if the gate is open). State changes are triggered by a `Clock` actor. These variables are set by signals from the output ports of the train and the gate controller.

The `Train` actor has five states, and a local variable `distance` denoting the distance between the train and the beginning of the intersection. The `Train` has one input port `Sec`, and three output ports `Tin`, `Tleave`, and `Tapproaching`. Initially, the state is `Tinit`. The actor stays in state `far` as long as `distance` < -3, while the value of `distance` increases by 2 each time there is input in the `Sec` port. When `distance` $= -3$, it takes a transition to state `approaching`, and outputs 1 through its `Tapproaching` port. The distance increases by 1 for each time unit in state `approaching` (as well as in states `within` and `leaving`). When `distance` $= 0$, the actor goes to state `within`, and emits a signal through its `Tin` port. When `distance` ≥ 3, the train is `leaving` the intersection, and an output is emitted through the `Tleave` port. When `distance` > 10, the train disappears and a signal with value 0 is output through all three output ports. Finally, the actor goes to state `far` again, and the value of `distance` is set to -10.

The `Gate` actor responds to input from the `Train` actor through its three input ports `Tapproaching`, `Tin`, and `Tleave` by the necessary signal through its `Gopen` output port. In particular, the `Gate` actor outputs 1 through the `Gopen` output port whenever it goes to state `open`.

The main safety requirement of `RailroadSystem` is that whenever a train is in the intersection, the gate must be closed. In our model, a train is in the intersection when the `'Train` actor is in state `'within`, and the gate is closed when the `'Gate` actor is in state `'closed`. We want to verify that it is *always* the case that the former *implies* the latter, expressed as the following LTL formula using the predefined propositions in Section 8.4:

$$\Box\ ((\texttt{'RailroadSystem . 'Train @ 'within}) \\ \rightarrow (\texttt{'RailroadSystem . 'Gate @ 'closed}))$$

Verification of this property through the Real-Time Maude code generation and analysis interface in Ptolemy II yielded the expected result `true`, proving that the desired property is satisfied in this Ptolemy II model.

We have also verified that the `Train` actor will reach the state `within` within 7 time units from the start of system execution:

$$\Diamond\ (\texttt{'RailroadSystem . 'Train @ 'within})\quad \textit{in time} <= 7$$

The execution of each verification command in this case study took less than one second on a 2.4 GHz Intel Core 2 Duo processor.

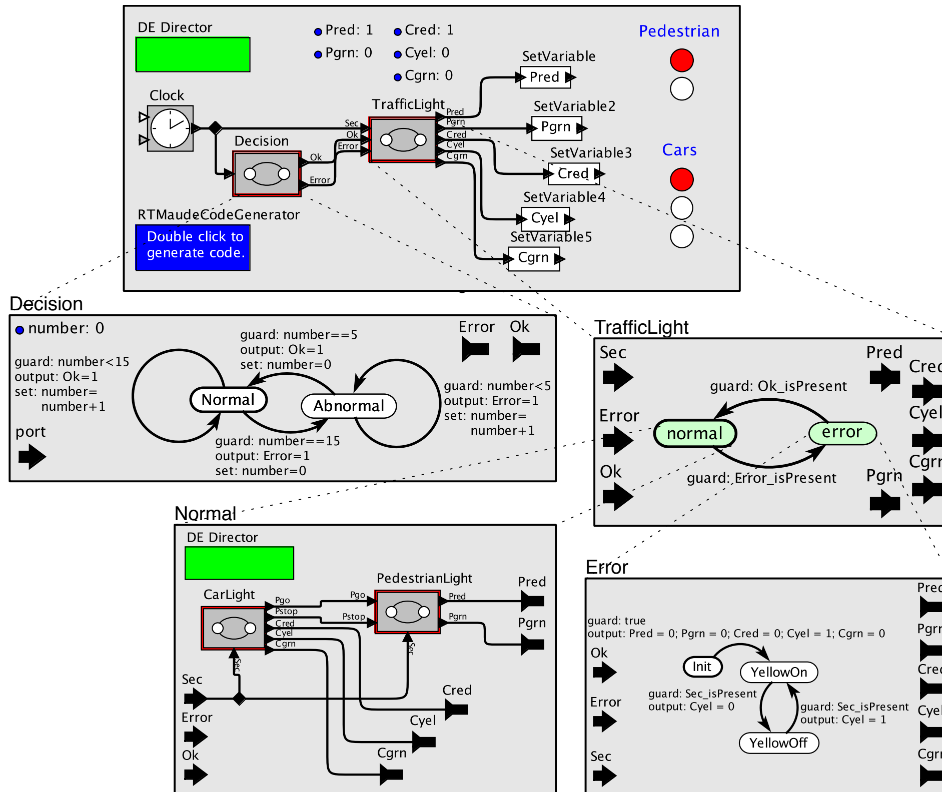


Figure 8.8: A hierarchical fault-tolerant traffic light system.

8.5.2 Hierarchical Traffic Light

This section describes the verification of the *hierarchical* Ptolemy II DE model in [44], illustrated in Figure 8.8, that extends the flat traffic light system in Example 8.1 to a fault-tolerant traffic light system consisting of one car light and one pedestrian light.

The FSM actor `Decision` "generates" failures and repairs by alternating between staying in state `Normal` for 15 time units and staying in state for `Abnormal` for 5 time units. Whenever the actor takes a transition with target `Normal`, it sends a signal through its `Ok` port, and whenever it reaches, or stays in, state `Abnormal`, the actor sends a signal through its `Error` port.

The actor `TrafficLight` is a *modal model*; whenever it is in `error` mode and receives a signal through its `Ok` port, the actor goes to `normal` mode, and vice versa when it receives an `Error` event in `normal` mode. The FSM actor refining the `error` mode has three states. In this mode, all lights are turned off except for the *blinking* yellow light of the car light. The refinement of the `normal` mode is the composite actor that consists of the FSM actors `CarLight` and `PedestrianLight`, explained in Example 8.1, defining the behavior of the two lights during normal operations.

The main properties we have verified are the safety and liveness properties described in Section 8.4:

$$\Box\,(\texttt{'HierarchicalTrafficLight | ('Pgrn = \# 1, 'Cgrn = \# 1)}),$$

$$\texttt{'HierarchicalTrafficLight} : (\Box\Diamond\,(\texttt{this | 'Pgrn = \# 1, 'Cgrn = \# 0}) \wedge \Box\Diamond\,(\texttt{this | 'Pgrn = \# 0, 'Cgrn = \# 1})).$$

The following bounded response property states that if some error has occurred (i.e., the decision actor generates an error), then the car light turns yellow within one time unit:

```
'HierarchicalTrafficLight : (('Decision | port 'Error is present)
   => <>le(1) (this | 'Cyel = # 1))
```

The following property states that not only will the car light turn yellow within 1 time unit of a failure, but the other car lights will be turned off:

```
'HierarchicalTrafficLight : (('Decision | port 'Error is present)
   => <>le(1) (this | 'Cyel = # 1, 'Cgrn = # 0, 'Cred = # 0))
```

Model checking this property returns a counter-example which shows that, after a failure, the car light may also show red or green in addition to blinking yellow. The reason for this flaw is that each time we enter the *error* mode, the `Error` actor is not re-initialized.

The final bounded response property is that whenever the traffic light goes to an error state, it is repaired within at most 6 time units:

```
'HierarchicalTrafficLight :
  (('TrafficLight @ 'error) => <>le(6) ('TrafficLight @ 'normal))
```

Model checking the following property verifies that there is at least 16 time units between a repair of an error and the emergence of the next error:

```
('HierarchicalTrafficLight : 'TrafficLight @ 'error)
                                            separated by >= 16
```

Finally, model checking the following property verifies that there is at least 3 time units between consecutive red pedestrian lights:

```
('HierarchicalTrafficLight | 'Pred = # 1) separated by >= 3
```

The execution of each verification command took around seven seconds in this case study on the same machine.

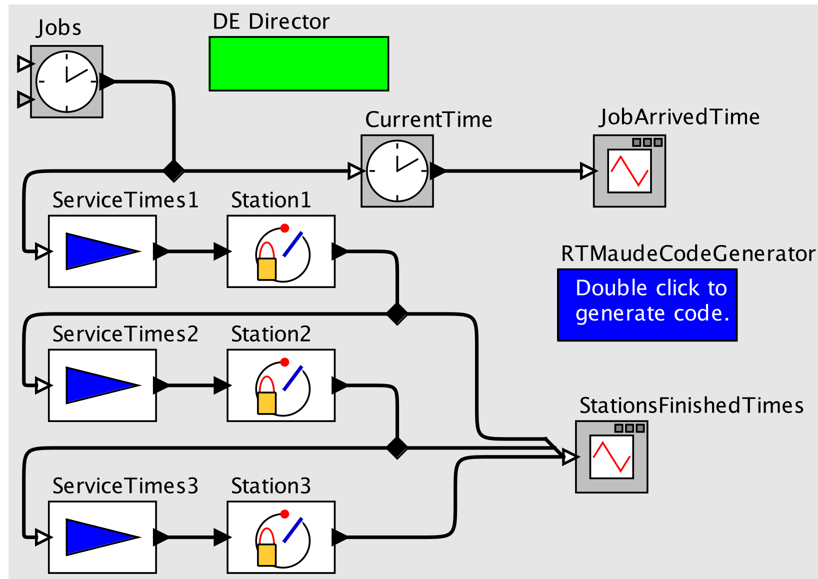


Figure 8.9: The assembly line example.

8.5.3 Assembly Line

We have simulated in Real-Time Maude the "assembly line" example of Misra [144] in Figure 8.9. In this model, a clock `Jobs` generates a set of *jobs* at certain times. The timed plotter `JobArrivedTime` records the actual times (obtained through the `currentTime`) when the jobs arrived.

Each job is executed in three different ways (at `Station1`, `Station2`, and `Station3`). First, a job gets assigned the time that it takes to execute the first task of the job. This is done by the `Ramp` actor `ServiceTimes1`. The actual "wait" is first done at the *noninterruptible timer* `Station1`. The point of using a noninterruptible timer is that the count down does not start if some other job is serviced.[4] After finishing the first part,the job is then assigned a duration of the second part in the ramp `ServiceTimes2`, and waits accordingly at the noninterrruptible timer `Station2`. Finally, when that wait is over, the process repeats for the third part of the task. The timed plotter `StationsFinishedTimes` records the times when jobs finish executing the first, the second, and the third part of the jobs.

In order to simulate the system up to time t in Real-Time Maude, we write the time bound t in the `Simulation bound` item of the dialog window (see Figure 8.6). The output shows the final state, where the timed plotter object `'StationsFinishedTimes` shows the times when events happened:

[4]For example, this can be compared to a gas station. It takes so long to fill up the gas tank of your car, but if someone else is already pumping gas, you must also wait for that car to stop pumping and to drive away.

```
{< 'AssemblyLine : CompositeActor | innerActors : (
    < 'StationsFinishedTimes : TimedPlotter |
      currentTime : 49,
      event-history :
        (source: 'Station1 ! 'output time:  9 value: # 1) ++
        (source: 'Station1 ! 'output time: 19 value: # 1) ++
        (source: 'Station2 ! 'output time: 21 value: # 2) ++
        (source: 'Station3 ! 'output time: 23 value: # 3) ++
        (source: 'Station1 ! 'output time: 31 value: # 1) ++
        (source: 'Station2 ! 'output time: 36 value: # 2) ++
        (source: 'Station1 ! 'output time: 37 value: # 1) ++
        (source: 'Station2 ! 'output time: 38 value: # 2) ++
        (source: 'Station3 ! 'output time: 39 value: # 3) ++
        (source: 'Station3 ! 'output time: 40 value: # 3) ++
        (source: 'Station2 ! 'output time: 45 value: # 2) ++
        (source: 'Station3 ! 'output time: 49 value: # 3),
      parameters : none, ... > ...), ... >
< global : EventQueue | queue : nil >} in time 49
```

For example, we see that `Station2` finish each job at time 21, 36, 38 and 45, respectively. These results are the same as the results shown in the Ptolemy II timed plotters after the Ptolemy II executions.

8.6 Concluding Remarks

This chapter has explained how the semantics of Ptolemy II DE models can be formalized in Real-Time Maude. The expressiveness of Real-Time Maude is necessary to define this semantics, including the use of unbounded data structures, nested objects, and advanced membership equational logic features such as partial functions and the `owise` construct. An additional contribution of our work is the clarification of the semantics of modal models, for which we have given a composite-actor semantics in Ptolemy II. We have integrated Real-Time Maude verification into Ptolemy II, and have defined useful atomic propositions, so that a Ptolemy II DE model can be easily verified in Ptolemy II. This enables a model-engineering process that combines the convenience of Ptolemy II modeling and simulation with formal verification in Real-Time Maude.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

Rewriting-based model checking integrates the power of modeling languages with the automatic nature of model checking, using the formal semantics of different system specification languages, and exploiting the expressive power of rewriting logic. The rewriting-based formal semantics automatically gives the mathematical model $\mathcal{R}_M$ of a system specification M.

In order to establish this rewriting-based model checking method, we have needed to develop: (i) well-matched property specification languages with efficient model checking algorithms, (ii) methods to deal with infinite-state systems and the state space explosion problem, and (iii) fully integrated model-engineering methods for practical applications.

Therefore, in this dissertation we presented the model checking algorithms and tools for LTLR properties under localized fairness, various infinite-state model checking techniques, the Multirate PALS methodology to reduce the system complexity of multirate distributed cyber-physical systems, and fully integrated model checking environments for two embedded system modeling languages based on their formal rewriting-based semantics.

9.1 Summary

We presented the automata-theoretic foundations and a tool for model checking properties of rewriting logic specifications in the linear temporal logic of rewriting (LTLR), where LTLR extends LTL by just adding spatial action patterns. We illustrated how spatial action patterns can be easily defined by means of equations in a similar way as state propositions. This strengthens and makes practical the claim that rewriting logic and LTLR are well-matched as a tandem of logics.

We presented an efficient on-the-fly algorithm for model checking LTLR properties under parameterized fairness. Parameterized fairness frequently occurs in many concurrent systems, but model checking under parameterized fairness has not been supported by existing model checking methods and tools before our work. We have implemented this algorithm in the Maude Fair LTLR model checker, developed as an extension of the Maude system at the C++ level, and have shown that it has comparable performance to that of other explicit-state model checkers that support fairness.

For infinite-state systems, we presented a number of infinite-state model checking techniques to verify *LTLR* properties: equational abstractions to define quotients of the system, folding abstractions to reduce the system by folding preorders, narrowing-based symbolic model checking, and predicate abstractions using E-unification. We showed that equational abstractions can be bisimilar, and that folding abstractions can be faithful. We also showed that narrowing-based model checking can be used together with equational abstraction to further reduce the system's logical state space.

For virtual synchronous multirate cyber-physical systems, we presented Multirate PALS to drastically reduce the system complexity. We defined a Multirate PALS transformation $\mathfrak{E} \mapsto \mathcal{MA}(\mathfrak{E}, T, \Gamma)$ from a multirate ensemble $\mathfrak{E}$ to its distributed real-time implementation $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ with global period T and performance parameters Γ, where $\mathfrak{E}$ and $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ are bisimilar. We showed how the Multirate PALS methodology can be used to formally specify and verify multirate distributed *hybrid* systems.

In order to make the Multirate PALS methodology available within the industrial modeling standard AADL, we defined the Multirate Synchronous AADL language. We defined the formal semantics of Multirate Synchronous AADL in rewriting logic. We have then developed the MR-SynchAADL plugin, together with a simple requirements specification language, to fully integrate rewriting-based model checking into the OSATE tool environment for AADL. We illustrated the effectiveness of our tool with case studies.

Finally, we presented the fully integrated modeling and model checking environment for Ptolemy II DE models. We defined the formal semantics of a significant subset of Ptolemy II DE models in rewriting logic, and also defined a simple property specification language for Ptolemy II DE models. We have integrated rewriting-based model checking into Ptolemy II using Ptolemy II's code generation infrastructure, so that Ptolemy II DE models can be easily verified from within Ptolemy II. This combines Ptolemy II's graphical modeling capability with rewriting-based formal verification.

9.2 Future Work

Richer Property Specification Languages. In this thesis LTLR under parameterized fairness was mainly considered as a well-matched property specification language for rewriting logic. Similarly, we can consider other classes of property specification languages. For example, model checking algorithms of TLR* under parameterized fairness have not been developed yet. We can consider similar extensions for probabilistic or timed temporal logics. Moreover, we can develop an efficient model checking algorithm for a richer class of *parameterized LTLR formulas*, extending the class of parameterized fairness assumptions, using parameter abstraction.

Infinite State Model Checking. In this thesis we presented bisimilar and faithful abstractions to deal with spurious counterexamples. Instead, we can also develop counterexample guided abstraction refinement methods [55] for rewriting-based abstraction methods. Another promising research direction is to combine our infinite-state model checking methods, based on narrowing and E-unification, with SMT techniques [34]. For example, some E-equality constraint that cannot be decided in the current framework can be directly solvable using SMT solvers.

Multirate PALS. The Multirate PALS pattern is fully constructive and can be automated as a theory transformation. Therefore, an obvious next step is to automate the Multirate PALS transformation for the generic rewriting-based framework in Section 6.4.2. Another immediate future work is to generalize Multirate PALS to distributed hybrid system with *physically correlated* environments. As discussed in Section 6.4, this is currently not allowed in the Multirate PALS methodology. Additionally, we can combine Multirate PALS with abstraction methods for hybrid systems [54].

Modeling Languages. For Multirate Synchronous AADL and Ptolemy II DE models, larger subsets of the languages can be formalized. Also, other model checking methods, such as statistical model checking, can be applied to analyze probabilistic models of embedded systems. One useful but missing feature in the current tools is to visualize counterexamples generated by model checking in the modeling tools: this should be fairly easy to achieve since our semantics preserves the structure of the design models. Finally, developing rewriting-based model checking environments for other modeling languages is also an interesting future research direction.

Part IV

Appendix

APPENDIX A

MORE LTLR CASE STUDIES AND IMPLEMENTATION

A.1 More Case Studies

All the case studies for this chapter, including those in Chapters 3 and 4, are available at `http://maude.cs.illinois.edu/tools/tlr`.

A.1.1 Fault-Tolerant Client-Server Communication

This section shows how the simple fault-tolerant client-server communication model in Example 4.2 can be verified under localized fairness using our tool. The configuration of the system is defined by the following module, where a client is represented as a term `[C,S,N,W]`, a server is represented as a term `[S]`, and a message is represented as a term `I <- {J,N}`:

```
fmod CLIENT-SERVER-SYNTAX is
  protecting NAT .
  sort Oid Nat? Conf .              subsort Nat < Nat? .
  op nil : -> Nat? [ctor] .
  op f : Oid Oid Nat -> Nat .       ops a b c : -> Oid [ctor] .
  op null : -> Conf [ctor] .
  op __ : Conf Conf -> Conf [ctor assoc comm id: null] .
  op [_] : Oid -> Conf [ctor] .                       *** servers
  op _<-{_,_} : Oid Oid Nat -> Conf [ctor] .          *** messages
  op [_,_,_,_] : Oid Oid Nat Nat? -> Conf [ctor] .    *** clients
endfm
```

For example, the term $[a]\ [b, a, 1, \texttt{nil}]\ [c, a, 2, \texttt{nil}]\ a$ `<-{`$b, 1$`}` describes a state containing one server, two clients, and one message.

The behavior is defined by the following system module CLIENT-SERVER in which the metadata attributes declare the localized fairness specification ($\mathcal{J} = \{ \{'req : 'C \backslash C\} \}$, $\mathcal{F} = \{ \{'reply : 'S \backslash S; 'C \backslash C\}, \{'rec : 'C \backslash C\} \}$):

```
mod CLIENT-SERVER is
  including CLIENT-SERVER-SYNTAX .
  vars C S I J : Oid .       vars N M : Nat .
  rl [req]  : [C,S,N,nil]
          => [C,S,N,nil] S <-{C,N}         [metadata "just(C)"] .
  rl [reply]: S <-{C,N} [S]
          => [S] C <-{S,f(S,C,N)}          [metadata "fair(S,C)"] .
  rl [rec]  : C <-{S,M} [C,S,N,nil]
          => [C,S,N,M]                     [metadata "fair(C)"] .
  rl [dupl] : I <-{J,M}      =>     I <-{J,M} I <-{J,M} .
  rl [loss] : I <-{J,M}      =>     null  .
endm
```

This system has an infinite number of reachable states from any initial state due to the *req* and *dupl* rules, but we can define a finite-state equational abstraction (see Section 5.3) by adding extra equations and rules in the following module (with similar metadata attributes):

```
mod CLIENT-SERVER-ABS is
  including CLIENT-SERVER .  vars C S I : Oid .    vars N M : Nat .
  eq I <-{C,N} I <-{C,N}  =  I <-{C,N} .
  rl [reply]: S <-{C,N} [S] =>  S <-{C,N} [S]  C <-{S,f(S,C,N)}
      [metadata "fair(S,C)"] .
  rl [rec]  : C <-{S,M} [C,S,N,nil] =>  C <-{S,M} [C,S,N,M]
      [metadata "fair(C)"] .
  rl [loss] : I <-{C,N} =>  I <-{C,N} .
endm
```

One of the key properties of this system is that any client should eventually receive the answer from its server, expressed as the formula $\Diamond answered(b)$ for client b, where the state proposition $answered(b)$ means that client b has a non-`nil` value in its fourth component:

```
mod CLIENT-SERVER-CHECK is
  protecting CLIENT-SERVER-ABS .    including LTLR-MODEL-CHECKER .
  subsort Conf < State .
  vars C S : Oid .    vars N M : Nat .      var CF : Conf .
  op answered : Oid -> Prop [ctor] .        op init : -> State .
  eq [C,S,N,M] CF |= answered(C) = true .
  eq CF |= answered(C) = false [owise] .
  eq init = [a] [b,a,1,nil] [c,a,2,nil] .
endm
```

By model checking the formula $\Diamond answered(b)$ without fairness, we can find a counterexample in the abstract model, where the rule *rec* has never been applied for client b even though it is infinitely often enabled:

```
Maude> red modelCheck(init,  <> answered(b)) .
result ModelCheckResult: counterexample(
{[a][b,a,1,nil][c,a,2,nil], {'req}}
{[a] a<-{b,1} [b,a,1,nil] [c,a,2,nil],  {'loss}}
{[a] [b,a,1,nil] [c,a,2,nil],  {'req}}
{[a] a<-{c,2} [b,a,1,nil] [c,a,2,nil],  {'reply}}
{[a] c<-{a,f(a,c,2)} [b,a,1,nil] [c,a,2,nil],  {'rec}}
{[a] [b,a,1,nil] [c,a,2,f(a,c,2)],  {'req}}
{[a] a<-{b,1} [b,a,1,nil][c,a,2,f(a,c,2)], {'reply}}
,
{[a] b<-{a,f(a,b,1)} [b,a,1,nil][c,a,2,f(a,c,2)], {'req}}
{[a] a<-{b,1} b<-{a,f(a,b,1)} [b,a,1,nil][c,a,2,f(a,c,2)], {'reply}})
```

The following command gives the model checking result of the formula $\Diamond answered(b)$ under the given localized fairness specification:

```
Maude> (pfmc init |= <> answered(b) .)
ltlr model check under localized fairness in CLIENT-SERVER-CHECK:
  init |= <> answered(b)
result Bool :
  true
```

A.1.2 An Unordered Communication Channel

This section illustrates a simple communication channel in which messages can get out of order. We specify the protocol as a rewrite theory slightly modified from [75]. In this protocol, a sender communicates with a receiver through an unordered channel. In order to achieve in-order communication in spite of the unordered nature of the channel, the sender sends each data item in a message together with a sequence number; and the receiver sends back an `ack` message indicating that it has received the item.

The state of the system is a 5-tuple $\{Buf_S, Ctr_S \mid Channel \mid Buf_R, Ctr_R\}$, where Buf_S is a buffer for the sender containing the current list of items to be sent, Buf_R is a buffer for the receiver storing the sequence of items already received, Ctr_S and Ctr_R are counters keeping track of the sequence number, and *Channel* is the unordered channel modeled by a multiset of messages. The protocol is specified by the following system module, where the imported module `NAT-LIST` defines a list data structure for numbers with the list-constructor `;` and the list-append operator `@`:

```
mod UNORDERED-CHANNEL is
  protecting NAT-LIST .
  vars N M J K : Nat . vars L P : List .   var C : Conf .

  sorts Conf Msg ChannelState .      subsort Msg < Conf .
  op null : -> Conf [ctor] .
  op __ : Conf Conf -> Conf [ctor assoc comm id: null] .
  op ack : Nat -> Msg [ctor] .
  op [_,_] : Nat Nat -> Msg [ctor] .
  op {_,_|_|_,_} : List Nat Conf List Nat -> ChannelState [ctor] .

  rl [snd]: {N ; L, M | C | P, K} => {N ; L, M | [N,M] C | P, K} .
  rl [rec]: {L, M | [N,J] C | P, J}
         => {L, M | ack(J) C | P @ (N ; nil), s(J)} .
  rl [ack]: {N ; L, J | ack(J) C | P, M} => {L, s(J) | C | P, M} .
endm
```

Since the system is infinite-state, we define an equational abstraction [139] (see Section 5.3) by adding an extra equation, and by adding an extra rule to keep the system coherent [75], where an unbounded number of duplicate messages are abstracted into a single message:

```
mod UNORDERED-CHANNEL-ABS is
  including UNORDERED-CHANNEL .
  vars N M J K : Nat .   vars L P : List .    var C : Conf .

  eq {L, M | [N,J] [N,J] C | P, K} = {L, M | [N,J] C | P, K} .
  rl [rec]: {L, M | [N,J] C | P, J}
        =>  {L, M | [N,J] ack(J) C | P @ (N ; nil), s(J)} .
endm
```

We are interested in the liveness property $\Diamond recQ(L)$, meaning that the receiver eventually gets the entire sequence L from the sender, where the state proposition $recQ(L)$ is equationally defined in the following module:

```
mod UNORDERED-CHANNEL-ABS-CHECK is
  protecting UNORDERED-CHANNEL-ABS .  including LTLR-MODEL-CHECKER .
  including SPATIAL-ACTION-PATTERN .  subsort ChannelState < State .
  vars L L1 L2 : List .      vars N K : Nat .      var C : Conf .
  op recQ : List -> Prop .   op init : -> State .
  eq {L1, N | C | L2, K} |= recQ(L) = (L == L2) .
  eq init = {0 ; 1 ; 2 ; nil , 0 | null | nil , 0} .
endm
```

However, the property $\Diamond recQ(L)$ is not satisfied without appropriate fairness assumptions, since the `snd` rule can keep resending a given element forever.

```
Maude> red modelCheck(init, <> recQ(0 ; 1 ; 2 ; nil)) .
result ModelCheckResult :
  counterexample({{0 ; 1 ; 2 ; nil,0 | null | nil,0},{'snd}}
                 {{0 ; 1 ; 2 ; nil,0 |[0,0]| nil,0},{'rec}}
                 {{0 ; 1 ; 2 ; nil,0 | ack(0)| 0 ; nil,1},{'ack}}
                 {{1 ; 2 ; nil,1 | null | 0 ; nil,1},{'snd}}
                 {{1 ; 2 ; nil,1 |[1,1]| 0 ; nil,1},{'rec}}
                 {{1 ; 2 ; nil,1 | ack(1)| 0 ; 1 ; nil,2},{'ack}}
                 {{2 ; nil,2 | null | 0 ; 1 ; nil,2},{'snd}}
                 ,
                 {{2 ; nil,2 |[2,2]| 0 ; 1 ; nil,2},{'snd}})
```

By simply assuming that such a situation cannot take place, we can verify the property $\diamond recQ(L)$ under the fairness assumption $\Box\diamond\neg\{'snd\}$:

```
Maude> red modelCheck(init, []<> ~ {'snd} -> <> recQ(0 ; 1 ; 2 ; nil)) .
result Bool: true
```

This command can be equivalently expressed as the following Full Maude command with an explicit fairness assumption:

```
Maude> (mc init |= <> recQ(0 ; 1 ; 2 ; nil)
         under (just : {'snd} => False) .)
ltlr model check in UNORDERED-CHANNEL-ABS-CHECK :
  init |= <> recQ(0 ; 1 ; s 1 ; nil)
under fairness :
  just : {'snd} => False
result Bool :
  true
```

Note that the spatial action pattern `{'snd}` cannot be directly expressed in LTL without modifying the given protocol specification.

A.1.3 Dekker's Algorithm

This section illustrates Dekker's algorithm, one of the earliest solutions to the mutual exclusion problem. Dekker's algorithm has two processes with entirely symmetric code. Process 1 sets variable `c1` to 1 to indicate that it wishes to enter its critical section. Process 2 does the same with variable `c2`. If one process, after setting its variable to 1, finds that the variable of its competitor is 0, then it enters its critical section. In case of a tie, the tie is broken using variable `turn` that takes values in $\{1, 2\}$.

Using a simple parallel language presented in [61, 81], the state of the system is modeled as a pair `{S, M}` with `S` a multiset of processes and `M` a shared memory. Each process is represented as a pair `[I,R]` with `I` its process id and `R` its program code to be executed next by process. The following system module `PARALLEL`, adapted from [61, 81], defines a simple parallel language, where the language's operational semantics is defined by rewrite rules for each language feature.

```
mod PARALLEL is
  protecting SEQUENTIAL .
  sorts Pid Process Soup MachineState .    subsort Process < Soup .
  op [_,_] : Pid Program -> Process [ctor] .
  op empty : -> Soup [ctor] .
  op _|_ : Soup Soup -> Soup [ctor comm assoc id: empty] .
  op {_,_} : Soup Memory -> MachineState [ctor] .

  var S : Soup .             var Q : Qid .         vars N X : Int .
  vars I J : Pid .           vars P R : Program .
  var T : Test .             var M : Memory .
  var U : UserStatement .    var L : LoopingUserStatement .

  rl [user]: {[I, U ; R] | S, M} => {[I, R] | S, M} .
  rl [loop]: {[I, L ; R] | S, M} => {[I, L ; R] | S, M} .
  rl [asgn]: {[I, Q := N ; R] | S, [Q,X] M}
        =>  {[I, R] | S, [Q,N] M} .
 crl [conT]: {[I, if T then P fi ; R] | S, M}
         => {[I, P ; R] | S, M}
   if eval(T,M) .
 crl [conF]: {[I, if T then P fi ; R] | S, M}
         => {[I, skip ; R] | S, M}
   if not eval(T,M) .
 crl [whiT]: {[I, while T do P od ; R] | S, M}
         => {[I, P ; while T do P od ; R] | S, M}
   if eval(T,M) .
 crl [whiF]: {[I, while T do P od ; R] | S, M}
         => {[I, R] | S, M}
   if not eval(T,M) .
  rl [rept]: {[I, repeat P forever ; R] | S, M}
         => {[I, P ; repeat P forever ; R] | S, M} .
endm
```

We refer to [61, 81] for the details about the module `SEQUENTIAL`, which defines the syntax of the language and some auxiliary functions such as `eval : Test Memory -> Bool`.

The initial state with two processes `p1` and `p2` for Dekker's algorithm is represented by the term `{[1,p1] | [2,p2], ['c1,0] ['c2,0] ['turn,1]}`, where `p1` and `p2` are defined in the following module:

```
mod DEKKER is
  including PARALLEL .
  subsort Int < Pid .                ops p1 p2 : -> Program .
  op crit : -> UserStatement .
  op rem : -> LoopingUserStatement .

  eq p1 = repeat
            'c1 := 1 ;
            while 'c2 = 1 do
                if 'turn = 2 then
                  'c1 := 0 ;
                  while 'turn = 2 do skip od ;
                  'c1 := 1
                fi
            od ;
            crit ;
            'turn := 2 ; 'c1 := 0 ; rem
         forever .

  eq p2 = repeat
            'c2 := 1 ;
            while 'c1 = 1 do
                if 'turn = 1 then
                  'c2 := 0 ;
                  while 'turn = 1 do skip od ;
                  'c2 := 1
                fi
            od ;
            crit ;
            'turn := 1 ; 'c2 := 0 ; rem
         forever .
endm
```

The code fragment of the critical section is abstracted as the constant `crit`, and the code fragment of the remaining part is abstracted as `rem`. We assume that `crit` is *terminating*, but `rem` may not be. This is achieved by declaring `crit` as a constant of sort UserStatement, and `rem` as one of sort LoopingUserStatement, where LoopingUserStatement is a subsort of UserStatement. Notice that both rules *user* and *loop* can be applied to the constant `rem` of sort LoopingUserStatement.

There are two requirements for Dekker's algorithm in LTL as follows, where the state proposition $\textit{in-rem}(i)$ holds iff Process i is in the `rem` section, the state proposition $\textit{in-crit}(i)$ holds iff Process i is in the `crit` section:

- $\Box\neg(\textit{in-crit}(1) \wedge \textit{in-crit}(2))$: two processes should not be in the critical section at the same time; and
- $\bigwedge_{i=1,2}(\neg\Diamond\Box\textit{in-rem}(i)) \rightarrow \Box\Diamond\textit{in-crit}(i)$: each process should enter its `crit` section infinitely often if it does *not* loop forever in its `rem` section.

The following functional module also declares the spatial action pattern $\textit{exec}(i)$, meaning that Process i has just been executed:

```
mod DEKKER-CHECK is
  protecting DEKKER .          including LTLR-MODEL-CHECKER .
  subsort MachineState < State .
  var I : Pid .                var P : Program .
  var S : Soup .               var M : Memory .
  var CXT : StateContext .     var R : RuleName .
  var SUB : StateSubstitution .

  ops in-crit in-rem : Pid -> Prop [ctor] .
  eq {[I, crit ; P] | S, M}  |= in-crit(I) = true .
  eq {[I,  rem ; P] | S, M}  |= in-rem(I) = true .

  op exec : Pid -> Action [ctor] .
  eq {CXT | R : 'I \ I ; SUB} |= exec(I)
   = (R == 'user) or (R == 'loop) or (R == 'asgn) or (R == 'conT) or
     (R == 'conF) or (R == 'whiT) or (R == 'whiF) or (R == 'rept) .

  op init : -> State .
  eq init = {[1, p1] | [2, p2], ['c1,0] ['c2,0] ['turn,1]} .
endm
```

The following command shows the model checking result of the mutual exclusion $\Box\neg(\textit{in-crit}(1) \wedge \textit{in-crit}(2))$, a state-based property that can also be verified using the existing Maude LTL model checker:

```
Maude> red modelCheck(init, [] ~ (in-crit(1) /\ in-crit(2))) .
result Bool: true
```

However, the property $\bigwedge_{i=1,2}(\neg\Diamond\Box\textit{in-rem}(i)) \rightarrow \Box\Diamond\textit{in-crit}(i)$ cannot be verified without suitable fairness assumptions, as shown by the following counterexample in which Process 2 is busy-waiting for Process 1 to change the value of `turn` but Process 1 is continuously idle:

```
Maude> red modelCheck(init,
               ((~ <>[] in-rem(1)) -> []<> in-crit(1)) /\
               ((~ <>[] in-rem(2)) -> []<> in-crit(2))) .
result ModelCheckResult :
  counterexample(
    {{[1,repeat ... forever]|[2,repeat ... forever],
      ['c1,0]['c2,0]['turn,1]},  {'repeat}}
    {{[1,'c1 := 1 ; ...]|[2,repeat ... forever],
      ['c1,0]['c2,0]['turn,1]},  {'assign}}
    {{[1,while 'c2 = 1 ...]|[2,repeat ... forever],
      ['c1,1]['c2,0]['turn,1]},  {'repeat}}
    {{[1,while 'c2 = 1 ...]|[2,'c2 := 1 ; ...],
      ['c1,1]['c2,0]['turn,1]},  {'assign}}
    {{[1,while 'c2 = 1 ...]|[2,while 'c1 = 1 ...],
      ['c1,1]['c2,1]['turn,1]},  {'while}}
    {{[1,while 'c2 = 1 ...]|[2,if 'turn = 1 then ...],
      ['c1,1]['c2,1]['turn,1]},  {'if}}
    {{[1,while 'c2 = 1 ...]|[2,'c2 := 0 ; ...],
      ['c1,1]['c2,1]['turn,1]},  {'assign}}
    ,
    {{[1,while ...]|[2,while 'turn = 1 do skip od ...],
      ['c1,1]['c2,0]['turn,1]},  {'while} })
```

To avoid such a "infinite busy-waiting" scenario, for each process i, we need to assume the weak fairness condition

$$\Diamond\Box \mathit{enabled}(\mathit{exec}(i)) \rightarrow \Box\Diamond \mathit{exec}(i).$$

The desired property can then be verified by the following command:[1]

```
Maude> red modelCheck(init,
             ((<>[] enabled(exec(1)) -> []<> exec(1)) /\
              (<>[] enabled(exec(2)) -> []<> exec(2)))
           ->
              ((~ <>[] in-rem(1)) -> []<> in-crit(1)) /\
              ((~ <>[] in-rem(2)) -> []<> in-crit(2))) .
result Bool: true
```

Since the spatial action pattern $\mathit{exec}(i)$ cannot be directly expressed in LTL, the original specification in [61, 81] had to be "cooked" by injecting event information into states to express $\mathit{exec}(i)$ in LTL, which leads to a less natural and more complex specification.

[1] Recall that for each spatial action pattern δ, the enabled state proposition $\mathit{enabled}(\delta)$ is automatically declared in our tool (see Section 4.2).

In our tool, a *renamed* basic action pattern $\{'\hat{l} : 'x_1 \backslash x_1 ; \ldots ; 'x_n \backslash x_n\}$ can be defined by a fairness item of one of the forms:

$$\mathit{just}[\hat{l}(x_1, \ldots, x_n)] \qquad \mathit{fair}[\hat{l}(x_1, \ldots, x_n)]$$

where $x_1, \ldots, x_n \in \mathit{vars}(q)$. and $\hat{l}$ is any name for a basic action pattern. For such a renamed pattern, the correspondence relation with a one-step proof term γ is automatically declared by our tool as follows:

$$\gamma \models \{'\hat{l} : 'x_1 \backslash x_1 ; \ldots ; 'x_n \backslash x_n\} \iff \gamma \models \{'l : 'x_1 \backslash x_1 ; \ldots ; 'x_n \backslash x_n\}.$$

A fairness item of the form $\mathit{just}['\hat{l}(x_1, \ldots, x_n)]$ declares the renamed pattern $\{'\hat{l} : 'x_1 \backslash x_1 ; \ldots ; 'x_n \backslash x_n\}$ in $\mathcal{J}$, and $\mathit{fair}['\hat{l}(x_1, \ldots, x_n)]$ declares the same pattern in $\mathcal{F}$. These renamed fairness items are useful to specify fairness conditions for a *group* of rules with different labels.

The localized fairness conditions for Dekker's algorithm can be succinctly specified using renamed fairness items. In the following redeclared rules, the metadata attribute `just[exec(I)]` in each rule declares the localized fairness specification $\mathcal{J} = \{\{'\mathit{exec} : 'I \backslash I\}\}$, where $\{'\mathit{exec} : 'I \backslash I\}$ corresponds to every rule l by renaming its rule label to *exec* such that for any one-step proof term γ, $\gamma \models \{'\mathit{exec} : 'I \backslash I\} \iff \gamma \models \{'l : 'I \backslash I\}$:

```
 rl [user]: {[I, U ; R] | S, M}
         => {[I, R] | S, M}         [metadata "just[exec(I)]"] .
 rl [loop]: {[I, L ; R] | S, M}
         => {[I, L ; R] | S, M}     [metadata "just[exec(I)]"] .
 rl [asgn]: {[I, Q := N ; R] | S, [Q,X] M}
         =>  {[I, R] | S, [Q,N] M}  [metadata "just[exec(I)]"] .
crl [conT]: {[I, if T then P fi ; R] | S, M}
         => {[I, P ; R] | S, M}
   if eval(T,M)                     [metadata "just[exec(I)]"] .
crl [conF]: {[I, if T then P fi ; R] | S, M}
         => {[I, skip ; R] | S, M}
   if not eval(T,M)                 [metadata "just[exec(I)]"] .
crl [whiT]: {[I, while T do P od ; R] | S, M}
         => {[I, P ; while T do P od ; R] | S, M}
   if eval(T,M)                     [metadata "just[exec(I)]"] .
crl [whiF]: {[I, while T do P od ; R] | S, M}
         => {[I, R] | S, M}
   if not eval(T,M)                 [metadata "just[exec(I)]"] .
 rl [rept]: {[I, repeat P forever ; R] | S, M}
         => {[I, P ; repeat P forever ; R] | S, M}
                                    [metadata "just[exec(I)]"] .
```

Under the given localized fairness specification $\mathcal{J} = \{\{'exec : 'I \backslash I\}\}$, the liveness property $\bigwedge_{I=1,2}(\neg\Diamond\Box\texttt{in-rem}(I)) \to \Box\Diamond\texttt{in-crit}(I)$ can be simply verified by using the parameterized-fair model checking command:

```
Maude> (pfmc init |= ((~ <>[] in-rem(1)) -> []<> in-crit(1)) /\
                     ((~ <>[] in-rem(2)) -> []<> in-crit(2)) .)
ltlr model check under localized fairness in DEKKER-CHECK :
  init |= ((~ <>[] in-rem(1)) -> []<> in-crit(1)) /\
          ((~ <>[] in-rem(2)) -> []<> in-crit(2))
result Bool :
  true
```

A.1.4 Position Fairness

Position weak fairness means that if a position in a term t is eventually always enabled,[2] then that position of t is rewritten infinitely many times. Likewise, position strong fairness means that if a position in a term t is enabled infinitely often, then that position of t is rewritten infinitely often. Such position fairness can be expressed by using spatial action patterns parameterized by context terms, e.g., $\{CXT \mid 'l\}$. If a one-step proof term $u[l(\theta)]_p$ corresponds to the *ground* spatial action pattern $\theta\{CXT \mid 'l\}$, then by definition $\theta CXT =_E u[\Box]_p$, which represents the position p of the term u. In this case, a fairness condition for $\{CXT \mid 'l\}$ localized to a context variable CXT is parameterized to such positions. For example, consider the following simple model with two rewrite rules.

```
mod POSITION-MODEL is
  including SATISFACTION .
  sort Conf .                     subsort Conf < State .
  ops a b : -> Conf [ctor] .      op reach : Conf -> Prop [ctor] .
  op f : Conf Conf -> Conf .      eq f(b,b) = b .
  eq C:Conf |= reach(D:Conf) = (C:Conf == D:Conf) .

  rl [toB]: a => b .              rl [toA]: b => a .
endm
```

Even under the strong *rule* fairness specification $\mathcal{F} = \{\{'toA\}, \{'toB\}\}$, the LTL formula $\Diamond reach(b)$ does *not* hold from the initial state $f(a,a)$, since there exists the counterexample that never visits the state b:

$$f(a,a) \longrightarrow f(a,b) \longrightarrow f(a,a) \longrightarrow f(b,a) \longrightarrow f(a,a) \longrightarrow \cdots$$

[2] A term position p in a term t is enabled iff some rewrite rule can be applied at the position p of the term t.

However, $\Diamond reach(b)$ holds under the strong *position* fairness specification $\mathcal{F} = \{\{CXT \mid {'toB}\}\}$. Given the initial state $f(a, a)$, there are five reachable states: $f(a, a)$, $f(a, b)$, $f(b, a)$, a, and $f(b, b) = b$. If the state $f(a, b)$ is visited infinitely often, then the spatial action pattern $\{f(\square, b) \mid {'toB}\}$ is enabled infinitely often. By the strong position fairness assumption, the rule toB must be infinitely often applied where the resulting term $f(b, b)$ is immediately reduced to b by the equation. Similarly, if the state $f(b, a)$ is visited infinitely often, then the state b will also be visited infinitely often. Therefore, any infinite path from the state $f(a, a)$ will visit the state b infinitely often, and thus satisfies the formula $\Diamond reach(b)$.

In our tool, the context terms for this model can be declared automatically by the module expression CONTEXT[POSITION-MODEL] as follows:

```
(mod POSITION-MODEL-CONTEXT is
  including LTLR-MODEL-CHECKER .
  including SPATIAL-ACTION-PATTERN .
  protecting CONTEXT[POSITION-MODEL] .
  subsort Context$Conf < StateContext .
endm)
```

The following model checking command shows that without fairness, the formula $\Diamond reach(b)$ does not hold from the initial state $f(f(a, a), f(a, a))$:

```
Maude> (mc f(f(a,a),f(a,a))  |=  <> reach(b) .)
ltlr model check in POSITION-MODEL-CONTEXT :
  f(f(a,a),f(a,a)) |= <> reach(b)
result ModelCheckResult :
  counterexample(
    {f(f(a,a),f(a,a)),{'toB}}    {f(f(b,a),f(a,a)),{'toB}}
    {f(b,f(a,a)),{'toA}}         {f(a,f(a,a)),{'toB}}
    ,
    {f(a,f(b,a)),{'toB}}         {f(b,f(b,a)),{'toA}})
```

Under the localized fairness specification $\mathcal{F} = \{\{CXT \mid {'toB}\}\}$, the formula $\Diamond reach(b)$ can be verified by our model checker as follows:

```
Maude> (mc f(f(a,a),f(a,a))  |=  <> reach(b)
          under fair({CXT:StateContext | 'toB}) .)
ltlr model check in POSITION-MODEL-CONTEXT :
  f(f(a,a),f(a,a)) |= <> reach(b)
under fairness :
  fair({CXT:StateContext | 'toB})
result Bool :
  true
```

A.1.5 Sliding Window Protocol

The balanced sliding window protocol is a symmetric protocol to ensure reliable communication in both directions of a communication channel [162]. The verification task for this protocol is not simple, since the specification involves unbounded queue data structures and dynamic fairness conditions. The model checking of a liveness property requires a fairness assumption for each message in transit as well as for each process.

In the balanced sliding window protocol, there are two entirely symmetric processes P and Q connected to each other through a lossy channel. Packets exchanged by the processes are pairs $[j, w]$, with j an index number and w a data word. Process P contains an array I_P of packets to be sent, another array O_P of received packets, and a FIFO queue F_P of packets in transit to be received. Process P also has three variables to describe a state of the process as follows: s_P is the lowest index of a packet not yet received from the other process, a_P is the lowest index of a packet sent but not yet acknowledged (s_P and a_P are initially 0), and l_P is a fixed "window" bound allowing sending packets before being acknowledged. The acknowledgement is implicitly provided by sending and receiving messages. That is:

1. P has received all the packets of process Q indexed from 0 to $s_p - 1$ and stored them in O_P;
2. P has sent all the packets in I_P from 0 to $a_p - 1$, and received their acknowledgement from Q; and
3. P can send a packet $[j, w]$ to acknowledge the receipt of all the packets of Q indexed from 0 to $j - l_P$, so that process Q need not send those packets $0, \ldots, j - l_P$ again.

A state of process P is expressed as a term $[P : s_P, a_P, l_P, l_Q, I_P, O_P, F_P]$, where l_Q is a fixed bound of the other process. The signature is defined in the following functional module in Maude, where the necessary data structures are defined in the module `QID-ARRAY&LIST`:

```
fmod SLIDING-WINDOW-SYNTAX is
  protecting INT .                    protecting QID-ARRAY&LIST .
  sorts Pid NodeQState GlobalState .
  op _&_ : NodeQState NodeQState -> GlobalState [ctor comm] .
  op [_:_,_,_,_,_,_,_] :
        Pid Nat Nat NzNat NzNat QidArray QidArray MsgList
    ->  NodeQState [ctor] .
endfm
```

$\underline{\text{send}}_P$: { $a_P \leq j < s_P + l_P$ }
  begin enqueue $I_P[j]$ to F_Q **end**

$\underline{\text{loss}}_P$: { $[j, w] \in F_P$ }
  begin remove $[j, w]$ from F_P **end**

$\underline{\text{recv}}_P$: { $[j, w] :=$ dequeue(F_P) }
  begin
    if $O_P[j] = \perp$ **then** $O_P[j] :=$ w;
      $s_P := \min\{k \mid O_P[k] = \perp\}$;
      $a_P := \max(a_P,\ j - l_Q + 1)$ **fi** **end**

Figure A.1: The balanced sliding window protocol (for process P) [162].

The behavior of this protocol can be summarized by the pseudo code in Figure A.1. Process P can *send* any packet $[j, w]$ in its array I_P if no acknowledgement for the packet has yet been received ($j \geq a_P$) but all the packets of the other process indexed from 0 to $j - l_P$ have already been received ($j - l_P < s_P$), that is, $a_P \leq j < s_P + l_P$. When *receiving* a packet $[j, w]$, an already received packet in O_P is ignored; otherwise, the packet is added to O_P, s_P is set to the smallest index that has not yet been received, and a_P is set to $max(a_P, j - l_Q + 1)$ to stop retransmission of an acknowledged packet whose index is less than or equal to $j - l_Q$. Finally, the *loss* of a packet can happen at any time.

The behavior is specified in the following system module, where an array is a semicolon-separated set of $[J, W]$ with index J, the term I_p `<` $j : k$ `>` denotes a sliced array that returns all entities between j and k, and `noValue(`O_P`,`J`)` returns `true` if the array O_P has no value at index J:

```
mod SLIDING-WINDOW is
  including SLIDING-WINDOW-SYNTAX .
  vars P Q : Pid .            var W : Qid .
  vars J SP SQ AP AQ : Nat . vars LP LQ : NzNat .
  vars FP FQ L G : MsgList . vars IP OP IQ OQ ARRAY : QidArray .
 crl [send]:  [P : SP, AP, LP, LQ, IP, OP, FP]
            & [Q : SQ, AQ, LQ, LP, IQ, OQ, FQ]
          =>  [P : SP, AP, LP, LQ, IP, OP, FP]
            & [Q : SQ, AQ, LQ, LP, IQ, OQ, FQ [J,W]]
    if SP + LP > AP /\ [J,W] ; ARRAY := IP < AP : SP + LP - 1 >
        [metadata "just(P,J,W)"] .
  rl [recv]: [P : SP, AP, LP, LQ, IP, OP, [J,W] FP]
          => if noValue(OP,J)
             then [P : 1st-undef(OP ; [J,W]), max(AP, J - LQ + 1),
                     LP, LQ, IP, OP ; [J,W], FP]
             else [P : SP, AP, LP, LQ, IP, OP, FP] fi
        [metadata "fair(P,J,W)"] .
  rl [loss]: [P : SP, AP, LP, LQ, IP, OP, L [J,W] G]
          =>  [P : SP, AP, LP, LQ, IP, OP, L G] .
endm
```

Notice that when executing the *send* rule with a substitution θ, by the matching equation `[J,W] ; ARRAY := IP < AP : SP + LP - 1 >`, the variables `J`, `W`, and `ARRAY` are instantiated by matching `[J,W] ; ARRAY` with the term θ(`IP < AP : SP + LP - 1 >`) modulo $E \cup B$.

This system module also declares the localized fairness specification using the rule attributes in which each fairness condition is parameterized over each process P and packet $[J, W]$:

$$\mathcal{J} = \{\{'send : 'P \backslash P; 'J \backslash J; 'W \backslash W\}\}, \quad \mathcal{F} = \{\{'recv : 'P \backslash P; 'J \backslash J; 'W \backslash W\}\},$$

meaning that:

1. if a process P is continuously able to send a packet $[J, W]$, then the process must send the packet $[J, W]$ infinitely many times, and

2. if a process P can receive a packet $[J, W]$ infinitely often, then the process must receive the packet $[J, W]$ infinitely many times.

Because the number of states is infinite due to the unbounded queue, we define an equational abstraction to obtain a finite state space as follows:

1. the FIFO queues are abstracted into bags by adding a commutativity equation, and

2. duplicate messages in transit are collapsed into a single message by identifying repeated packets.

Although a commutativity equation is *not* terminating, we can gain the same effect using commutativity axioms. The following functional module adds the commutativity axioms, where the fifo queue in the last component is abstracted into a set of packets, i.e., QidArray:

```
fmod SLIDING-WINDOW-ABSTRACTION-SYNTAX is
  protecting INT .
  protecting QID-ARRAY&LIST .
  sorts Pid NodeQState GlobalState .
  op _&_ : NodeQState NodeQState -> GlobalState [comm ctor] .
  op [_:_,_,_,_,_,_,_] :
        Pid Nat Nat NzNat NzNat QidArray QidArray QidArray
    ->  NodeQState [ctor] .
endfm
```

The following system module then adds the abstraction equation, where all the rewrite rules are redeclared since the signature has been changed:

```
fmod SLIDING-WINDOW-ABSTRACTION is
  including SLIDING-WINDOW-ABSTRACTION-SYNTAX .
  vars P Q : Pid .           var W : Qid .
  vars J SP SQ AP AQ : Nat . vars LP LQ : NzNat .
  vars FP FQ L G : QidArray . vars IP OP IQ OQ ARRAY : QidArray .

 crl [send]:  [P : SP, AP, LP, LQ, IP, OP, FP]
            & [Q : SQ, AQ, LQ, LP, IQ, OQ, FQ]
          =>  [P : SP, AP, LP, LQ, IP, OP, FP]
            & [Q : SQ, AQ, LQ, LP, IQ, OQ, FQ ; [J,W]]
    if SP + LP > AP /\ [J,W] ; ARRAY := IP < AP : SP + LP - 1 >
        [metadata "just(P,J,W)"] .
  rl [recv]: [P : SP, AP, LP, LQ, IP, OP, [J,W] ; FP]
          => if noValue(OP,J)
             then [P : 1st-undef(OP ; [J,W]), max(AP, J - LQ + 1),
                   LP, LQ, IP, OP ; [J,W], FP]
             else [P : SP, AP, LP, LQ, IP, OP, FP] fi
        [metadata "fair(P,J,W)"] .
  rl [loss]: [P : SP, AP, LP, LQ, IP, OP, L ; [J,W]]
          => [P : SP, AP, LP, LQ, IP, OP, L] .

  *** abstraction equations and rules
  eq [P : SP, AP, LP, LQ, IP, OP, FP ; [J,W] ; [J,W]]
   = [P : SP, AP, LP, LQ, IP, OP, FP ; [J,W]] .

  rl [recv]: [P : SP, AP, LP, LQ, IP, OP, [J,W] ; FP]
          => if noValue(OP,J)
             then [P : 1st-undef(OP ; [J,W]), max(AP, J - LQ + 1),
                   LP, LQ, IP, OP ; [J,W], [J,W] ; FP]
             else [P : SP, AP, LP, LQ, IP, OP, [J,W] ; FP] fi
        [metadata "fair(P,J,W)"] .

  rl [loss]: [P : SP, AP, LP, LQ, IP, OP, L ; [J,W]]
          => [P : SP, AP, LP, LQ, IP, OP, L ; [J,W]] .
endfm
```

Notice that two new rules are added at the end to ensure coherence of the rules with respect to the abstraction equations and thus the executability of the abstract model (see Section 5.3).

The liveness property we are interested in is that all messages should be eventually delivered, which is specified by the formula $\Diamond$*success*. The state proposition *success* holds iff Process P receives all the messages in I_Q from Process Q and Process Q receives all the messages in I_P from Process P:

```
(mod SLIDING-WINDOW-CHECK is
  including SATISFACTION .
  protecting SLIDING-WINDOW-ABSTRACTION .
  subsort GlobalState < State .
  ops p q : -> Pid .

  vars P Q : Pid .            vars I SP SQ AP AQ : Nat .
  vars LP LQ : NzNat .        vars FP FQ : MsgList .
  vars IP OP : QidArray .

  op success : -> Prop .
  eq [P : SP,AP,LP,LQ,IP,IQ,FP] & [Q : SQ,AQ,LQ,LP,IQ,IP,FQ] |=
       success = true .

  ops initP initQ : -> NodeQState .
  eq initP = [p : 0,0,2,2,[0,'C];[1,'B];[2,'A],empty,empty] .
  eq initQ = [q : 0,0,2,2,[0,'X];[1,'Y];[2,'Z],empty,empty] .
endm)
```

However, $\Diamond$*success* is not satisfied without the fairness assumptions; the rule *loss* can preclude all communications without fairness. But if we assume the localized fairness specification declared in the rule attributes, we can verify the formula $\Diamond$*success* by the following model checking command:

```
Maude> (pfmc initP & initQ |= <> success .)
ltlr model check under localized fairness in SLIDING-WINDOW-CHECK :
  initP & initQ |= <> success
result Bool :
  true
```

Table A.1 presents the model checking results from the initial states with different input array sizes and window bounds, conducted on an Intel Core 2 Duo 2.66 GHz running Mac OS X 10.6, where "#Fairness" denotes the total number of fairness instances generated during model checking.

Size	Bound	States	Time	#Fairness
3	1	420	0.2	
3	2	1596	1.7	12
3	3	4095	5.7	
5	1	6900	5.5	
5	2	32256	42.6	20
5	3	123888	223.8	

Table A.1: Results for models with bounded sliding window protocol

A.2 The Model Checker Implementation

The Maude Fair LTLR model checker (introduced in Chapters 3 and 4) has been implemented at the C++ level within the Maude system [61], by extending the existing LTL model checker [81]. Our tool generally consists of the following four components:

1. the module to construct a Büchi automaton $\mathscr{B}_{\neg\varphi}$ that recognizes the negation of a given LTLR formula φ;
2. the graph traversal engine that constructs the corresponding LKS from a computable rewrite theory;
3. the model checking algorithms (under parameterized fairness); and
4. the user interface of the model checker in Full Maude.

For efficiency reasons, the first three components are implemented at the C++ level. But the user interface of the model checker, particularly for metadata rule annotations, has been implemented by extending Full Maude [73] using Maude's reflective capabilities [61], since it involves several theory transformations to automate such an interface.

For generating a Büchi automaton $\mathscr{B}_{\neg\varphi}$ from the negated LTLR formula φ, the Maude LTLR model checker reuses the existing LTL model checker implementation (see [81, 80] for details). For automata-based verification of LTLR formulas, as discussed in Section 3.3.2, spatial action patterns do not need to be distinguished from state propositions. Therefore, the LTLR formula $\neg\varphi$ is first transformed into the syntactically equivalent LTL formula by regarding spatial action patterns as state propositions, and the same algorithm as in the LTL case is then used to generate $\mathscr{B}_{\neg\varphi}$.

Given a Maude specification of a rewrite theory $\mathcal{R}$, the associated LKS $\bar{\mathcal{K}}(\mathcal{R}, k)_\Pi$ is generated *on-the-fly* from an initial state $[t]_E$, and therefore only *requested* states and transitions are created during model checking. Each state (resp., transition) generated in the LKS keeps two bit vectors to record: (i) which state propositions (resp., spatial action patterns) have been tested, and (ii) which state propositions (resp., spatial action patterns) were satisfied in the state (resp., transition). The state/event synchronous product $\bar{\mathcal{K}}(\mathcal{R}, k)_\Pi \otimes \mathscr{B}_{\neg\varphi}$ is also constructed on-the-fly whenever a new state or transition is explored, while the nested depth-first search algorithm [109] is applied to the product automaton to find a counterexample, in the same way as the existing Maude LTL model checker [81].

For space optimization purposes, whenever a one-step proof term λ is generated for each one-step rewrite, all spatial action patterns in the input formula φ are tested at once and the full term representation of λ is then discarded from memory (when a counterexample is found, such one-step proof terms can be re-computed to display them). On the other hand, a full term representation of each state is preserved during model checking, as is the existing LTL model checker [81], to check whether a newly generated state has already been included in the currently generated LKS.

Different model checking algorithms are applied by the tool for handling different kinds of fairness conditions, because the general algorithms are more computationally expensive. All model checking algorithms are based on the on-the-fly emptiness checking algorithms, where fairness assumptions are incorporated into their acceptance conditions:

- For an LTLR formula with no fairness requirements, we use the nested depth first search algorithm [109] for Büchi automata.
- If only weak fairness conditions are provided, we use the SCC-based algorithm [66] for generalized Büchi automata, where weak fairness conditions are directly incorporated as acceptance conditions.
- For strong fairness conditions, we use the SCC-based algorithm for Streett automata, described in Section 4.3.3, where strong fairness conditions are directly incorporated as acceptance conditions.

If some of the strong/weak fairness conditions are parameterized, then those model checking algorithms are combined with the parameter abstraction algorithm that computes realized substitutions in SCCs.

Our Fair model checker assumes that the system specification satisfies the sufficient conditions for FIP in Theorem 4.1, and computes the realized substitutions according to the matching processes described in the proof of Theorem 4.1. Besides dealing with fairness, the Maude Fair LTLR model checker also generates shorter counterexamples than the previous model checkers in Maude. When a counterexample is found, we perform a backward breadth-first search from loop states to the initial states, using only *already visited* states to find the shortest prefix in the explored state space.

APPENDIX B

MORE DETAILS ON MULTIRATE PALS

B.1 Formalizing Specification of Asynchronous Models

This section presents the formal specification of the asynchronous real-time system $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ as a rewrite theory in Real-Time Maude. For simplicity reasons we assume that all components in the entire system have different identifiers. We assume that the local clock of a component j in $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ is given by a continuous and monotonic (with respect to the relation $\leq$) function $c_j : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ satisfying $|c_j(x) - x| < \epsilon$ for all $x \in \mathbb{R}_{\geq 0}$.

B.1.1 Generic Machines and Wrappers

All typed machines and wrappers have an input buffer of incoming messages of sort MsgConfiguration denoting multisets of messages, so we declare a generic class `Machine`:

```
class Machine | inBuffer : MsgConfiguration .
```

A wrapper is a machine with an inner configuration consisting of the inner machine/wrapper object and the messages sent by that object, where sort NEConfiguration denotes non-empty multisets of objects and messages:

```
class Wrapper | innerConf : NEConfiguration,
                ignoreNbs : OidSet .
subclass Wrapper < Machine .
```

The attribute `ignoreNbs` is used for the wrapper structure (in Figure 6.9) of slow components in a fast subensemble. When a wrapper receives a set of messages, it immediately relays those messages whose senders or receivers are in the `ignoreNbs`, a semicolon-separated set of object identifiers.

The following rewrite rules move the messages that should be taken care of by *other* wrappers up and down the wrapper hierarchy. The point is of course that the wrappers dealing with the communication with the faster subcomponents (i.e., the two innermost wrappers in Figure 6.9) should not "restrict" the movement of messages to/from the environment, which should be taken care of by the other wrappers, and vice versa:

```
var P : Nat .  var D : Data .  vars O O' : Oid .    var OS : OidSet .
var CONF : Configuration .    vars MSGS MSGS' : MsgConfiguration .

rl [forward-in-ignored-messages]:
    < O : Wrapper |
        inBuffer : (to O from O' (P,D)) MSGS,
        innerConf : (< O : Machine | inBuffer : MSGS' > CONF),
        ignoreNbs : O' ; OS >
  =>
    < O : Wrapper |
        inBuffer : MSGS,
        innerConf : (< O : Machine | inBuffer : (to O from O' (P,D))
                                                MSGS' > CONF) > .

rl [forward-out-ignored-messages]:
    < O : Wrapper | innerConf : (to O' from O (P,D)) CONF,
                    ignoreNbs : O' ; OS >
  =>
    < O : Wrapper | innerConf : CONF >
    (to O' from O (P,D)) .
```

All actions in the wrappers moving messages back and forth should now only deal with messages that should not be ignored. Since each message will "sift up/down" to its appropriate wrapper in zero time, we can wait to apply a rewrite rule (other than the two above) until the message set to be treated contains only "relevant" messages. That is, each rule moving a set `MSGS` of messages into the input buffer of an inner object or moving them out from the inner configuration has a condition `onlyRelevantMsgs(MSGS, OS)` that returns `true` if the message set `MSGS` does not contain any message whose sender/receiver belongs to the ignore set `OS`:

```
op onlyRelevantMsgs : MsgConfiguration OidSet -> Bool .
eq onlyRelevantMsgs((to O from O' (P,D)) MSGS, O ; OS) = false .
eq onlyRelevantMsgs((to O from O' (P,D)) MSGS, O' ; OS) = false .
eq onlyRelevantMsgs(MSGS, OS) = true [owise] .
```

B.1.2 Typed Machines

A typed machine in an asynchronous distributed real-time system goes in rounds according to its local clock and its given period, and has an input buffer in which it collects messages. When it executes a new round, it reads these messages, performs a transition, and sends out the generated output messages (to be picked up by the surrounding wrapper) whenever its execution finishes. Each machine M_j is modeled by an object instance of a subclass $C_{[j]}$ of the following class `Machine`:

```
class TypedMachine |  period : Time,
                      clock : Time,
                      roundTimer : Time,
                      state : DlyState,
                      outBuffer : DlyConfiguration,
                      localWiring : LocalWiring .

class C_1 .
...
class C_k .
subclass C_1 ... C_k < TypedMachine .
subclass TypedMachine < Machine .
```

Several typed machines, say, $M_{j_1}, \ldots, M_{j_r}$, can all be of the same type, and can therefore all belong to the same subclass, i.e.,

$$C_{[j_1]} = \cdots = C_{[j_r]}.$$

The attribute `period` denotes the period of the typed machine, the `clock` attribute shows the value of the local clock of the component, and the `roundTimer` is the timer that expires at the start of each new round.

The `state` attribute denotes the local state of the machine. The state component of machine j has sort S_j. For convenience, we add a supersort State of all such states. It may take some time to compute the next local state of a machine. During this computation time, the local state has the value $[s, t]$, where s is the next state, and t is the time remaining until the execution of the transition is finished. Such a term $[s, t]$ is called a *delayed state*, where the sort DlyState is defined as follows:

```
sorts State DlyState .
subsort S_1 ... S_|J_s ∪ J_F| < State < DlyState .
op [_,_] : State Time -> DlyState [ctor right id: 0] .
```

Notice that `0` has been defined as a *right identity* of the operator `[_,_]`, meaning that $[s, 0] = s$.

The messages produced by performing a transition can only be propagated to the surrounding wrapper after the execution of the transition has finished. Meanwhile, they are stored in the `outBuffer`. The messages have the form

`to` o `from` o' `(`p`,`d`)`,

where o and o' are the identifiers of the receiver and sender components, respectively, p is the number of the receiving "port," d is the data element sent, and Data is a supersort of the sorts $D_1 \ldots D_n$ of the data in the wires:

```
sort Data .
subsorts D1 ... Dn < Data .
msg to_from_(_,_) : Oid Oid NzNat Data -> Msg .
```

Likewise, during the execution of a transition, the messages *msgs* being generated by the transition are not yet ready, and have the form $[msgs, t]$:

```
sort DlyConfiguration .
subsort Configuration < DlyConfiguration .
op [_,_] : Configuration Time -> DlyConfiguration [ctor right id: 0] .
```

When `roundTimer` expires, the messages `NEMSGS` in the `inBuffer` are read,[1] and a transition is taken. Since different classes will have different transitions, executing transitions is modeled by a *family* of rewrite rules, one for each class $C_{[j]}$. Notice that the resulting state and messages are delayed by a value $\alpha_{min} \leq$ `X-DLY` $\leq \alpha_{max}$. In addition, the `roundTimer` must be reset to expire at the same time in the next round (i.e., to the `PERIOD`):

```
var X-DLY : Time .            vars S NEXT-STATE : State .
var W : LocalWiring .         var NEMSGS : NEMsgConfiguration .
var PERIOD : NzTime .         var d_{j_1} : D^j_{o_1} . ... var d_{j_{m_j}} : D^j_{o_{m_j}} .

crl [applyTrans]:
    < j : C[j] | inBuffer : NEMSGS,  state : S,   period : PERIOD,
                 roundTimer : 0,     localWiring : W >
  =>
    < j : C[j] | inBuffer : none,
                 state : [NEXT-STATE, X-DLY],
                 roundTimer : PERIOD,
                 outBuffer : [makeMsg(j, W, (d_{j_1},...,d_{j_{m_j}})), X-DLY] >
  if inputInAllPorts(NEMSGS, j)
  /\ X-DLY >= α_min and X-DLY <= α_max
  /\ ((vect_[j](NEMSGS), S), (NEXT-STATE, (d_{j_1},...,d_{j_{m_j}}))) ∈ δ_{M_j} .
```

[1]By Definition 6.1, each typed machine has non-empty inputs and non-empty outputs. The sort NEMsgConfiguration of `NEMSGS` denotes non-empty multisets of messages.

The condition `inputInAllPorts(NEMSGS,`j`)`, meaning that the machine has received inputs in all its input "ports," is needed for the hierarchical case in which there are *two* PALS wrappers surrounding the machine.[2] Given a *complete* set `NEMSGS` of messages of the form

$$\texttt{(to } j \texttt{ from } j'_1 \texttt{ } (1, d_1)\texttt{)} \ldots \texttt{(to } j \texttt{ from } j'_{n_j} \texttt{ } (n_j, d_{n_j})\texttt{)},$$

the function $vect_{[j]}$(`NEMSGS`) returns the vector of inputs $(d_1, \ldots, d_{n_j})$. The function `makeMsg` looks at the local wiring diagram `W`, takes the vector of output data from j, and produces the set of messages for the machines and environment getting inputs from that wire.

When the output messages are "ready" (i.e., the execution time of the transition producing the messages is over), those messages are sent out of the machine object, and should be picked up by the surrounding wrapper:

```
rl [sendMsgs]:
    < O : TypedMachine | outBuffer : NEMSGS >
   =>
    < O : TypedMachine | outBuffer : none > NEMSGS .
```

B.1.3 Local Wiring Diagrams

Each component is also assumed to know its local wiring diagram in the `localWiring` attribute; that is, which components and ports are connected to its output ports in the synchronous system. This knowledge is stored in a data structure called a *local wiring*, defined as follows:

```
sort LocalWiring .
op _-->_._ : Nat Oid Nat -> LocalWiring [ctor] .
op noWiring : -> LocalWiring [ctor] .
op _;_ : LocalWiring LocalWiring
                      -> LocalWiring [ctor assoc comm id: noWiring] .
```

A connection p `-->` j `.` p' says that the output port p of the current component is connected to the input port p' of component j. A *local wiring* is then a set of such single connections formed with the associative-commutative union operator `_;_` with identity the empty set constant `noWiring`. However, a connection is here only a *reference* for asynchronous message passing, and not a real "wired" connection as in the synchronous model.

[2]Since the PALS wrappers expires at the same time as the typed machine's timer, the machine must "wait" for zero time for the messages to trickle down from the PALS wrappers before executing a transition.

For a slow component in a fast subensemble, the "short-circuiting" of messages across hierarchical boundaries is formalized by modifying the local wiring diagrams in the `localWiring` attribute. For example, consider a hierarchical ensemble $\mathfrak{E}$ with a subensemble $\mathfrak{E}_{se}$, where the environment index and source function of $\mathfrak{E}_{se}$ are, respectively, env_{se} and src_{se}. Output from a component in $\mathfrak{E}$ to the "port" (se, p) (i.e., the pth input port of $\mathfrak{E}_{se}$) is considered by components inside $\mathfrak{E}_{se}$ to be input from the environment output port (env_{se}, p). That is, on the subdomain of input ports of $\mathfrak{E}_{se}$, the source function of the whole system is:

$$
\begin{array}{ll}
(j,p) \mapsto (src[se/env_{se}])(src_{se}(j,p)) & \text{if } (\exists p')\ src_{se}(j,p) = (env_{se}, p') \\
(j,p) \mapsto src_{se}(j,p) & \text{otherwise.}
\end{array}
$$

Likewise, we must also "short-circuit" links *from* the inner subsystem to an outer component. That is, for any (k, p), with k a concrete machine index in $\mathfrak{E}$, the real source in the composed system is:

$$
\begin{array}{ll}
(k,p) \mapsto (src_{se}[se/env_{se}])(src(k,p)) & \text{if } (\exists p')\ src(k,p) = (se, p') \\
(k,p) \mapsto src(k,p) & \text{otherwise.}
\end{array}
$$

B.1.4 The k-Machine Wrapper

The k-machine is the distributed version of the more abstract k-step deceleration pattern of Definition 6.5. This wrapper should get a k-tuple of inputs for each input port of the underlying machine at the start of each slow period. The first data item of each message should immediately be sent to the layer below. When the next fast period starts, the second data item in each incoming messages is sent to the object that it wraps around, and so on. Output handling is equally simple:

1. get output from the fast machine inside;
2. store it in an output buffer; and
3. whenever it has received the kth such chunk of messages, it composes them into k-tuples and sends them upstream immediately.

Since the k-machine wrapper may not be able to wait for the latest outputs, due to communication delays, its timer provides a cutoff time when the wrapper sends out whatever outputs it has by this cutoff time, padded with $\perp$ values to form k-tuples. The class `K-machine` is declared as follows:

```
class K-machine |
   fastPeriod : NzTime,            rate : NzNat,
   slowPeriodTimer : Time,         fastPeriodTimer : Time,
   outBuffer : MsgConfiguration,   outputDeadline : TimeInf,
   clock : Time,                   currentFastRound : Nat .
   prevInput : MsgConfiguration .
subclass K-machine < Wrapper .
```

The `fastPeriod` attribute denotes the period of the fast (inner) machine; `rate` is the rate of the fast machine; `slowPeriodTimer` denotes the time until a new "slow" period begins; `fastPeriodTimer` does the same for the fast period; `outBuffer` is the wrapper's output buffer; the `outputDeadline` timer expires at the cutoff time when messages from the output buffer must be sent; `clock` is the local clock, which should be the same as that of its inner typed machine; `currentFastRound` denotes which fast period within a slow period the machine is in; and `prevInput` stores the input received at the beginning of the current slow round.

We define a data type for message tuples. First we define lists of data:

```
sorts DataList NeDataList .
subsort Data < NeDataList < DataList .
op nil : -> DataList
op _::_ : DataList DataList -> DataList [ctor assoc id: nil] .
op _::_ : NeDataList DataList -> NeDataList [ctor ditto] .
op _::_ : DataList NeDataList -> NeDataList [ctor ditto] .
```

Messages can now also take tuples as data:

```
msg to_from_(_,_) : Oid Oid PortId DataList -> Msg .
```

We assume some functions on such messages:

- `selectData(`msgs, i`)` takes a *set* of messages *msgs*, where the content of each message m in *msgs* is a *list* of data $(d_{m_1}, \ldots, d_{m_k})$, and returns the same set of *msgs* but the where the content in each message m is only the ith element d_{m_i}.

- `append(`$\mathit{msgs}_1, \mathit{msgs}_2$`)` appends the data list (the "content") in each message in msgs_2 to the data list content in the corresponding message in the set msgs_1 (by "corresponding" we mean the message with the same receiver and receiver port id).

- `addBottom(`msgs, n`)` appends $n - i$ copies of the specific "bottom" element $\perp$ to the content of each message in *msgs*, where i is the length of the data list in the message.

When a new slow period begins (`slowPeriodTimer` is `0`), the wrapper reads its non-empty input buffer, stores these messages in its `prevInput` buffer and puts the received messages—but only with the *first* data item for each message—into the input buffer of the internal machine:

```
var NEMSGS : NEMsgConfiguration .      var MSGS : MsgConfiguration .
vars FASTPERIOD SLOWPERIOD : NzTime .  vars RATE CFR : Nat .

crl [start-slow-period]:
   < O : K-machine |
       fastPeriod : FASTPERIOD,      rate : RATE,
       inputBuffer : NEMSGS,
       slowPeriodTimer : 0,
       innerConf : < O : Machine | inputBuffer : MSGS > CONF,
       ignoreNbs : OS >
  =>
   < O : K-machine |
       inputBuffer : none,           outBuffer : none,
       slowPeriodTimer : SLOWPERIOD, fastPeriodTimer : FASTPERIOD,
       innerConf : < O : Machine |
                       inputBuffer : MSGS selectData(NEMSGS,1) >
                   CONF,
       prevInput : NEMSGS,           currentFastRound : 1,
       outputDeadline : SLOWPERIOD − 2 · ε − μmax >
  if onlyRelevantMsgs(NEMSGS, OS)
  /\ SLOWPERIOD := FASTPERIOD * RATE .
```

The following rule models the beginning of the next *fast* period that does *not* coincide with the beginning of a slow period; i.e., the `currentFastRound` is less than the rate factor:

```
crl [start-fast-period]:
   < O : K-machine |
       fastPeriod : FASTPERIOD,      rate : RATE,
       innerConf : < O : Machine | > CONF,
       fastPeriodTimer : 0,          currentFastRound : CFR,
       prevInput : NEMSGS >
  =>
   < O : K-machine |
       innerConf : < O : Machine |
                       inputBuffer : selectData(NEMSGS, CFR + 1) >
                   CONF,
       fastPeriodTimer : FASTPERIOD,
       currentFastRound : CFR + 1 >
  if CFR < RATE .
```

We next model receiving messages (`NEMSGS`) from the inner object. If the wrapper has received all expected tuples from below, it should immediately send them out to its outer wrapper. The term `allDataRcvd(`*msgs*`,` n`, OS)` becomes `true` if the content of each message in *msgs* is a list of length n, except for irrelevant messages denoted by the set `OS`. To forward received messages immediately, the wrapper must be in the *last* fast round within the slow period and `allDataRcvd` must be true.[3] An additional consequence is that the output deadline timer is turned off (set to the infinity value `INF`):

```
vars NEMSGS OUTPUT-MSGS : NEMsgConfiguration .    var T : Time .

crl [rcv-and-send]:
    < O : K-machine |  ignoreNbs : OS,
                       rate : RATE,
                       innerConf : < O : Machine | > NEMSGS,
                       outBuffer : MSGS,
                       currentFastRound : RATE >
   =>
    < O : K-machine |  innerConf : < O : Machine | >,
                       outBuffer : none,
                       outputDeadline : INF > OUTPUT-MSGS
  if onlyRelevantMsgs(NEMSGS, OS)
  /\ OUTPUT-MSGS := append(MSGS, NEMSGS)
  /\ allDataRcvd(OUTPUT-MSGS, RATE, OS) .
```

If the system is not in the last fast round, then the received messages cannot be all the messages that must be sent out:[4]

```
crl [rcv-and-store]:
    < O : K-machine |  ignoreNbs : OS,
                       rate : RATE,
                       outputDeadline : T,
                       currentFastRound : CFR,
                       innerConf : < O : Machine | > NEMSGS,
                       outBuffer : MSGS >
   =>
    < O : K-machine |  innerConf : < O : Machine | >,
                       outBuffer : OUTPUT-MSGS >
  if onlyRelevantMsgs(NEMSGS, OS)
  /\ OUTPUT-MSGS := append(MSGS, NEMSGS)
  /\ not allDataRcvd(OUTPUT-MSGS, RATE, OS) or CFR < RATE .
```

[3]If the output deadline timer of the `K-machine` expires *before* the `K-machine` gets the last messages, then the rule `send-msgs` is applied, and since it empties the output buffer, `allDataRcvd` will be `false` when the last messages are received.

[4]Since `T` has sort `Time`, the timer cannot have `INF`, and is hence not turned off.

When the output deadline timer expires, the messages must be sent out, even though all of them may not have been received yet. We make sure that there are no messages waiting to be received by the wrapper in zero time:

```
rl [send-msgs]:
   < O : K-machine |
        rate : RATE,
        innerConf : < O : TypedMachine | outBuffer : none >,
        outputDeadline : 0,
        outBuffer : MSGS >
  =>
   < O : K-machine |
        outputDeadline : INF,
        outBuffer : none >
   addBottom(MSGS, RATE) .
```

The final action we need to consider is what to do when the wrapper receives messages after it has sent out the messages because its deadline timer expired. Our deadline timer takes a worst case scenario into account: the network delay on the messages will be the maximal, the clocks of the sender and receiver will have the worst relationship, etc. Therefore, it *could* happen that the additional messages sent after the deadline could reach the receiver in time. However, since we assume that the adaptors ensure that these message will not matter, the wrapper will not forward them:

```
crl [rcv-and-ignore]:
   < O : K-machine | ignoreNbs : OS,
                     outputDeadline : INF,
                     innerConf : < O : Machine | > NEMSGS >
  =>
   < O : K-machine | innerConf : < O : Machine | > >
  if onlyRelevantMsgs(NEMSGS, OS) .
```

B.1.5 The Input Adaptor Wrapper

The input adaptor for a machine is the distributed version of the input adaptor pattern in Definition 6.7; it propagates input, without any further delay, to its inner machine or wrapper after applying the input adaptor function. Output from the inner object is just propagated outwards. The `InputAdaptor` class adds no new attributes to `Wrapper` it inherits:

```
class InputAdaptor .
subclass InputAdaptor < Wrapper .
```

The rewrite rules are also straight-forward, but the input adaptor for the hierarchical case must be defined carefully: let $adap(se) = (\alpha_1, \ldots, \alpha_n)$ (in $\mathfrak{E}$), and let o be a (relatively) slow component in a subensemble $\mathfrak{E}_{se}$. Then the input adaptor that should be applied to the input in the *outer* `InputAdaptor` wrapper should be given by $\beta = (\beta_1, \ldots, \beta_k)$, where $\beta_i = \alpha_j$ if $src_{se}(o, i) = (e_{se}, j)$, for e_{se} the environment index in $\mathfrak{E}_{se}$.

```
rl [forward]:  < O : InputAdaptor | innerConf : OBJECT NEMSGS >
           =>  < O : InputAdaptor | innerConf : OBJECT > NEMSGS .

crl [apply-adaptor] :
   < O : InputAdaptor |
        inBuffer : NEMSGS,    ignoreNbs : OS,
        innerConf : < O : Machine | inBuffer : MSGS > >
  =>
   < O : InputAdaptor |
        inBuffer : none,
        innerConf : < O : Machine | inBuffer : MSGS
                                    applyAdaptor(O, NEMSGS) > >
  if onlyRelevantMsgs(NEMSGS, OS) .

op applyAdaptor : Oid MsgConfiguration -> MsgConfiguration .
eq applyAdaptor(O, (to O from O' (P,D)) MSGS)
 = (to O from O' (P, adap(O)_P(D))) applyAdaptor(O, MSGS) .
eq applyAdaptor(O, none) = none .
```

B.1.6 The PALS Wrapper

The PALS wrapper receives messages from the other components and stores them in its input buffer. When a new period begins, the PALS wrapper propagates the received messages to the inner machine/wrapper. The PALS wrapper also needs to avoid sending out messages *from* its inner layer into the network too early. It therefore has a backoff timer for the output: if the output from the inner layer arrives before the backoff timer expires, this output is sent into the network (with appropriate delay) when the backoff timer expires; otherwise, this output is immediately sent into the network. This PALS wrapper is defined as an object of the following class:

```
class PALS-wrapper |  period : Time,
                      roundTimer : Time,
                      clock : Time,
                      outputBuffer : MsgConfiguration,
                      outputBackoffTimer : TimeInf .
subclass PALS-wrapper < Wrapper .
```

Messages received by the PALS wrapper from the other components in the network are stored in its input buffer:

```
rl [receiveMsg] :
   (to O from O' (P, D))
   < O : PALS-wrapper | inBuffer : MSGS >
  =>
   < O : PALS-wrapper | inBuffer : MSGS (to O from O' (P, D)) > .
```

When the `roundTimer` expires, a new PALS period begins: the messages in the input buffer are propagated to the inner object, the output backoff timer is set, and the `roundTimer` is reset to its period `PERIOD`:

```
crl [startRound] :
   < O : PALS-wrapper |
       period : PERIOD,      ignoreNbs : OS,
       roundTimer : 0,
       inBuffer : NEMSGS,
       innerConf : < O : Machine | inBuffer : MSGS > >
  =>
   < O : PALS-wrapper |
       roundTimer : PERIOD,
       inBuffer : none,
       outputBackoffTimer : 2·ε monus μmin ,
       innerConf : < O : Machine | inBuffer : MSGS NEMSGS > >
  if onlyRelevantMessages(NEMSGS, OS) .
```

When the output backoff timer expires, if there are messages in the output buffer, they are sent into the network, with a network delay in the interval $[\mu_{\min}, \mu_{\max}]$, and the backoff timer is turned off (set to `INF`):

```
rl [backoffTimerExpires] :
   < O : PALS-wrapper |  outputBackoffTimer : 0,
                         outputBuffer : MSGS >
  =>
   < O : PALS-wrapper |  outputBackoffTimer : INF,
                         outputBuffer : none >
   multiMsg(MSGS, μmin , μmax) .
```

The `multiMsg` operator takes a set of messages, with lower/upper network delay bounds, and creates messages with the delay interval, where the term `dly(`*msg*`,` t_1`,` t_2`)` denotes a "delayed" message that can become the ("ripe") message *msg* at any time in the time interval $[t_1, t_2]$:

```
op multiMsg : MsgConfiguration Time TimeInf -> DlyMsgConfiguration .
eq multiMsg(MSG MSGS, T, TI) = dly(MSG, T, TI) multiMsg(MSGS,T,TI) .
eq multiMsg(none, T, TI) = none .
```

When the PALS wrapper gets messages from its inner object, two things can happen: if the output backoff timer has *not* expired (i.e., not `INF`), then the messages are put into the output buffer; otherwise, they are immediately sent into the network, with appropriate network delay:

```
crl [getMessagesAndStore] :
   < O : PALS-wrapper |  ignoreNbs : OS,    outputBackoffTimer : T,
                         innerConf : (< O : Machine | > NEMSGS) >
  =>
   < O : PALS-wrapper |  innerConf : < O : Machine | >,
                         outputBuffer : NEMSGS > .
  if onlyRelevantMessages(NEMSGS, OS) .

crl [getMessagesFromInnerAndSendIntoNetwork] :
   < O : PALS-wrapper |  ignoreNbs : OS,    outputBackoffTimer : INF,
                         innerConf : (< O : Machine | > NEMSGS) >
  =>
   < O : PALS-wrapper | innerConf : < O : Machine | > >
   multiMsg(NEMSGS, μmin, μmax) .
  if onlyRelevantMessages(NEMSGS, OS) .
```

B.1.7 The Environment

We assume that the environment can generate any output and satisfies the same timing requirements as all other (slow) objects. Therefore, we can formalize the environment as a typed machine surrounded only by a PALS wrapper. Since we do not explicitly represent the state of the environment, we assume that its `state` attribute has the value `*`. The environment typed machine can therefore be defined as follows:

```
class Env .
subclass Env < TypedMachine .
var D1 : D^e_o1 . ...      var D_me : D^e_o_me .

crl [envApplyTrans] :
    < e : Env |  inBuffer : NEMSGS,  roundTimer : 0,
                 period : PERIOD,    localWiring : W >
   =>
    < e : Env |  inBuffer : none,    roundTimer : PERIOD,
                 outBuffer : [makeMsg(e, W, (D1,...,D_me)), X-DLY] >
   if X-DLY >= α_min and X-DLY <= α_max .
```

B.1.8 Time Behavior

In a similar way to single-rate PALS, the global state of the distributed real-time system $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ has the form $\{conf\ ;\ t\}$, where *conf* is the configuration consisting of the hierarchical objects and messages traveling between the different distributed components, and t is the global time. The reason for carrying around the global time in the state is that the values of the local clocks depend on the global time.

The tick rule, advancing the global time in the system, is the following modification of the "usual" tick rule for object-oriented systems [149]:

```
crl [tick] : {CONF ; T}
         => {timeEffect(CONF, T, T') ; T + T'} in time T'
  if T' <= mte(CONF, T) .
```

where `timeEffect` is the function that defines how the *passage of time affects the state*, and `mte` is the function that defines the *maximum amount of time that can elapse* before an instantaneous action must be performed:

```
op timeEffect : Configuration Time Time -> Configuration [frozen(1)] .
op mte : Configuration Time -> TimeInf [frozen(1)] .
```

These functions distribute over the objects and messages in the state in the expected way and must be defined for individual objects and messages.

We first define `timeEffect`: how does time elapse affect the timers? We assume that the timers associated to a machine or its wrappers operate according to the rate of the machine's local clock, and *not* according "real" time. Therefore, if from time t, time advances by Δ, then the local clock of machine j has advanced from $c_j(t)$ to $c_j(t+\Delta)$, that is, by $c_j(t+\Delta) - c_j(t)$, where $c_j(t)$ is the value of the `clock` attribute for machine j at time t. Likewise, an object's local timer values decrease according its local clock (using the function `monus`, defined by x `monus` $y = \max(x - y, 0)$; since `INF` `monus` $x =$ `INF`, a timer that is turned off is unaffected by time elapse):

```
vars t Δ T1 T2 T3 TD : Time .            var TI : TimeInf .
ceq timeEffect(
   < O : PALS-wrapper | roundTimer : T1,    clock : T2,
                        innerConf : CONF,
                        outputBackoffTimer : TI >, t, Δ)
 =
   < O : PALS-wrapper | roundTimer : T1 monus TD,
                        clock : c0(t + Δ),
                        innerConf : timeEffect(CONF, t, Δ),
                        outputBackoffTimer : TI monus TD >
 if TD := (c0(t + Δ) - T2) .
```

```
eq timeEffect(< O : InputAdaptor | innerConf : CONF >, t, Δ)
 = < O : InputAdaptor | innerConf : timeEffect(CONF, t, Δ) > .

eq timeEffect(
   < O : K-machine |  clock : T3,
                      slowPeriodTimer : T1,
                      fastPeriodTimer : T2,
                      outputDeadline : TI,
                      innerConf : CONF >, t, Δ)
 =
   < O : K-machine |  clock : c0(t + Δ),
                      slowPeriodTimer : T1 monus (c0(t + Δ) - T3),
                      fastPeriodTimer : T2 monus (c0(t + Δ) - T3),
                      outputDeadline : TI monus (c0(t + Δ) - T3),
                      innerConf : timeEffect(CONF, t, Δ) > .
```

The remaining execution time of a typed machine does not depend on the object's local clock:

```
eq timeEffect(
   < O : TypedMachine |  clock : T3,
                         roundTimer : T1,
                         state : [S, T2],
                         outBuffer : [MSGS, T2] >, t, Δ)
 =
   < O : TypedMachine |  clock : c0(t + Δ),
                         roundTimer : T1 monus (c0(t + Δ) - T3),
                         state : [S, T2 monus Δ],
                         outBuffer : [MSGS, T2 monus Δ] > .
```

Time elapse decreases the remaining delay interval of messages traveling in the network; other messages are not affected by time elapse:

```
eq timeEffect(dly(MSG, T1, T2), t, Δ)
 = dly(MSG, T1 monus Δ, T2 monus Δ) .
eq timeEffect(MSGS, t, Δ) = MSGS .
```

Regarding `met`, denoting how much time can advance in the system before some action must be taken, it equals 0 if some message must be treated or moved up or down the wrapper hierarchy. Otherwise, `met` will be the smallest of the `met`'s of the objects and messages in the configuration, where the `met` of an object is just defined as the smallest time until one of the timers become zero, or until the delay on the outgoing messages in the output buffer reaches 0. The assumption of monotonicity of the local clock functions is crucial to make our definition of `met` well-defined.

First, the operator `met` distributes over the elements in a configuration in the expected way, where an "undelayed" message in the network should be picked up by a rewrite rule immediately:

```
eq mte(OBJECT CONF, t) = min(mte(OBJECT, t), min(CONF, t)) .
eq mte(none, t) = INF .

eq mte(dly(MSG, T1, T2) CONF, t) = min(T2, mte(CONF, t)) .
eq mte(MSG CONF, t) = 0 .
```

In a typed machine, time cannot advance if it has a message in its input buffer or "ready" messages in its output buffer. The first case forces a machine transition to be performed when a machine has inputs:

```
eq mte(< O : TypedMachine | inBuffer : NEMSGS >, t) = 0 .
```

The other case, a special case of the equation below when `T2` is `0`, forces the wrapper enclosing the typed machine to read the messages from the machine's output buffer before time can advance:

```
eq mte(< O : TypedMachine | inBuffer : none,
                            clock : T1,
                            state : [S,T2],
                            roundTimer : T3,
                            outBuffer : [NEMSGS,T2] >, t)
 =
   min(T2, localClockElapse_c0(t, T1, T3)) .
```

where $\texttt{localClockElapse}_{c_0}(t_1, t_2, t_3)$ denotes the amount of time needed for the local clock c_0 to advance by t_3 time units, when the current time according to c_0 is t_2 and the current time according to a perfect clock is t_1. This function is defined as follows:

$$\texttt{localClockElapse}_{c_0}(t_1, t_2, t_3) = \min\{t' : \mathit{Time} \mid c_0(t_1 + t') - t_2 = t_3\}.$$

Otherwise, time can advance to whichever comes first of the expiration of the `roundTimer` and the end of the execution of an internal "transition":

```
eq mte(< O : TypedMachine | inBuffer : none,
                            clock : T1,
                            state : S,
                            roundTimer : T2,
                            outBuffer : none >, t)
 =
   localClockElapse_c0(t, T1, T2) .
```

Time cannot advance if a k-machine wrapper has messages in its input buffer (these must be taken care of immediately) or if its inner configuration has any messages, which must be read by the wrapper without any delay:

```
eq mte(< O : K-machine | inBuffer : NEMSGS >, t) = 0 .
eq mte(< O : K-machine | innerConf : < O : Machine | > NEMSGS >, t)
 = 0 .
```

Otherwise, time can advance until the next timer expires, or until its inner object must perform an action:

```
eq mte(< O : K-machine | clock : T1,
                         inBuffer : none,
                         innerConf : OBJECT,
                         slowPeriodTimer : T2,
                         fastPeriodTimer : T3,
                         outputDeadline : TI >, t)
 = min(mte(OBJECT, t), localClockElapse_c0(t, T1, min(T2, T3, TI))) .
```

The input adaptor wrapper should immediately forward (or propagate) incoming messages or messages from its inner object:

```
eq mte(< O : InputAdaptor | inBuffer : NEMSGS >, t) = 0 .
eq mte(< O : InputAdaptor | innerConf : CONF NEMSGS >, t) = 0 .
eq mte(< O : InputAdaptor | inBuffer : none,
                            innerConf : OBJECT >, t)
 = mte(OBJECT, t) .
```

The PALS wrapper must perform an action when one of its timers expire, and it must also immediately read output from its inner object:

```
eq mte(< O : PALS-wrapper | innerConf : MSG CONF >, t) = 0 .

eq mte(< O : PALS-wrapper | inBuffer : MSGS,
                            ignoreNbs : OS
                            clock : T1,
                            roundTimer : T2,
                            innerConf : OBJECT,
                            outputBackoffTimer : TI >, t)
 = if onlyRelevantMessages(MSGS, OS)
   then min(mte(OBJECT, t), localClockElapse_c0(t, T1, min(T2, TI)))
   else 0 fi .
```

Finally, a message in the network *may* be ready for consumption whenever its smallest remaining network delay is zero:

```
rl [messageRipe] : dly(MSG, 0, T) => MSG .
```

B.1.9 Initial States

As explained in [138] for single-rate PALS, it is convenient to define the initial states to be those at time $T-\epsilon$, just before the start of a new "round" of the entire system, when all the PALS wrapper input buffers are full:

Definition B.1. Admissible initial states *of distributed real-time system* $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ *are states* $\{state\,;\, T-\epsilon\}$ *as described above, where:*

1. *the value of the* `clock` *attribute of each object in* state *with object identifier* j *is* $c_j(T-\epsilon)$*;*
2. *each* `PALS-wrapper` *object contains a complete set of messages in its* `inBuffer`*;*
3. *there are no other messages in the system, neither in the network, nor in other buffers or in the inner configurations of wrappers;*
4. *the* `roundTimer`*,* `fastPeriodTimer`*, and* `slowPeriodTimer` *attributes are initialized to* $T-c_j(T-\epsilon)$ *(for the appropriate local clock* c_j*), which is always greater 0; and*
5. *all other timers are turned off (i.e., equal* `INF`*).*

B.1.10 Example

We give an example showing how states in $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ can be represented. Let Γ define the following performance parameters: the maximum network delay $\mu_{\max} = 2$, the maximal clock drift $\epsilon = 0.5$, and the maximum transition execution times $\alpha_{max} = \alpha_{max_e} = \alpha_{max_g} = 2$. Let the period T of the whole system be 12 time units. These parameter values satisfy the PALS constraint $T \geq \mu_{\max} + 2\epsilon + \max(2\epsilon \texttt{ monus } \mu_{\min}, \alpha_{max})$, since $12 \geq 5$. Consider the multirate ensemble $\mathfrak{E}_1$ in Figure B.1, i.e.:

$$\begin{aligned}\mathfrak{E}_1 = \; & (\{c,d\},\{e,g\}, env_1, \{M_j\}_{j\in\{c,d,e,g\}}, \\ & (D^{env_1}_{i_1} \times D^{env_1}_{i_2} \times D^{env_1}_{i_3}, D^{env_1}_{o_1} \times D^{env_1}_{o_2}), src, \{e \mapsto 2, g \mapsto 3\}, adap)\end{aligned}$$

where:

- *src*, assigning to each input port the source output port, is the function

$$\begin{aligned}&\{(c,1) \mapsto (env_1,1),\ (c,2) \mapsto (d,3),\ (c,3) \mapsto (g,1),\ (c,4) \mapsto (e,1),\\ &(d,1) \mapsto (env_1,2),\ (d,2) \mapsto (e,2),\ (d,3) \mapsto (c,3),\\ &(e,1) \mapsto (c,4),\ (g,1) \mapsto (d,2),\\ &(env_1,1) \mapsto (c,1),\ (env_1,2) \mapsto (c,2),\ (env_1,3) \mapsto (d,1)\}.\end{aligned}$$

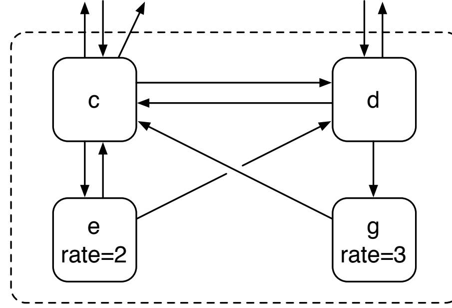


Figure B.1: A multirate ensemble $\mathfrak{E}_1$, when ports are counted clockwise from the upper left corner.

- *adap* is a function that maps each machine (index) j to the input adaptor $adap_j$. Here, $(adap_c)_1$ and $(adap_c)_2$ are functions that take a *single* data element (of the appropriate type) and return a single data element (input from env_1 and d, respectively), whereas $(adap_c)_3$ maps a three-tuple of data elements (input from g) to a single element, $(adap_c)_4$ maps a pair (input from e) to a single element, and so on. Likewise, $(adap_g)_1$ maps a single data value (from d) to a triple of values (inputs for the fast machine g).

Since the period of e is 6 and that the period of g is 4, the parameter values Γ and T satisfy the condition that each fast component can finish a fast transition before, n the worst case, having to perform another transition, $T/k \geq 2\epsilon + \alpha_{max}$, since $4 \geq 3$. However, when we consider the formula

$$k'_f = 1 + \lfloor \frac{(T \texttt{ monus } (2\epsilon + \mu_{max} + \alpha_{max_f})) \cdot rate(f)}{T} \rfloor,$$

stating how many outputs a fast machine f is guaranteed to be able to send, we get that for the fastest component g,

$$k'_g = 1 + \lfloor \frac{(12 - 5) \cdot 3}{12} \rfloor = 2.$$

That is, it cannot be guaranteed that the component g can finish all 3 transitions in a (slow) round before having to transmit its messages into the network (by way of its outer wrappers).[5]

[5]The third fast period of g begins at local time 8, and if its local clock is slow, and it takes two time units to execute the transition and two time units to transmit the messages in the network, the messages will not arrive by time 12, i.e., the start of the next global round, at the nodes with faster clocks.

```
< c : PALS-wrapper |
    inBuffer : possibly input msgs from other components ,
    period : 12,            ignoreNbs : emptySet,
    clock : cc(t),          roundTimer : ...,
    outputBuffer : ...,     outputBackoffTimer : ...,
    innerConf :
        < c : InputAdaptor |
            inBuffer : possibly input msgs from PALS wrapper ,
            ignoreNbs : emptySet,
            innerConf :
                < c : C[c] |
                    inBuffer : possibly input msgs from adaptor ,
                    period : 12,
                    clock : cc(t),     roundTimer : ...,
                    state : s,         outBuffer : ...,
                    localWiring : (1 --> env1.1 ; 2 --> env1.2 ;
                                   3 --> d.3 ;  4 --> e.1)
                >
                --- possible output msgs from machine to adaptor
        >
        --- possible output msgs from adaptor to PALS wrapper
>
```

Figure B.2: The slow component c in $\mathcal{MA}(\mathfrak{E}_1, 12, \Gamma)$.

Therefore, to ensure predictable behavior, the input adaptor(s) handling inputs from g must be $(k'+1)$-oblivious; that is, they must ignore the third element from g. In our case, this means that for all values d_1, d_2, d_3, d_4 of the appropriate data type, the input adaptor $(adap_c)_3$ must satisfy the condition $(adap_c)_3(d_1, d_2, d_3) = (adap_c)_3(d_1, d_2, d_4)$. For the other fast component, e, its rate 2 equals its k'_e value $1 + \lfloor \frac{(12-5)\cdot 2}{12} \rfloor$, and hence it can always finish both of its transitions in a fast round before having to send the generated messages into the network. The $(k'+1)$-obliviousness condition therefore does not impose any further restrictions on the adaptors $(adap_c)_4$ and $(adap_d)_2$.

Since we assume that all objects defining/wrapping a machine j have the object identifier j and have the same local clock c_j, the slow component c in $\mathfrak{E}_1$ is modeled by a term of the form in Figure B.2, where there are input in at most one of the input buffers. Likewise, the fast component g in $\mathfrak{E}_1$ can be represented by a term of the form in Figure B.3.

We now consider a hierarchical multirate ensemble $\mathfrak{E}$ that has $\mathfrak{E}_1$ as a subcomponent (with index f in $\mathfrak{E}$), where $rate(\alpha) = 5$ and the top-level machines in $\mathfrak{E}$ have indices a and b, as shown in Figure B.4; i.e.:

$$\mathfrak{E} = (\{a, b\},\ \emptyset,\ \{f\},\ env,\ \{M_a, M_b\},\ \{\mathfrak{E}_f\},\ (\{*\}, \{*\}),\ src,\ \{f \mapsto 5\},\ adap)$$

$$src = \{(a,1) \mapsto (f,1),\ (b,1) \mapsto (f,3),\ (b,2) \mapsto (f,2),\ (b,3) \mapsto (f,1),\\ (f,1) \mapsto (a,1),\ (f,2) \mapsto (b,1)\}.$$

```
< g : PALS-wrapper |
    inBuffer : possibly input messages from other components ,
    period : 12,            ignoreNbs : emptySet,
    clock : c_g(t),         roundTimer : ...,
    outputBuffer : ...,     outputBackoffTimer : ...,
    innerConf :
        < g : InputAdaptor |
            inBuffer : possibly input msgs from PALS wrapper ,
            ignoreNbs : emptySet,
            innerConf :
                < g : K-machine |
                    inBuffer : input msgs from input adaptor ,
                    fastPeriod : 4,           ignoreNbs : emptySet,
                    clock : c_g(t),           fastPeriodTimer : ...,
                    slowPeriodTimer : ...,    rate : 3,
                    outBuffer : ...,          outputDeadline : ...,
                    prevInput : ...,          currentFastRound : ...,
                    innerConf:
                        < g : C_[g] |
                            inBuffer : possibly input msgs from K-machine ,
                            period : 4,
                            clock : c_g(t),     roundTimer : ...,
                            state : s,          outBuffer : ...,
                            localWiring : (1 --> c.3)
                        >
                        --- possible output msgs from machine to K-machine
                >
                --- possible output msgs from K-machine to adaptor
        >
        --- possible output msgs from adaptor to PALS wrapper
>
```

Figure B.3: The fast component g in $\mathcal{MA}(\mathfrak{E}_1, 12, \Gamma)$.

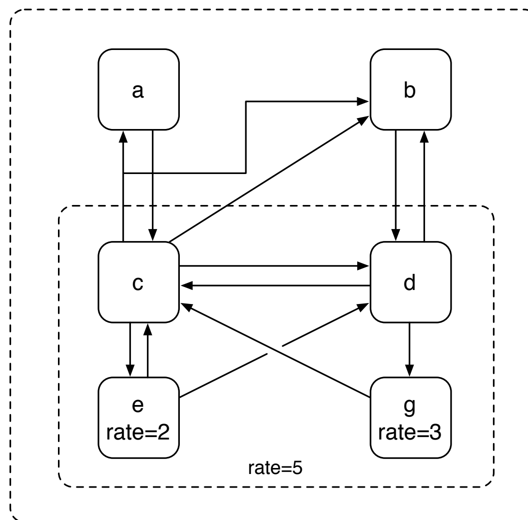


Figure B.4: A hierarchical multirate ensemble $\mathfrak{E}$, with some connections to/from the nodes a and b omitted.

We define the states of $\mathcal{MA}(\mathfrak{E}, 60, \Gamma)$. The representation of the local fast components e and g of $\mathfrak{E}_1$ in the flat case is completely unchanged (since they do not communicate with the environment, we do not even need to change the local wiring for these objects), except for adding the attribute/value pair `ignoreNbs : emptySet` to the wrappers. But the slow components c and d in the fast subcomponent $\mathfrak{E}_1$ must be updated by adding new wrappers for the communication with the outside. That is, we add a new PALS wrapper, a new `InputAdaptor` wrapper (for the input adaptor around $M_{SC(\mathfrak{E}_1)}$ in the definition of $\mathfrak{E}$), and a new `K-machine` wrapper for the communication with the "environment". Note that these *new* wrappers ignore communication with the other components in $\mathfrak{E}_1$ (i.e., c or d, and e and g), whereas the "old" wrappers should ignore communication with the external components a and b. For example, if $src(a, 1) = src(b, 3) = (f, 1)$, $src(b, 1) = (f, 3)$, and $src(b, 2) = (f, 2)$, then the component c has the form in Figure B.5.

Finally, the last components we have to consider are the top-level machine components, a and b in our case. They are modeled exactly as the slow components in the flat case, with the exceptions of "short-circuiting" the connections that go via the environment and adding `ignoreNbs : emptySet` to the wrappers. For example, if $src_1(c, 1) = (env_1, 1)$ and $src(1, 1) = (a, 1)$, then the node a in our system is modeled by the object in Figure B.6.

To summarize, given a hierarchical multirate ensemble $\mathfrak{E}$ with period T and performance parameters Γ, we have provided the following procedure for constructing the corresponding (flat) model of its asynchronous distributed real-time realization $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$:

1. Recursively construct the asynchronous model $\mathcal{MA}(\mathfrak{E}_{se}, T/rate(se), \Gamma)$ for each subensemble $\mathfrak{E}_{se}$ of $\mathfrak{E}$.

2. For each *slow* machine in $\mathfrak{E}$, construct the corresponding node, like we did for a above.

3. For each *fast* machine in $\mathfrak{E}$, construct the corresponding node in the same way as in the flat setting.

4. For each slow component in each subsystem $\mathcal{MA}(\mathfrak{E}_{se}, T/rate(se), \Gamma)$, add the new wrappers for communicating with the global top-level machines, and modify its existing wrappers to ignore messages to/from these outermost slowest node (like we did for node c above).

5. "Short-circuit" the connections between the top-level components in $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ and the top-level nodes in each subensemble $\mathfrak{E}_{se}$.

```
< c : PALS-wrapper | --- NEW! Communication with outer components
  inBuffer : ...,
  period : 60,            ignoreNbs : d ; e ; g,
  clock : c_c(t),         roundTimer : ...,
  outputBuffer : ...,     outputBackoffTimer : ...,
  innerConf :
    < c : InputAdaptor | --- NEW! transforms single inputs to 5-tuples,
      inBuffer : ...,
      ignoreNbs : d ; e ; g,
      innerConf :
        < c : K-machine | --- NEW! Slows down by factor 5
          inBuffer : ...,
          fastPeriod : 12,          ignoreNbs : d ; e ; g,
          clock : c_c(t),           fastPeriodTimer : ...,
          slowPeriodTimer : ...,    rate : 5,
          outBuffer . ...,          outputDeadline : ...,
          prevInput : ...,          currentFastRound : ...,
          innerConf:  --- "old wrappers slightly modified"
            < c : PALS-wrapper |
              inBuffer : ...,
              period : 12,          ignoreNbs : a ; b,
              clock : c_c(t),       roundTimer : ...,
              outputBuffer : ...,   outputBackoffTimer : ...,
              innerConf :
                < c : InputAdaptor |
                  inBuffer : ...,
                  ignoreNbs : a ; b,
                  innerConf :
                    < c : C_[c] |
                      inBuffer : possibly input from all nodes,
                      period : 12,
                      clock : c_c(t),     roundTimer : ...,
                      state : s,          outBuffer : ...,
                      localWiring : (1 --> a.1 ; 1 --> b.3 ; 2 --> b.2 ;
                                     3 --> d.3 ; 4 --> e.1)
                    >
                    --- possible output messages from machine
                >
                --- possible output messages from adaptor
            >
            --- possible output messages from fast PALS-wrapper
        >
        --- possible output messages from K-machine wrapper
    >
    --- possible output messages from outer input adaptor
>
--- possible output messages from outer PALS wrapper
```

Figure B.5: The relatively slow component c in $\mathcal{MA}(\mathfrak{E}, 60, \Gamma)$.

```
< a : PALS-wrapper |
    inBuffer : ...,
    period : 60,            ignoreNbs : emptySet ,
    clock : c_a(t),         roundTimer : ...,
    outputBuffer : ...,     outputBackoffTimer : ...,
    innerConf :
      < a : InputAdaptor |
          inBuffer : ...,
          ignoreNbs : emptySet ,
          innerConf :
            < a : C_[a] |
                inBuffer : ...,
                period : 60,
                clock : c_a(t),     roundTimer : ...,
                state : s,          outBuffer : ...,
                localWiring : (1 --> c.1 ; ...)
            >
            --- possible output messages from machine to adaptor
       >
       --- possible output messages from adaptor to PALS wrapper
>
```

Figure B.6: The slow component a in $\mathcal{MA}(\mathfrak{E}, 60, \Gamma)$.

B.2 More Details on the Proof

This section provides the proofs of two lemmas in Section 6.3.3 for the correctness of Multirate PALS, the proof sketch for the hierarchical case of Theorem 6.1. We recall some of the assumptions that enable us to establish the relationship between the synchronous composition MRSC($\mathfrak{E}$) and the asynchronous system $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$ for a multirate ensemble $\mathfrak{E}$, global period T, and performance parameters $\Gamma = (\alpha_{min}, \alpha_{max}, \mu_{\min}, \mu_{max}, \epsilon, \vec{c}, \alpha_F)$, where $\alpha_F = \{\alpha_{max_f}\}_{f \in J_F}$ denotes the maximal execution time of a transition of all the fast machines in the ensemble $\mathfrak{E}$:

1. The PALS condition: $T \geq \mu_{\max} + 2\epsilon + \max(2\epsilon \texttt{ monus } \mu_{\min}, \alpha_{max})$;

2. Each fast component f can always finish executing a transition before having to start executing the next one: $T/\mathit{rate}(f) \geq \alpha_{max_f} + 2\epsilon$;

3. For each input "port" (j, p) of a slow component $j \in J_S$ connected to an output port (f, p') of a fast component $f \in J_F$ (i.e., $\mathit{src}(j, p) = (f, p')$), its input adaptor $\mathit{adap}(j)_p$ is $(k'_f + 1)$-oblivious for

$$k'_f = 1 + \lfloor \frac{T \texttt{ monus } (2\epsilon + \mu_{\max} + \alpha_{max_f}) \cdot \mathit{rate}(f)}{T} \rfloor,$$

the maximal number of elements in each $\mathit{rate}(f)$-tuple of outputs that f can be guaranteed to be transmitted in each slow round.

For a flat multirate ensemble $\mathfrak{E}$, in the single-rate system $\mathcal{A}(SR(\mathfrak{E}), T, \Gamma)$, a *slow* component $j \in J_S$ is represented by a `TypedMachine` object modeling $(M_j)_{adap(j)}$ enclosed by a PALS wrapper: i.e., $PALS_{T,\Gamma}((M_j)_{adap(j)})$. In the multirate system $\mathcal{MA}(\mathfrak{E}, T, \Gamma)$, it is modeled by a `TypedMachine` object modeling M_j, enclosed by an `InputAdaptor` wrapper for $adap(j)$, which in itself is enclosed by a PALS wrapper: i.e., $PALS_{T,\Gamma}(adap(j)(M_j))$.

Lemma 6.3. *For a slow machine M_j in a flat multirate machine ensemble $\mathfrak{E}$, $j \in J_S$, the correlated local asynchronous systems $\mathcal{MA}(adap(j)(M_j), T, \Gamma)$ and $\mathcal{A}((M_j)_{adap(j)}, T, \Gamma)$ are behaviorally $\sim_{obl}$-equivalent.*

Proof. Consider stable states s_{ma} and s_a in $Stable(\mathcal{MA}(adap(j)(M_j), T, \Gamma))$ and $Stable(\mathcal{A}((M_j)_{adap(j)}, T, \Gamma))$, respectively, such that

$$s_{ma} \sim_{obl} s_a.$$

Then, the `state` attribute of the typed machine is the same in both cases, but their inputs are only $\equiv_{obl}$-equivalent. Suppose that there is a transition $s_a \longrightarrow_{st} s'_a$ for some $s'_a \in Stable(\mathcal{A}((M_j)_{adap(j)}, T, \Gamma))$. Then, there is a corresponding transition $s_{ma} \longrightarrow_{st} s'_{ma}$, because:

1. both immediately apply the input adaptor $adap(j)$ to the received input without any delay, and then immediately apply the transition of M_j to the resulting input; and
2. we assume $(k' + 1)$-obliviousness of our input adaptors, so it follows from Lemma 6.1 that the input values seen by the innermost "machine" will be the same.

Clearly, for M_j, the `TypedMachine` object in s'_{ma} has the same `state` with one in s'_a, and the input buffers of the PALS wrappers can also be the same, since the local environments can give the same output. Once the output is ready, in the single-rate case, the output is grabbed immediately by the PALS wrapper, and in the multirate case, it is grabbed immediately by the `InputAdaptor` wrapper. However, in the latter case, the `InputAdaptor` wrapper immediately outputs it for the PALS wrapper to pick it up. The timing is the same in both cases, since the input adaptor wrapper reads messages, propagate messages up and down, and applies adaptors eagerly (i.e., without any wait) and in zero time. Therefore, the outputs will be the same in both cases. That is, the input buffer of the PALS wrapper for the local environment in s'_{ma} has received the same output as one in s'_a. Consequently, $s'_{ma} \sim_{obl} s'_a$. The other direction is similar. □

On the other hand, a *fast* component $j \in J_F$ of a flat multirate ensemble $\mathfrak{E}$ is represented by $PALS_{T,\Gamma}((M_j^{\times rate(j)})_{adap(j)})$ in the single-rate system, i.e., a `TypedMachine` object corresponding to the machine $(M_j^{\times rate(j)})_{adap(j)}$ enclosed by a PALS wrapper, and in the multirate system, represented by $PALS_{T,\Gamma}(adap(j)(rate(j)(M_j)))$, i.e., a `TypedMachine` object modeling M_j, enclosed by a `K-machine` wrapper with rate $rate(j)$, enclosed by an `InputAdaptor` wrapper for $adap(j)$, again enclosed by a PALS wrapper.

Lemma 6.4. *For a fast machine M_f in a flat multirate machine ensemble $\mathfrak{E}$, $f \in J_F$, the local asynchronous systems $\mathcal{MA}(adap(f)(rate(f)(M_f)), T, \Gamma)$ and $\mathcal{A}((M_f^{\times rate(f)})_{adap(f)}, T, \Gamma)$ are behaviorally $\sim_{obl}$-equivalent.*

Proof. Consider states $s_{ma} \in Stable(\mathcal{MA}(adap(f)(rate(f)(M_f)), T, \Gamma))$ and $s_a \in Stable(\mathcal{A}((M_f^{\times rate(f)})_{adap(f)}, T, \Gamma))$ with $s_{ma} \sim_{obl} s_a$, and suppose that $s_a \longrightarrow_{st} s'_a$ for some $s'_a \in Stable(\mathcal{A}((M_f^{\times rate(f)})_{adap(f)}, T, \Gamma))$. In s_{ma} and s_a, the typed machine objects for M_f have the same state and the PALS objects have $\equiv_{obl}$-equivalent inputs in their input buffers. However, since a fast component only gets inputs from slow components, there is no "cutoff" in these received messages; that is, the PALS wrappers for M_f in s_{ma} and s_a actually have the same inputs in their input buffers.

The single-rate component for M_f in s_a performs all $rate(f)$ transitions in one step,[6] according to Definition 6.5, and the multirate component for M_f in s_{ma} executes them one by one,[7] using the `K-machine` wrapper. As stated in the proof of Lemma 6.3, the input adaptors (either as machine closures, in the single-rate case, or as explicit wrappers, in the multirate case) achieve the same thing in the same amount of time. Therefore, for the `K-machine` wrapper in s_{ma}, we can choose to apply the same transitions in both cases, so that the resulting state of the machine will be the same, *provided* that: (i) the multirate machine will have time to execute all $rate(f)$ transitions before reaching the time when it needs to become a stable state, and (ii) it can finish one local transition before the next fast input arrives.

[6] $PALS_{T,\Gamma}((M_f^{\times rate(f)})_{adap(f)})$ first applies $adap(f)$ to the inputs, and starts from its state and performs a single transition, which "simulates" $rate(f)$ steps of M_f as described in Definition 6.5. This takes time less than α_{max} (maximum time to perform a single transition). The resulting outputs, which are $rate(f)$-tuples of outputs for every output port of M_f are sent out of the machine once they are ready, and are picked up by the PALS wrapper, and are sent out into the configuration in time.

[7] For $PALS_{T,\Gamma}(adap(f)(rate(f)(M_f)))$, the input adaptor wrapper transforms the input and immediately sends it to the `K-machine` layer below. The `K-machine` wrapper feeds the $rate(f)$-tuples of inputs, one element from each $rate(f)$-tuple at a time, at each "fast period". When the innermost machine gets an input, it performs one transition from M_f, and when the output is ready, it is thrown out by the typed machine and collected by the `K-machine` object. When the `K-machine` object has received all $rate(f)$ sets of outputs, or when its deadline timer expires, it sends out the messages.

Following the results of the single-rate PALS timing properties in [138], the period will expire some time in the time interval $(i \cdot T - \epsilon, i \cdot T + \epsilon)$, and the following *fast* transition will occur some time in the time interval $(i \cdot T - \epsilon + T/rate(f), i \cdot T + \epsilon + T/rate(f))$. Therefore, to make sure that the fast machine can indeed perform transitions when the previous state has already finished, we must have

$$i \cdot T + \epsilon + \alpha_{max_f} \leq i \cdot T - \epsilon + T/rate(f),$$

that is, $\alpha_{max_f} \leq T/rate(f) - 2\epsilon$, which was an assumption made at the beginning of this section. Therefore, the fast machine will have time to perform a transition, and hence perform the following transition with the updated new internal state. Furthermore, this also directly implies that the fast machine can finish all of its $rate(f)$ executions before entering the start of the next round, i.e., the next stable state. Consequently, there exists a corresponding transition $s_{ma} \longrightarrow_{st} s'_{ma}$, where the `TypedMachine` object in s'_{ma} has the same `state` attribute as one in s'_a, and has generated the same output. The input buffers of the PALS wrappers for M_j in s'_a and s'_{ma} can also be the same, since their local environments can give the same output.

Now let us consider the outputs, which determine the input buffers of the PALS wrappers for the local environments in s'_a and s'_{ma}. The single-rate machine will output the $rate(f)$-tuple of outputs resulting from executing a single transition of $M_f^{\times rate(f)}$, which simulates k transitions of M_f. The multirate component will send either all of these (since as we saw above, the exact same transitions can be performed in the multirate case), or must send away its messages *before* it has gotten all k sets of outputs, when its deadline timer expires. If so, it "pads" the outgoing message tuples with the selected bottom values until it it has sent k-tuples of outputs. That is, the resulting outputs only differ in $(k' + 1)$-oblivious data, and hence the outputs will be $\equiv_{obl}$-equivalent. Finally, we need to show that these messages indeed will be arrive at the PALS wrapper in time to reach all recipients before the next round. At the beginning of a slow round, this `outputDeadline` timer is set to $T - 2 \cdot \epsilon - \mu_{\max}$, which means that it will expire *before* time $(i + 1) \cdot T - \epsilon - \mu_{\max}$, which means that the messages will reach their recipients *before* "real" time $(i + 1) \cdot T - \epsilon$, which is exactly what we want [138]. Therefore, the input buffer of the PALS wrapper for the local environment in s'_{ma} has received the same output as one in s'_a. Consequently, $s'_{ma} \sim_{obl} s'_a$. The other direction is entirely similar. □

As explained in Section 6.3.3, the above two lemmas and Lemma 6.2 imply the following theorem for a flat multirate ensemble $\mathfrak{E}$, and therefore also imply that $\sim_{obl} ; sync$ is a bisimulation between two asynchronous systems $ts(Stable(\mathcal{MA}(\mathfrak{E}, T, \Gamma)))$ and $ts(Stable(\mathcal{A}(M_{SR(\mathfrak{E})}, T, \Gamma)))$:

Theorem 6.1. *For a flat multirate ensemble $\mathfrak{E}$, the relation $\sim_{obl} ; sync$ is a bisimulation between $ts(Stable(\mathcal{MA}(\mathfrak{E}, T, \Gamma)))$ and $ts(SR(\mathfrak{E}))$.*

Consider an immediate (fast) subensemble $\mathfrak{E}_{se}$ of a hierarchical multirate ensemble $\mathfrak{E} = (J_S, J_F, K, e, \{M_j\}_{j \in J_S \cup (J_F - K)}, \{\mathfrak{E}_{se}\}_{se \in K}, E, src, rate, adap)$. Let $\mathcal{MA}((\mathfrak{E}_{se}^{\times rate(se)})_{adap(se)}, T, \Gamma)$ be the distributed asynchronous model in which each relatively slow component in $\mathfrak{E}_{se}$ is enclosed by outer wrappers as illustrated in Figure 6.9. That is, if $PALS_{T,\Gamma}(\alpha_s(k(PALS_{T/k,\Gamma}(\alpha_f(M)))))$ denotes a layered object[8] for a slow component in $\mathfrak{E}_{se}$, then:

$$\begin{aligned}
&\mathcal{MA}((\mathfrak{E}_{se}^{\times rate(se)})_{adap(se)}, T, \Gamma) = \\
&\qquad \{PALS_{T,\Gamma}(\alpha_s(k(PALS_{T/k,\Gamma}(adap_{se}(j)(M_j))))) \mid j \in J_S\} \cup \\
&\qquad \{PALS_{T/k,\Gamma}(adap_{se}(j)(rate_{se}(j)(M_j))) \mid j \in J_F\} \cup \\
&\qquad \{PALS_{T,\Gamma}(E)\},
\end{aligned}$$

where $\alpha_s = adap(se)$, $k = rate(se)$, $adap_{se}(j)$ is the input adaptor function for a machine j in $\mathfrak{E}_{se}$, and $rate_{se}$ is the rate of a machine j in $\mathfrak{E}_{se}$. By the above theorem, $ts(Stable(\mathcal{MA}(\mathfrak{E}_{se}, T, \Gamma)))$ and $ts(Stable(\mathcal{A}(M_{SR(\mathfrak{E}_{se})}, T, \Gamma)))$ are bisimilar, provided that $\mathfrak{E}_{se}$ is flat (i.e., $\mathfrak{E}$ has depth 2). Therefore, by a similar argument used in the proof of Lemma 6.4, we have:

Lemma B.1. *Consider an immediate fast subensemble $\mathfrak{E}_{se}$ in a hierarchical multirate ensemble $\mathfrak{E}$ of depth 2. Then, the following local asynchronous distributed systems are also behaviorally $(\sim_{obl} ; sync)$-equivalent:*

$$\mathcal{MA}((\mathfrak{E}_{se}^{\times rate(se)})_{adap(se)}, T, \Gamma) \quad \textit{and} \quad \mathcal{A}((M_{SR(\mathfrak{E}_{se})}^{\times rate(se)})_{adap(se)}, T, \Gamma).$$

As a consequence, by using a slightly extended version of Lemma 6.2 to $(\sim_{obl} ; sync)$-equivalent components, and by using induction on depth d of $\mathfrak{E}$, we can obtain the following main result:

Theorem B.1. *For a hierarchical multirate ensemble $\mathfrak{E}$, $(\sim_{obl} ; sync)$ is a bisimulation between $ts(Stable(\mathcal{MA}(\mathfrak{E}, T, \Gamma)))$ and $ts(SR(\mathfrak{E}))$.*

[8]The slow `PALS-wrapper` object with period T encloses an outer `InputAdaptor` object for input adaptor α_s, which encloses a `K-machine` object with rate k, which encloses the fast `PALS-wrapper` object with period T/k, which encloses an inner `InputAdaptor` object for adaptor α_f, which finally encloses a `TypedMachine` object for M.

B.3 More Details on the Real-Time Maude Framework

This section provides the definitions of the three functions `clearOutputs`, `transferInputs`, and `transferResults`, used in Section 6.4.2. Note that more "optimized" versions of the `transferInputs` and `transferResults` functions are provided in the paper [16].

Definition of the `clearOutputs` Function. The `clearOutputs` function that clears every output port in the configuration is declared as follows:

```
var C : ComponentId .                      var P : PortId .
vars PORTS COMPS : Configuration .         var DL : List{Data} .

op clearOutput : Configuration -> Configuration .

eq clearOutput(< C : Component | ports : PORTS > COMPS)
 = < C : Component | ports : clearOutput(PORTS) >
   clearOutput(COMPS) .

eq clearOutput(< P : OutPort | content : DL > PORTS)
 = < P : OutPort | content : nil > clearOutput(PORTS) .

eq clearOutput(PORTS) = PORTS [owise] .
```

Definition of the `transferInputs` Function. For a multirate ensemble component, the `transferInputs` function transfers data in the input ports of the ensemble or the feedback output ports of the subcomponents into their connected input ports at the beginning of each step. We model transferring data by a message passing mechanism. A message has a list of data to be delivered and a comma-separated set of its target port names.

```
op transIn : NeList{Data} PortName ~> Msg .
```

Notice that messages are declared to only have a kind `[Msg]`, and thus no other operation can be applied until every message is delivered.

The following equations formalize that `transIn` messages are delivered to the input ports of the subcomponents, where `KPS` and `KCS` are variables at the kind level to capture traveling messages as well as objects:

```
vars PN PN' : PortName .            vars KPS KCS : [Configuration] .
vars PORTS COMPS : Configuration .  var CONXS : Set{Connection} .

eq < C : Ensemble | ports : KPS transIn(NDL,PN), machines : KCS >
 = < C : Ensemble | ports : KPS, machines : KCS transIn(NDL,PN) > .
```

```
eq transIn(NDL, C . P)
   < C : Component | ports : < P : InPort | content : nil > PORTS >
 = < C : Component | ports : < P : InPort | content : NDL > PORTS > .
```

The `transferInputs` function just generates those `transIn` messages from each ensemble input port and feedback output port:

```
eq transferInputs(
      < C : Ensemble | ports : PORTS,
                       machines : COMPS,
                       connections : CONXS >)
 =
      < C : Ensemble | ports : transEnvIn(CONXS, PORTS),
                       machines : transFBOut(CONXS, COMPS) > .
```

The `transEnvIn` function produces the message `transIn(D, C' . P')` from the *first* item `D` in each ensemble input port `P`, and the `transFBOut` function produces the message `transIn(NDL, C . P)` from data `NDL` in each feedback output port `P` of subcomponent `C`. Since one source port may correspond to many destination ports, we also define two auxiliary functions: given a source port name `PN`, data `NDL`, and a connection set `CONXS`, the function `genIn` returns the set of all resulting `transIn` messages, and the function `remove` returns the connection set obtained from `CONXS` by removing each connection `PN --> PN'` with source port `PN`:

```
op transEnvIn : Set{Connection} Configuration ~> Configuration .
eq transEnvIn((P --> C' . P') ; CONXS,
      < P : InPort | content : D DL > KPS)
  =
    transEnvIn(remove(P, CONXS),
      < P : InPort | content : DL > KPS
      genIn(P, D, (P --> C' . P') ; CONXS)) .
eq transEnvIn(CONXS, KPS) = KPS [owise] .

op transFBOut : Set{Connection} Configuration ~> Configuration .
eq transFBOut((C . P --> C' . P') ; CONXS,
     < C : Component |
         ports : < P : OutPort | content : NDL > PORTS > KCS)
  =
   transFBOut(remove(C . P, CONXS),
     < C : Component |
         ports : < P : OutPort | content : nil > PORTS > KCS
     genIn(C . P, NDL, (C . P --> C' . P') ; CONXS)) .
eq transFBOut(CONXS, KCS) = KCS [owise] .
```

```
op genIn : PortName NeList{Data} Set{Connection} ~> Configuration .
eq genIn(PN, NDL, (PN --> PN') ; CONXS)
 = genIn(PN, NDL, CONXS) transIn(NDL, PN') .
eq genIn(PN, NDL, CONXS) = none [owise] .

op remove : PortName Set{Connection} -> Set{Connection} .
eq remove(PN, (PN --> PN') ; CONXS) = remove(PN, CONXS) .
eq remove(PN, CONXS) = CONXS [owise] .
```

Definition of the `transferResults` Function. The `transferResults` function transfers data in the output ports of the subcomponents to their connected output ports in an ensemble component. Similarly, we model transferring data by means of a message passing mechanism using `transOut` messages, which are delivered to the output ports of the ensemble:

```
op transOut : NeList{Data} PortName ~> Msg .

eq < C : Ensemble | ports : KPS, machines : transOut(NDL,PN) KCS >
 = < C : Ensemble | ports : KPS transOut(NDL,PN), machines : KCS > .

eq transOut(NDL, P)  < P : OutPort | content : DL >
 = < P : OutPort | content : DL NDL > .
```

The `transferResults` function generates such `transOut` messages from the output ports of the subcomponents, where the function `transEnvOut` produces the message `transOut(NDL, P')` from data `NDL` in each output port `P` of subcomponent `C`: if the port `C . P` is also involved in feedback output connections (by using the `hadFeedback` function), then the data `NDL` remains for the next step, and otherwise removed:

```
eq transferResults(< C : Ensemble | machines : COMPS,
                                    connections : CONXS >)
 = < C : Ensemble | machines : transEnvOut(CONXS, COMPS) > .

op transEnvOut : Set{Connection} Configuration ~> Configuration .
ceq transEnvOut((C . P --> P') ; CONXS
     < C : Component |
         ports : < P : OutPort | content : NDL > PORTS > KCS)
  = transEnvOut(remove(C . P, CONXS),
     < C : Component |
         ports : < P : OutPort | content : DL > PORTS > KCS
     genOut(C . P, NDL, (C . P --> P') ; CONXS))
 if DL := if hasFeedback(C . P, CONXS) then NDL else nil fi .
eq transEnvOut(CONX, KCS) = KCS [owise] .
```

The auxiliary functions `hasFeedback`—that checks if the given port is a feedback output port—and `genOut`—that returns the set of all resulting `transOut` messages—are defined as follows:

```
op hasFeedback : PortName Set{Connection} -> Bool .
eq hasFeedback(PN, (PN --> C . P) ; CONXS) = true .
eq hasFeedback(PN, CONXS) = false [owise] .

op genOut : PortName NeList{Data} Set{Connection} ~> Configuration .
eq genOut(PN, NDL, (PN --> P) ; CONXS)
 = genOut(PN, NDL, CONXS) transOut(NDL, P) .
eq genOut(PN, NDL, CONXS) = none [owise] .
```

B.4 The Simplified Asynchronous Model

This section presents the formal specification of *highly simplified* Multirate PALS asynchronous model used in Section 6.5.5. This specification supports a *hierarchical system design* in which a component may communicate with both faster and slower components. For example, the main controller (period 60 ms) in the airplane turning control system is connected to both the pilot console (period 600 ms) and the left wing subcontroller (period 15 ms).

In the asynchronous real-time model, each component runs according to its own period, and communicates with other components by sending and receiving messages asynchronously. When a component begins its new local period, it reads the incoming messages from its input ports, performs its local transition, and places the generated outputs in its output ports. The input adaptor functions are applied to deal with inputs from components with different periods. To ensure that those output messages are used in the *next* round of the recipient, they are sent into the network as follows:

- A component with period $k \cdot T$ sends a k-tuple of data at the same time to the *slower* component with period T, *one time unit after* all its k local transitions are performed.
- A component sends an output message to a *faster* (or equally fast) component *one time unit after* each local transition is performed.

An output message generated in one round can therefore not be used in the same round, since it is sent one time unit after the beginning of the round.

An asynchronous component is an object instance of (a subclass of) the class `AsyncComponent`, which also integrates all its *wrappers* and the inner component specified by an object instance of `Component`:

```
class AsyncComponent | rate : NzNat,
                       counter : Nat,
                       timer : Time,
                       fastInputs : Set{PortName},
                       slowInputs : Set{PortName},
                       fastOutputs : Set{Connection},
                       slowOutputs : Set{Connection},
                       fastOutputTimer : TimeInf,
                       slowOutputTimer : TimeInf,
                       fastBuffer : Configuration,
                       slowBuffer : Configuration .
subclass AsyncComponent < Component .
```

The `rate` attribute denotes the rate of the component compared to the slower components. The `counter` denotes the number of fast transitions that have been taken in the current slow period, and the `timer` denotes the time until a new *fast* period begins (that is, a new *slow* round of the asynchronous component begins when both `timer` and `counter` attributes are 0). The other attributes of `AsyncComponent` are used to control the input and output of the component: the `fast` attributes are related to *faster* (or equal) components, and the `slow` attributes are related to *slower* components. For example, `fastInputs` denotes a set of input ports that are connected to faster (or equal) components, and `fastOutputs` denotes a set of connections from the component's output ports to input ports of faster components. The output message to a faster component generated during its round is stored in `fastBuffer`, until it is sent into the network when the `fastOutputTimer` expires. The `slow` attributes are similar, but are used for communication with slower components.

In the aircraft turning control system, each controller component is an instance of both its corresponding controller class and `AsyncComponent`. Therefore, we define the common "asynchronous" controller classes:

```
class AsyncSubController .
class AsyncMainController .
class AsyncPilotConsole .
subclass AsyncSubController   <   AsyncComponent SubController .
subclass AsyncMainController  <   AsyncComponent MainController .
subclass AsyncPilotConsole    <   AsyncComponent PilotConsole .
```

The state of the asynchronous model is then represented as a configuration of `AsyncComponent` objects. For example, a state of the aircraft turning control system in the asynchronous model is represented in Figure B.7. Notice that the structure is *flattened*, unlike the synchronous model.

```
< pilot : AsyncPilotConsole |
  period : 600,    rate : 1,    counter : 0,    timer : 0,
  ports : < input : InPort | content : bot >
          < output : OutPort | content : nil >,
  scenario : nil,
  fastOutputTimer : INF,                  slowOutputTimer : INF,
  fastBuffer : none,                      slowBuffer : none,
  fastInputs : input,                     slowInputs : empty,
  fastOutputs : output --> main . input,   slowOutputs : empty >

< left-wing : AsyncSubController |
  period : 15,    rate : 4,    counter : 0,    timer : 0,
  ports : < input : InPort | content : bot >
          < output : OutPort | content : nil >,
  curr-angle : 0.0,  goal-angle : 0.0 ,  diff-angle : 1.0,
  fastOutputTimer : INF,    slowOutputTimer : INF,
  fastBuffer : none,        slowBuffer : none,
  fastInputs : empty,       slowInputs : input,
  fastOutputs : empty,      slowOutputs : output --> main . inLW >

< right-wing : AsyncSubController |
  period : 15,    rate : 4,    counter : 0,    timer : 0,
  ports : < input : InPort | content : bot >
          < output : OutPort | content : nil >,
  curr-angle : 0.0,  goal-angle : 0.0,  diff-angle : 1.0,
  fastOutputTimer : INF,    slowOutputTimer : INF,
  fastBuffer : none,        slowBuffer : none,
  fastInputs : empty,       slowInputs : input,
  fastOutputs : empty,      slowOutputs : output --> main . inRW >

< virt-wing : AsyncSubController |
  period : 20,    rate : 3,    counter : 0,    timer : 0,
  ports : < input : InPort | content : bot >
          < output : OutPort | content : nil >,
  curr-angle : 0.0,  goal-angle : 0.0,  diff-angle : 0.5,
  fastOutputTimer : INF,    slowOutputTimer : INF,
  fastBuffer : none,        slowBuffer : none,
  fastInputs : empty,       slowInputs : input,
  fastOutputs : empty,      slowOutputs : output --> main . inTW >

< main : AsyncMainController |
  period : 60,    rate : 10,    counter : 0,    timer : 0,
  ports :
    < input : InPort | content : bot > < output : OutPort | content : nil >
    < inLW : InPort | content : 0.0 >  < outLW : OutPort | content : nil >
    < inRW : InPort | content : 0.0 >  < outRW : OutPort | content : nil >
    < inTW : InPort | content : 0.0 >  < outTW : OutPort | content : nil >,
  velocity : 50.0,    goal-dir : 0.0,
  curr-yaw : 0.0,     curr-rol : 0.0,    curr-dir : 0.0,
  fastOutputTimer : INF,                 slowOutputTimer : INF,
  fastBuffer : none,                     slowBuffer : none,
  fastInputs : (inLW, inRW, inTW),       slowInputs : input,
  fastOutputs : outLW --> left-wing . input ; outRW --> right-wing . input ;
                outTW --> virt-wing . input,
  slowOutputs : output --> pilot . input >
```

Figure B.7: An initial state of the asynchronous model.

Communication between different components is formalized by explicit message passing, using messages of the form (`msg` *Data* `to` *Target*):

```
op msg_to_ : NeList{Data} PortName -> Msg [ctor] .
```

The rewrite rules for asynchronous communications are straightforward. In the `recv` rule, the component `C` receives the message `(msg NDL to C . P)` and puts it in the corresponding input port `P`:

```
rl [recv]:
   (msg NDL to C . P)
   < C : AsyncComponent |
       ports : < P : InPort | content : DL > PORTS >
 =>
   < C : AsyncComponent |
       ports : < P : InPort | content : DL NDL > PORTS > .
```

There are two rules for sending messages. The rule `sendFast` sends the messages `MSGS` in `fastBuffer` for faster components into the network when `fastOutputTimer` expires, and turns `fastOutputTimer` off. The `sendSlow` rule is similar but uses `slowBuffer` and `slowOutputTimer` to send messages generated for slower components into the network:

```
rl [sendFast]:
   < C : AsyncComponent | fastOutputTimer : 0,
                          fastBuffer : MSGS >
 =>
   < C : AsyncComponent | fastOutputTimer : INF,
                          fastBuffer : none >  MSGS .

rl [sendSlow]:
   < C : AsyncComponent | slowOutputTimer : 0,
                          slowBuffer : MSGS >
 =>
   < C : AsyncComponent | slowOutputTimer : INF,
                          slowBuffer : none >  MSGS .
```

When a component begins a new round (`timer` = 0), the appropriate input adaptors are applied to its input ports, the transition is performed by using the `delta` operator after setting `timer` to its period, and output is put into the relevant output buffers. Since we assume in our simplified asynchronous model that the execution time is zero, the operator `delta` is defined by rewrite rules in the same way as the synchronous semantics. The asynchronous transition of each component is specified by the following rewrite rule, where `rem` is the reminder function:

```
vars PNS FNS SNS : Set{PortName} .  var NZ : NzNat .   var N : Nat .
vars FCXS SCXS : Set{Connection} .  vars FMGS SMGS : Configuration .
vars NCF NCF' : NEConfiguration .   vars T T' TI TI' : Time .

crl [step]:
    < C : AsyncComponent | timer : 0,    period : T,    rate : NZ,
                           counter : N,       ports : PORTS,
                           fastInputs : FNS,  slowInputs : SNS >
 => procBufferOut(OBJ)
 if PNS := if N == 0 then (FNS, SNS) else FNS fi
 /\ TI := if NZ == s(N) then 1 else INF fi
 /\ delta(< C : AsyncComponent |
              ports : applyAdaptors(C,PORTS,NS),
              timer : T,            counter : s(N) rem NZ,
              fastOutputTimer : 1,  slowOutputTimer : TI >) => OBJ .
```

In the condition of the rule, `fastOutputTimer` is set to 1 so that the messages in `fastBuffer` are sent one time unit later; but `slowOutputTimer` is set to 1 only if all its fast transitions are performed in the slow round (i.e., `rate` $=$ `counter` $+ 1$). If an input port is connected to a slower component, the adaptor for the input port is applied only if the current round is a slow round (`counter` $= 0$), where the `applyAdaptors` function now takes an extra argument to denote a set of chosen input port names.

```
op applyAdaptors : ComponentId Configuration Set{PortName}
                -> Configuration .
eq applyAdaptors(C, < P : InPort | content : NDL > PORTS, (P, PNS))
 = applyAdaptors(C, PORTS, PNS)
   < P : InPort | content : adaptor(C,P,NDL) > .
eq applyAdaptors(C, PORTS, empty) = PORTS .
```

After performing the `delta` operator, all the data in the output ports are transferred to the corresponding output buffers according to the connection information in `fastOutputs` and `slowOutputs`:

```
op procBufferOut : Object -> Object .
eq procBufferOut(< C : AsyncComponent |
                  ports : < P : OutPort | content : NDL > PORTS,
                  fastOutputs : FCXS,         fastBuffer : FMGS,
                  slowOutputs : SCXS,         slowBuffer : SMGS >)
 = procBufferOut(< C : AsyncComponent |
                  ports : < P : OutPort | content : nil > PORTS,
                  fastBuffer : merge(FMGS, genMsgs(P,NDL,FCXS)),
                  slowBuffer : merge(SMGS, genMsgs(P,NDL,SCXS)) >) .
eq procBufferOut(OBJECT) = OBJECT [owise] .
```

The genMsgs function generates messages from output ports based on connections, and the merge function combines two multisets of messages so as to generate messages with k-tuples of data for slow components:

```
op genMsgs : PortId NeList{Data} Set{Connection} -> Configuration .
eq genMsgs(P, NDL, (P --> PN) ; CONXS)
 = genMsgs(P, NDL, CONXS) (msg NDL to PN) .
eq genMsgs(P, NDL, CONXS) = none [owise] .

op merge : Configuration Configuration -> Configuration .
eq merge(MSGS (msg NDL to PN), MSGS' (msg NDL' to PN))
 = mergeMsgs(MSGS (msg NDL NDL' to PN), MSGS') .
eq merge(MSGS, MSGS') = MSGS MSGS' [owise] .
```

Finally, the following tick rule advances time until some asynchronous event must happen, where the function timeEffect defines how the system state changes according to the time elapsed, and mte defines the maximum amount of time that may elapse in the system until some timer expires:

```
crl [tick] :  {< C : AsyncComponent | >}
          =>  {timeEffect(< C : AsyncComponent | >, T)} in time T
 if T := mte(< C : AsyncComponent | >) .
```

The function timeEffect is distributed over the objects and messages in the state, and decreases the timers of each component in the system by the amount of time elapsed (given by the variable TE):

```
op timeEffect : Configuration Time -> Configuration [frozen] .
eq timeEffect(NCF NCF', T) = timeEffect(NCF,T) timeEffect(NCF',T) .
eq timeEffect(none, T) = none .           eq timeEffect(M, T) = M .
eq timeEffect(< C : AsyncComponent | timer : T',
                                     fastOutputTimer : TI,
                                     slowOutputTimer : TI' >, T)
 =   < C : AsyncComponent |  timer : T' monus T,
                             fastOutputTimer : TI monus T,
                             slowOutputTimer : TI' monus T > .
```

Similarly, mte of a configuration is the smallest mte value of an object or a message in the configuration, where mte of a component is the smallest value in its timers, and mte of a message is 0:

```
op mte : Configuration -> TimeInf [frozen] .     eq mte(M) = 0 .
eq mte(NCF NCF') = min(mte(NCF), mte(NCF')) .    eq mte(none) = INF .
eq mte(< C : AsyncComponent |
          timer : T,
          fastOutputTimer : TI,
          slowOutputTimer : TI' >) = min(T,TI,TI') .
```

APPENDIX C

MORE DETAILS ON MULTIRATE SYNCHRONOUS AADL

C.1 More Details on the Real-Time Maude Semantics

This section shows the definitions of some semantic functions for Multirate Synchronous AADL, summarized in Chapter 7. The entire semantics is available at in `http://maude.cs.illinois.edu/tools/synchaadl`.

Reading Features. The function `readFeature` returns a map from each input port to its current value for the `execute` rule. Given a set of port objects, the `readFeature` function "consumes" the current value of each input port, and constructs a map from port identifiers to their current values.

First, if an input port `P` has value `V`, then the port `P` is related to the pair '`V : true`' in the resulting map `FMAP` (indicating that `P'fresh` is `true`) and the `cache` attribute of the input port `P` is also updated to the value `V`:

```
var P : FeatureId .  var DCL : List{DataContent} .  var V : Value .
eq readFeature(< P : InPort | content : V DCL > PORTS, PORTS', FMAP)
 = readFeature(PORTS,
               < P : InPort | content : DCL, cache : V > PORTS',
               insert(P, V : true, FMAP)) .
```

Second, if an input port `P` has the value `bot` (i.e., no "actual" value has been received in the latest dispatch), then the port `P` is related to the pair '`V : false`' (indicating that `P'fresh` is `false`) with `V` the `cache`-ed value:

```
eq readFeature(< P : InPort | content : bot DCL, cache : V > PORTS,
               PORTS', FMAP)
 = readFeature(PORTS, < P : InPort | content : DCL > PORTS',
               insert(P, V : false, FMAP)) .
```

Finally, each output port P is related to the "don't care" value bot, since transitions cannot read a value from such an output port P:

```
eq readFeature(< P : OutPort | > PORTS, PORTS', FMAP)
 = readFeature(PORTS, < P : OutPort | > PORTS', insert(P, bot, FMAP)) .

eq readFeature(none, PORTS', FMAP) = PORTS' | FMAP .
```

Writing Features. The function writeFeature updates the content of each output port from the result for the execute rule. The definition of writeFeature is straightforward; for each output port P, if some value V (other than bot) is written for P in the map FMAP, then V is added to the end of the data content of the output port P, and otherwise, bot is added:

```
eq writeFeature(FMAP, < P : OutPort | content : DCL > PORTS, PORTS')
 = if $hasMapping(FMAP, P) and FMAP[P] :: Value then
      writeFeature(FMAP, PORTS,
                   < P : OutPort | content : DCL FMAP[P] > PORTS')
   else
      writeFeature(FMAP, PORTS,
                   < P : OutPort | content : DCL bot > PORTS') fi .

eq writeFeature(FMAP, PORTS, PORTS') = PORTS PORTS' [owise] .
```

Enabled Transitions. The function enabledTrans is defined as follows. Any transition guarded by on dispatch is enabled. A transition guarded by a Boolean expression E is enabled only if E is evaluated to true. If there are no enabled transitions from the current state L by the above cases, then all transitions from L guarded by otherwise are enabled:

```
eq enabledTrans(L, (L -[on dispatch]-> L' ACTION) ; TRS,
                FMAP | DATA | PROPS, TRS')
 = enabledTrans(L, TRS, FMAP | DATA | PROPS,
                TRS' ; (L -[on dispatch]-> L' ACTION)) .

eq enabledTrans(L, (L -[E]-> L' ACTION) ; TRS,
                FMAP | DATA | PROPS, TRS')
 = if eval(E, empty | FMAP | DATA | PROPS) == [true]
   then enabledTrans(L, TRS, FMAP | DATA | PROPS,
                     TRS' ; (L -[E]-> L' ACTION))
   else enabledTrans(L, TRS, FMAP | DATA | PROPS, TRS') fi .

eq enabledTrans(L, TRS, FMAP | DATA | PROPS, TRS')
 = if TRS' == empty then owiseTrs(L,TRS,empty) else TRS' fi [owise] .
```

```
eq owiseTrs(L, (L -[otherwise]-> L' ACTION) ; TRS, ETRS)
 = owiseTrs(L, TRS, ETRS ; (L -[otherwise]-> L' ACTION)) .
eq owiseTrs(L, TRS, ETRS) = ETRS [owise] .
```

Evaluating Behavior Expressions. An expression E is evaluated to a "value" by the function `eval`, based on the configuration of the temporary variable values `VAL`, the input port values `FMAP`, the state variables `DATA`, and the property values `PROPS`. For example, the following equations defines the basic cases: a value `V`, a temporary variable `VI`, a state variable `C`, a port identifier `P`, a property name `PR`, and a fresh expression:

```
var VI : VarId .      var VAL : VarValuation .     var B : Bool .

eq eval(V, VAL | FMAP | DATA | PROPS) = V .
eq eval([VI], (VI |-> V) ; VAL | FMAP | DATA | PROPS) = V .
eq eval([C], VAL | FMAP | < C : Data | value : V > DATA | PROPS) = V .
eq eval([P], VAL | (P |-> (V : B), FMAP) | DATA | PROPS) = V .
eq eval([PR], VAL | FMAP | DATA | (PR => PV) ; PROPS) = value(PV) .
eq eval(fresh(P), VAL | (P |-> (V : B), FMAP) | DATA | PROPS) = [B] .
```

The cases for the other expressions are defined by propagating `eval` to their subexpressions; for example, the semantics of negation expressions, conjunction expressions, and addition expressions is defined by:

```
eq eval(not(E), VAL | FMAP | DATA | PROPS)
 = not(eval(E, VAL | FMAP | DATA | PROPS)) .

eq eval(E and E', VAL | FMAP | DATA | PROPS)
 = eval(E, VAL | FMAP | DATA | PROPS) and
   eval(E', VAL | FMAP | DATA | PROPS) .

eq eval(E + E', VAL | FMAP | DATA | PROPS)
 = eval(E, VAL | FMAP | DATA | PROPS) +
   eval(E', VAL | FMAP | DATA | PROPS) .
```

Executing Behavior Actions. Whenever a behavior transition executes its action block `ACTION`, it uses the *default* valuation for local *temporary* variables in which each variable `VI` is mapped to `bot` as follows:

```
eq execAction(ACTION, VARS, FMAP | DATA | PROPS)
 = execAction(ACTION, defaultValuation(VARS) | FMAP | DATA | PROPS)
eq defaultValuation(VI ; VARS)
 = (VI |-> bot) ; defaultValuation(VARS) .
eq defaultValuation(empty)  = empty .
```

The function `execAction` then executes an action block using the current configuration `VAL | FMAP | DATA | PROPS`, and returns a new configuration. For example, an assignment action *id* `:=` *exp* assigns the evaluated value of *exp* to the identifier *id* as follows:

```
var DC : DataContent .  var A : Action .  var SEQ : ActionSequence .

ceq execAction({VI} := E, (VI |-> DC) ; VAL | FMAP | DATA | PROPS)
  = (VI |-> V) ; VAL | FMAP | DATA | PROPS .
 if V := eval(E, (VI |-> DC) ; VAL | FMAP | DATA | PROPS) .

ceq execAction({P} := E, VAL | (P |-> DC, FMAP) | DATA | PROPS)
 = VAL | (P |-> V, FMAP) | DATA | PROPS .
 if V := eval(E, VAL | (P |-> DC, FMAP) | DATA | PROPS) .

ceq execAction({C} := E,
               VAL | FMAP | < C : Data | value : DC > DATA | PROPS)
  = VAL | FMAP | < C : Data | value : V > DATA | PROPS .
 if V := eval(E, VAL | FMAP | < C : Data | value : DC > DATA | PROPS)
```

For a sequence of actions $\{Action_1 \;;\; \cdots \;;\; Action_n\}$, an action $Action_k$ in the sequence is executed based on the result of the previous actions:

```
eq execAction({A ; SEQ}, VAL | FMAP | DATA | PROPS)
 = execAction({SEQ}, execAction(A, VAL | FMAP | DATA | PROPS)) .
eq execAction({A}, VAL | FMAP | DATA | PROPS)
 = execAction(A, VAL | FMAP | DATA | PROPS) .
```

A conditional action can also be straightforwardly specified as follows:

```
eq execAction(if (E) SEQ else SEQ' end if, VAL | FMAP | DATA | PROPS)
 = if eval(E, VAL | FMAP | DATA | PROPS) == [true]
   then execAction({SEQ}, VAL | FMAP | DATA | PROPS)
   else execAction({SEQ'}, VAL | FMAP | DATA | PROPS) fi .

eq execAction(if (E) SEQ (elsif (E') SEQ' ELSIFS) else SEQ'' end if,
             VAL | FMAP | DATA | PROPS)
 = if eval(E, VAL | FMAP | DATA | PROPS) == [true]
   then execAction({SEQ}, VAL | FMAP | DATA | PROPS)
   else execAction(if (E') SEQ' ELSIFS else SEQ'' end if,
                  VAL | FMAP | DATA | PROPS) fi .
```

Finally, math library functions are separately defined in Maude; e.g.:

```
ceq execAction(MathLib::sqrt ! (E, E'), VAL | FMAP | DATA | PROPS)
 = execAction(target(E') := [sqrt(F)], VAL | FMAP | DATA | PROPS) .
 if F := float(eval(E,VAL | FMAP | DATA | PROPS)) .
```

Applying Input Adaptors. For each input port `P` of the subcomponents of an ensemble `C`, the function `applyAdaptors` applies a (predefined) input adaptor to the input port `P` if it is declared. First, `applyAdaptors` calls the auxiliary function `applyAdaptors(GT,COMPS)` with `GT` the period of the ensemble `C` and `COMPS` the subcomponents of `C`:

```
eq applyAdaptors(< C : Ensemble |
                    subcomponents : COMPS,
                    properties : (Period => GT) ; PROPS >)
 = < C : Ensemble | subcomponents : applyAdaptors(GT, COMPS) > .
```

Next, for each subcomponent `C'` with period `T`, it calls another auxiliary function `applyAdaptors(GT quo T, PORTS, none)`, where `quo` is the integer quotient function (i.e., `GT quo T` is the "rate" of the component `C'`):

```
eq applyAdaptors(GT,
    < C' : Component | features : PORTS,
                       properties : (Period => T) ; PROPS > REST)
 = applyAdaptors(GT, REST)
   < C' : Component |
      features : applyAdaptors(GT quo T, PORTS, none) > .

eq applyAdaptors(GT, none) = none .
```

If an input adaptor `IA` is defined for an input port `P` as its property, then the input adaptor `IA` is applied to the data content list `DL` of the port `P`:

```
eq applyAdaptors(N,
    < P : InPort |
       content : DL,
       properties : (MRSynchAADL::InputAdaptor => {IA}); PROPS >
    PORTS, PORTS')
 = applyAdaptors(N,
    PORTS, PORTS' < P : InPort | content : adaptor(IA, DL, N) >) .

eq applyAdaptors(N, PORTS, PORTS') = PORTS PORTS' [owise] .
```

The semantics of predefined input adaptors is defined by the function `adaptor(`*id*`,`*data*`,`*rate*`)`, where *data* is a data content list in the input port and *rate* is the rate of the component. For example, the 1-to-k input adaptor `repeat input` can be defined by the following equations:

```
eq adaptor(repeat input, D, N) = adaptor(repeat input, D, N, nil) .
eq adaptor(repeat input, D, s(N), DL)
 = adaptor(repeat input, D, N, DL D) .
eq adaptor(repeat input, D, 0, DL) = DL .
```

Semantics of Requirements Specification Language. The semantics of the requirements specification language is defined by using equations. For example, the meaning of state-lockup propositions are defined as follows using the auxiliary function `lookupState`:

```
eq {< C : Ensemble | subcomponents : COMPS >} |= PATH @ L
   = lookupState(COMPS, PATH, L) .
eq lookupState(< C : Ensemble | subcomponents : COMPS > REST,
               C . PATH, L)
 = lookupState(COMPS, PATH, L) .
eq lookupState(< C : Thread | currState : L > REST, C, L) = true .
eq lookupState(REST, PATH, L) = false [owise] .
```

Likewise, the meaning of expression propositions are defined as follows, where the function `feedbackOutputs` returns a map from each feedback output port to its current value:

```
eq {< C : Ensemble | subcomponents : COMPS >} |= PATH | E
 = lookupExp(COMPS, PATH, E) .
eq lookupExp(< C : Ensemble | subcomponents : COMPS > REST,
             C . PATH, E)
 = lookupExp(COMPS, PATH, E) .
eq lookupExp(< C : Component | features : PORTS, properties : PROPS,
                                subcomponents : DATA, > REST, C, E)
 = eval(E, empty | feedbackOutputs(PORTS) | DATA | PROPS) == [true] .
eq lookupExp(REST, PATH, E) = false [owise] .

eq feedbackOutputs(< P : OutPort | content : DCL V > PORTS, FMAP)
 = feedbackOutputs(PORTS, insert(P, V, FMAP)) .
eq feedbackOutputs(PORTS, FMAP) = FMAP [owise] .
```

Each formula declaration in the MR-SynchAADL tool automatically adds the corresponding equations. For example, the `safeYaw` formula for the airplane controller example generates the following Maude declarations:

```
op safeYaw : -> Formula .
eq safeYaw = turningCtrl . mainController . ctrlProc . ctrlThread |
                abs([currYaw]) < [1.0] .
```

Finally, each requirement declaration gives the corresponding Real-Time Maude verification command. For example, the `safety` requirement gives the following command where `initial` will be reduced to the term representation of the initial state of the entire model.

```
(mc {initial} |=u [] safeYaw .)
```

C.2 The Active Standby System Requirements

For the active standby system explained in Section 7.5.2, the paper [143] lists the following requirements that the active standby system must satisfy:

R_1: Both sides should agree on which side is active (provided that neither side has failed, the availability of a side has not changed, and the pilot has not made a manual selection).

R_2: A side that is not fully available should not be the active side if the other is fully available (when neither side has failed, the availability of a side has not changed, and the pilot has not made a manual selection).

R_3: The pilot can always change the active side (unless a side is failed or the availability of a side has changed).

R_4: If a side is failed, then the other side should become active.

R_5: The active side should not change unless the availability of a side changes, the failed status of a side changes, or manual selection is selected by the pilot.

A more detailed discussion of the LTL representation of the above properties can be found in [138]. The requirement R_1 is explained in Section 7.5.2, and this section explains the other requirements R_1–R_4.

Requirement R_2. *A side that is not fully available should not be the active side if the other side is fully available (provided that neither side has failed, the availability of a side has not changed, and the pilot has not made a manual selection).* This property does not hold as stated, since a standby side monitors full availability and therefore the change of active side can be delayed by one round [138]. Instead, we have verified the following property:

```
requirement R2a:
   O ([] ((noChangeAssumptionNextState /\
              O (side1FullyAvailable /\ ~ side2FullyAvailable)) ->
         O (~ side2Active \/
             (noChangeAssumptionNextState -> O (~ side2Active)))));
```

where the proposition `side`i`Active` holds if side i has received the value i in its `side`i`ActiveSide` port:

```
formula side1Active:
   sideOne.sideProcess.sideThread | side1ActiveSide = 1;
formula side2Active:
   sideTwo.sideProcess.sideThread | side2ActiveSide = 2;
```

Requirement R_3**.** *The pilot can always change the active side (unless a side is failed or the availability of a side has changed).* Since R_3 does not satisfied again as explained in [138], we have verified the following variant of R_3: if the two sides are fully available and do not receive a manual switch request for two consecutive rounds, and stay faultless and receive a manual switch request in the third round, then the active side will switch:

```
requirement R3g:
  [] ((~ manSelectPressed /\ agreeOnActiveSide /\
       bothFullyAvailable /\ noChangeAssumptionNextState)
     ->
      ((side1Active ->
           O (O (   (manSelectPressed /\ bothFullyAvailable)
                 -> side2Active))) /\
       (side2Active ->
           O (O (   (manSelectPressed /\ bothFullyAvailable)
                 -> side1Active)))));

formula bothFullyAvailable:
   side1FullyAvailable /\ side2FullyAvailable;
```

Requirement R_4**.** *If a side is failed, then the other side should become active.* We have verified this requirement R_4, represented as the following LTL formula considering the one-step communication delay:

```
requirement R4:  [] (
 ((side1Failed /\ ~ side2Failed) -> O (~ side2Failed -> side2Active))
/\
 ((side2Failed /\ ~ side1Failed) -> O (~ side1Failed -> side1Active)));
```

Requirement R_5**.** *The active side should not change, unless the failed status or the availability of a side changes or manual selection is selected by the pilot.* We have verified the requirement R_5 for active side 1, represented in the requirement specification language as follows (i.e., if side 1 is active, then it stays active forever, or until something changes [138]):

```
requirement R5side1:  [] (
   ((side1Active /\ side1FullyAvailable /\ ~ manSelectPressed)
     -> (side1Active W (~ side1FullyAvailable \/ manSelectPressed)))
/\ ((side1Active /\ ~ side1FullyAvailable /\ side1Stable)
     -> (side1Active W (~ side1Stable))));

formula side1Stable:
   ~ side2FullyAvailable /\ ~ manSelectPressed /\ ~ side1Failed;
```

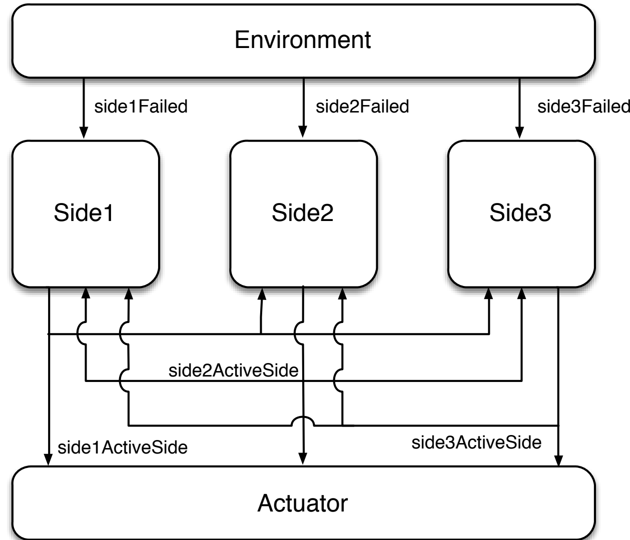


Figure C.1: The architecture of the 3-node active standby system.

We have verified every requirement of the Multirate Synchronous AADL model that has 203 reachable states. Each model checking analysis took 0.6 seconds on an Intel Xeon 2.93 GHz with 24GB RAM. Note that it is unfeasible to model check the corresponding *asynchronous* design: as shown in [138], the *simplest possible* asynchronous model (no message delays, no execution times, etc.) has 3,047,832 reachable states. If the message delay can be 1 then no model checking terminates in reasonable time.

C.3 The Three–Node Active Standby System

This section shows another version of the active standby system with three nodes in Multirate Synchronous AADL. In this version, there exists *three* side components, instead of two. Each side can fail, but the active side should always be one of non-failed sides. The architecture of this model is shown in Figure C.1. The entire Multirate Synchronous AADL specification is available at `http://maude.cs.illinois.edu/tools/synchaadl`.

Top-level Component. The top-level AADL system component contains the five subcomponents. The three sides and forward their output to the `actuator` component. The `env` component injects failure into the three side components by sending 3 Boolean data nondeterministically, with the input constraint that all sides cannot fail at the same time.

```
system implementation MainSystem.impl
  subcomponents
    sideOne: system Side1.impl;     sideTwo: system Side2.impl;
    sideThree: system Side3.impl;   env: system Environment.impl;
    actuator: system Actuator.impl;
  connections
    C1: data port sideOne.side1Active -> sideTwo.side1Active;
    C2: data port sideOne.side1Active -> sideThree.side1Active;
    C3: data port sideTwo.side2Active -> sideOne.side2Active;
    C4: data port sideTwo.side2Active -> sideThree.side2Active;
    C5: data port sideThree.side3Active -> sideOne.side3Active;
    C6: data port sideThree.side3Active -> sideTwo.side3Active;
    C7: data port env.side1Failed -> sideOne.side1Failed;
    C8: data port env.side2Failed -> sideTwo.side2Failed;
    C9: data port env.side3Failed -> sideThree.side3Failed;
    C10: data port sideOne.side1Active -> aileron.side1Active;
    C11: data port sideTwo.side2Active -> aileron.side2Active;
    C12: data port sideThree.side3Active -> aileron.side3Active;
  properties
    MR_SynchAADL::Synchronous => true;      Period => 2 Ms;
    Timing => Delayed applies to
      C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12;
    Data_Model::Initial_Value => ("false") applies to
      env.side1Failed, env.side2Failed, env.side3Failed;
end MainSystem.impl;
```

Environment. The environment component is declared as an instance of the following system component:

```
system Environment
  features
    side1Failed: out data port Base_Types::Boolean;
    side2Failed: out data port Base_Types::Boolean;
    side3Failed: out data port Base_Types::Boolean;
end Environment;

system implementation Environment.impl
  subcomponents
    envProcess: process EnvironmentProcess.impl;
  connections
    data port envProcess.side1Failed -> side1Failed;
    data port envProcess.side2Failed -> side2Failed;
    data port envProcess.side3Failed -> side3Failed;
end Console.impl;
```

This component contains an instance of the following thread component defining the environment behavior, which sends any values satisfying the input constraint '`not s1F or not s2F or not s3F`' to the corresponding output ports, stating that all sides cannot fail at the same time:

```
thread implementation EnvironmentThread.impl
  properties
    MR_SynchAADL::InputConstraints =>
                    ("not s1F or not s2F or not s3F");
    MR_SynchAADL::Nondeterministic => true;
    Dispatch_Protocol => Periodic;
  annex behavior_specification {**
    states
      s0 : initial complete state;
    variables
      s1F, s2F, s3F: Base_Types::Boolean;
    transitions
        s0 -[on dispatch]-> s0 {
          side1Failed := s1F;
          side2Failed := s2F; side3Failed := s3F;};
**};
  end EnvironmentThread.impl;
```

Sides. We only show the specification of the `sideTwo` component. The `sideTwo` is declared as an instance of the following system component with three input ports and one output port:

```
system Side2
  features
    side3Active: in data port Base_Types::Integer;
    side1Active: in data port Base_Types::Integer;
    side2Failed: in data port Base_Types::Boolean;
    side2Active: out data port Base_Types::Integer;
end Side2;

system implementation Side2.impl
  subcomponents
    sideProcess: process Side2Process.impl;
  connections
    data port side3Active -> sideProcess.side3Active;
    data port side1Active -> sideProcess.side1Active;
    data port side2Failed -> sideProcess.side2Failed;
    data port sideProcess.side2Active -> side2Active;
end Side2.impl;
```

The system `Side2` contains an instance of the thread component defining the behavior of the side below. The output of `Side2` is determined by the "local view" of the three sides' statuses perceived by `Side2`, represented by three state variables `g_side1`, `g_side2`, and `g_side3`. The synchronous behavior of the entire system guarantees that these local views of all the three sides are identical, so that all sides are consistent:

```
thread implementation Side2Thread.impl
  subcomponents
    g_side1: Base_Types::Integer;    g_side2: Base_Types::Integer;
    g_side3: Base_Types::Integer;    prev: Base_Types::Integer;
  properties
    Dispatch_Protocol => Periodic;
    Data_Model::Initial_Value => ("-1") applies to
      g_side1, g_side2, g_side3, prev;
  annex behavior_specification {**
    states
      s0: initial complete state;           s1: state;
    transitions
      s0 -[on dispatch]-> s1 {
        g_side2 := prev;
        if (side1Active'fresh) g_side1 := side2Active;
        else                   g_side1 := -1;      end if;
        if (side3Active'fresh) g_side3 := side3Active;
        else                   g_side3 := -1;      end if; };

      s1 -[side2Failed = true]-> s0 {prev := -1; side2Active := -1;};

      s1 -[side2Failed = false]-> s0 {
        if (g_side2 = -1)
          prev := 0;   side2ActiveSide := 0;
        elsif (g_side2 = 0 and g_side1 = -1 and g_side3 != 1)
          prev := 1;   side2ActiveSide := 1;
        else
          side2ActiveSide := prev;
        end if; };
   **};
end Side2Thread.impl;
```

By the *dispatch* transition, the value of `g_side2` becomes the value of `prev` computed at the previous step, and the 'fresh' values of `side1Active` and `side3Active` are assigned to `g_side1` and `g_side3`, respectively. Then, `Side2` becomes active (`prev` = 1) if the others are not active.

C.3.1 System Requirements

The requirements of the active standby system with three nodes can be summarized as follows, in a similar way to the case of two nodes:

R_1: At most one side is the active side, to simplify the logic of the actuator.

R_2: A new active side should be chosen after the failure of an active side.

R_3 Every side should agree on which side is the active side.

We have also verified these requirements for the Multirate Synchronous AADL model of the three–node active standby system within OSATE.

Requirement R_1. *At most one side is the active side.* Side i is active if the side system component produces the value 1 in its `side`*i*`Active` port. Therefore, this property R_1 can be simply represented as follows:

```
requirement R1:
    [] ((~ side1Active \/ ~ side2Active) /\
        (~ side1Active \/ ~ side3Active) /\
        (~ side2Active \/ ~ side3Active));
formula side1Active: sideOne | side1Active = true;
formula side2Active: sideTwo | side2Active = true;
formula side3Active: sideThree | side3Active = true;
```

Requirement R_2. *A new active side should be chosen after the failure of an active side.* This property requires that the failure status of a side is not “flipping.” Side i is flipping iff the side does not fail in the next step whenever it fails in one step, and vice versa. Then, the property R_2 can be written in MR-SynchAADL as follows:

```
requirement R2:
   []<> (~ flipOne /\ ~ flipTwo /\ ~ flipThree)
-> []((~ side1Active /\ ~ side2Active /\ ~ side3Active) ->
       <> (side1Active \/ side2Active \/ side3Active));
formula flipOne:
     (env | side1Failed = true <-> O env | side1Failed = false)
 \/  (env | side1Failed = false <-> O env | side1Failed = true);
formula flipTwo:
     (env | side2Failed = true <-> O env | side2Failed = false)
 \/  (env | side2Failed = false <-> O env | side2Failed = true);
formula flipThree:
     (env | side3Failed = true <-> O env | side3Failed = false)
 \/  (env | side3Failed = false <-> O env | side3Failed = true);
```

Requirement R_3**.** *Every side should agree on which side is the active side.* In the three–node active standby system, each side thinks that side i is active if the state variable `g_side`i has the value 1. Such "local views" for side i are identical when the variables `g_side`i have the same value in all the sides. Therefore, the property R_3 can be defined as follows:

```
requirement R3: [] (sameOne /\ sameTwo /\ sameThree);

formula sameOne:
    (sideOne.sideProcess.sideThread | g_side1 = 0 <->
      sideThree.sideProcess.sideThread | g_side1 = 0)
 /\ (sideOne.sideProcess.sideThread | g_side1 = 0 <->
      sideTwo.sideProcess.sideThread | g_side1 = 0)
 /\ (sideOne.sideProcess.sideThread | g_side1 = 1 <->
      sideThree.sideProcess.sideThread | g_side1 = 1)
 /\ (sideOne.sideProcess.sideThread | g_side1 = 1 <->
      sideTwo.sideProcess.sideThread | g_side1 = 1);

formula sameTwo:
    (sideOne.sideProcess.sideThread | g_side2 = 0 <->
      sideThree.sideProcess.sideThread | g_side2 = 0)
 /\ (sideOne.sideProcess.sideThread | g_side2 = 0 <->
      sideTwo.sideProcess.sideThread | g_side2 = 0)
 /\ (sideOne.sideProcess.sideThread | g_side2 = 1 <->
      sideThree.sideProcess.sideThread | g_side2 = 1)
 /\ (sideOne.sideProcess.sideThread | g_side2 = 1 <->
      sideTwo.sideProcess.sideThread | g_side2 = 1);

formula sameThree:
    (sideOne.sideProcess.sideThread | g_side3 = 0 <->
      sideThree.sideProcess.sideThread | g_side3 = 0)
 /\ (sideOne.sideProcess.sideThread | g_side3 = 0 <->
      sideTwo.sideProcess.sideThread | g_side3 = 0)
 /\ (sideOne.sideProcess.sideThread | g_side3 = 1 <->
      sideThree.sideProcess.sideThread | g_side3 = 1)
 /\ (sideOne.sideProcess.sideThread | g_side3 = 1 <->
      sideTwo.sideProcess.sideThread | g_side3 = 1);
```

We have verified the requirements R_1–R_3 for the Multirate Synchronous AADL model. Each model checking analysis took less than 1 seconds on the same machine, and the number of reachable states was 134.

APPENDIX D

MORE DETAILS ON PTOLEMY II DE MODELS

D.1 More Ptolemy II Actors

In addition to the Ptolemy II actors explained in Section 8.2.2, we have also defined the Real-Time Maude semantics for the following actors.

- *Pulse.* When an input is received, a *pulse* actor outputs pulses with values given by the *values* parameter; the parameter *indexes* specifies when those values should be produced.[1] A zero is produced when the iteration count does not match an index. After that, the output is always 0, unless yet another parameter, *repeat*, is set to true, in which case the output is repeated. The *init* action does nothing, *fire* outputs a value, and *postfire* updates the number of times *fire* has been invoked.

- *Ramp.* A *ramp* actor simulates the behavior of a "for loop" in a conventional programming language. Each time a *ramp* actor fires, it generates an event with a value that is incremented by the specified *step* each iteration. The first output is given by the *init* parameter.

- *Variable Delay.* A *variable delay* actor works in a similar way as a timed delay actor, except that the amount of time delay is specified by an incoming token through the *delay port*.

- *Timer.* The difference between a *timer* actor and a delay actor is that the value of the generated output of a timer is not the same as the input, but is given by the *output* parameter of this actor. The length of the delay is specified by the input received in the actor's input port.

[1] For example, if the *indexes* parameter is "{1, 3, 0, 2, 4}", and the *values* are stored in array A, then the output in the first 5 invocation of *fire* is $A[1]$, $A[3]$, $A[0]$, $A[2]$, and $A[4]$.

- *Noninterruptible Timer.* A *noninterruptible timer* is quite similar to a normal timer, but with the difference that a noninterruptible timer delays the processing of a new input if it has not finished processing a previous input. That is, other input events are queued, while an input event is being delayed and the corresponding output has not been sent.

- *Timed Plotter.* A *timed plotter* records its received events and the times they were received (by the *postfire*).

- *Single Event.* A *single event* produces an event with the specified value at the specified time (by the *init*).

- *Expression.* An *expression* actor contains an expression that specifies the value of its output as a function of its inputs.

In sum, we support the following actors: timed delay, variable delay, clock, current time, timer, noninterruptible timer, FSM, pulse, timed plotter, set variable, expression, ramp, single event actors, composite actors, modal models, and algebraic actors (e.g., add, subtract, const, and scale).

D.2 Real-Time Maude Code Generation

Ptolemy II provides an *adapter* infrastructure to support the generation of code into any target language. In particular, Ptolemy II provides a Java class `CodeGeneratorHelper` that contains several utility methods such as `getComponent()`, which returns a (Java) object containing all information about an actor, including its name, parameters, ports, inner actors, etc. This class furthermore contains "skeleton" functions, including:

- `String generateFireCode()`, generating the code executed when the actor is "fired," and

- `Set getSharedCode()`, generating code shared by multiple instances of the same actor class.

For each kind of actor, we must define a *adapter* class that extends the class `CodeGeneratorHelper`. An adapter class consists of a Java class file and a *code template* file—containing code blocks written in the target language—that together specify the actor's behavior. Such a template file contains code blocks of the following form, where the *code pattern* is code written in the target language (a code pattern can be parametrized with variables, and also have macro functions):

```
/***header(parameters)***/
  code_pattern
/**/
```

For the Real-Time Maude code generation, each adapter class A has its template file that includes a code block with header `semantics_A`, which is just the Real-Time Maude module defining the formal semantics of the actor A. The template file also includes a code block with header `attr_A` that defines the attributes of the actor and their initial values. Moreover, if the actor A has its own atomic proposition pattern, then a code block with header `formal_A` is included for the definition of such a proposition. In Ptolemy II, each actor class is a subclass of the class `Entity`. Therefore, we defined an adapter class for `Entity` that is a superclass of every actor adapter class. The template file for `Entity` hence contains the following code blocks, where macros are prefixed with '`$`':

```
/***semantics_Entity***/
(tomod ACTOR is
...
  class Actor | ports : ObjectConfiguration,
                parameters : ObjectConfiguration,
                status : ActorStatus,
                computation : Computation .
...
endtom)
/**/

/***fireBlock($attr_terms)***/
< '$info(name) : $info(class) | $attr_terms >
/**/

/***attr_Entity***/
computation : noComputation,
status : enabled,
ports : ($info(ports)),
parameters : ($info(parameters))
/**/

/***formal_Entity***/
(tomod CHECK-ACTOR is
...
endtom)
/**/
```

The parameter `attr_terms` of the code block `fireBlock` will be replaced by set of `attr_`*`Actor`* code blocks for each *`Actor`* a super class of the given actor. `$info` is a macro that uses Ptolemy's `getComponent()` to extract information, such as the name, the class, etc., about the actor instance. Likewise, the template file for `currentTime` contains

```
/***semantics_CurrentTime***/
(tomod CURRENT-TIME is inc ACTOR .
  ...
  class CurrentTime | current-time : Time .
  subclass CurrentTime < AtomicActor TimeActor .
  ...
  eq portFixPoints(...) = ... .
endtom)
/**/

/***attr_CurrentTime***/
current-time : 0
/**/
```

The Real-Time Maude code generation is implemented by redefining the functions `getSharedCode()` and `generateFireCode()` in the adapter class for each type of actor. The function `getSharedCode()` is used to generate the Real-Time Maude modules defining the semantics of those actors, and is defined as the following Java function that returns the set of all code blocks whose header starts with '`semantics`' and '`formal`':

```
public Set getSharedCode() throws IllegalActionException {
    // Use LinkedHashSet to give order to the shared code.
    Set sharedCode = new LinkedHashSet();
    semanticsIncludes = getModuleCode("semantics");
    formalIncludes = getModuleCode("formal");

    for (String m : semanticsIncludes)
        sharedCode.add(getRTMmodule().get(m));
    for (String m : formalIncludes)
        sharedCode.add(getRTMmodule().get(m));
    return sharedCode;
}
```

The auxiliary function `getModuleCode(`*`header`*`)` reads the code blocks that starts with *header*, from the templates of the adepter class including those of its all super classes. Hence, for a `currentTime` actor, `getSharedCode()` returns the above Real-Time Maude modules `Actor` and `CURRENT-TIME` (and adds modules for LTL model checking in the same way).

```
******** include basic definitions ********
load ptolemy-base.maude

******** semantics modules ********
(tomod ACTOR is ... endtom)
(tomod COMPOSITE-ACTOR is ... endtom)
(tomod ATOMIC-ACTOR is ... endtom)
(tomod CLOCK is ... endtom)
(tomod FSM-ACTOR is ... endtom)
(tomod SET-VARIABLE is ... endtom)
(tomod DELAY-ACTOR is ... endtom)

******** formal analysis modules ********
(tomod CHECK-ACTOR is ...  endtom)
(tomod CHECK-COMPOSITE-ACTOR is ... endtom)
(tomod CHECK-FSM-ACTOR is ...  endtom)

******** Initial model modules ********
(tomod INIT is
...
  op init : -> Configuration .
  eq init
   = < global EventQUEUE | queue : nil >
     init(< 'DE_SimpleTrafficLight : ConpositeActor |
                status : enabled,
                ports : none,
                innerActors : (
                  < 'Clock : Clock | ... >
                  < 'CarLightNormal : FSM-Actor | ... >
                  < 'PedestrianLightNormal : FSM-Actor | ... >
                  < 'TimedDelay : Delay | ... >
                  < 'TimedDelay2 : Delay | ... >
                  < 'SetVariable : SetVariable | ... >
                  ('Clock ! 'output) ==> ('PedestrianLightNormal ! 'Sec ;
                                          'CarLightNormal ! 'Sec)
                  ... ),
                parameters : < 'Pred : Parameter | exp : # 1, ... >
                             < 'Pgrn : Parameter | exp : # 0, ... >
                             < 'Cred : Parameter | exp : # 1, ... >
                             < 'Cyel : Parameter | exp : # 0, ... >
                             < 'Cgrn : Parameter | exp : # 0, ... >,
                computation : noComputation >) .
endtom)
(tomod PTOLEMY-MODELCHECK is
    including INIT + CHECK-ACTOR + CHECK-COMPOSITE-ACTOR + CHECK-FSM-ACTOR .
endtom)

******** verification commands ********
(mc {init} |=u [] ~ ('DE_SimpleTrafficLight | ('Pgrn = # 1, 'Cgrn = # 1)) .)
(mc {init} |=u 'DE_SimpleTrafficLight : (
                   []<>(this | 'Pgrn = #1, 'Cgrn = #0) /\
                   []<>(this | 'Pgrn = #0, 'Cgrn = #1)) .)
quit
```

Figure D.1: Dialog window for the Real Time Maude code generation

The function `generateFireCode()` is used to generate the Real-Time Maude term representing the (initial state of the) given Ptolemy II model. It generates the code from the code templates with header `fireBlock()` and `$attr` in the appropriate adapter classes; that is, a Real-Time Maude object corresponding to the initial state of the actor. For example, given a Ptolemy II `currentTime` actor with the name CT, the `generateFireCode()` function returns the term

```
< 'CT : CurrentTime |
    current-time : 0,
    computation : noComputation,
    status : enabled,
    ports : < 'output : OutPort | value : # 0, status : absent >
            < 'trigger : InPort | value : # 0, status : absent >,
    parameters : emptyMap >
```

The generated Real-Time Maude code consists of semantics modules, formal analysis modules, the initial state module, and verification commends, as illustrated in Figure D.1 for the simple traffic light system.

D.3 More Details on the Ptolemy II DE Semantics

D.3.1 Formal Definitions of Semantic functions

This section presents the formal definitions of several semantic functions for the Real-Time Maude semantics of Ptolemy II DE models in Section 8.3.2, including `add` (for event queues), `makeEnv` (for variable environments), and `clearPorts`, `update`, and the "inactive" cases for `portFixPoints`.

Adding Events to the Queue. Recall that each event is ordered by its tag (T, N) in the event queue, where $(T, N) \leq (T', N')$ iff $T < T'$ or $T = T'$ and $N \leq N'$. The function `add` inserts the new event in the correct place:

```
op add : Event Time Nat EventQueue ~> EventQueue .
eq add(EVENT, T, N, (EVTS ; T' ; N') :: QUEUE)
  = if T < T' or (T == T' and N < N') then      --- strictly smaller
       ((EVENT ; T ; N) :: (EVTS ; T' ; N') :: QUEUE)
    else (if T == T' and N == N' then            --- simultaneous
       (EVENT EVTS ; T ; N) :: QUEUE
    else                                         --- otherwise
       ((EVTS ; T' ; N') :: add(EVENT, T, N, QUEUE)) fi) fi
eq add(EVENT, T, nil) = (EVENT ; T ; 0) .
```

Creating Variable Environments. First of all, the syntax of variable environments, $x_1 \leftarrowtail v_1; \cdots ; x_n \leftarrowtail v_n$, is declared in Real-Time Maude:

```
sort EnvAssignment .
op _<-|_ : VarId Value -> EnvAssignment [ctor] .
op _<-? : VarId -> EnvAssignment [ctor] .

sort EnvMap .
subsort EnvAssignment < EnvMap .
op emptyEnv : -> EnvMap [ctor] .
op _;_ : EnvMap EnvMap
      -> EnvMap [ctor assoc comm id: emptyEnv prec 90] .
```

Then, the function `makeEnv` constructs a variable environment from given ports and parameters as follows:

```
op makeEnv : Configuration ~> ConfigItem .
op makeEnv : Configuration EnvMap ~> EnvMap .
eq makeEnv(REST) = env(makeEnv(REST, emptyEnv)) .
eq makeEnv(<R : Parameter | value : V > REST, ENV)
 = makeEnv(REST, ENV[R <-| V]) .
eq makeEnv(< P : InPort | status : PS, value : V > REST, ENV)
 = makeEnv(REST, ENV[if PS == present then P <-| V else P <-? fi]) .
eq makeEnv(REST, ENV) = ENV [owise] .
```

Initializing Ports. The `clearPorts` function clears all the ports of each actor in the state (that is, sets the `status` to `unknown`). For *composite* actors, it just propagates to the inner actors:

```
op clearPorts : Configuration ~> Configuration .
eq clearPorts(< O : AtomicActor | ports : PORTS > OBJS)
 = < O : AtomicActor | ports : clearPorts(PORTS) > clearPorts(OBJS) .
eq clearPorts(< O : CompositeActor |
                  innerActors : OBJS, ports : PORTS > OBJS')
 = < O : CompositeActor |
      innerActors : clearPorts(OBJS), ports : clearPorts(PORTS) >
   clearPorts(OBJS') .
eq clearPorts(< P : Port | > PORTS)
 = < P : Port | status : unknown > clearPorts(PORTS) .
eq clearPorts(OBJS) = OBJS [owise] .
```

The `releaseEvt` function generates the `active-evt` message for each event scheduled to fire.

```
op releaseEvt : Events ~> Configuration .
eq releaseEvt(EVENT EVTS) = active-evt(EVENT) releaseEvt(EVTS) .
eq releaseEvt(noEvent) = none .
```

Updating Parameters. The `update` function computes new parameter values for the next iteration. This function first computes the (possibly new) value of each parameter of an actor using its `next-value` attribute, and then replaces the old `value` with the new value in the `next-value`. Similarly, `update` distributes over the actor objects in the configuration:

```
eq update(NOBJS NOBJS') = update(NOBJS) update(NOBJS') .
eq update(OBJS) = OBJS [owise] .
```

For a composite actor, a variable in an expression of an inner actor may refer to a parameter of the composite actor. Such a parameter should *not* be updated until the parameters of every inner actor is updated. To ensure this, we define the auxiliary function `update`(B, *actor*) with an extra Boolean argument B. For an atomic actor, B is always set to `true`:

```
op update : Bool Configuration ~> Configuration .
eq update(< O : AtomicActor | status : enabled >)
 = update(true, < O : AtomicActor | >) .
```

However, for a composite actor, B is initially set to `false`, and becomes `true` after the `update` computation of every inner actor is completed:

```
eq update(< O : CompositeActor | status : enabled,
                                 innerActors : OBJS >)
 = update(false,
     < O : CompositeActor | innerActors : update(OBJS) >) .

eq update(false, < O : CompositeActor | innerActors : OBJS >)
 = update(true, < O : CompositeActor | >) .
```

Notice that these equations are applied for only *enabled* actors, since disabled actors should not change their states, including the parameters.

When B is `true`, if the `next-value` attribute of any parameter of an actor has not been computed yet (i.e., `noValue`), then the computation configuration for the parameter is created in the corresponding actor:

```
var R : ParamId .  var PARAMS : ObjectConfiguration . var E : Exp .
eq update(true,
      < O : Actor |
          computation : noComputation, ports : PORTS,
          parameters : < R : Parameter | next-value : noValue,
                                         exp : E > PARAMS >)
 = update(true,
      < O : Actor |
          computation : #param(R) / k(E) makeEnv(PARAMS PORTS) >) .
```

The resulting value of the expression, given by the rewriting-based formal semantics of the Ptolemy II expression language (in Section 8.3.4), is then stored in the `next-value` attribute by the following equation:

```
eq update(true,
      < O : Actor | parameters : < R : Parameter | > PARAMS,
                    computation : #param(R) / result(V) >)
 =
   update(true,
      < O : Actor |
           parameters : < R : Parameter | next-value : V > PARAMS,
           computation : noComputation >) .
```

If every `next-value` attribute is computed and thus the above equations cannot be applied any more, then the `write` function is applied for updating the values and clearing the `next-value` attribute of the parameters:

```
ceq update(true, OBJS) = write(OBJS)
 if noComputation(OBJS) [owise] .

op write : Configuration ~> Configuration .
eq write(< O : Actor | parameters : PARAMS > OBJS)
 = < O : Actor | parameters : write(PARAMS) >
   write(OBJS) .
eq write(< R : Parameter | value : V, next-value : V' > PARAMS)
 = < R : Parameter | value : V', next-value : noValue >
   write(PARAMS) .
eq write(none) = none .
```

More Details on `portFixPoints`. First of all, the function `filterMsg`, separating the events toward inside from the others, is defined as follows, where sort NEActorID denotes non-empty global actor identifiers:

```
var PORTS OBJS REST : ObjectConfiguration .
var CF : Configuration .     var NAI : NEActorID .

op filterMsg : Oid Configuration MsgConfiguration ~> FilterResult .
eq filterMsg(O, active-evt(event((O . NAI) ! P, V)) CF, MSGS)
 = filterMsg(O, CF, active-evt(event(NAI ! P, V)) MSGS) .
eq filterMsg(O, CF, MSGS) = fr(MSGS, CF) [owise] .
```

If all input ports of an actor are absent or the status of an actor is disabled, then the actor should not generate any output, unless it has a scheduled event from the global event queue. In these cases, the `status` of each output port of the actor is set to `absent`:

```
ceq portFixPoints(
     < O : Actor |
         ports : < P : OutPort | status : unknown > PORTS > REST)
  = portFixPoints(
     < O : Actor | ports : < P : OutPort | status : absent >
                           unknownOutPortsAbsent(PORTS) > REST)
  if allInputPortsAbsent(PORTS) .

eq portFixPoints(
    < O : Actor |
        status : disabled,
        ports : < PI : OutPort | status : unknown > PORTS > REST)
 = portFixPoints(
    < O : Actor |
        ports : < PI : OutPort | status : absent >
                setUnknownOutPortsAbsent(PORTS) > REST) .
```

The function `allInputPortsAbsent` returns `true` iff every input port has status `absent`, and the function `unknownOutPortsAbsent` returns a set of ports obtained by setting the status of each output port to `absent`:

```
op allInputPortsAbsent : Configuration -> Bool .
eq allInputPortsAbsent(< P : InPort | status : PS > PORTS)
 = (PS == absent) and allInputPortsAbsent(PORTS) .
eq allInputPortsAbsent(PORTS) = true [owise] .

op unknownOutPortsAbsent : Configuration ~> Configuration .
eq unknownOutPortsAbsent(< P : OutPort | status : unknown > PORTS)
 = < P : OutPort | status : absent > unknownOutPortsAbsent(PORTS) .
eq unknownOutPortsAbsent(PORTS) = PORTS [owise] .
```

It is also possible that some actor has an isolated input port that has no incoming connection. Obviously, the input port has no value, and therefore its `status` should be `absent`:

```
ceq portFixPoints(
     < O : Actor |
         ports : < P : InPort | status : unknown > PORTS > REST)
  =
   portFixPoints(
     < O : Actor |
         ports : < P : InPort | status : absent > PORTS > REST)
  if not connectedInPort(O ! P, REST) .
```

The function `connectedInPort(O ! P, REST)` returns `true` iff there is an incoming connection `O' ! P' ==> (O ! P) ; EPIS` to the port `O ! P`:

```
op connectedInPort : EPortId Configuration -> Bool .
eq connectedInPort(O ! P, (O' ! P' ==> (O ! P) ; EPIS)
                    < O' : Actor | status : enabled > REST) = true .
eq connectedInPort(O ! P, REST) = false [owise] .
```

If an output port of a composite actor has no incoming connections, it is then *isolated* and should be `absent`.

```
ceq portFixPoints(
      < O : CompositeActor | status : enabled,
         innerActors : OBJS,
         ports : < P : OutPort | status : unknown > PORTS > REST)
   = portFixPoints(
      < O : CompositeActor |
         ports : < P : OutPort | status : absent > PORTS > REST)
  if not connectedInPort(parent ! P, OBJS) .
```

If some output port of a composite actor is directly connected to its input port, the status (and the value if the status is present) of the input port is transferred to the output port after the inner fixed-point is finished:

```
ceq portFixPoints(
      < O : CompositeActor | status : enabled,
         innerActors : (parent ! P) ==> (parent ! P' ; EPIS) OBJS
         ports : < P : InPort | status : PS, value : V >
                < P' : OutPort | status : unknown > PORTS > REST)
= portFixPoints(
      < O : CompositeActor |
         ports : < P : InPort | >
                < P' : OutPort | status : PS, value : V > PORTS >
      REST) if PS =/= unknown .
```

All input and output ports of inner actors in *disabled* composite actors become `absent`, since there is no computation for disabled actors, where the `setAllPortsAbsent` function makes the status of every port `absent`:

```
eq portFixPoints(
      < O : CompositeActor | status : disabled,
         innerActors :
           < O' : Actor |
              ports : < P : Port | status : unknown > PORTS >
           OBJS > REST)
=
    portFixPoints(
      < O : CompositeActor |
         innerActors :
           setAllPortsAbsent(< O' : Actor | > OBJS) > REST) .
```

Property Specification Language Semantics. The semantics of the predefined propositions is defined by equations in a usual way. In particular, the proposition `_@_` for locations is defined by:

```
eq {< O : FSM-Actor | currState : L > CF} |= O @ L = true .
eq {< O : ModalModel | controller : CO,
                       innerActors : OBJS > CF} |= O @ L
 = {OBJS} |= CO @ L .
eq {< O : CompositeActor | innerActors : OBJS > CF} |= (O . AI) @ L
 = {OBJS} |= AI @ L .
eq {< O : Actor | > CF} |= O @ L = false [owise] .
```

D.3.2 The Semantics of Actors in DE Models

This section presents the DE semantic of additional actors in Real-Time Maude, not shown in Section 8.3.3. Recall that the syntax of each actor is specified by declaring a subclass of the class `Actor` in Real-Time Maude, and the semantics of the actor is specified by declaring three semantic functions for the actor: `init`, `portFixPoints`, and `postfire`.

More Details on FSM Actors. Some constructs and functions for the semantics of FSM actors in Section 8.3.3 are defined as follows. A location is the sort of the local "states" of the transition system:

```
sort Location .
subsorts Qid < Location .
```

In Real-Time Maude, a set of transitions are declared as follows:

```
sorts Transition TransBody .
op _:_-->_`{_`} : TransId Location Location TransBody
               -> Transition [ctor] .
op guard:_output:_set:_ : Exp ExpMap ExpMap -> TransBody [ctor] .

sort TransitionSet .
subsort Transition < TransitionSet .
op emptyTransitionSet : -> TransitionSet [ctor] .
op _;_ : TransitionSet TransitionSet
      -> TransitionSet [ctor assoc comm id: emptyTransitionSet] .
```

The function `noGuardTrue` is defined as follows, which returns *true* if there is no enabled transition:

```
eq noGuardExpTrue((TI <- # true, VREC)) = false .
eq noGuardExpTrue(VREC) = true [owise] .
```

The `updateOutPorts` function is defined as follows. Each output port is assigned an expression of the corresponding output action, and all remaining output ports are set to be absent in the end of the update process:

```
op updateOutPorts : ExpMap Configuration ~> Configuration .
eq updateOutPorts(VI |-> E ; OL,
     < VI : OutPort | status : unknown > PORTS)
 = updateOutPorts(OL,
     < VI : OutPort | status : present, value : E > PORTS) .
eq updateOutPorts(OL, PORTS)
 = setUnknownOutPortsAbsent(PORTS) [owise] .
```

The guard expressions of the transitions are computed again for the `postfire` function as follows, provided there exists at least one input to the actor (notice that the status of every output port is determined to be either `present` or `absent` during `portFixPoints`):

```
ceq postfire(< O : FSM-Actor |
                status : enabled,     parameters : PARAMS,
                currState : STATE,    transitions : TRANSSET,
                ports : < P : InPort | status : present > PORTS,
                computation : noComputation >)
 = postfire(< O : FSM-Actor | computation : #guards / k(E) EV >)
 if E := makeGuardExp(STATE, TRANSSET)
 /\ EV := makeEnv(PARAMS PORTS < P : InPort | status : present >) .
```

The `updateParam` function updates the `exp` attribute of each parameter. At the end of the current iteration, the `update` function will change the `value` according to the new `exp`:

```
op updateParam : ExpMap Configuration -> Configuration .
eq updateParam(VI |-> E ; AL, < VI : Parameter | > PARAMS)
 = < VI : Parameter | exp : E > updateParam(AL, PARAMS) .
eq updateParam(AL, PARAMS) = PARAMS [owise] .
```

Single Event. A *single event* actor produces a single event once according to its *time* and *value* parameters. Therefore, it only define the *init* action:

```
class SingleEvent .
subclass SingleEvent < AtomicActor .

eq init(< O : SingleEvent |
          parameters : < 'time : Parameter | value : V1 >
                       < 'value : Parameter | value : V2 > PARAMS >)
  = < O : SingleEvent | >
   schedule-evt(event(O ! 'output, V2), toTime(V1), 0) .
```

Pulse. A *pulse* actor has an extra attribute `index` that keeps track of the current iteration count of the actor, while the other parameters (*values* and *indexes*) are represented in the `parameters` attribute:

```
class Pulse | index : Nat .
subclass Pulse < AtomicActor .
```

When a *pulse* actor gets input through its *trigger* port, it should generate immediate output through its *output* port:

```
eq portFixPoints(
      < O : Pulse |
          parameters : < 'indexes : Parameter | value : V1 >
                       < 'values : Parameter | value : V2 > PARAMS,
          ports : < 'output : OutPort | status : unknown >
                  < 'trigger : InPort | status : present > PORTS,
          index : N, status : enabled > REST)
 = portFixPoints(
      < O : Pulse |
          ports : < 'output : OutPort | status : present,
                                        value : getValue(V1, V2, N) >
                  < 'trigger : InPort | > PORTS > REST) .
```

where the function `getValue(V1,V2,N)` gives an output value as described in Section 8.2.2 (that is, if $|\texttt{V1}| < N$, then `V2(V1(N))`, and otherwise 0).

Whenever a pulse actor produces an output, then the `postfire` function should increase the `index` attribute by 1:

```
eq postfire(< O : Pulse |
              ports : < 'trigger : InPort | status : present > PORTS,
              current-index : N, status : enabled >)
 = < O : Pulse | current-index : s(N)> .
```

Timed Delay. A timed delay actor propagates an incoming event after a delay specified by its parameter, and does not need any new attribute:

```
class Delay .
subclass Delay < AtomicActor .
```

Since a delay actor does not produce any output as a result of any input, every `unknown` output port should be set to `absent`:

```
eq portFixPoints(
      < O : Delay |
          ports : < P : OutPort | status : unknown > PORTS > REST)
 = portFixPoints(
      < O : Delay |
          ports : < P : OutPort | status : absent > PORTS > REST) .
```

If a *timed delay* actor has input in its **'input** port, then it generates an event with delay equal to the current value of the **'delay** parameter. If this delay is 0, then the microstep is 1, otherwise the microstep is 0:

```
eq postfire(
     < O : Delay |
         status : enabled,
         parameters : < 'delay : Parameter | value : TV > PARAMS,
         ports : < 'input : InPort | status : present, value : V >
                  < 'output : OutPort | > PORTS >)
 =
   < O : Delay | >
   schedule-evt(event(O ! 'output, V),
                toTime(TV),
                if toTime(TV) == 0 then 1 else 0 fi) .
```

Variable Delay. A *variable delay* actor has a *delay* port to specify time delay. If this port is **absent**, the behavior is the same as the delay actor.

```
class VariableDelay .
subclass VariableDelay < AtomicActor .
```

However, if the *delay* port receives some value, then the value of the port is used instead of the **'delay** parameter:

```
eq postfire(
     < O : VariableDelay |
         ports : < 'input : InPort | status : present, value : V >
                 < 'delay : InPort | status : present, value : TV >
                 < 'output : OutPort | > PORTS, status : enabled >)
 =
  < O : VariableDelay | >
  schedule-evt(event(O ! 'output, V), toTime(TV),
               if toTime(TV) == 0 then 1 else 0 fi) .
```

Timer. A *timer* actor is similar to a variable delay actor, but its output value is determined by the **output** parameter, instead of an input port:

```
class Timer .
subclass Timer < AtomicActor .
```

If a timer actor received input at its **input** port, it generates an event with value equal to the current value of the **output** parameter. The event is scheduled to fire in the time given by the value of the **input** port:

```
eq postfire(
     < O : Timer |
         parameters : < 'output : Parameter | value : V > PARAMS,
         ports : < 'input : InPort | status : present, value : TV >
                 PORTS, status : enabled >)
 =
   < O : Timer | >
    schedule-evt(event(O ! 'output, V),
                 toTime(TV),
                 if toTime(TV) == 0 then 1 else 0 fi) .
```

Noninterruptible Timer. A *noninterruptible timer* actor is similar to a timer, but needs some attributes to keep track of the state. The `processing` attribute is `true` when the timer has not finished processing previous inputs. The `waitQueue` attribute denotes a list that stores (the values of) the inputs received while the timer is *busy*:

```
class NonInterruptibleTimer | processing : Bool,
                              waitQueue : TimeList .
subclass NonInterruptibleTimer < AtomicActor .

sort TimeList .
subsort Time < TimeList .
op emptyList : -> TimeList [ctor] .
op __ : TimeList TimeList -> TimeList [ctor assoc id: emptyList] .
```

The `portFixPoints` function just sets every `unknown` output port to `absent`, and the `postFire` function generates actual events. If there is no scheduled output (i.e., the output port is `absent`) and no processing input (i.e., the `processing` attribute is `false`), then the behavior is the same as a normal timer, except that the `processing` attribute is set to `true`:

```
eq postfire(
    < O : NonInterruptibleTimer |
       ports : < 'input : InPort | status : present, value : TV >
               < 'output : OutPort | status : absent > PORTS,
       parameters : < 'value : Parameter | value : V > PARAMS,
       processing : false,  status : enabled >)
 =
    < O : NonInterruptibleTimer | processing : true >
    schedule-evt(event(O ! 'output, V),
                 toTime(TV),
                 if toTime(TV) == 0 then 1 else 0 fi) .
```

If there is no scheduled output but the actor is still processing previous inputs (that is, the `processing` attribute is `true`), then a received input is just added to the `waitQueue`:

```
eq postfire(
   < O : NonInterruptibleTimer |
      ports : < 'input : InPort | status : present, value : TV >
              < 'output : OutPort | status : absent > PORTS,
      parameters : < 'value : Parameter | value : V > PARAMS,
      processing : true,  waitQueue : TL,  status : enabled >)
 =
   < O : NonInterruptibleTimer | waitQueue : (TL toTime(TV)) > .
```

If a scheduled event has been arrived (i.e., the output port is `present`, and the `processing` attribute is `true`), the `waitQueue` is empty, and no new input has been received, then there will be no processing input in the next step; therefore, the `processing` attribute is set to `false`:

```
eq postfire(
    < O : NonInterruptibleTimer |
       ports : < 'input : InPort | status : absent, value : TV >
               < 'output : OutPort | status : present > PORTS,
       parameters : < 'value : Parameter | value : V > PARAMS,
       waitQueue : emptyList, status : enabled >)
  =
   < O : NonInterruptibleTimer | processing : false > .
```

For the other cases when a scheduled event has been arrived, any received input is added to the `waitQueue`, and the first event in the `waitQueue` is scheduled to fire, where the function `head` returns the first item in the queue and the function `tail` returns the rest of the queue except for the head:

```
ceq postfire(
    < O : NonInterruptibleTimer |
       ports : < 'input : InPort | status : PS, value : TV >
               < 'output : OutPort | status : present > PORTS,
       parameters : < 'value : Parameter | value : V > PARAMS,
       waitQueue : TL,  status : enabled >)
  =
   < O : NonInterruptibleTimer | waitQueue : tail(UQ) >
   schedule-evt(event(O ! 'output, V),
                head(UQ),
                if head(UQ) == 0 then 1 else 0 fi)
 if UQ := if PS == present then (TL toTime(TV)) else TL fi
 /\ UQ =/= emptyList .
```

Expression. An expression actor has an output port `output` and may have several input ports. It has also the additional attribute `expression` for an expression that defines the value of the output as a function of the values in the input ports of the actor:

```
class Expression | expression : Exp .
subclass Expression < AtomicActor .
```

The `portFixPoints` of expression actors are straightforward and very similar to the case for ports and parameters. If the output port is unknown, then the configuration for the expression is created, and the output port will finally have the evaluated value of the expression:

```
eq portFixPoints(
     < O : Expression |
         status : enabled,
         parameters : PARAMS,
         expression : E,
         ports : < 'output : OutPort | status : unknown > PORTS,
         computation : noComputation > REST)
 =
   portFixPoints(
     < O : Expression |
         computation : #port('output) / k(E) makeEnv(PORTS PARAMS) >
     REST) .

eq portFixPoints(< O : Expression |
                   ports : < 'output : OutPort | > PORTS,
                   computation : #port('output) / result(V) > REST)
 =
   portFixPoints(< O : Expression |
                   ports : < 'output : OutPort | status : present,
                                                 value : V > PORTS,
                   computation : noComputation > REST) .
```

Set Variable. A *set variable* actor contains a name of a parameter of the composite actor that contains the actor:

```
class SetVariable | variableName : ParamId .
subclass SetVariable < AtomicActor .
```

The `portFixPoints` outputs the value of the corresponding parameter. Since a variable name is a non-value expression, it just sets the `value` of the output port to the variable name `R`, which will be automatically evaluated to a value by another `portFixPoints` equation defined above:

```
eq portFixPoints(
     < O : SetVariable |
        ports : < 'input : InPort | status : present >
                < 'output : OutPort | status : unknown > PORTS,
        variableName : R,  status : enabled > REST)
 =
   portFixPoints(
     < O : SetVariable |
        ports : < 'input : InPort | >
                < 'output : OutPort |
                    status : present, value : R > PORTS > REST) .
```

The `postfire` function updates the value of the corresponding parameter in the composite actor, if a new value has been received in its input port:

```
eq postfire(
    < O : SetVariable |
        ports : < P : InPort | status : present, value : V > PORTS,
        variableName : R,  status : enabled >)
  = < O : SetVariable | > setv(R, V) .
```

It generates a `setv` message that is propagated towards its container actors to change the `exp` attribute of the parameter in the composite actor:

```
msg setv : ParamId Value -> Msg .

eq < O : CompositeActor |
       innerActors : CF setv(R, V),
       parameters : < R : Parameter | exp : E > PARAMS >
  = < O : CompositeActor |
       innerActors : CF,
       parameters : < R : Parameter | exp : V > PARAMS > .

ceq < O : CompositeActor | innerActors : CF setv(R,V),
                           parameters : PARAMS >
  = < O : CompositeActor | innerActors : CF > setv(R,V)
 if not R in PARAMS .
```

Timed Plotter. A *timed plotter* records its received data values and the times they were received. These values are recorded as a `++`-separated list

`(source:` s_1 `time:` t_1 `value:` v_1`) ++ ... ++ (source:` s_n `time:` t_n `value:` v_n`)`

of triples `(source:` s `time:` t `value:` v`)`, denoting, respectively, the port from which the data was received, the time it was received, and the received data value. The `TimedPlotter` class is a subclass of `TimeActor`:

```
class TimedPlotter | eventHistory : EventHistory .
subclass TimedPlotter < AtomicActor TimeActor .

sort EventTriple EventHistory .
subsort EventTriple < EventHistory .
op source:_time:_value:_ : EPortId Time Value -> EventTriple [ctor] .
op emptyHistory : -> EventHistory [ctor] .
op _++_ : EventHistory EventHistory
       -> EventHistory [ctor assoc id: emptyHistory] .
```

At the end of an iteration, the timed plotter records any input through its input port by adding a triple `source:` *`channel`* `time:` *`current time`* `value:` *`value of input`* for each input to its `eventHistory` attribute. This job is done by the auxiliary function `genHistory` which traverses its input ports and generates a "history triple" for those ports which had input:

```
eq postfire(< O : TimedPlotter | currentTime : T, status : enabled,
                                 eventHistory : EH, ports : PORTS >)
 = < O : TimedPlotter | eventHistory : EH ++ genHistory(T, PORTS) > .

op genHistory : Time Configuration ~> EventHistory .
eq genHistory(T, < 'input # (O ! P) : InPort | status : present,
                                               value : V > PORTS)
 = (source: O ! P time: T value: V) ++ genHistory(T, PORTS) .
eq genHistory(T, PORTS) = emptyHistory [owise] .
```

Modal Models. Modal models are represented as equivalent composite actors according to the frozen-composite-actor semantics for modal models described in Section 8.2. The class `ModalModel` has an additional attribute `controller` pointing to the controller FSM actor in `innerActors`, and the additional `refinementSet` attribute mapping each state in the modal model to its refinement:

```
class ModalModel | controller : Oid,
                   refinement : RefinementSet .
subclass ModalModel < CompositeActor .
```

Most of the semantics for modal models is borrowed from the semantics of composite actors, except for frozen actors, coupled ports, and the evaluation order between the controller and refinements. For modal models, `postfire` also sets the `status` attribute of the inner actors according to the current state of the controller to freeze all refinement actors except the refinement of the current state, where the function `setStateRefinement` disables all refinement actors except the refinement of the current state:

```
ceq postfire(
    < O : ModalModel | status : enabled, controller : CO,
                       refinement : REFS, innerActors : CF >)
 = < O : ModalModel |
       innerActors : < CO : FSM-Actor | >
                      setStateRefinement(STATE, REFS, OBJS) >
 if < CO : FSM-Actor | currStatus : STATE > OBJS := postfire(CF) .

op setStateRefinement : Location RefinementSet Configuration
                     -> Configuration .
eq setStateRefinement(STATE, refine-state(STATE', O) REFS,
                      < O : Actor | > REST)
 = < O : Actor | status : if STATE == STATE'
                           then enabled else disabled fi >
   setStateRefinement(STATE, REFS, REST) .
eq setStateRefinement(STATE, empty, REST) = REST .
```

If the controller actor depends on the result of `portFixPoints` of some refinement actors, then the result must be transferred through some coupled input port of the controller actor. Hence the evaluation order between the controller and refinements is automatically treated in our representation. The only part not yet covered is to handle coupled input/output ports in the controller FSM actor of a modal model. In our representation, the coupled input/output ports have the same name, and the value of the input port will be copied only if the coupled output port is *absent*:

```
eq portFixPoints(
      < O : ModalModel |
          status : enabled, controller : CO,
          innerActors :
              < CO : FSM-Actor |
                  ports : < P : InPort | status : present, value : V >
                          < P : OutPort | status : absent > PORTS,
                  status : enabled > OBJS >
      REST)
 =
   portFixPoints(
      < O : ModalModel |
          innerActors : portFixPoints(
              < CO : FSM-Actor |
                  ports : < P : InPort | >
                          < P : OutPort | status : present,
                                          value : V > PORTS > OBJS >)
      REST) .
```

This equation can be only applied after the inner fixed-point computation triggered by the controller FSM actor has been finished. Therefore, an output port copies a value from its coupled input port only if no value is generated at the output port when the controller is computed.

However, because of the above equation, the absent status of coupled output ports should not be transferred to the parent until we can decide whether the associated coupled input port is absent or not. For this reason we do not explicitly represent the connections between coupled output ports of the controller and the output ports of the parent modal model. Instead, the following equations propagate the value of the coupled output ports:

```
eq portFixPoints(
     < O : ModalModel |
         status : enabled, controller : CO,
         ports : < P : OutPort | status : unknown > PORTS,
         innerActors :
             < CO : FSM-Actor |
                 ports : < P : OutPort | status : present,
                                          value : V > PORTS2 > OBJS >
     REST)
 =
   portFixPoints(
     < O : ModalModel |
         ports : < P : OutPort | status : present,
                                 value : V > PORTS > REST) .
```

Similarly, the absent status of a coupled output port is propagated only if the associated input port is also absent:

```
eq portFixPoints(
     < O : ModalModel |
         status : enabled, controller : CO,
         ports : < P : OutPort | status : unknown > PORTS,
         innerActors :
             < CO : FSM-Actor |
                 ports : < P : InPort | status : absent >
                          < P : OutPort | status : absent > PORTS2 >
             OBJS >
     REST)
 =
   portFixPoints(
     < O : ModalModel |
         ports : < P : OutPort | status : absent > PORTS > REST) .
```

D.3.3 More Details on the Expression Language Semantics

This section shows more details on the syntax and the semantics of the Ptolemy II expression language in Section 8.3.4. Ptolemy II expressions consist of constants, algebraic operators, and variables. A constant can be a number, a Boolean value, or a string. Operators can be arithmetic (e.g., `+`, `-`, `*`, `/`, `^`, `%`), bitwise (e.g., `&`, `|`, `#`, `~`), logical (e.g., `&&`, `||`, `!`, `&`, `|`), shift (e.g., `<<`, `>>`, `>>>`), or conditional (*condition* ? exp_1 : exp_2). Variables are references to parameters or ports of actors (and may refer to parameters of composite actors that contain the actors).

The Ptolemy II expression language supports composite data types such as arrays, records, and matrices. As explained in Section 8.3.4, arrays are lists of expressions in curly brackets, and records are lists of fields in which each field is a pair of a name and a value. A matrix data structure describes a usual $n \times m$ matrix. Matrices are specified with square brackets, using commas to separate row elements and semicolons to separate rows, e.g., the expression `[1, 2 ; 3, 4]` represents the matrix $\left(\begin{smallmatrix}1 & 2\\ 3 & 4\end{smallmatrix}\right)$.

Algebraic Semantics. Ptolemy II expressions are terms of sort Exp as explained in Section 8.3.4. *Values* are expressions that cannot be further evaluated, and are represented as terms of sort Value, a subsort of Exp. Variables are terms of sort VarId in our semantics. Constants have sort Value, and are represented by values in Real-Time Maude, prefixed with the `#` symbol. Numerical constants are either rational numbers (including the integers) or fixed-point constants. The constants `Infinity` and `NaN` are special types of numbers to denote an infinite number and a numerical error such as 1.0 / 0.0, respectively.

```
op #_ : Bool -> Value [ctor] .
op #_ : String -> Value [ctor] .
op #_ : Rat -> Value [ctor] .
op #_ : Float -> Value [ctor] .
ops NaN Infinity : -> Value [ctor] .
```

Besides algebraic operators shown in Section 8.3.4, array and method call operators are defined as follows, where sort ExpList denotes a comma-separated list of expressions (with the empty expression list `()`):

```
op {_} : ExpList -> Exp .                                --- array
op _._(_) : Exp VarId ExpList -> Exp [prec 12] .       --- method call
```

For the algebraic semantics of data structure, the semantics of array and record expressions is defined as follows:

```
var N : Nat .   vars E E' : Exp .
var EL : ExpList .   var ER : ExpRow

--- an array expression A(N) that returns the (N-1)-th element
eq {E, EL}(# 0) = E .
eq {E, EL}(# s(N)) = {EL}(# N) .

--- a method call A.length() that returns the length of an array A
eq {()} . 'length(()) = # 0 .
eq {E, EL} . 'length(()) = # 1 + ({EL} . 'length(())) .

--- a record expression R.I() that returns the value of the field I
eq {(I <- E), ER} . I(()) = E .
```

The algebraic semantics defines the meaning of expressions that contain no variables and no user-defined functions.

Other Variable Expressions. When a variable in an expression refers to a parameter higher in the actor hierarchy, this hierarchical scope is handled using messages in a similar way to the event handling in composite actors. If a variable is not available in the current variable environment, then a query is sent to the parent actor by a message `query-var`:

```
ceq < O : Actor | computation : CI / k(I -> K) env(ENV) >
 =
    query-var(O, I)
    < O : Actor | computation : CI / k(request(I) -> K) env(ENV) >
  if not I in ENV .
```

If the variable is not available in the current composite actor, then the message is passed to its parent:

```
ceq < O : CompositeActor | parameters : PARAMS,
                           innerActors : query-var(AI, I) KCF >
 = query-var(O . AI, I)
   < O : CompositeActor | innerActors : KCF >
  if not I in PARAMS .
```

Otherwise, the corresponding value in a composite actor is returned by using another message `return-var` as follows:

```
eq < O : CompositeActor |
       parameters : < I : Parameter | value : V > PARAMS,
       innerActors : query-var(AI, I) KCF >
 =
   < O : CompositeActor |
       innerActors : return-var(AI, I, V) KCF > .
```

The returned value is delivered into the corresponding inner actor, and plugged back into the computation configuration:

```
eq return-var(O . O' . AI, I, V)
   < O : CompositeActor | innerActors : KCF >
 = < O : CompositeActor |
       innerActors : return-var(O' . AI, I, V) KCF > .

eq return-var(O, I, V)
    < O : Actor | computation : CI / k(request(I) -> K) env(ENV) >
 = < O : Actor | computation : CI / k(V -> K) env(ENV[I <-| V]) > .
```

During `portFixPoints`, `postfire`, and `update`, such messages can freely move between different hierarchies:

```
eq portFixPoints(query-var(AI, I) KCF)
 = query-var(AI, I) portFixPoints(KCF) .
eq return-var(AI, I, V) portFixPoints(KCF)
 = portFixPoints(return-var(AI, I, V) KCF) .

eq postfire(query-var(AI, I) KCF)
 = query-var(AI, I) postfire(KCF) .
eq return-var(AI, I, V) postfire(KCF)
 = postfire(return-var(AI, I, V) KCF) .

eq update(B, query-var(AI, I) KCF)
 = query-var(AI, I) update(B, KCF) .
eq return-var(AI, I, V) update(B, KCF)
 = update(B, return-var(AI, I, V) KCF) .
```

Note that the variable `KCF` in the above equations is defined at the *kind* level so that those equations can be applied when `portFixPoints`, `postfire`, and `update` is executed further down in the hierarchy.

Actor-specific Expressions. There are also *actor-specific* expressions that are only meaningful under a certain type of actors. For example, the expression P`_isPresent` for a FSM actor, expressed as `isPresent(`P`)` in our representation, returns `true` iff the status of an input port P of the actor is determined to be *present*:

```
ceq < O : FSM-Actor |
      ports : < P : Port | status : PS > PORTS,
      computation : CI / k(isPresent(P) -> K) env(ENV) >
 = < O : FSM-Actor |
      computation : CI / k(#(PS == present) -> K) env(ENV) > .
 if PS =/= unknown .
```

Functional Expressions. In Ptolemy II, a user-defined function is also considered as a value, called a *closure*. The closure of a function definition `function`$(i_1, \ldots, i_n)$ E is the triple `closure`$((i_1, \ldots, i_n), E, \mathit{env})$ together with the variable environment *env*. Whenever a function is invoked, its value is computed using the closure environment *env* and argument values, and the old environment is restored after the function computation is completed:

```
var IL : VarIdList .          var VL : ValueList .

op closure : VarIdList Exp EnvMap -> Value [ctor] .

eq k(function(IL) E -> K) env(ENV)
 = k(closure(IL,E,ENV) -> K) env(ENV) .

eq k(closure(IL,E,ENV)(VL) -> K) env(ENV')
 = k(E -> restore(ENV') -> K) env(ENV[IL <-| VL]) .
eq k(V -> restore(ENV') -> K) env(ENV) = k(V -> K) env(ENV') .
```

Other Structural Equations. The heating and cooling equations for data structures can be similarly defined by identifying an arbitrary non-value item. For example, the heating equation for arrays picks a non-value item *PE* from an array $\{E_1, \ldots, E_m, PE, E_{m+2}, \ldots, E_n\}$, and constructs the computation $PE \curvearrowright \{E_1, \ldots, E_m \;\square\; E_{m+2}, \ldots, E_n\}$, and the cooling equation reduces $V \curvearrowright \{E_1, \ldots, E_m \;\square\; E_{m+2}, \ldots, E_n\}$ with a value V to the array $\{E_1, \ldots, E_m, V, E_{m+2}, \ldots, E_n\}$. The heating/cooling equations for records and matrices are defined in a similar way:

```
vars EL EL' : ExpList .  var ER : ExpRow .  vars EM EM' : Matrix .

--- arrays
eq k(  {EL, PE, EL'} -> K) = k(PE -> {EL [] EL'} -> K) .
eq k(V -> {EL [] EL'} -> K) = k(    {EL, V, EL'} -> K) .

--- records
eq k(     {I <- PE, ER} -> K)  = k(PE -> {I <-[] ER} -> K) .
eq k(V ->   {I <-[] ER} -> K)  = k(     {I <- V, ER} -> K) .

--- matrices
eq k(     [EM ; EL, P, EL' ; EM'] -> K)
 = k(P -> [EM ; EL [] EL' ; EM'] -> K) .
eq k(V -> [EM ; EL [] EL' ; EM'] -> K)
 = k(     [EM ; EL, V, EL' ; EM'] -> K) .
```

Operations for data structures have the form of either a function call or a method call. For example, the function call $A(n)$ returns the n-th element of an array A, and the method call $A.\mathit{length}()$ returns the length of A. Similarly, for a record R, the method call $R.a()$ returns the value of the field a. Their heating/cooling equations are similarly defined in the obvious way:

```
--- function calls
eq k(     PE(EL) -> K)  = k(PE -> [](EL) -> K) .
eq k(V -> [](EL) -> K)  = k(       V(EL) -> K) .

eq k(    E(EL, PE, EL') -> K) = k(PE -> E(EL [] EL') -> K) .
eq k(V -> E(EL [] EL') -> K)  = k(      E(EL, V, EL') -> K) .

--- method calls
eq k(    PE . I(EL) -> K) = k(PE -> []. I(EL) -> K) .
eq k(V -> []. I(EL) -> K) = k(      V . I(EL) -> K) .

eq k(   E . I(EL, PE, EL') -> K) = k(PE -> E . I(EL [] EL') -> K) .
eq k(V -> E . I(EL [] EL') -> K) = k(     E . I(EL, V, EL') -> K) .
```

Example D.1. *The conditional expression* $x < A.\mathit{length}()\ ?\ A(x) : A(0)$ *with the environment* $x \mapsfrom 0\,; A \mapsfrom \{1\}$ *can generate the rewrite sequence:*

$$
\begin{array}{lrl}
 & k(x < A.\mathit{length}()\ ?\ A(x) : A(0)) & \mathit{env}(x \mapsfrom 0\,; A \mapsfrom \{1\}) \\
\longrightarrow^* & k(x \curvearrowright (\Box < A.\mathit{length}()) \curvearrowright \Box\ ?\ A(x) : A(0)) & \mathit{env}(x \mapsfrom 0\,; A \mapsfrom \{1\}) \\
\longrightarrow & k(0 \curvearrowright (\Box < A.\mathit{length}()) \curvearrowright \Box\ ?\ A(x) : A(0)) & \mathit{env}(x \mapsfrom 0\,; A \mapsfrom \{1\}) \\
\longrightarrow & k(0 < A.\mathit{length}() \curvearrowright \Box\ ?\ A(x) : A(0)) & \mathit{env}(x \mapsfrom 0\,; A \mapsfrom \{1\}) \\
\longrightarrow^* & k(A \curvearrowright \Box.\mathit{length}() \curvearrowright (0 < \Box) \curvearrowright \Box\ ?\ A(x) : A(0)) & \mathit{env}(x \mapsfrom 0\,; A \mapsfrom \{1\}) \\
\longrightarrow & k(\{1\} \curvearrowright \Box.\mathit{length}() \curvearrowright (0 < \Box) \curvearrowright \Box\ ?\ A(x) : A(0)) & \mathit{env}(x \mapsfrom 0\,; A \mapsfrom \{1\}) \\
\longrightarrow & k(\{1\}.\mathit{length}() \curvearrowright (0 < \Box) \curvearrowright \Box\ ?\ A(x) : A(0)) & \mathit{env}(x \mapsfrom 0\,; A \mapsfrom \{1\}) \\
\longrightarrow & k(1 \curvearrowright (0 < \Box) \curvearrowright \Box\ ?\ A(x) : A(0)) & \mathit{env}(x \mapsfrom 0\,; A \mapsfrom \{1\}) \\
\longrightarrow & k(0 < 1 \curvearrowright \Box\ ?\ A(x) : A(0)) & \mathit{env}(x \mapsfrom 0\,; A \mapsfrom \{1\}) \\
\longrightarrow & k(\mathit{true} \curvearrowright \Box\ ?\ A(x) : A(0)) & \mathit{env}(x \mapsfrom 0\,; A \mapsfrom \{1\}) \\
\longrightarrow & k(\mathit{true}\ ?\ A(x) : A(0)) & \mathit{env}(x \mapsfrom 0\,; A \mapsfrom \{1\}) \\
\longrightarrow & k(A(x)) & \mathit{env}(x \mapsfrom 0\,; A \mapsfrom \{1\}) \\
\longrightarrow & k(A \curvearrowright \Box(x)) & \mathit{env}(x \mapsfrom 0\,; A \mapsfrom \{1\}) \\
\longrightarrow & k(\{1\} \curvearrowright \Box(x)) & \mathit{env}(x \mapsfrom 0\,; A \mapsfrom \{1\}) \\
\longrightarrow & k(\{1\}(x)) & \mathit{env}(x \mapsfrom 0\,; A \mapsfrom \{1\}) \\
\longrightarrow & k(x \curvearrowright \{1\}(\Box)) & \mathit{env}(x \mapsfrom 0\,; A \mapsfrom \{1\}) \\
\longrightarrow & k(0 \curvearrowright \{1\}(\Box)) & \mathit{env}(x \mapsfrom 0\,; A \mapsfrom \{1\}) \\
\longrightarrow^* & k(1) & \mathit{env}(x \mapsfrom 0\,; A \mapsfrom \{1\}) \\
\longrightarrow & & \mathit{result}(1)
\end{array}
$$

REFERENCES

[1] P. Abdulla, A. Annichini, and A. Bouajjani. Symbolic verification of lossy channel systems: Application to the bounded retransmission protocol. In *TACAS*, volume 1579 of *LNCS*, pages 208–222. Springer, 1999.

[2] P. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular tree model checking. In *CAV*, volume 2404 of *LNCS*, pages 555–568. Springer, 2002.

[3] P. A. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *LICS*, pages 313–321. IEEE, 1996.

[4] P. A. Abdulla, Y.-F. Chen, G. Delzanno, F. Haziza, C.-D. Hong, and A. Rezine. Constrained monotonic abstraction: A CEGAR for parameterized verification. In *CONCUR*, volume 6269 of *LNCS*, pages 86–101. Springer, 2010.

[5] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[6] A. Al-Nayeem, L. Sha, D. D. Cofer, and S. M. Miller. Pattern-based composition and analysis of virtually synchronized real-time distributed systems. In *ICCPS*, pages 65–74. IEEE, 2012.

[7] A. Al-Nayeem, M. Sun, X. Qiu, L. Sha, S. P. Miller, and D. D. Cofer. A formal architecture pattern for real-time distributed systems. In *RTSS*, pages 161–170. IEEE, 2009.

[8] J. Anderson. *Introduction to flight*. McGraw-Hill, 2005.

[9] J. Avenhaus and C. Loría-Sáenz. On conditional rewrite systems with extra variables and deterministic logic programs. In *LPAR*, volume 822 of *LNCS*, pages 215–229. Springer, 1994.

[10] B. Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4), 1985.

[11] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[12] F. Baader and W. Snyder. Unification theory. *Handbook of automated reasoning*, 1:445–532, 2001.

[13] K. Bae. Source code for a Multirate PALS framework in Real-Time Maude. `http://hdl.handle.net/2142/49977`, 2014.

[14] K. Bae, S. Escobar, and J. Meseguer. Abstract logical model checking of infinite-state systems using narrowing. In *RTA*, volume 21 of *LIPIcs*, pages 81–96. Schloss Dagstuhl, 2013.

[15] K. Bae, J. Krisiloff, J. Meseguer, and P. C. Ölveczky. PALS-based analysis of an airplane multirate control system in Real-Time Maude. In *FTSCS*, pages 5–21, 2012.

[16] K. Bae, J. Krisiloff, J. Meseguer, and P. C. Ölveczky. Designing and verifying distributed cyber-physical systems using Multirate PALS: An airplane turning control system case study. *Science of Computer Programming*, 2014. To appear.

[17] K. Bae and J. Meseguer. The Linear Temporal Logic of Rewriting Maude Model Checker. In *WRLA*, volume 6381 of *LNCS*, pages 208–225. Springer, 2010.

[18] K. Bae and J. Meseguer. State/event-based LTL model checking under parametric generalized fairness. In *CAV*, volume 6806 of *LNCS*, pages 132–148. Springer, 2011.

[19] K. Bae and J. Meseguer. Model checking LTLR formulas under localized fairness. In *WRLA*, volume 7571 of *LNCS*, pages 99–117. Springer, 2012.

[20] K. Bae and J. Meseguer. A rewriting-based model checker for the temporal logic of rewriting. *Electronic Notes in Theoretical Computer Science*, 290:19–36, 2012. Proc. RULE'2008.

[21] K. Bae and J. Meseguer. Infinite-state model checking of LTLR formulas using narrowing. In *WRLA*, LNCS. Springer, 2014. To Appear.

[22] K. Bae and J. Meseguer. Model checking linear temporal logic of rewriting formulas under localized fairness. *Science of Computer Programming*, 2014. To appear. `http://dx.doi.org/10.1016/j.scico.2014.02.006`.

[23] K. Bae and J. Meseguer. Predicate abstraction of rewrite theories. In *RTA-TLCA*, volume 8560 of *LNCS*, pages 61–76. Springer, 2014.

[24] K. Bae, J. Meseguer, and P. C. Ölveczky. Formal patterns for multi-rate distributed real-time systems. In *FACS*, volume 7684 of *LNCS*, pages 1–18. Springer, 2012.

[25] K. Bae, J. Meseguer, and P. C. Ölveczky. Formal patterns for multirate distributed real-time systems. *Science of Computer Programming*, 91, Part A:3 – 44, 2014.

[26] K. Bae, P. Ölveczky, and J. Meseguer. Definition, semantics, and analysis of multirate synchronous aadl. In *FM*, volume 8442 of *LNCS*, pages 94–109. Springer, 2014.

[27] K. Bae and P. C. Ölveczky. Extending the Real-Time Maude semantics of Ptolemy to hierarchical DE models. In *RTRTS*, volume 36 of *EPTCS*, pages 46–66, 2010.

[28] K. Bae, P. C. Ölveczky, A. Al-Nayeem, and J. Meseguer. Synchronous AADL and its formal analysis in Real-Time Maude. In *ICFEM*, volume 6991 of *LNCS*. Springer, 2011.

[29] K. Bae, P. C. Ölveczky, T. H. Feng, E. A. Lee, and S. Tripakis. Verifying hierarchical Ptolemy II discrete-event models using Real-Time Maude. *Science of Computer Programming*, 77(12):1235–1271, 2012.

[30] K. Bae, P. C. Ölveczky, T. H. Feng, and S. Tripakis. Verifying Ptolemy II discrete-event models using Real-Time Maude. In *ICFEM*, volume 5885 of *LNCS*, pages 717–736. Springer, 2009.

[31] K. Bae, P. C. Ölveczky, J. Meseguer, and A. Al-Nayeem. The SynchAADL2Maude tool. In *FASE*, volume 7212 of *LNCS*, pages 59–62. Springer, 2012.

[32] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2007.

[33] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. *ACM SIGPLAN Notices*, 36(5):203–213, 2001.

[34] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.

[35] A. Benveniste, P. Caspi, R. Lublinerman, and S. Tripakis. Actors without Directors: a Kahnian View of Heterogeneous Systems. In *HSCC*, volume 5469 of *LNCS*, pages 46–60. Springer, 2009.

[36] B. Berthomieu, J.-P. Bodeveix, C. Chaudet, S. Dal Zilio, M. Filali, and F. Vernadat. Formal verification of AADL specifications in the Topcased environment. In *Reliable Software Technologies*, volume 5570 of *LNCS*, pages 207–221. Springer, 2009.

[37] M. Bezem, J. W. Klop, and R. de Vrijer. *Term rewriting systems*. Cambridge University Press, 2003.

[38] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[39] P. Borovanskỳ, C. Kirchner, H. Kirchner, and P. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285(2):155–185, 2002.

[40] A. Bouajjani and J. Esparza. Rewriting models of boolean programs. In *RTA*, volume 4098 of *LNCS*, pages 136–150. Springer, 2006.

[41] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *CAV*, volume 1855 of *LNCS*, pages 403–418. Springer, 2000.

[42] A. Bouajjani and T. Touili. Widening techniques for regular tree model checking. *International Journal on Software Tools for Technology Transfer*, 14(2):145–165, 2012.

[43] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Nguyen, T. Noll, M. Roveri, and R. Wimmer. A model checker for AADL. In *CAV*, volume 6174 of *LNCS*, pages 562–565. Springer, 2010.

[44] C. Brooks, C. Cheng, T. H. Feng, E. A. Lee, and R. v. Hanxleden. Model engineering using multimodeling. In *International Workshop on Model Co-Evolution and Consistency Management*, 2008.

[45] R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1-3):386–414, 2006.

[46] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In *CAV*, volume 1254 of *LNCS*, pages 400–411. Springer, 1997.

[47] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures. In *Handbook of Process algebra*, pages 545–623. Elsevier, 2001.

[48] A. Cataldo, E. Lee, X. Liu, E. Matsikoudis, and H. Zheng. A constructive fixed-point theorem and the feedback semantics of timed systems. In *International Workshop on Discrete-Event Systems*, pages 27–32. IEEE, 2006.

[49] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/event-based software model checking. In *IFM*, volume 2999 of *LNCS*, pages 128–147. Springer, 2004.

[50] S. Chaki, E. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. Concurrent software verification with states, events, and deadlocks. *Formal Aspects of Computing*, 17:461–483, 2005.

[51] C. P. Cheng, T. Fristoe, and E. A. Lee. Applied verification: The Ptolemy approach. Technical Report UCB/EECS-2008-41, EECS Department, University of California, Berkeley, 2008.

[52] M. Y. Chkouri, A. Robert, M. Bozga, and J. Sifakis. Translating AADL into BIP - application to the verification of real-time systems. In *MoDELS Workshops*, volume 5421 of *LNCS*, pages 5–19. Springer, 2008.

[53] A. Cholewa, J. Meseguer, and S. Escobar. Variants of variants and the finite variant property. Technical report, University of Illinois at Urbana-Champaign, 2014. `http://hdl.handle.net/2142/47117`.

[54] E. Clarke, A. Fehnker, Z. Han, B. Krogh, J. Ouaknine, O. Stursberg, and M. Theobald. Abstraction and counterexample-guided refinement in model checking of hybrid systems. *International Journal of Foundations of Computer Science*, 14(04):583–604, 2003.

[55] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[56] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[57] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS*, volume 3440 of *LNCS*, pages 570–574. Springer, 2005.

[58] E. Clarke, M. Talupur, and H. Veith. Environment abstraction for parameterized verification. In *VMCAI*, volume 3855 of *LNCS*, pages 126–141. Springer, 2006.

[59] E. M. Clarke. The birth of model checking. In *25 Years of Model Checking*, volume 5000 of *LNCS*, pages 1–26. Springer, 2008.

[60] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2001.

[61] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude – A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.

[62] M. Clavel, F. Durán, J. Hendrix, S. Lucas, J. Meseguer, and P. Ölveczky. The maude formal tool environment. In *CALCO*, volume 4624 of *LNCS*, pages 173–178. Springer, 2007.

[63] R. P. G. Collinson. *Introduction to avionics*. Chapman & Hall, 1996.

[64] H. Comon-Lundh and S. Delaune. The finite variant property: How to get rid of some algebraic properties. In *RTA*, volume 3467 of *LNCS*, pages 294–307. Springer, 2005.

[65] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of logic and computation*, 2(4):511–547, 1992.

[66] J. Couvreur, A. Duret-Lutz, and D. Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In *SPIN*, volume 3639 of *LNCS*, pages 169–184. Springer, 2005.

[67] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19:253–291, 1997.

[68] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *CAV*, volume 1633 of *LNCS*, pages 160–171. Springer, 1999.

[69] G. Delzanno and A. Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, 2001.

[70] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 243–320. North-Holland, 1990.

[71] R. Diaconescu and K. Futatsugi. *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification.* World Scientific Pub Co Inc, 1998.

[72] F. Durán, S. Eker, S. Escobar, J. Meseguer, and C. Talcott. Variants, unification, narrowing, and symbolic reachability in Maude 2.6. In *RTA*, volume 10 of *LIPIcs*, pages 31–40. Schloss Dagstuhl, 2011.

[73] F. Durán and J. Meseguer. Maude's module algebra. *Science of Computer Programming*, 66:125–153, 2007.

[74] F. Durán and J. Meseguer. A Church-Rosser checker tool for conditional order-sorted equational Maude specifications. In *WRLA*, volume 6381 of *LNCS*, pages 69–85. Springer, 2010.

[75] F. Durán and J. Meseguer. A maude coherence checker tool for conditional order-sorted rewrite theories. In *WRLA*, volume 6381 of *LNCS*, pages 86–103. Springer, 2010.

[76] A. Duret-Lutz, D. Poitrenaud, and J.-M. Couvreur. On-the-fly emptiness check of transition-based Streett automata. In *ATVA*, volume 5799 of *LNCS*. Springer, 2009.

[77] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Păsăreanu, H. Zheng, and W. Visser. Tool-supported program abstraction for finite-state verification. In *ICSE*, pages 177–187. IEEE, 2001.

[78] S. Edwards and E. A. Lee. The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, 48:21–42(22), July 2003.

[79] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(2):127–144, 2003.

[80] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker and its implementation. In *SPIN*, volume 2648 of *LNCS*, pages 230–234. Springer, 2003.

[81] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. *Electronic Notes in Theoretical Computer Science*, 71:162–187, 2004.

[82] C. Ellison and G. Roşu. An executable formal semantics of C with applications. In *POPL*, pages 533–544. ACM, 2012.

[83] E. A. Emerson and C. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, 1987.

[84] E. A. Emerson and K. S. Namjoshi. On model checking for non-deterministic infinite-state systems. In *LICS*, pages 70–80. IEEE, 1998.

[85] S. Escobar, C. Meadows, and J. Meseguer. Maude-NPA: cryptographic protocol analysis modulo equational properties. In *Foundations of Security Analysis and Design V*, volume 5705 of *LNCS*, pages 1–50. Springer, 2009.

[86] S. Escobar and J. Meseguer. Symbolic model checking of infinite-state systems using narrowing. In *RTA*, volume 4533 of *LNCS*, pages 153–168, 2007.

[87] S. Escobar, R. Sasse, and J. Meseguer. Folding variant narrowing and optimal variant termination. *Journal of Logic and Algebraic Programming*, 81:898–928, 2012.

[88] A. Farzan, F. Chen, J. Meseguer, and G. Rosu. Formal analysis of Java programs in JavaFAN. In *CAV*, volume 3114 of *LNCS*, pages 501–505. Springer, 2004.

[89] A. Farzan and J. Meseguer. State space reduction of rewrite theories using invisible transitions. In *AMAST*, volume 4019 of *LNCS*, pages 142–157. Springer, 2006.

[90] A. Farzan and J. Meseguer. Partial order reduction for rewriting semantics of programming languages. *Electronic Notes in Theoretical Computer Science*, 176(4):61–78, 2007.

[91] P. H. Feiler and D. P. Gluch. *Model-Based Engineering with AADL*. Addison-Wesley, 2012.

[92] M. Fernández. AC complement problems: Satisfiability and negation elimination. *Journal of Symbolic Computation*, 22(1):49–82, 1996.

[93] M. Fernández. Negation elimination in empty or permutative theories. *Journal of Symbolic Computation*, 26(1):97–133, 1998.

[94] M. Filali and J. Lawall. Development of a synchronous subset of AADL. In *Proc. ASM'10*, volume 5977 of *LNCS*. Springer, 2010.

[95] A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1-2):63–92, 2001.

[96] G. S. Fishman. *Discrete-Event Simulation: Modeling, Programming, and Analysis*. Springer, 2001.

[97] R. França, J.-P. Bodeveix, M. Filali, J.-F. Rolland, D. Chemouil, and D. Thomas. The AADL behaviour annex - experiments and roadmap. In *ICECCS*, pages 377–382. IEEE, 2007.

[98] N. Francez. *Fairness*. Springer, 1986.

[99] T. Genet and V. Rusu. Equational approximations for tree automata completion. *Journal of Symbolic Computation*, 45(5):574–597, 2010.

[100] T. Genet and V. Tong. Reachability analysis of term rewriting systems with timbuk. In *LPAR*, volume 2250 of *LNCS*, pages 695–706. Springer, 2001.

[101] A. Girault and C. Ménier. Automatic production of globally asynchronous locally synchronous systems. In *EMSOFT*, volume 2491 of *LNCS*, pages 266–281. Springer, 2002.

[102] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[103] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–172, 1985.

[104] M. R. Henzinger and J. A. Telle. Faster algorithms for the nonemptiness of Streett automata and for communication protocol pruning. In *SWAT*, volume 1097 of *LNCS*, pages 16–27. Springer, 1996.

[105] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *ACM SIGPLAN Notices*, volume 37, pages 58–70. ACM, 2002.

[106] T. Henzinger and J. Sifakis. The discipline of embedded systems design. *IEEE*, 40(10):32–40, 2007.

[107] T. A. Henzinger. Two challenges in embedded systems design: predictability and robustness. *Philosophical Transactions of the Royal Society A*, 366(1881):3727–3736, 2008.

[108] G. Holzmann. *The SPIN model checker: Primer and reference manual*. Addison Wesley Publishing Company, 2004.

[109] G. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *SPIN*, pages 23–32. American Mathematical Society, 1996.

[110] J. M. Hullot. Canonical forms and unification. In *CADE*, volume 87 of *LNCS*. Springer, 1980.

[111] E. Jahier, N. Halbwachs, P. Raymond, X. Nicollin, and D. Lesens. Virtual execution of AADL models via a translation into synchronous programs. In *EMSOFT*, pages 134–143. ACM, 2007.

[112] J. P. Jouannaud, C. Kirchner, and H. Kirchner. Incremental construction of unification algorithms in equational theories. In *ICALP*, volume 154 of *LNCS*. Springer, 1983.

[113] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

[114] Y. Kesten and A. Pnueli. Control and data abstraction: The cornerstones of practical formal verification. *International Journal on Software Tools for Technology Transfer*, 2(4):328–342, 2000.

[115] Y. Kesten, A. Pnueli, L. Raviv, and E. Shahar. Model checking with strong fairness. *Formal Methods in System Design*, 28(1):57–84, 2006.

[116] H. Kopetz and G. Grünsteidl. TTP - a protocol for fault-tolerant real-time systems. *Computer*, 27(1):14–23, 1994.

[117] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 2002.

[118] O. Kupferman and M. Vardi. Model checking of safety properties. *Formal Methods in System Design*, 19(3):291–314, 2001.

[119] A. L. Lafuente, J. Meseguer, and A. Vandin. State space c-reductions of concurrent systems in rewriting logic. In *ICFEM*, volume 7635 of *LNCS*, pages 430–446. Springer, 2012.

[120] S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In *CAV*, volume 2725 of *LNCS*, pages 141–153. Springer, 2003.

[121] J.-L. Lassez and K. Marriott. Explicit representation of terms defined by counter examples. *Journal of Automated Reasoning*, 3(3):301–317, 1987.

[122] T. Latvala. Model checking LTL properties of high-level Petri nets with fairness constraints. In *ICATPN*, volume 2075 of *LNCS*, pages 242–262. Springer, 2001.

[123] E. A. Lee. Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, 7(1-4):25–45, 1999.

[124] E. A. Lee and A. Sangiovanni-Vincentelli. A unified framework for comparing models of computation. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, 1998.

[125] E. A. Lee and H. Zheng. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT*, pages 114–123. ACM, 2007.

[126] X. Liu and E. A. Lee. Cpo semantics of timed interactive actor networks. *Theoretical Computer Science*, 409(1):110–125, 2008.

[127] X. Liu, E. Matsikoudis, and E. A. Lee. Modeling timed concurrent systems. In C. Baier and H. Hermanns, editors, *CONCUR*, volume 4137 of *LNCS*, pages 1–15, August 2006.

[128] N. Lynch, R. Segala, and F. Vaandrager. Hybrid i/o automata. *Information and Computation*, 185(1):105–157, 2003.

[129] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[130] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems – Specification*. Springer, 1992.

[131] P. O. Meredith, M. Katelman, J. Meseguer, and G. Roşu. A formal executable semantics of verilog. In *MEMOCODE*, pages 179–188. IEEE, 2010.

[132] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[133] J. Meseguer. Membership algebra as a logical framework for equational specification. In *WADT*, volume 1376 of *LNCS*, pages 18–61. Springer, 1997.

[134] J. Meseguer. Localized fairness: A rewriting semantics. In *RTA*, volume 3467 of *LNCS*, pages 250–263. Springer, 2005.

[135] J. Meseguer. The temporal logic of rewriting. Technical report, University of Illinois at Urbana-Champaign, 2007. `http://hdl.handle.net/2142/11293`.

[136] J. Meseguer. The temporal logic of rewriting: A gentle introduction. In *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 354–382. Springer, 2008.

[137] J. Meseguer. Twenty years of rewriting logic. *Journal of Logic and Algebraic Programming*, 81:721–781, 2012.

[138] J. Meseguer and P. C. Ölveczky. Formalization and correctness of the PALS architectural pattern for distributed real-time systems. *Theoretical Computer Science*, 451:1–37, 2012.

[139] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. *Theoretical Computer Science*, 403(2-3):239–264, 2008.

[140] J. Meseguer and G. Rosu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007.

[141] J. Meseguer and G. Roşu. The rewriting logic semantics project: A progress report. *Information and Computation*, 231:38–69, 2013.

[142] J. Meseguer and P. Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *Higher-Order and Symbolic Computation*, 20(1–2):123–160, 2007.

[143] S. P. Miller, D. D. Cofer, L. Sha, J. Meseguer, and A. Al-Nayeem. Implementing logical synchrony in integrated modular avionics. In *Digital Avionics Systems Conference*. IEEE, 2009.

[144] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, 1986.

[145] R. D. Nicola and F. W. Vaandrager. Action versus state based logics for transition systems. In *Semantics of Systems of Concurrent Processes*, volume 469 of *LNCS*, pages 407–419. Springer, 1990.

[146] H. Ohsaki, H. Seki, and T. Takai. Recognizing boolean closed A-tree languages with membership conditional rewriting mechanism. In *RTA*, volume 2706 of *LNCS*, pages 483–498. Springer, 2003.

[147] P. C. Ölveczky, A. Boronat, and J. Meseguer. Formal semantics and analysis of behavioral AADL models in Real-Time Maude. In *FMOODS/FORTE*, volume 6117 of *LNCS*, pages 47–62. Springer, 2010.

[148] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285:359–405, 2002.

[149] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.

[150] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, 1982.

[151] M. Palomino. A predicate abstraction tool for maude. available at `http://maude.sip.ucm.es/~miguelpt/bibliography.html`, 2005.

[152] A. Pnueli, J. Xu, and L. Zuck. Liveness with $(0,\ 1,\infty)$-counter abstraction. In *CAV*, volume 2404 of *LNCS*, pages 93–111. Springer, 2002.

[153] D. Potop-Butucaru and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. *Fundamenta Informaticae*, 78(1):131–159, 2007.

[154] G. Roşu and T. F. Şerbănuţă. An overview of the k semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[155] J. Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, 1999.

[156] H. Saïdi and N. Shankar. Abstract and model check while you prove. In *CAV*, volume 1633 of *LNCS*, pages 443–454. Springer, 1999.

[157] W. Steiner and J. Rushby. TTA and PALS: Formally verified design patterns for distributed cyber-physical systems. In *Digital Avionics Systems Conference*. IEEE, 2011.

[158] J. Sun, Y. Liu, J. Dong, and J. Pang. PAT: Towards flexible verification under fairness. In *CAV*, volume 5643 of *LNCS*, pages 709–714. Springer, 2009.

[159] J. Sztipanovits and G. Karsai. Model-integrated computing. *Computer*, 30(4):110–111, 1997.

[160] J. Sztipanovits and G. Karsai. Embedded software: Challenges and opportunities. In *EMSOFT*, volume 2211 of *LNCS*, pages 403–415. Springer, 2001.

[161] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[162] G. Tel. *Introduction to distributed algorithms.* Cambridge University Press, 2000.

[163] G. Tel, E. Korach, and S. Zaks. Synchronizing ABD networks. *IEEE/ACM Transactions on Networking*, 2(1):66–69, 1994.

[164] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Using language inference to verify omega-regular properties. In *TACAS*, volume 3440 of *LNCS*, pages 45–60. Springer, 2005.

[165] M. Vardi. Automata-theoretic model checking revisited. In *VMCAI*, volume 4349 of *LNCS*, pages 137–150. Springer, 2007.

[166] P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285:487–517, 2002.

[167] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[168] H. Völzer, D. Varacca, and E. Kindler. Defining fairness. In *CONCUR*, volume 3653 of *LNCS*, pages 458–472. Springer, 2005.

[169] Y. Zhao, E. A. Lee, and J. Liu. A programming model for time-synchronized distributed real-time systems. In *Real-Time and Embedded Technology and Applications*, pages 259–268. IEEE, 2007.