Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic

Dr. Scient. thesis, Department of Computer Science, University of Bergen

Peter Csaba Ölveczky

December 18, 2000

Dedicated with affection to my beloved parents Cecilia and Miklós

Contents

Introduction

The work presented in this thesis is a part of a long term research program which explores the application of rewriting logic [Mes92] to the *executable formal modeling* of computing systems. The general conceptual advantage of using a logic instead of using a specific model is that many different models can be specified in the same logic, each in its own terms, rather than by means of possibly awkward translations into a fixed model. Since rewriting is executable, and is supported by implementations of the rewriting logic languages Maude [CDE+99a], ELAN [BKK+96], and CafeOBJ [FD98], these models can be formally analyzed using a flexible range of formal analysis techniques including rapid prototyping, model checking, and symbolic simulation.

The advantages of using rewriting logic as a semantic framework for concurrent systems, in which a wide range of concurrent system models can naturally be represented, has been amply demonstrated (see the surveys [Mes96, Mes98b]). On the pragmatic level, rewriting logic—due to its being based on a fairly natural and well-known formalism, its integration of static and dynamic system aspects, its abstract modeling of communication and concurrency, its support for concurrent object-oriented programming, and its possibilities of allowing user-defined execution strategies in the logic itself—has been shown to be an appealing formalism in which to specify and analyze a variety of applications such as communication and security protocols, which are complex enough to warrant prototyping and formal analysis, and whose operational nature fits very well with rewriting logic.

This thesis investigates whether rewriting logic is similarly well-suited to model and analyze real-time and hybrid systems. The first important research contribution exploring the application of rewriting logic to real-time specification has been the work of Kosiuczenko and Wirsing on *timed rewriting logic* (TRL) [KW97], an extension of rewriting logic where the rewrite relation is labeled with time stamps. Although TRL has been shown to be adequate for specifying a variety of complex real-time systems [ÖKW96, SK98], it suffers from having rigid synchrony requirements and from changing the logic, so that TRL specifications cannot be directly executed in a rewriting logic language implementation.

Our work indicates that rewriting logic is indeed well-suited to model and analyze real-time and hybrid system without having to change the inference rules of the logic. We show how a wide range of models of real-time and hybrid systems, including timed automata [AD94], hybrid automata [ACH+95], timed and phase transition systems [MP93], timed extensions of Petri nets [Aal93, MPT91], and object-oriented real-time systems, can indeed be expressed in rewriting logic quite naturally and directly as *real-time rewrite theories*. Such theories are rewrite theories where real-time aspects are highlighted. However, real-time rewrite theories are easily reducible to ordinary rewrite theories.

Thus, rewriting logic can be used to specify many different formal models of such systems in a unified logic. But, since rewriting logic is executable, these models can be executed and can be formally analyzed in a variety of ways. This is in contrast to the most well-known formal methods tools for real-time and hybrid systems such as Kronos [Yov97], STeP [Sta, BMSU97, MS98], and UPPAAL [LPY97]. These are model checking tools which require the user to specify both the system and the formal properties the system should satisfy. The tools then try to check whether the system satisfies a given abstract property. Besides having somewhat restrictive specification languages for purposes of remaining as much as possible in decidable fragments, these tools are not well suited for directly executing the system itself. The same can be said about HyTech [HHWT97], which takes a hybrid system description with some parameters unspecified, and returns the concrete values of the parameters which would make the system satisfy some given property. Of course, different tools have different important strengths of their own. The point is that executable specification methods and tools can complement those strengths in new ways.

To see how rewriting logic complements more abstract specification formalisms such as temporal logic as well as more concrete, automaton-based ones, one can think of it as covering an *intermediate operational level*, that can substantially help in bridging the gap between more abstract, property-oriented, specifications and actual implementations by providing:

- a precise mathematical model of the system (the initial model [Mes92]) against which more abstract specifications can be proved correct by means of inductive theorem proving, model checking, and other techniques;
- support for other useful techniques of automated or semi-automated formal reasoning and analysis at the rewriting logic and equational logic levels, such as coherence [Vir94], confluence, and strategy-based formal analysis;
- support for executable specification, symbolic simulation, and a wide range of formal analyses;
- the possibility of generating correct implementations from specifications by theory transformations and code generation techniques.

Indeed, after showing that rewriting logic is a suitable semantic framework for real-time and hybrid systems, we investigate how well the specification and analysis methods of rewriting logic already known to work well for concurrent and distributed systems (see for example [Mes00a]) specialize to real-time and hybrid systems. For this purpose, we have defined and implemented the Real-Time Maude specification language and analysis tool. The Real-Time Maude language extends the Full Maude language to support the specification of real-time rewrite theories in timed modules and object-oriented timed modules. The Real-Time Maude tool supports a wide range of techniques for formally analyzing timed modules. Besides supporting Maude's default execution strategy, it provides special-purpose strategies for the symbolic simulation of timed modules, and additional rewrite strategies and search commands for model checking timed modules with respect to properties stated in a subset of real-time temporal logic. Furthermore the tool's strategy library can be easily extended to define new application-specific analysis strategies.

For building the Real-Time Maude tool, we used Maude as a formal meta-tool [CDE$^+$99b], in which other tools—in this case the Real-Time Maude tool itself—can be represented and executed. Specifically, the specification of Real-Time Maude extends the specification of Full Maude, and makes extensive use of Maude's *reflective* features. Besides accruing great saving in implementation effort, the reflective design of the tool as an extension of Full Maude also means that the Real-Time Maude specification language inherits and extends the rich module algebra of Full Maude, including parameterized modules, theories, views, module expressions, and object-oriented modules, all of which are supported for real-time specifications.

The usefulness of a specification language and a formal methodology is measured by how well different examples and applications can be expressed, and by how effectively the formal analysis methods can be used to uncover flaws, or indicate the absence of errors, in informal specifications. We have tested the Real-Time Maude specification language and analysis tool on a variety of real-time systems, including a method for solving scheduling problems presented in this thesis. The most important case study we have undertaken has been the specification and analysis of a sophisticated protocol suite for reliable and efficient multicast in active networks, namely, the AER/NCA suite of active network protocols [KBK+99, AER00], developed by S. K. Kasera, S. Bhattacharyya, M. Keaton, D. Kiwior, J. Kurose, D. Towsley, and S. Zabele, which combines adaptability, fault-tolerance, real-time, performance, and composability of protocol aspects, and therefore presents a challenging specification and analysis problem of a kind typically not included in formal methods treatments of protocols.

We have used the tool to specify all protocols in the AER/NCA suite, as well as their composition, in an object-oriented style which is naturally supported by Real-Time Maude. We then have used the Real-Time Maude tool to formally analyze the protocols and their composition by executing them using the tool's standard interpreter and by extending the strategy library to model check the specification with respect to relevant temporal logic properties. This process has uncovered many missing assumptions, ambiguities, and flaws in the original informal specification.

This thesis is organized as follows: Chapter 1 introduces rewriting logic and the languages Maude and Full Maude. Chapter 2 defines the notion of real-time rewrite theories, how such theories can be mapped to ordinary rewrite theories in a way that preserves all their expected properties, and how many well-known models of real-time and hybrid systems can be naturally represented as real-time rewrite theories. Chapter 3 describes the Real-Time Maude specification language and the execution and search commands of Real-Time Maude. Chapter 4 presents the Maude specification of Real-Time Maude's extensible strategy library, which is also used to execute Real-Time Maude commands, and shows how the Maude specification of Full Maude has been extended to a specification of Real-Time Maude. Chapter 5 presents the Real-Time Maude specification of the AER/NCA protocol suite, and describes how Real-Time Maude was used to formally analyze the specification. Chapter 6 summarizes the thesis and outline some directions for future work. Appndix A gives the Real-Time Maude specification of some of the time domains used in the thesis, namely the natural numbers, and the rational numbers, as well as the fixed point real numbers used in the AER/NCA protocol suite. Finally, Appendix B presents the informal specification of the AER/NCA protocol suite upon which the Real-Time Maude specification of the protocol suite given in Chapter 5 is based.

Acknowledgments

The work presented in this thesis has been done at the Computer Science Laboratory, SRI International, where Dr. José Meseguer has been my thesis advisor.

I would like to express a deeply felt gratitude to José. Although I was a different kind of student than the ones he usually advises, he guided me in a most friendly, competent, and patient way. The thesis presented here is truly joint work, and is partly based on the joint publications [ÖM00b] and [ÖM00a].

Although scientific guidance is very important, José, as a true friend, has also taught me, in words and by his actions, the rôle of work in both social and (not necessarily disjointly) spiritual contexts. I consider him not only as my thesis advisor but as a true and very dear friend. Unfortunately, a three-year relationship will inevitably have a moment of disappointment: Just before finishing this thesis, I lost for the first time our prestigious weekly tennis encounter.

I am also greatly indebted to Professor Martin Wirsing at Ludwig-Maximilians-Universität, München, where I stayed and learnt a lot during a too short, but very nice and hectic, four months in the beginning of my doctoral studies. That stay introduced me to rewriting logic and real-time systems. I also thank Professor Sigurd Meldal for inviting me to Bergen and for sending me to Stanford University and SRI International, and for his guidance in the initial phase of my studies, and Michal Walicki, who was a very friendly mentor in Bergen during the brief time I didn't prepare for teaching there. I also thank my master's thesis advisor, Professor Olav Lysne in Oslo, for introducing me to term rewriting, and Harry Hartmann, an excellent teacher in Malmö who always motivated me to learn more.

This thesis is dedicated to my parents, Cecilia and Miklós, two completely wonderful human beings, whose love was always of the best—and for a parent probably the hardest—kind: unselfish. My dear family is completed by my great little brother Bence, a most gracious host in Cambridge, Mass., during my trip to the East coast to work with Mark Keaton on parts of this thesis. I also thank my beloved grandmother Babkó for our special relationship. I am particularly grateful and happy to have stayed as much as possible with the "Ball Community," that is, my aunt Eszter, my uncle Peter, and my cousins Zsófi and Ágnes ("Ifi"), in San Francisco. They all welcomed me with generosity in abundance—there was never a lack of chocolate croissants in the morning and of beer in the evening—and provided me with a home away from home where I was always a (spoilt) son and brother, never a nephew and cousin. It is symptomatic of their generosity that I was spoilt even more, if possible, after three years with them than on my first day in San Francisco.

I imagine most doctoral students feel frustrated during their studies. I had to suddenly change advisor, commute across the Atlantic to fulfill teaching and other obligations, and was, in addition, not used to hard, consistent work. Still, I remember my studies as some of the best years of my life—both for being able to closely follow the challenging research done by my advisor and others at SRI, but mostly because of all the great friends I have made during these years. It is "beyond the scope of this thesis" to thank all these truly great friends individually. I still feel the need to mention some individuals on whose shoulders I leaned too frequently, and who I hope always will remain dear friends. They include Sonia Tsui and Mark-Oliver "Ollie" Stehr for some wonderful time in 2000, including a classic trip to L.A. which will never be forgotten, Christiano Braga, Hassen Saïdi, with particular thanks for his support during the toughest times in 1999, Peter Habermehl, a fellow sports nut in California and a generous host in Paris, Francisco Durán, and Manuel "El Functor" Garcia Clavel and his (and my!) friends at Menlough. I enjoyed tremendously living for extended periods of time with my generous housemates Denise West and George Talbot, the most intelligently funny man I've met.

In München, a favorite city, I was well received and introduced to the city's pleasures by Bernhard Reus, Robert Stabl, Rolf Hennicker, and Stephan Merz. I owe them a lot for enjoying my four months in München so much. In Bergen, I truly enjoyed the company of Michal Walicki, with whom I had the most pleasant disagreements on some teaching material, not to talk about the beer, the jokes, the movies, etc., my office-mate and close friend, "Bambino" Luka Onesti, and Jan Arne Telle. In Oslo, I feel a deep friendship to Imtiaz and Maiken Ahmad, and I thank them for the one month they spent in California (you should visit me more often!), Tor Skeie and his family who managed to visit me three times in California, and my Oslo, Marktoberdorf, and München friend Henrik Linnestad. I am especially grateful for the support I received from Ingrid Gravseth during more than five years. Her support and presence were invaluable in the early years of my doctoral studies. Her visits in München and our trip to the various natural wonders in Arizona and Utah are among my dearest memories.

My only regret seeing this list of wonderful friends and family, is that I wish I could be in California, Oslo, Hungary, Cambridge, Bergen, and München at the same time, all the time!

I have benefitted enormously from being at the remarkable Computer Science Laboratory at SRI International. I thank the lab director and my friend Pat Lincoln for letting me work here, and all the people who without hesitation or arrogance were most happy to help me at any time (that is, *everyone* at SRI). I abused in particular the kindness of Ollie Stehr, Franciso Durán, Steven Eker, Pat Lincoln, Grit Denker, and Natarajan Shankar. Narciso Martí-Oliet was always an encouraging friend and colleague who not only did not hesitate to help—he almost demanded to help! My papers improved significantly thanks to his comments. The dry-run of a talk in Asilomar will be remembered for a long time for the good-natured humiliation, cleverly disguised as "help," by Narciso, Manuel, and Pat. I enjoyed it! I really liked working with Mark Keaton and Steve Zabele in Reading, if only for one week, and have benefitted from discussions with Carolyn Talcott and Joseph Sifakis. Lee Iverson took upon him the seemingly impossible task of teaching me how to play basketball—thank you for your effort, even though clearly in vain. I also thank head of the department Terje Espelid, and teaching boss Khalid Mughal, at the Department of Computer Science, University of Bergen, for being as flexible as possible with my teaching responsibilities, allowing me to spend the time necessary at SRI. Michal Walicki helped me with a lot in Bergen, and Sigurd Meldal in Bergen/Stanford. Thanks also to my friend Piotr Kosiuczenko in München for the work on our paper [ÖKW96].

Finally, I would like to thank to thank Dr. Garth O. George in St. George's, Grenada, Dr. Jamie S. Barkin in Miami, and Professor Dr. Hansjochen Wilke in Essen for their help, without which it would have been difficult to finish this thesis.

I feel truly blessed to have seen so much friendship and love in only five years!

Partial support from the following research contracts is gratefully acknowledged: Support from the Norwegian Research Council, support from DARPA through Rome Laboratories Contract F30602-97-C-0312, from DARPA and NASA through Contract NAS2-98073, from Office of Naval Research Contracts N00014-96-C-0114 and N00014-99-C-0198, and from National Science Foundation Grants CCR-9633363 and CCR-9900334.

Chapter 1

Preliminaries on Rewriting Logic, Maude, and Full Maude

This section gives some background material on rewriting logic (see e.g., [Mes92, Mes96, Mes98b] for an introduction to rewriting logic and its applications), the rewriting logic language Maude [CDE$^+$99a, CDE$^+$00], and the Full Maude language and tool [CDE$^+$99a, Dur99, DM99]. We assume familiarity with the basic concepts of algebraic specification (see e.g., [Wir90]) and term rewriting (see e.g., [DJ90]). Sections 1.1 and 1.2 are adapted from [Mes00b].

1.1 Rewriting Logic

Rewriting logic is a logic of change where the dynamic parts of the system are specified by labelled conditional rewrite rules and the static and structural parts are specified by equations. A *signature* in rewriting logic is an equational theory[1] (Σ, E), where Σ is an equational signature and E is a set of Σ-equations. Rewriting will operate on equivalence classes of terms modulo E.

Given a signature (Σ, E), *sentences* of rewriting logic are sequents of the form

$$[t]_E \longrightarrow [t']_E,$$

where t and t' are Σ-terms possibly involving some variables, and $[t]_E$ denotes the equivalence class of the term t modulo the equations E. A *rewrite theory* $\mathcal{R}$ is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$ where Σ is a ranked alphabet of function symbols, E is a set of Σ-equations, L is a set of *labels*, and R is a set of pairs $R \subseteq L \times T_{\Sigma,E}(X)^2$ whose first component is a label and whose second component is a pair of E-equivalence classes of terms, with $X = \{x_1, \ldots, x_n, \ldots\}$ a countably infinite set of variables. Elements of R are called *rewrite rules*.[2] We understand a rule $(l, ([t], [t']))$ as a labeled

[1] Rewriting logic is parameterized by the choice of its underlying equational logic, that can be unsorted, many-sorted, order-sorted, membership equational logic, and so on. To ease the exposition we give an *unsorted* presentation.

[2] To simplify the exposition the rules of the logic are given for the case of *unconditional* rewrite rules. However, all the ideas presented here have been extended to conditional rules in [Mes92] with very general rules of the form

$$l : [t] \longrightarrow [t'] \ \ \textit{if} \ \ [u_1] \longrightarrow [v_1] \ \wedge \ldots \wedge \ [u_k] \longrightarrow [v_k].$$

This increases considerably the expressive power of rewrite theories. More generally, one can consider rewrite rules that include also equations and (for membership equational logic) memberships in their conditions. Indeed, Maude supports the specification of these more general rewrite rules.

sequent and use for it the notation $l : [t] \longrightarrow [t']$. To indicate that $\{x_1, \ldots, x_n\}$ is the set of variables occurring in either t or t', we write $l : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$, or in abbreviated notation $l : [t(\overline{x})] \longrightarrow [t'(\overline{x})]$.

Given a rewrite theory $\mathcal{R}$, we say that $\mathcal{R}$ *entails* a sentence $[t] \longrightarrow [t']$, or that $[t] \longrightarrow [t']$ is a *(concurrent)* $\mathcal{R}$*-rewrite*, and write $\mathcal{R} \vdash [t] \longrightarrow [t']$ if and only if $[t] \longrightarrow [t']$ can be obtained by finite application of the following *rules of deduction* (where we assume that all the terms are well formed and $t(\overline{w}/\overline{x})$ denotes the simultaneous substitution of w_i for x_i in t):

1. **Reflexivity**. For each $[t] \in T_{\Sigma,E}(X)$, $\dfrac{}{[t] \longrightarrow [t]}$.

2. **Congruence**. For each $f \in \Sigma_n$, $n \in \mathbb{N}$,
$$\frac{[t_1] \longrightarrow [t'_1] \quad \ldots \quad [t_n] \longrightarrow [t'_n]}{[f(t_1, \ldots, t_n)] \longrightarrow [f(t'_1, \ldots, t'_n)]}.$$

3. **Replacement**. For each rule $l : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$ in R,
$$\frac{[w_1] \longrightarrow [w'_1] \quad \ldots \quad [w_n] \longrightarrow [w'_n]}{[t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w'}/\overline{x})]}.$$

4. **Transitivity**
$$\frac{[t_1] \longrightarrow [t_2] \qquad [t_2] \longrightarrow [t_3]}{[t_1] \longrightarrow [t_3]}.$$

A sentence $[t] \longrightarrow [t']$ in $\mathcal{R}$ is called a *one-step rewrite* iff it can be derived from $\mathcal{R}$ by a finite number of applications of the rules (1) – (3), with at least one application of rule (3). If rule (3) was applied exactly once, then we say that the sentence is a *one-step sequential rewrite*.

Rewriting logic is a logic for reasoning correctly about *concurrent systems* having *states*, and evolving by means of *transitions*. The signature of a rewrite theory describes a particular structure for the states of a system—e.g., multiset, binary tree, etc.—so that its states can be distributed according to such a structure. The rewrite rules in the theory describe which *elementary local transitions* are possible in the distributed state by concurrent local transformations. The rules of rewriting logic allow us to reason correctly about which *general* concurrent transitions are possible in a system satisfying such a description. Thus, computationally, each rewriting step is a parallel local transition in a concurrent system.

Alternatively, however, we can adopt a logical viewpoint instead, and regard the rules of rewriting logic as *metarules* for correct deduction in a *logical system*. Logically, each rewriting step is a logical *entailment* in a formal system.

The initial model of a rewrite theory $\mathcal{R}$ is a category $\mathcal{T}_\mathcal{R}$ whose objects are equivalence classes $[t]$ of *ground* terms modulo the equations in E, and whose arrows are *proofs* $\alpha : [t] \longrightarrow [t']$ in rewriting logic—that is, concurrent computations in the system so described. Proofs are generated by "decorating" the deduction rules of rewriting logic as follows:

1. **Identities**. For each $[t] \in T_{\Sigma,E}(X)$, $\dfrac{}{[t] : [t] \longrightarrow [t]}$.

2. **Σ-structure**. For each $f \in \Sigma_n$, $n \in \mathbb{N}$,

$$\frac{\alpha_1 : [t_1] \longrightarrow [t'_1] \quad \ldots \quad \alpha_n : [t_n] \longrightarrow [t'_n]}{f(\alpha_1, \ldots, \alpha_n) : [f(t_1, \ldots, t_n)] \longrightarrow [f(t'_1, \ldots, t'_n)]}.$$

3. **Replacement**. For each rewrite rule $l : [t(\overline{x}^n)] \longrightarrow [t'(\overline{x}^n)]$ in R,

$$\frac{\alpha_1 : [w_1] \longrightarrow [w'_1] \quad \ldots \quad \alpha_n : [w_n] \longrightarrow [w'_n]}{l(\alpha_1, \ldots, \alpha_n) : [t(\overline{w}/\overline{x})] \longrightarrow [t'(\overline{w'}/\overline{x})]}.$$

4. **Composition** $\dfrac{\alpha : [t_1] \longrightarrow [t_2] \qquad \beta : [t_2] \longrightarrow [t_3]}{\alpha; \beta : [t_1] \longrightarrow [t_3]}$.

Proofs are equated modulo the following notion of *proof equivalence* that computationally corresponds to the "true concurrency" of the computation:

1. **Category**
   (a) *Associativity.* For all α, β, γ, $\quad (\alpha; \beta); \gamma = \alpha; (\beta; \gamma)$.
   (b) *Identities.* For each $\alpha : [t] \longrightarrow [t']$, $\quad \alpha; [t'] = \alpha$ and $[t]; \alpha = \alpha$.

2. **Functoriality of the Σ-algebraic structure**. For each $f \in \Sigma_n$,
   (a) *Preservation of composition.* For all $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n$,

   $$f(\alpha_1; \beta_1, \ldots, \alpha_n; \beta_n) = f(\alpha_1, \ldots, \alpha_n); f(\beta_1, \ldots, \beta_n).$$

   (b) *Preservation of identities.* $\quad f([t_1], \ldots, [t_n]) = [f(t_1, \ldots, t_n)]$.

3. **Axioms in E**. For $t(x_1, \ldots, x_n) = t'(x_1, \ldots, x_n)$ an axiom in E, for all $\alpha_1, \ldots, \alpha_n$, $t(\alpha_1, \ldots, \alpha_n) = t'(\alpha_1, \ldots, \alpha_n)$.

4. **Exchange**. For each $l : [t(x_1, \ldots, x_n)] \longrightarrow [t'(x_1, \ldots, x_n)]$ in R,

$$\frac{\alpha_1 : [w_1] \longrightarrow [w'_1] \quad \ldots \quad \alpha_n : [w_n] \longrightarrow [w'_n]}{l(\overline{\alpha}) = l(\overline{[w]}); t'(\overline{\alpha}) = t(\overline{\alpha}); l(\overline{[w']})}.$$

We will often use the result in [Mes92] that a proof $\alpha : [t] \longrightarrow [t']$ can be factorized into a proof corresponding to a sequence of one-step sequential rewrites. That is, if $\alpha : [t] \longrightarrow [t']$, then either α is the identity proof, or there is a sequence of one-step sequential rewrites

$$[t] \xrightarrow{\alpha_0} [t_1] \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{n-1}} [t_n] \xrightarrow{\alpha_n} [t']$$

where α and $\alpha_1; \ldots; \alpha_n$ are equivalent modulo the equivalence of proof terms described above.

For most specifications in this thesis the equational theory (Σ, E) will be assumed to be *order-sorted* [GM92], that is, the set of sorts comes equipped with a partial order relation, with $s \leq s'$ interpreted as subset inclusion $A_s \subseteq A_{s'}$ in a model A. Furthermore, operation symbols can be *subsort overloaded* (as for example the addition symbol $+$ for naturals, integers, and rationals). Such overloaded operators are required to yield the same result for the same arguments, regardless of the overloaded operator that is applied. In addition, the order-sorted signature Σ is assumed to be *coherent* [GM92]. This ensures that all terms have a least sort and that the poset of sorts has good properties. In Section 2.2.6 we give a rewriting logic specification that uses an even more expressive equational logic, namely, *membership equational logic* [Mes98a].

The initial model $\mathcal{T}_{\mathcal{R}}$ has to be understood in an order-sorted sense, so that for each sort s we have an associated category $(\mathcal{T}_{\mathcal{R}})_s$, with arrows $\alpha : t \longrightarrow t'$ equivalence classes of proofs with t, t' ground terms of sort s.

1.2 Maude

Maude [CDE+99a, CDE+00] is a language and system developed at SRI International whose modules are theories in rewriting logic. The most general Maude modules are called *system modules*. They have the syntax `mod` $\mathcal{R}$ `endm` with $\mathcal{R}$ the rewrite theory in question, expressed with a syntax quite close to the corresponding mathematical notation[3]. The equations E in the equational theory (Σ, E) underlying the rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$ are presented as a union $E = E' \cup A$, with A a set of *equational axioms* introduced as *attributes* of certain operators in the signature Σ—for example, an operator $+$ can be declared associative and commutative by keywords `assoc` and `comm`—and where E' is a set of equations that are assumed to be Church-Rosser and terminating *modulo* the axioms A. Furthermore, $\mathcal{R}$ is assumed to be weakly (ground) coherent [Vir94]. Maude supports rewriting modulo different combinations of such equational attributes A: operators can be declared with any combination of associative, commutative, and with left, right, or two-sided identity attributes.

Since we can view an equational theory as a rewrite theory whose set L and R of labels and rules are both empty, Maude contains a sublanguage of *functional modules* of the form `fmod` (Σ, E) `endfm`, where, as before, $E = E' \cup A$, with E' Church-Rosser and terminating modulo A. The equational logic of functional modules and of the equational part of system modules is membership equational logic [Mes98a], an expressive logic whose atomic formulas are equations $t = t'$ and membership predicates $t : s$ with s a sort. Membership equational logic is quite expressive. It supports sorts, subsorts, operator overloading, and equational partiality. But this expressiveness is achieved while being efficiently executable by rewriting and having suitable techniques for completion and for equational theorem proving.

Maude has a third class of modules, namely, *object-oriented modules* that specify concurrent object-oriented systems and have syntax `omod` $\mathcal{R}$ `endom`, with $\mathcal{R}$ a sugared form for a rewrite theory. That is, object-oriented modules are internally translated into ordinary system modules, but Maude provides a more convenient syntax for them, supporting concepts such as objects, messages, object classes, and multiple class inheritance.

Modules in Maude have an *initial semantics*. Therefore, a system module `mod` $\mathcal{R}$ `endm` specifies the initial model $\mathcal{T}_{\mathcal{R}}$ of the rewrite theory $\mathcal{R}$. Similarly, a functional module `fmod` (Σ, E) `endfm` specifies the initial algebra $T_{\Sigma,E}$ of the equational theory (Σ, E).

In addition, Maude supports a module algebra with parameterized modules, parameter theories (with "loose" semantics) views (that is, theory interpretations) and module expressions. In Maude, this module algebra, called Full Maude, is defined inside the language by reflection, and is easily

[3]See [CDE+99a] for a detailed description of Maude's syntax.

extensible with new module composition operations [DM98, Dur99]. Such a module algebra is one important concrete application of the general *metaprogramming* capabilities made possible by Maude's support of reflection through the META-LEVEL module. Another important application of reflection is the capacity for defining internal strategy languages that can guide the execution of rewrite theories by means of rewrite rules at the metalevel.

1.3 The Reflective Features of Maude and the Module META-LEVEL

Clavel and Meseguer has shown that rewriting logic is *reflective* [CM96a, CM96b, Cla98, CDE+98]. That is, there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal* in the sense that we can represent any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) and any terms t, t' in $\mathcal{R}$ as terms $\overline{\mathcal{R}}$ and $\overline{t}$, $\overline{t'}$ in $\mathcal{U}$, and that we then have the following equivalence

$$\mathcal{R} \vdash t \longrightarrow t' \;\;\Leftrightarrow\;\; \mathcal{U} \vdash \langle \overline{R}, \overline{t} \rangle \longrightarrow \langle \overline{R}, \overline{t'} \rangle.$$

The theory $\mathcal{U}$ can be seen as a universal interpreter for rewriting logic which can simulate the rewrites of any rewrite theory $\mathcal{R}$. It may, however, be quite costly to simulate $\mathcal{R}$-rewrites in (a naïve implementation of) $\mathcal{U}$, since the simulation of a single step in $\mathcal{R}$ may involve many $\mathcal{U}$-steps. The universal theory $\mathcal{U}$ can therefore be extended with *descent functions* [CDE+98] which simulate some operations, say a rewrite step, on parameters $\overline{t}$ and $\overline{R}$ in an efficient way by "going down" one level of reflection and use the rewrite engine to compute in $\mathcal{R}$ the result t' of applying the rewrite to t. The resulting term t' is then transformed into its representation $\overline{t'}$. An extension of a universal theory with descent functions is conservative under certain conditions, and is therefore itself a universal theory [CDE+98].

In the Maude language, the predefined module META-LEVEL is intended to model a universal theory $\mathcal{U}$. In the module META-LEVEL terms are reified as elements of the data type Term, having the following signature

```
subsort Qid < Term .
subsort Term < TermList .
op {_}_ : Qid Qid -> Term .
op _[_] : Qid TermList -> Term .
op _,_ : TermList TermList -> TermList [assoc] .
op error* : -> Term .
```

The first declaration, making the sort Qid of quoted identifiers a subsort of Term, is used to represent variables in a term by the corresponding quoted identifiers. Thus, the variable N is represented by 'N. The operator {_}_ is used for representing constants as pairs, with the first argument the constant, in quoted form, and the second argument the sort of the constant, also in quoted form. For example, a constant zero of sort Time is represented as {'zero}'Time. The operator _[_] corresponds to the recursive construction of terms out of subterms, where list concatenation is denoted _,_. For example, the term f(a,b) (for a and b constants of a sort s) is represented in META-LEVEL as 'f[{'a}'s, {'b}'s] and the term {f(a,b)} (for {_} an infix operator) is represented as '`{_`}['f[{'a}'s, {'b}'s]].

Maude functional and system modules are represented in a syntax very similar to their original syntax:

```
sorts FModule Module .
subsort FModule < Module .

op fmod_is_______endfm : Qid ImportList SortDecl SubsortDeclSet OpDeclSet
                         VarDeclSet MembAxSet EquationSet -> FModule .
op mod_is________endf : Qid ImportList SortDecl SubsortDeclSet OpDeclSet
                        VarDeclSet MembAxSet EquationSet RuleSet -> Module .
```

The Maude system manual [CDE+99a] provides all the details about the representation of modules. For example, the set of rules in a module is represented using the following syntax:

```
sorts Rule RuleSet .
subsort Rule < RuleSet .
op rl[_]:_=>_. : Qid Term Term -> Rule .
op crl[_]:_=>_if_=_. : Qid Term Term Term Term -> Rule .
op none -> RuleSet .
op __ : RuleSet RuleSet -> RuleSet [assoc comm id: none] .
```

The module `META-LEVEL` provides key functionality in efficient descent functions. For example, the process of reducing a term to its normal form is reified by a function

```
op meta-reduce : Module Term -> Term [special ... ]
```

where `meta-reduce`$(\overline{M}, \overline{t})$ returns the meta-representation $\overline{u}$ of the normal form u of the term t in the module M. The process of applying a rule is reified by a descent function

```
op meta-apply : Module Term Qid Substitution MachineInt -> ResultPair [special ... ] .
```

where `meta-apply`$(\overline{M}, \overline{t}, \overline{l}, \overline{\sigma}, n)$ gives the result of applying the rule l, partially instantiated with σ, in M to the (normal form of the) term t, using the $(n+1)$th successful match. The result returned is a term `{` $\overline{t'}, \overline{\sigma'}$ `}` where t' is the normal form of the resulting term, and σ' is the match used in the rule application. The result `{error*,none}` is returned if there are fewer than $n+1$ successful matches. Observe that the rule is only applied at the "top" of the (M-normal form of the) term t. The process of rewriting a term in a system using Maude's default interpreter is reified by a descent function `meta-rewrite`. Parsing is reified by a function

```
op meta-parse : Module QidList -> Term [special ... ] .
```

which parses a list of quoted identifiers in a given module.

Maude also contains a module `LOOP-MODE` which provides built-in facilities for keeping a persistent state while performing input and output. Input is given at Maude's command line by writing a pair of parentheses '`(`' and '`)`' around a list of identifiers. This input can be read by a Maude application extending `LOOP-MODE` as the corresponding list of quoted identifiers, which can be manipulated and stored in the persistent state. The Maude application can output text by placing it as a list of quoted identifiers in the output buffer.

1.4 Full Maude

Since Maude modules are first-class data objects in Maude, it is relatively easy to define, in Maude modules extending `META-LEVEL`, different operations on modules. This is taken advantage of in the Full Maude system, which is a Maude specification developed by Durán and Meseguer [Dur99, DM99], and defines a language and a tool for the specification and execution of Maude modules extended with a rich module algebra with module hierarchies, parameterization, views, theories, and module expressions. In addition, Full Maude supports the specification of *object-oriented* modules using a convenient syntax. Full Maude uses the `LOOP-MODE` facilities to read modules and Full Maude "system commands," to store in the persistent state a database of modules read so far, and to output the result of the commands by placing the appropriate list of quoted identifiers in the output buffer.

More precisely, the input is parsed in two stages using the `meta-parse` function. In the first stage of say, the parsing of a module specification, the input is parsed with respect to the top-level grammar defining the syntax of Full Maude commands and modules. The parts of input representing terms in the module being defined, cannot be parsed with respect to the top-level grammar, and are "protected" from being parsed by being enclosed in "bubbles" [CDE$^+$99a]. In the second stage, when the signature part of the module being defined is determined, then the bubbles can be parsed by `meta-parse` taking as grammar the signature of the module being parsed. The module is then completely parsed, and is entered into the Full Maude's module database in the persistent loop state. It is not yet "flattened," and must be flattened to become the meta-representation of a (Core) Maude module before being executed by `meta-rewrite`.

The treatment is similar for system commands, such as rewrite commands, when Full Maude must first retrieve from its database the module in which to rewrite, flatten that module, and then parse the input representing the term to rewrite. The rewrite command is executed by calling the descent function `meta-rewrite`, and the result of the rewrite is transformed by `meta-pretty-print` to a list of quoted identifiers, and placed in the output buffer.

Chapter 2

Specification of Real-Time and Hybrid Systems in Rewriting Logic

In this chapter, which is based on our joint paper [ÖM00b], we show that rewriting logic can be used to specify real-time systems in a natural way. Essentially, all we need is to include in the specification a *Time* data type satisfying appropriate equational properties. However, it is sometimes useful to highlight the real-time aspect by making explicit the duration information for some rewrite rules. We formalize this idea in Section 2.1 by means of *real-time rewrite theories*; but we show that, by adding an explicit clock, they are reducible to ordinary rewrite theories in a way that preserves all the expected properties.

The naturalness of the specification method, and its smooth integration with rewriting logic's support for object-oriented specification, is explored and illustrated with examples in Section 2.2, in which we also address the question of how generally and naturally rewriting logic can be used to express a variety of real-time and hybrid system models. We show in detail how, besides object-oriented real-time systems, a wide range of such models, including timed automata [AD94], hybrid automata [ACH$^+$95], timed and phase transition systems [MP93], and timed extensions of Petri nets [Aal93, MPT91], can indeed be expressed in rewriting logic quite naturally and directly.

As mentioned, The first important research contribution exploring the application of rewriting logic to real-time specification has been the work of Kosiuczenko and Wirsing on *timed rewriting logic* (TRL) [KW97], an extension of rewriting logic where the rewrite relation is labeled with time stamps. TRL has been shown well-suited for giving object-oriented specifications of complex hybrid systems such as the steam-boiler [ÖKW96], and has also been applied to give semantics to the SDL telecommunications specification language [SK98]. In fact, rewriting logic object-oriented specifications in the Maude language [CDE$^+$99a] have a natural extension to TRL object-oriented specifications in *Timed Maude* [KW97, ÖKW96].

The approach taken here is different. We argue that real-time systems can be specified in ordinary rewriting logic, and that reasoning about their behavior does not require a special inference system of their own, such as the one proposed in TRL. Even when special notation highlighting real-time aspects—such as that provided by real-time rewrite theories—is used, we show that this can essentially be regarded as syntactic sugar. This has the conceptual advantage of remaining within a simpler theoretical framework, and the practical advantage of being able to use the existing language implementations of rewriting logic to execute specifications. Therefore, it seems both conceptually and practically useful to study the relationships between our approach and TRL. We do so in Section 2.3, where we show that there is a map of logics $\mathcal{M} : TRL \longrightarrow RWL$ sending each TRL specification to a corresponding rewrite theory in such a way that logical entailment is preserved. However, the translated theory $\mathcal{M}(\mathcal{T})$ can in general prove additional sentences. This is due to some intrinsic conceptual differences between both formalisms that our analysis reveals.

Throughout the text in this chapter we often use Maude-like notation [CDE+99a] to present specific rewrite theories. For the most part this notation is self-explanatory. For example, we use for (conditional) rewrite rules the notation $[l] : t \longrightarrow t'$ **if** C.

2.1 Time Models and Real-Time Rewrite Theories

After specifying equationally the general requirements for the models of time that we will consider (Section 2.1.1) we propose a general notion of *real-time rewrite theory*, consisting of an ordinary rewrite theory, where rewrite rules affecting the whole system have associated time-duration expressions (Section 2.1.2). We then show that real-time rewrite theories form a category (Section 2.1.3) and that they can be reduced to *ordinary* rewrite theories by adding an explicit clock to the global state in a way that preserves all the expected properties (Section 2.1.4). We finish the section with a discussion of several issues and specification techniques for real-time rewrite theories (Section 2.1.5).

2.1.1 Time Models

Time is modeled abstractly by a commutative monoid $(Time, +, 0)$ with additional operators $\leq$, $<$, and $\dot{-}$ ("monus") satisfying the following Maude theory.[1]

fth *TIME* **is**
protecting *BOOL*
  sort *Time*
  op $0 : \to Time$
  op $\_+\_ : Time\ Time \to Time\quad [assoc\ \ comm\ \ id : 0]$
  ops $\_<\_, \_\leq\_ : Time\ Time \to Bool$
  op $\_\dot{-}\_ : Time\ Time \to Time$
  vars $x_r, y_r, z_r, w_r : Time$
  ceq $y_r = z_r$ **if** $x_r + y_r == x_r + z_r$
  eq $(x_r < x_r) = false$
  ceq $x_r < z_r = true$ **if** $x_r < y_r$ and $y_r < z_r$
  eq $(x_r \leq y_r) = (x_r < y_r)$ or $(x_r == y_r)$
  eq $0 \leq x_r = true$
  ceq $x_r + y_r \leq z_r + w_r = true$ **if** $x_r \leq z_r$ and $y_r \leq w_r$
  eq $x_r \leq (x_r \dot{-} y_r) + y_r = true$
  ceq $(x_r \dot{-} y_r) + y_r = x_r$ **if** $y_r \leq x_r$
  ceq $x_r \dot{-} z_r \leq y_r \dot{-} z_r = true$ **if** $x_r \leq y_r$

[1]Note that conditional equations (**ceq**) having as condition a Boolean term b are shorthand notation for the same equations with condition $b = true$. In particular, such Boolean expressions can contain the Boolean-valued equality predicate denoted $==$.

endft

In this theory, it can for example be proved that the relation $\leq$ is a partial order with least element 0, and that for all $x_r, y_r : Time$, $y_r \leq x_r$ if and only if there exists a unique z_r (namely $x_r \dot{-} y_r$) such that $x_r = y_r + z_r$.

For simulation and executable specification purposes we will be interested in *computable* models of the above theory *TIME*. This means that all the operations are computable. By the Bergstra-Tucker Theorem [BT87], any computable algebra is finitely specifiable as an initial algebra for a set E of Church-Rosser and terminating equations. For example, the nonnegative rational numbers can be so specified as a model of *TIME* by adding a subsort Rat_+ to the specification of rationals in [GM92], and extending it with order and monus operations in the obvious way. Similarly, the real algebraic numbers with the standard order are also computable [Rab60], and therefore have a finite algebraic specification with Church-Rosser and terminating equations. Note that just taking a constructive version of the real numbers will not yield a computable data type, because the equality and order predicates on the constructive reals are not computable [Bee85].

In some examples in this thesis we will need to extend the time domain with a new value ∞ and/or require that the time domain is linear. The following theory gives an abstract specification of the time domain extended with a value ∞.

fth $TIME_\infty$ **is**
including *TIME*
  sort $Time_\infty$
  subsort $Time \leq Time_\infty$
  op $\infty : \to Time_\infty$
  op $\_ \leq \_ : Time_\infty\ Time_\infty \to Bool$
  op $\_ \dot{-} \_ : Time_\infty\ Time \to Time_\infty$
  op $\_ + \_ : Time_\infty\ Time_\infty \to Time_\infty\ \ [assoc\ \ comm\ \ id : 0]$
  var $x_r : Time$
  var $y_r : Time_\infty$
  eq $y_r \leq \infty = true$
  eq $\infty \leq x_r = false$
  eq $x_r < \infty = true$
  eq $\infty < y_r = false$
  eq $\infty \dot{-} x_r = \infty$
  eq $\infty + y_r = \infty$
endft

Linear time can be specified by the following theory:

fth *LTIME* **is**
including *TIME*
  op $\min : Time\ Time \to Time\ \ [comm]$
  vars $x_r, y_r : Time$
  ceq $x_r = y_r$ **if** $not(x_r < y_r)$ and $not(y_r < x_r)$
  ceq $\min(x_r, y_r) = y_r$ **if** $y_r \leq x_r$
endft

This theory can also be extended with a time value ∞ as follows:

fth *LTIME*$_\infty$ **is**
including *LTIME, TIME*$_\infty$
  op $\min : Time_\infty\ Time_\infty \to Time_\infty$ [*comm*]
  var $x_r : Time_\infty$
  eq $\min(\infty, x_r) = x_r$
endft

Notation: We will use symbols r, r', $r_1, \ldots$ to denote time values and x_r, $y_r, \ldots$ to denote *variables* of the sort of the time domain.

2.1.2 Real-Time Rewrite Theories

We define in this section real-time rewrite theories; they are used to specify real-time systems in rewriting logic and contain duration information for some rules. Rules are divided into *tick rules*, that model the elapse of time on a system, and *instantaneous rules*, that model instantaneous change and are assumed to take zero time. Having a tick rule $[tick] : t \longrightarrow t'$ could lead to rewrites $f(t, u) \longrightarrow f(t', u)$, i.e., rewrites where time only elapses in a part of the system under consideration. To ensure that time advances uniformly in all the parts of a state, we introduce a new sort *System*, with no subsorts, and a free constructor $\{\_\} : State \to System$ with the intended meaning that $\{t\}$ denotes the whole system, which is in state t. Uniform time elapse is then ensured if the global state always has the form $\{t\}$ and every tick rule is of the form $[tick] :\ \{t\} \longrightarrow \{t'\}$.

Before defining real-time rewrite theories, we recall the notion of a theory morphism between equational theories, which will be used to show how the time domain is interpreted in a real-time rewrite theory.

Definition 1 *An* equational theory morphism $H : (\Sigma, E) \to (\Sigma', E')$ *consists of a monotone map* $H : sorts(\Sigma) \to sorts(\Sigma')$*, and a mapping sending each function symbol*[2] $f : s_1 \ldots s_n \to s$ *in* Σ *to a* Σ'*-term* $H(f_{s_1 \ldots s_n, s})$ *of sort* $H(s)$*, such that its set of variables is contained in the set* $x_1 : H(s_1), \ldots, x_n : H(s_n)$*, and such that if* $f : s_1 \ldots s_n \to s$*, and* $f : s'_1 \ldots s'_n \to s'$ *are subsort overloaded operators with* $s_i \leq s'_i$*,* $s \leq s'$*, then the term* $H(f_{s_1 \ldots s_n, s})$ *can be obtained from the term* $H(f_{s'_1 \ldots s'_n, s'})$ *by replacing each variable* $x_i : H(s'_i)$ *by the corresponding variable* $x_i : H(s_i)$*. Furthermore, for each axiom* $(\forall y_1 : s_1, \ldots, y_k : s_k)\ u = v$ **if** C *in* E*,*

$$E' \models (\forall y_1 : H(s_1), \ldots, y_k : H(s_k))\ H^*(u) = H^*(v) \ \textbf{if}\ H^*(C)$$

holds, for H^* *the homomorphic extension of* H *to terms, and to equations in the condition* C*.*

A real-time rewrite theory can then be defined as a rewrite theory together with an interpretation of the abstract time domain and an assignment of duration to rules involving terms of sort *System*:

Definition 2 *A* real-time rewrite theory $\mathcal{R}_{\phi,\tau}$ *is a tuple* $(\mathcal{R}, \phi, \tau)$*, where* $\mathcal{R} = (\Sigma, E, L, R)$ *is a rewrite theory, such that*[3]*:*

[2] Since the variables $x_1, \ldots, x_n$ are ordered, the assignment $f : s_1 \ldots s_n \to s \mapsto H(f_{s_1 \ldots s_n, s})$ can alternatively be understood as an assignment $f(x_1 : s_1, \ldots, x_n : s_n) \mapsto H(f_{s_1 \ldots s_n, s})$.

[3] We give a definition based on loose semantics of rewrite theories. Real-time rewrite theories can be defined in a similar way in an initial semantics setting.

- *ϕ is an equational theory morphism $\phi : TIME \to (\Sigma, E)$ where TIME is the theory defined in Section 2.1.1,*
- *the time domain is functional; that is, whenever $\alpha : r \longrightarrow r'$ is a rewrite proof in $\mathcal{R}$ and r is a term of sort $\phi(Time)$, then $r = r'$ and α is equivalent to the identity proof r,*
- *(Σ, E) contains a designated sort that we usually call State and a specific sort System with no subsorts or supersorts and with only one operator*

  $$\{\_\} : State \to System$$

  which satisfies no non-trivial equations; furthermore, for any $f : s_1 \ldots s_n \to s$ in Σ, the sort System does not appear among the $s_1, \ldots, s_n$;
- *τ is an assignment of a term $\tau_l(x_1, \ldots, x_n)$ of sort $\phi(Time)$ to each rewrite rule in $\mathcal{R}$ of the form*

  $$(\dagger) \qquad [l] : u(x_1, \ldots, x_n) \longrightarrow u'(x_1, \ldots, x_n) \text{ if } C(x_1, \ldots, x_n)$$

  where u and u' are terms of sort System.

Notation: We will write

$$[l] : u(x_1, \ldots, x_n) \stackrel{\tau_l(x_1,\ldots,x_n)}{\longrightarrow} u'(x_1, \ldots, x_n) \text{ if } C(x_1, \ldots, x_n)$$

for a rule l of sort *System* with duration term τ_l. If $\tau_l(x_1, \ldots, x_n)$ equals $\phi(0)$, the rule l will often be written

$$[l] : u(x_1, \ldots, x_n) \longrightarrow u'(x_1, \ldots, x_n) \text{ if } C(x_1, \ldots, x_n).$$

We will also write $Time_\phi$, 0_ϕ, and $+_\phi$ for, respectively, $\phi(Time)$, $\phi(0)$, and $\phi(+)$. In the rest of the thesis we will often further simplify the notation and write *Time*, 0, and + instead of $Time_\phi$, 0_ϕ, and $+_\phi$. Finally, we will often write $Time_\infty$, ∞, ... instead of $\phi'(Time_\infty)$, $\phi'(\infty)$, ... when $\phi' : TIME_\infty \to (\Sigma, E)$ is an extension of ϕ.

We call rules of the form (†) *global rules.* A global rule l is a *tick rule* if its duration $\tau_l(x_1, \ldots, x_n)$ is different from 0_ϕ for some instance of its variables, and is an *instantaneous rule* otherwise. Rules not of the form (†) are called *local* rules, because they do not act on the system as a whole, but only on some system components. Local rules are always viewed as instantaneous rules that take zero time.

The total time elapse $\tau(\alpha)$ of a rewrite $\alpha : \{t\} \longrightarrow \{t'\}$ of sort *System* is defined as the sum of the time elapsed in each tick rule application in α, and can easily be extracted from the proof:

Definition 3 *Let $(\mathcal{R}, \phi, \tau)$ be a real-time rewrite theory with $\mathcal{R} = (\Sigma, E, L, R)$ and let Time denote the time domain $(T_{\Sigma,E})_{Time_\phi}$ viewed as a monoid and therefore as a category with a single object 0, and with the time values as arrows in the usual way. The* time extraction functor

$$\tau : (\mathcal{T}_\mathcal{R})_{System} \to Time$$

which gives the total time elapse $\tau(\alpha)$ of a proof $\alpha : t \longrightarrow t'$, with t, t' ground terms of sort System, is defined as follows:

- $\tau(t) = 0_\phi$ *for every term (seen as an identity arrow) in* $(\mathcal{T}_\mathcal{R})_{System}$*,*
- $\tau(\{\alpha\}) = 0_\phi$ *for a proof term whose top operator is the constructor* $\{\_\}$*,*
- $\tau(l(\alpha_1, \ldots, \alpha_n)) = \tau_l(t_1, \ldots, t_n)$ *if* l *is a (system) rule of the form* $(\dagger)$ *and* $\alpha_1 : t_1 \longrightarrow t_1', \ldots, \alpha_n : t_n \longrightarrow t_n'$ *are proofs, and*
- $\tau(\alpha;\beta) = \tau(\alpha) +_\phi \tau(\beta)$ *for proofs* α *and* β*.*

This definition does not depend on the choice of representative proof terms. That is, if α and β are two equivalent proofs of terms of sort *System* in a real-time rewrite theory $(\mathcal{R}, \phi, \tau)$, then $\tau(\alpha) = \tau(\beta)$.

Given a real-time rewrite theory $\mathcal{R}$, a *computation* is a non-extensible sequence $t_0 \longrightarrow t_1 \longrightarrow \cdots \longrightarrow t_n$ (that is, one for which t_n cannot be further rewritten) or an infinite sequence $t_0 \longrightarrow t_1 \longrightarrow \cdots$ of one-step $\mathcal{R}$-rewrites $t_i \longrightarrow t_{i+1}$, with t_i and t_{i+1} ground terms, starting with a given initial term t_0 of sort *System*. It should be noted that, since we model time elapse *explicitly* (by rewrite rules), the usual requirement that the total time elapse in a computation is infinite is not needed. Time elapse is totally up to the specifier; we allow both terminating computations and infinite computations with finite total time elapse, that is, "Zeno" computations.

2.1.3 A Category of Real-Time Rewrite Theories

The notion of theory morphism—also called theory interpretation—between real-time rewrite theories plays an important role in this work. We give a definition of theory morphism between real-time rewrite theories based on loose semantics and preservation of durations of rewrites. Morphisms based on properties of the initial models of theories, and morphisms having less restrictive requirements on the relationships between the durations in the rewrites could be defined in a similar way. We begin by defining theory morphisms between ordinary rewrite theories.

Definition 4 *A* rewrite theory morphism *from a rewrite theory* $\mathcal{R} = (\Sigma, E, L, R)$ *to another rewrite theory* $\mathcal{R}' = (\Sigma', E', L', R')$ *consists of:*

- *an equational theory morphism* $H : (\Sigma, E) \to (\Sigma', E')$*, and*
- *a map* $H : L \longrightarrow L'$ *of labels such that for each rule* $[l] : t \longrightarrow t'$ **if** C *in* R *the rule*

$$[H(l)] : H^*(t) \longrightarrow H^*(t') \textbf{ if } H^*(C)$$

  is in R' *up to a renaming of its variables.*

Rewrite theory morphisms compose in the expected way and define a category *RWTh* of rewrite theories.

Definition 5 *A* real-time rewrite theory morphism *from a real-time rewrite theory* $(\mathcal{R}, \phi, \tau)$ *to a real-time rewrite theory* $(\mathcal{R}', \phi', \tau')$ *is a rewrite theory morphism* $H : \mathcal{R} \to \mathcal{R}'$ *such that:*

- $\phi' = H \circ \phi$*,*

- *H maps the designated sort of the states in $\mathcal{R}$ to the designated state sort in $\mathcal{R}'$, maps the sort System to itself, and leaves the constructor $\{\_\}$ unchanged, and*
- *H preserves the duration of the tick rules in the sense that for each rule l in $\mathcal{R}$ of sort System,*

$$E' \vdash H^*(\tau_l(x_1 : s_1, \ldots, x_n : s_n)) = \tau'_{H(l)}(x_1 : H(s_1), \ldots, x_n : H(s_n)).$$

It is easy to check that the usual composition of rewrite theory morphisms defines a category *RTRWTh* with real-time rewrite theories as objects and real-time rewrite theory morphisms as arrows.

2.1.4 Real-Time Theories Internalized in Rewriting Logic

By adding a clock to the state, a real-time rewrite theory $(\mathcal{R}, \phi, \tau)$ can be transformed into an ordinary rewrite theory without losing timing information. A state in such a clocked system is of the form $\langle t, r\rangle$ with t the global state of sort *System*, and r a value of sort $Time_\phi$, which intuitively is supposed to denote the total time elapsed in a computation if in the initial state the clock had value 0_ϕ.

Definition 6 *The* internalizing functor $(\_)^C$ *from the category RTRWTh of real-time rewrite theories to the category RWTh of rewrite theories takes a real-time rewrite theory $(\mathcal{R}, \phi, \tau)$ to a rewrite theory $\mathcal{R}^C_{\phi,\tau} = (\Sigma^C_{\phi,\tau}, E^C_{\phi,\tau}, L^C_{\phi,\tau}, R^C_{\phi,\tau})$ as follows:*

- *the sorts in $\Sigma^C_{\phi,\tau}$ are those in $\mathcal{R}$, together with a new sort ClockedSystem,*
- *the operations in $\Sigma^C_{\phi,\tau}$ are those in $\mathcal{R}$, together with a new free constructor*

$$\langle \_, \_\rangle : System\ Time_\phi \to ClockedSystem,$$

- *the axioms in $E^C_{\phi,\tau}$ are unchanged from those in $\mathcal{R}$,*
- *$R^C_{\phi,\tau}$ contains the local rules in $\mathcal{R}$ of sorts other than System, together with a rule*[4]

$$\begin{aligned}[l^C_{\phi,\tau}(x_1, \ldots, x_n, x_r)] :\ & \langle u(x_1, \ldots, x_n), x_r\rangle \longrightarrow \\ & \langle u'(x_1, \ldots, x_n), x_r +_\phi \tau_l(x_1, \ldots, x_n)\rangle \textbf{ if } C(x_1, \ldots, x_n)\end{aligned}$$

for each rule

$$[l(x_1, \ldots, x_n)] :\ u(x_1, \ldots, x_n) \longrightarrow u'(x_1, \ldots, x_n) \textbf{ if } C(x_1, \ldots, x_n)$$

in $\mathcal{R}$ of sort System, where x_r is a variable of sort $Time_\phi$ which is not in the list $x_1, \ldots, x_n$.

The internalizing functor is defined as expected on arrows in RTRWTh; i.e., an arrow H in RTRWTh is mapped to H^C, which coincides with H on $\mathcal{R}$, leaves the new sort and operator unchanged, and takes a label $l^C_{\phi,\tau}$ of a rule of sort ClockedSystem to the label $(H(l))^C$.

[4]In the unlikely case that any condition C of a rule in $\mathcal{R}$ contains a conjunct $v \longrightarrow v'$ of sort *System*, each such conjunct is replaced by a conjunct $\langle v, 0_\phi\rangle \longrightarrow \langle v', y_r\rangle$ in the condition in $\mathcal{R}^C_{\phi,\tau}$, where y_r is a fresh variable of sort $Time_\phi$.

For the sake of a simpler exposition, in the rest of the thesis we will assume that no condition of a rewrite rule in a real-time theory contains a rewrite conjunct of sort *System*. We also assume, without loss of generality, that no variable of sort *System* is introduced in the condition of a rule.

Proposition 7 *The mapping* $(\_)^C$ *above defines a functor from RTRWTh to RWTh.*

Proposition 8 *Let* V *be the forgetful functor from the category RTRWTh of real-time rewrite theories to the category RWTh of rewrite theories defined by*

$$V((\mathcal{R},\phi,\tau)) = \mathcal{R} \ \ \textit{and} \ \ V(H) = H.$$

Then the map of rewrite theories $\pi_{(\mathcal{R},\phi,\tau)} : \mathcal{R}^C_{\phi,\tau} \to \mathcal{R}$ *defined by:*

- *mapping each sort and operator in* $\Sigma^C_{\phi,\tau}$ *other than ClockedSystem and* $\langle \_, \_ \rangle$ *identically to themselves,*
- *mapping ClockedSystem to System, mapping the operator* $\langle \_, \_ \rangle$ *to the term* $x_1 : System$*, and*
- *mapping each label* $l^C_{\phi,\tau}$ *to the label* l

defines a natural transformation $\pi : (\_)^C \Rightarrow V$*.*

Since a rewrite theory morphism $H : \mathcal{R} \to \mathcal{R}'$ induces a forgetful functor $U_H : \mathcal{R}'\text{-}Sys \to \mathcal{R}\text{-}Sys$ in the opposite direction for the corresponding categories of models, our natural rewrite theory morphism $\pi : \mathcal{R}^C_{\phi,\tau} \to \mathcal{R}$ induces a forgetful functor $U_\pi : \mathcal{R}\text{-}Sys \to \mathcal{R}^C_{\phi,\tau}\text{-}Sys$. In particular, the initial model $\mathcal{T}_\mathcal{R}$ is sent to the $\mathcal{R}^C_{\phi,\tau}$-system $U_\pi(\mathcal{T}_\mathcal{R})$ and, by initiality of $\mathcal{T}_{\mathcal{R}^C_{\phi,\tau}}$, we have a unique $\mathcal{R}^C_{\phi,\tau}$-homomorphism $h : \mathcal{T}_{\mathcal{R}^C_{\phi,\tau}} \to U_\pi(\mathcal{T}_\mathcal{R})$ such that:

- h takes objects and arrows of every sort except *ClockedSystem* to themselves,
- $h(\langle t, r\rangle) = t$ for each object $\langle t, r\rangle$ in $(\mathcal{T}_{\mathcal{R}^C_{\phi,\tau}})_{ClockedSystem}$,
- $h(\langle \alpha, \beta\rangle : \langle t, r\rangle \longrightarrow \langle t', r'\rangle) = \alpha : t \longrightarrow t'$,
- $h(l^C_{\phi,\tau}(\alpha_1, \ldots, \alpha_n, \alpha_{n+1}) : \langle t, r\rangle \longrightarrow \langle t', r'\rangle) = l(\alpha_1, \ldots, \alpha_n) : t \longrightarrow t'$, and
- $h(\alpha; \beta) = h(\alpha); h(\beta)$.

The map h expresses the essential semantic equivalence between the initial model of a real-time theory $(\mathcal{R}, \phi, \tau)$ and that of its clocked representation $\mathcal{R}^C_{\phi,\tau}$ in the precise sense that, as we shall see:

1. if $\alpha : t \longrightarrow t'$ is an arrow in $(\mathcal{T}_\mathcal{R})_{System}$ with $\tau(\alpha) = r$, then, for each value r' of sort $Time_\phi$ there is a unique arrow $\alpha' : \langle t, r'\rangle \longrightarrow \langle t', r' +_\phi r\rangle$ in $(\mathcal{T}_{\mathcal{R}^C_{\phi,\tau}})_{ClockedSystem}$ such that $h(\alpha' : \langle t, r'\rangle \longrightarrow \langle t', r' +_\phi r\rangle) = \alpha : t \longrightarrow t'$, and
2. whenever $\alpha : \langle t, r\rangle \longrightarrow \langle t', r'\rangle$ is an arrow in $(\mathcal{T}_{\mathcal{R}^C_{\phi,\tau}})_{ClockedSystem}$ then $r' = r +_\phi \tau(h(\alpha))$.

These two properties are immediate consequences of the following

Theorem 9 *Let $(\mathcal{R}, \phi, \tau)$ be a real-time rewrite theory and let $\alpha : t \longrightarrow t'$ be an arrow in $(\mathcal{T}_{\mathcal{R}})_{System}$ (therefore, with t and t' ground terms of sort System). Then, for each value r in the time domain, there is a* unique *arrow*

$$\alpha' : \langle t, r\rangle \longrightarrow \langle t', r'\rangle$$

in $(\mathcal{T}_{\mathcal{R}^C_{\phi,\tau}})_{ClockedSystem}$ such that $h(\alpha') = \alpha$, and, in addition, $r' = r +_\phi \tau(\alpha)$.

The theorem can be proved by induction on the structure of the proof terms by first proving the theorem for one-step rewrites, and then proving it for all proofs between terms of sort *System* using the facts that every proof factorizes into a sequence of one-step rewrites and that h distributes over one-step rewrite proofs.

The above theorem implies that, whenever $\alpha : \langle t, r\rangle \longrightarrow \langle t', r'\rangle$ is an arrow in $\mathcal{T}_{\mathcal{R}^C_{\phi,\tau}}$, then the arrow $h(\alpha) : t \longrightarrow t'$ satisfies $r +_\phi \tau(h(\alpha)) = r'$. It also implies that h, viewed as a functor $h : (\mathcal{T}_{\mathcal{R}^C_{\phi,\tau}})_{ClockedSystem} \to (\mathcal{T}_{\mathcal{R}})_{System}$, is full and faithful, and is an opfibration [BW99].

2.1.5 Discussion

We discuss several system specification issues and techniques, including tick rules, eager and lazy rules, and time as an action on the system.

Specifying the Tick Rules

For simulation of a system having a continuous time domain, the tick rules will often be of the form

$$[tick] :\ \{t\} \xrightarrow{x_r} \{t'(x_r)\} \textbf{ if } x_r \leq mte(t) \text{ and } C(t)$$

or otherwise of the exact same form, but replacing $x_r \leq mte(t)$ by $x_r < mte(t)$, where x_r is a variable not occurring in t and denotes the time advanced by the tick, $mte(t)$ computes the *maximum time elapse* permissible to ensure timeliness of time-critical actions, and the condition $x_r \leq mte(t)$ (resp. $x_r < mte(t)$) ensures that time elapse *may* halt temporarily for the *possible* application of a non-time-critical rule, that is, a rule modeling an action which could occur somewhat "arbitrarily" in time. The introduction of the variable x_r in the righthand side requires additional execution strategies for its instantiations, which is not surprising, since it models behavior which is nondeterministic in time. Allowing for real nondeterminism in timed behavior in this way may lead to Zeno behavior of the system and it is up to the execution strategy to instantiate the righthand side variable so as to avoid that, whenever possible.

Eager and Lazy Rules

In general, it is not sufficient to ensure that time elapse "stops" whenever necessary. Often, it must also be ensured that time does not advance again before all the necessary instantaneous actions are performed. In particular, an application of a rule often enables a number of instantaneous rules that must be taken immediately, and it must be ensured that all these actions are performed before time elapses again. A rule may, for example, produce a message which must be consumed before time advances again. In many cases it is possible to add conditions on the tick rules such that time will not elapse if some time-critical rule is enabled, but this may considerably complicate the specification. Instead of computing the enabledness condition of every time-critical rule explicitly, it is often more convenient to use the rewriting logic notion of *internal rewrite strategy* [CDE+99a, Cla98, CM96b], whose execution is well supported by Maude's reflective features [CDE+98, CM96a], to deal with these enabledness and priority aspects using a simple strategy.

The idea is to divide the rules in a real-time rewrite theory into *eager* and *lazy* rules and to let the *admissible* rewrites be those rewrites where the application of eager rules takes precedence over the application of lazy rules.

Definition 10 *Let $\mathcal{R}_{\phi,\tau}$ be a real-time theory with its set R of rewrite rules partitioned into disjoint sets R_{eager} and R_{lazy}. Then, the set of* admissible rewrites *in $\mathcal{R}_{\phi,\tau}$ are those rewrites $\alpha : t \longrightarrow t'$ in which lazy rules are applied sequentially and, furthermore, they are only applied when no eager rule is enabled. That is, a rewrite $\alpha : t \longrightarrow t'$ is admissible if and only if α is an identity proof or is equivalent to a factorization $\alpha_1; \ldots ; \alpha_n$ of proof terms α_i corresponding to one-step* sequential *rewrites such that if some $\alpha_j : t_{j-1} \longrightarrow t_j$ is a proof of an application of a rule in R_{lazy}, then there is no one-step sequential rewrite $\beta : t_{j-1} \longrightarrow u$ applying a rule in R_{eager}.*

Tick rules and non-time-critical instantaneous rules should be lazy, which implies that time will not advance while some eager rule is enabled. Our treatment of timed Petri nets in Section 2.2.6 gives an example of the convenience of using this strategy.

Notation: Whenever an eager strategy should be used, the eager and lazy rules will be preceded by the keywords **eager** and **lazy**, respectively.

Time as an Action on the Whole System

When the state of a system has a rich structure, it may be both natural and necessary to have an explicit function denoting the effect of time elapse on the whole state. The function δ denoting the action of time on a system has the form

$$\delta : State\ Time \to State$$

involving the designated sorts *State* and *Time*. The action δ should be a *monoid action*, that is, it seems natural to require that it satisfies the axioms:

$$\delta(x, 0) = x$$
$$\delta(\delta(x, y_r), z_r) = \delta(x, y_r + z_r).$$

Tick rules should then be of the form

$$(\dagger) \qquad [tick] : \ \{t\} \xrightarrow{r} \{\delta(t, r)\} \ \textbf{if}\ C.$$

Using the action δ to describe the effect of the passage of time on a dynamic evolution of a system is not without possible pitfalls. If done carelessly, it may allow "going back in time" to perform a rewrite. Suppose, for example, that $t = \delta(t', r)$ holds, and that the term t' rewrites to t''. Then, there would also be an "aged" rewrite $\{t\} =_E \{\delta(t', r)\} \longrightarrow \{\delta(t'', r)\}$, as illustrated in the following example.

Example 11 *The following specification is intended to specify a discrete time "clock" (that is, Time is the sort Nat of natural numbers) which can be reset when it has reached the value 24.*

sorts *State System*
op $clock : Nat \to State$
op $\delta : State\ Nat \to State$
op $\{\_\} : State \to System$
vars $x_r, y_r : Nat$
var $z : State$
eq $\delta(clock(x_r), y_r) = clock(x_r + y_r)$
rl $[reset] : clock(24) \longrightarrow clock(0)$
rl $[tick] : \{z\} \xrightarrow{x_r} \{\delta(z, x_r)\}$

However, starting from $\{clock(0)\}$*, there is a rewrite sequence*

$$\begin{aligned}&\{clock(0)\} \overset{tick(\ldots,35)}{\longrightarrow} \{\delta(clock(0), 35)\} = \{clock(35)\} = \{\delta(clock(24), 11)\}\\ &\longrightarrow \{\delta(clock(0), 11)\} = \{clock(11)\}\end{aligned}$$

where the reset rule is applied at time 35. The problem is that the equation for δ *and the reset rule are not coherent as explained below.*

For executable specification purposes it is important to require that the set E of equations in a rewrite theory is divided into a set E' of simplifying equations and a set Ax of structural axioms, in such a way that the equations in E' define a Church-Rosser and terminating set of equations modulo the set Ax, and such that the set of rules R is *coherent* [CDE$^+$99a, Vir94] w.r.t. $E' \uplus Ax$. A rewrite theory is coherent if for every one-step sequential rewrite $t \longrightarrow t_1$ modulo the structural axioms Ax, there is also a rewrite $t!_{E'} \longrightarrow t'_1$ modulo Ax, for $t!_{E'}$ the E'-normal form of t modulo Ax, such that t_1 and t'_1 are E-equivalent. A coherent system in which ground terms of the form $\{\delta(t', r)\}$ reduce by the equations to terms of the form $\{t\}$ with no δ's does not allow "going back in time," in the sense that for any rewrite $t' \longrightarrow u'$ inducing a rewrite $\{\delta(t', r)\} \longrightarrow \{\delta(u', r)\}$ there must be an equivalent rewrite $\{t\} \longrightarrow \{w\}$ such that $\{\delta(u', r)\} =_{E' \cup Ax} \{w\}$.

A commonly occurring state structure for which we want the action of time to distribute over the different state components is a *multiset* distributed structure. For example, object-oriented systems and Petri nets have that structure. For multiset distributed systems we can give a general treatment of time actions that avoids lack of coherence problems.

A simple solution to avoid lack of coherence problems is to let each rule rewrite terms of sort *System* only, which would solve the coherence problem w.r.t. the symbol δ, since each rewrite would occur at the top. However, concurrency is lost by this solution. Our solution is to use special tokens of the form '$*$' and to let the extended state be a term in a supersort *ExtendedState* of the designated sort *State*, consisting of the multiset union of the original state and a multiset of tokens. The system operator $\{\_\}$ takes arguments of the sort *ExtendedState*, while δ is left unchanged, i.e., is a function $\delta : State\ \ Time \to State$. If multiset union is denoted by juxtaposition, the tick rules then take the form

$$(\ddagger) \qquad [tick] :\ \{T\ t\} \xrightarrow{r} \{T\ \delta(t, r)\} \textbf{ if } C,$$

for T a variable of a subsort $Tokens \leq ExtendedState$, denoting multisets of tokens, and t a term of sort $State$. Each local rule then has the form[5]

$$[l] : * \, t \longrightarrow * \cdots * \, t' \textbf{ if } C.$$

Since one token appears in the lefthand side of each local rule, the global state must contain at least n tokens for n local rewrites to fire concurrently. For object-oriented systems, the number of tokens in a configuration could suitably equal the number of objects in a configuration, since the number of rewrites firing concurrently is bounded by the number of objects present in the global state, under the assumption that at least one object appears in the lefthand side of each rule. Coherence w.r.t. the symbol δ is now trivially unproblematic, since every instance of a lefthand side of a local rule has least sort $ExtendedState$, and therefore cannot be an argument of δ. This general approach to deal with time as an action on multiset distributed states is specialized to object-oriented systems in Section 2.2.5, and is illustrated with a multi-thermostat system example in Section 2.2.5.

To summarize, a monoid action δ denoting the effect of time elapse on the whole state may be useful for specifying real-time systems where the state of the system can have a rich distributed structure, but we must require coherence, since this ensures that δ does not cause counterintuitive rewrites resulting from "going back in time."

2.2 Specifying Models of Real-Time and Hybrid Systems in Rewriting Logic

We show how some well-known models of real-time and hybrid systems can be naturally regarded as specializations of the real-time rewriting logic framework. Since we are interested in executable specifications, we place some computability restrictions on some models. Even though we do not present an exhaustive discussion of real-time models, we think that the models we have chosen are significantly varied and well-known to suggest that rewriting logic is a good semantic framework for real-time and hybrid systems.

2.2.1 Timed Automata

We show how a *timed automaton* (see, e.g., [AD94]) can be specified in rewriting logic. A timed automaton reads timed words—infinite words where each symbol in a word has a time stamp—and is equipped with a set of clocks. A transition may reset some clocks to 0, and can only be applied if the values of the clocks satisfy the clock constraint of the transition. For example, the timed automaton in Fig. 2.1 "accepts" timed words with alternating occurrences of the input symbols a and b, starting with a, such that an occurrence of a is followed by an occurrence of b no earlier than time 5 and no later than time 10 after the occurrence of a, and each occurrence of b is followed by an a in some time in $[4, 8]$. Furthermore, each occurrence of a must be followed by another occurrence of a within time 15.

The time model of timed automata is the set of nonnegative real numbers. However, to get a computable data type, we can replace it by the set of nonnegative elements $\mathbb{R}_+$ of a computable

[5]To ensure maximal possible concurrency in following rewrite steps, some tokens may have to be added to the righthand side of the rule if the rule increases the size of the state.

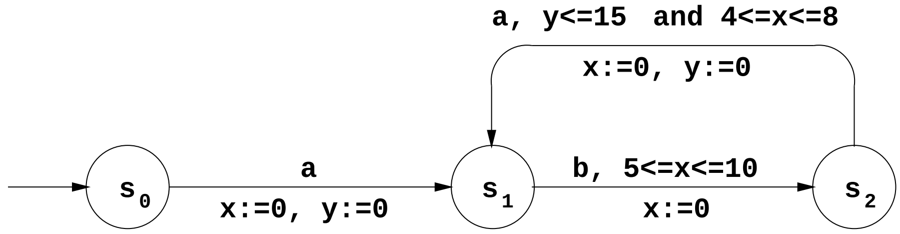

Figure 2.1: A timed automaton.

subfield $\mathbb{R}$, such as the rationals or the algebraic real numbers. Omitting details about initial states and acceptance conditions, a timed automaton is given by a tuple (Σ, S, C, E) where:

- Σ is a finite alphabet;
- S is a finite set of states;
- C is a finite set of clocks, defining a set $\Phi(C)$ of clock constraints inductively by

$$\varphi ::= c \leq k \mid k \leq c \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2$$

  where c is a clock in C, and k is a constant in the set of nonnegative rationals, and
- E is a set $E \subseteq S \times S \times \Sigma \times 2^C \times \Phi(C)$ of transitions. The tuple $(s, s', a, \lambda, \varphi)$ represents a transition from state s to state s' on input symbol a. The set $\lambda \subseteq C$ gives the clocks to be reset with this transition, and φ is a clock constraint over C.

Given a timed word (i.e., an infinite sequence of tuples (a_i, r_i) where a_i is an input symbol and r_i is the time at which it occurs, and where $r_i \leq r_{i+1}$ for all $i \geq 0$), the automaton starts at time 0 with all clocks initialized to 0. As time advances, the values of all clocks change, reflecting the elapsed time; that is, the state of the automaton can change not only by the above transitions, but also by the passage of time, with all the clocks being increased by the elapsed time. At time r_i the automaton changes state from s to s' using some transition of the form $(s, s', a_i, \lambda, \varphi)$ reading input a_i, if the current values of the clocks satisfy φ. With this transition the clocks in λ are reset to 0, and start counting time again.

A *run* ρ of a timed automaton is an *infinite*[6] sequence

$$\rho : \ (s_0, v_0) \xrightarrow[\tau_1]{a_1} (s_1, v_1) \xrightarrow[\tau_2]{a_2} (s_2, v_2) \xrightarrow[\tau_3]{a_3} \cdots$$

with states $s_0, s_1, \ldots$, valuations $v_0, v_1, \ldots \in [C \to \mathbb{R}_+]$ such that $v_0(c) = 0$ for all $c \in C$, and such that for each $i \geq 1$, $\tau_{i-1} \leq \tau_i$ and there is a transition $(s_{i-1}, s_i, a_i, \lambda_i, \varphi_i)$ where the clock valuation $v_{i-1} + (\tau_i - \tau_{i-1})$ (with $\tau_0 \stackrel{\text{def}}{=} 0$) satisfies the clock constraint φ_i, and $v_i(c)$ is 0 if clock c is in λ_i and $v_i(c) = v_{i-1}(c) + (\tau_i - \tau_{i-1})$ otherwise. The set of all runs of a timed automaton A is denoted $\mathit{Runs}(A)$.

We assume in the remainder of this section that the set of clocks is ordered as a sequence $c_1, \ldots, c_n$. A timed automaton $A = (\Sigma, S, C, E)$ can then be naturally represented by a real-time rewrite theory $\Psi_{TA}(A) = ((\Sigma_A, E_A, L_A, R_A), \phi_A, \tau_A)$ as follows:

[6]Runs are infinite, since timed words where the automaton cannot proceed upon reading a symbol with a timestamp can never be accepted by the automaton.

- (Σ_A, E_A) contains an equational axiomatization of the time domain $\mathbb{R}_+$. Furthermore, the signature Σ_A contains a sort *TAState* with a constant $s : \to \mathit{TAState}$ for each $s \in S$, a sort *State* with an $(n+1)$-ary operator $\_,\_,\ldots,\_ : \mathit{TAState}\ \mathit{Time}\ \ldots\ \mathit{Time} \to \mathit{State}$, and an operator $\{\_\} : \mathit{State} \to \mathit{System}$.

- The set L_A of labels is $\Sigma \cup \{\mathit{tick}\}$.

- The set of rules R_A and its associated duration assignment contains an instantaneous rule

$$[a] : \{s, x_1, \ldots, x_n\} \longrightarrow \{s', t_1, \ldots, t_n\} \textbf{ if } \varphi(x_1, \ldots, x_n)$$

for each transition $(s, s', a, \lambda, \varphi) \in E$, where the x_i's are variables of sort *Time*, where the term t_i is 0 if $c_i \in \lambda$ and $t_i = x_i$ otherwise, and where $\varphi(x_1, \ldots, x_n)$ is obtained from φ by substituting each clock c_i with x_i and by substituting $\wedge$ and $\neg$ with the Boolean operators "and" and "not". In addition, a rule

$$[\mathit{tick}] : \ \{z, x_1, \ldots, x_n\} \xrightarrow{y_r} \{z, x_1 + y_r, \ldots, x_n + y_r\}$$

(where $z, y_r, x_1, \ldots, x_n$ are all variables) is added to model time elapse.

The timed automaton in Fig. 2.1 is therefore represented by a real-time rewrite theory containing the following rules:

$$\begin{array}{llll}
[a]: & \{s_0, x, y\} & \longrightarrow & \{s_1, 0, 0\} \\
[b]: & \{s_1, x, y\} & \longrightarrow & \{s_2, 0, y\} \textbf{ if } 5 \leq x \leq 10 \\
[a]: & \{s_2, x, y\} & \longrightarrow & \{s_1, 0, 0\} \textbf{ if } y \leq 15 \text{ and } 4 \leq x \leq 8 \\
[\mathit{tick}]: & \{z, x, y\} & \xrightarrow{y_r} & \{z, x + y_r, y + y_r\}
\end{array}$$

for x, y, and y_r variables of sort *Time*, and z a variable of sort *TAState*.

A *TA-step* in $\Psi_{TA}(A)$, which simulates one step in a run ρ of a timed automaton $A = (\Sigma, S, C, E)$, is a ground rewrite $\alpha; b(\overline{\beta}) : t \longrightarrow t'$ in $\Psi_{TA}(A)$, where $b \in \Sigma$ and α is a proof corresponding to a (possibly empty) sequence of tick applications. Since the passage of time r can be modeled by an arbitrary sequence of tick applications with total time elapse r, we consider two TA-steps $\alpha; b(\overline{\beta}) : t \longrightarrow t'$ and $\alpha'; b'(\overline{\beta'}) : u \longrightarrow u'$ to be *equivalent* if and only if $t = u$, $t' = u'$, $\tau(\alpha) = \tau(\alpha')$, and $b = b'$ hold. Together, these conditions imply $\beta = \beta'$. A TA-step $\alpha; b(\overline{\beta}) : t \longrightarrow t'$ is, therefore, uniquely determined by t, t', $\tau(\alpha)$, and b, and will be written $t \xrightarrow[\tau(\alpha)]{b} t'$.

A *TA-computation* of the rewriting logic representation $\Psi_{TA}(A)$ of a timed automaton A is an infinite sequence

$$\tilde{\rho} : \{t_0, 0, \ldots, 0\} \xrightarrow[r_1]{b_1} \{t_1, r_{1_1}, \ldots, r_{1_n}\} \xrightarrow[r_2]{b_2} \{t_2, r_{2_1}, \ldots, r_{2_n}\} \xrightarrow[r_3]{b_3} \cdots$$

of TA-steps in $\Psi_{TA}(A)$. The set of all TA-computations in $\Psi_{TA}(A)$ (modulo TA-step equivalence) is denoted $\mathit{TA}\text{-}\mathcal{C}(\Psi_{TA}(A))$.

Theorem 12 *Let $A = (\Sigma, S, C, E)$ be a timed automaton, and let $\Psi_{TA}(A)$ be its rewriting logic translation. Then, there exists a bijective function $\widetilde{(\_)} : \mathit{Runs}(A) \to \mathit{TA}\text{-}\mathcal{C}(\Psi_{TA}(A))$ assigning to each run of A of the form ρ defined above a TA-computation of the form $\tilde{\rho}$ with $s_0 = t_0$, $s_i = t_i$, $\tau_i - \tau_{i-1} = r_i$, $a_i = b_i$, and $r_{i_j} = v_i(c_j)$ for each $i \geq 1$ and $j \in \{1, \ldots, n\}$.*

Proof sketch: The theorem follows from the fact that

$$(s_i, v_i) \xrightarrow[\tau_i+\Delta_i]{a_{i+1}} (s_{i+1}, v_{i+1})$$

is a "transition step" in A for each $i \geq 0, \Delta_i, \tau_i, s_i, s_{i+1}, v_i, v_{i+1}$ if and only if there is a TA-step

$$\{s_i, r_{i_1}, \ldots, r_{i_n}\} \xrightarrow[\Delta_i]{a_{i+1}} \{s_{i+1}, r_{i+1_1}, \ldots, r_{i+1_n}\}$$

in $\Psi_{TA}(A)$ with $r_{j_k} = v_j(c_k)$ for $j = i, i+1$.

Using the equational axioms defining equivalence of rewrite proofs in rewriting logic [Mes92], one can prove

Proposition 13

- *Each non-identity ground rewrite proof $\gamma : t \longrightarrow t'$ in $\Psi_{TA}(A)$ can be decomposed as a sequence $\gamma = \xi_1; \cdots; \xi_n; \delta_1; \cdots; \delta_m$, with $n, m \geq 0$ and $n + m \geq 1$, where $\xi_1, \ldots, \xi_n$ are all TA-steps and the $\delta_1, \ldots, \delta_m$ are all tick applications.*
- *Any infinite computation of $\Psi_{TA}(A)$ (see Section 2.1.2), starting with a term of the form $\{t_0, 0, \ldots, 0\}$, and involving an infinite number of instantaneous rule applications, can be rearranged in the form of a TA-computation $\tilde{\rho}$ by an appropriate composition of the arrows.*

There are at least two ways of modifying the specification to simulate the behavior of the automaton on only those timed words satisfying a given acceptance condition. It is possible to define a computable predicate *has_computation*, so that *has_computation*$(s, r_1, \ldots, r_n)$ holds if and only if there exists an *accepted* timed word "starting" in state s with (rational-numbered) values $r_1, \ldots, r_n$ of the clocks $c_1, \ldots, c_n$ (such a predicate is computable, and therefore finitely specifiable by Church-Rosser and terminating equations [BT87], since defining such a predicate reduces to the emptiness problem for timed automata which is decidable [AD94]). In this way, we obtain a rewrite theory whose computations simulate the behavior of the automaton on *accepted* timed words by adding the condition **if** *has_computation*$(t', t_1, \ldots, t_n)$ to every rule of the form $[l] : \{t, x_1, \ldots, x_n\} \longrightarrow \{t', t_1, \ldots, t_n\}$, including the tick rule.

A more modular, alternative way of restricting the rewrites to simulate automata behavior on accepted words only would be to encode the accepting states (or sets of states for Muller-automata) as predicates in the rewrite theory, and then use the internal strategies at the metalevel of rewriting logic to restrict the application of the rules, so that only accepted timed words are executed.

2.2.2 Hybrid Automata

A *hybrid automaton* [ACH+95] (also called a *hybrid system*) is a finite automaton equipped with variables that evolve continuously with time according to dynamical laws, and where each location is labeled with an invariant condition that must hold when control resides at that location. The hybrid automaton in Fig. 2.2 describes a simple thermostat which keeps the temperature in a room between m and M by turning a heater on and off. In our simplified model, the temperature in the room increases by 2 degrees per time unit when the heater is turned on, and decreases by 1 degree per time unit otherwise.

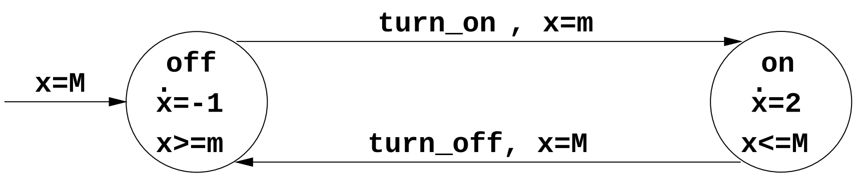

Figure 2.2: A hybrid automaton model of a thermostat.

The time model of hybrid automata is the set of nonnegative real numbers, but we replace it by the set $\mathbb{R}_+$ of nonnegative elements in a computable subfield $\mathbb{R}$, as we did for timed automata. A hybrid automaton is given by a tuple $(V_D, Loc, Lab, Act, Inv, Edg)$ where:

- V_D is a finite set of data variables, each ranging over a given data sort, defining the data space Σ_D, that is, Σ_D is the set of sort-consistent valuations v of V_D.
- *Loc* is a finite set of *locations* (corresponding to "states" in untimed automata).
- The state space of a hybrid automaton is $Loc \times \Sigma_D$.
- *Lab* is a set of *synchronization labels*, including the label τ.
- *Act* is a labeling function that assigns to each location $l \in Loc$ a set Act_l of *activities*. An activity is a function from $\mathbb{R}_+$ to Σ_D. For each activity f in Act_l and each time value r there is an activity f^r in Act_l defined by $f^r(r') = f(r + r')$.
- *Inv* is a labeling function that assigns to each location $l \in Loc$ an *invariant* $Inv(l) \subseteq \Sigma_D$.
- *Edg* is a finite set of *transitions*. Each transition $e = (l, \mu, l', a)$ consists of a source location l, a target location l', a transition relation $\mu \subseteq \Sigma_D^2$, and a synchronization label a. For each location l there is a *stutter transition* (l, Id, l, τ) where $Id = \{(v, v) \mid v \in \Sigma_D\}$.

The hybrid automaton in Fig. 2.2 has two locations, *on* and *off*, one ($\mathbb{R}_+$-valued) data variable x denoting the temperature in the room, and two non-stutter transitions $(\mathit{off}, (\{x \mapsto m\}, \{x \mapsto m\}), \mathit{on}, \mathit{turn\_on})$ and $(\mathit{on}, (\{x \mapsto M\}, \{x \mapsto M\}), \mathit{off}, \mathit{turn\_off})$. For location *on*, $Inv(on) = \{\, \{x \mapsto r\} \mid r \in \mathbb{R}_+ \wedge r \leq M\}$ and Act_{on} is the set of functions $\{f_r : \mathbb{R}_+ \to (\{x\} \to \mathbb{R}) \mid r \in \mathbb{R}\}$ defined by $f_r(r') = \{x \mapsto r + 2r'\}$. The invariant and the activities of location *off* are entirely similar.

The state of a hybrid automaton can change in two ways: (1) by an instantaneous transition that changes the entire state according to the transition relation, or (2) by elapse of time that changes only the values of data variables in a continuous manner, according to the activities of the current location, where the state (l, v) evolves to $(l, f(r))$ in time r whenever f is an activity of location l such that $v = f(0)$. The system may stay at a location only if the invariant at that location remains true. The invariants of a hybrid automaton thus enforce the progress of the underlying discrete transition system. That is, some transition must be taken before the invariant of the location becomes false.

A run ϱ of a hybrid automaton is a finite or infinite sequence

$$\varrho : \ (l_0, v_0) \overset{a_0\, r_0}{\mapsto_{f_0}} (l_1, v_1) \overset{a_1\, r_1}{\mapsto_{f_1}} (l_2, v_2) \overset{a_2\, r_2}{\mapsto_{f_2}} \cdots$$

where $l_0, l_1, \ldots$ denote locations, $v_0, v_1, \ldots$ denote valuations of the variables V_D, $a_0, a_1, \ldots$ denote synchronization labels, $r_0, r_1, \ldots$ denote time values, and where each f_i is either the constant id or an activity in location l_i, such that the following conditions hold: For all i, $v_{i+1} \in Inv(l_{i+1})$; furthermore, whenever f_i is id, we have $r_i = 0$, $v_i \in Inv(l_i)$, and $(v_i, v_{i+1}) \in \mu_i$ for some $(l_i, \mu_i, l_{i+1}, a_i) \in Edg$; and whenever f_i is an activity we have $f_i(0) = v_i$, $f_i(r) \in Inv(l_i)$ for all $0 \leq r \leq r_i$, and $(f_i(r_i), v_{i+1}) \in \mu_i$ for some $(l_i, \mu_i, l_{i+1}, a_i) \in Edg$. The set of all such runs of a hybrid automaton A is denoted $Runs(A)$, and $Runs_{Inv}(A)$ denotes the subset of these runs where $v_0 \in Inv(l_0)$. This definition of a run is a slight modification of the one appearing in [ACH+95], in that it exhibits the transition used at each step, uses the special symbol id to allow a transition to be taken in zero time from a state v_j, satisfying $Inv(l_j)$, even when there is no time-advancing activity acting on the state, and adds the requirement $v_{i+1} \in Inv(l_{i+1})$ to ensure that the final state in a run satisfies the invariant of its location.

In the remainder of this section, we assume that the set V_D of data variables is ordered as a sequence $x_1, \ldots, x_n$. Since the definition of hybrid automata is very general, we restrict our treatment to a subclass of hybrid automata satisfying some natural requirements. First of all, we require that the set of activities Act_l for a location l is generated by a finite set

$$ActGen_l = \{f_i^l : \Sigma_D \times \mathbb{R}_+ \to \Sigma_D \mid 1 \leq i \leq n_l\}$$

of computable functions, called activity generators, where each f_i^l satisfies the property

$$f_i^l(f_i^l(v, r), r') = f_i^l(v, r + r') \textbf{ if } f_i^l(v, 0) = v.$$

In the thermostat example, $ActGen_{on} = \{g\}$ and $ActGen_{off} = \{h\}$ for the functions g and h defined by $g(\{x \mapsto r\}, r') = \{x \mapsto r + 2r'\}$ and $h(\{x \mapsto r\}, r') = \{x \mapsto r - r'\}$.

The set Act_l of activities for a location l is generated from $ActGen_l$ as follows,

$$\begin{array}{rl} Act_l = \{f : \mathbb{R}_+ \to \Sigma_D \mid & (\exists f_i^l \in ActGen_l, v \in \Sigma_D, r \in \mathbb{R}_+) \\ & f_i^l(v, 0) = v \wedge f = \lambda x_r . f_i^l(v, r + x_r)\}. \end{array}$$

Furthermore, we require that for each location l and activity generator $f_i^l \in ActGen_l$, there is a computable function

$$max\_stay_{f_i^l} : \Sigma_D \to \mathbb{R}_+ \cup \{\infty\}$$

where $max\_stay_{f_i^l}(v)$ denotes the amount of time a system in state (l, v) can stay at location l performing the activity-function f_i^l, without violating the invariant of location l. We also require that there is a computable predicate

$$Inv : Loc \times \Sigma_D \to Bool$$

where $Inv(l, v)$ holds if and only if the state v does not violate the invariant of location l. Although not necessary, we also find it natural to assume that *all* τ-transitions are identity transitions, that is, $(l, \mu, l', \tau) \in Edg$ implies $l = l'$ and $\mu = Id$.

We furthermore require that each data type of A is computable. Before stating our last requirement about the transition of the hybrid automaton A, we explain some necessary details about its associated rewrite theory $\Psi_{HA}(A)$.

A hybrid automaton $A = (V_D, Loc, Lab, Act, Inv, Edg)$ with the variables ordered into $x_1, \ldots, x_n$ and satisfying the above requirements can be represented by a real-time rewrite theory $\Psi_{HA}(A) = ((\Sigma_A, E_A, L_A, R_A), \phi_A, \tau_A)$ as follows.

- The signature Σ_A contains a sort *Location* with a constant $l : \rightarrow \mathit{Location}$ for each $l \in \mathit{Loc}$, a sort s_i for each data sort s_i of the variable x_i, a sort *Valuation* with an n-ary operator $\langle \_, \dots, \_ \rangle : s_1 \dots s_n \rightarrow \mathit{Valuation}$, an operator $\_,\_ : \mathit{Location}\ \mathit{Valuation} \rightarrow \mathit{State}$, and an operator $\{\_\} : \mathit{State} \rightarrow \mathit{System}$. Furthermore, (Σ_A, E_A) contains finitary axiomatizations of the data types of the sorts $s_1, \dots, s_n$, and of the time domain, so that $\phi_A(\mathit{Time})$ specifies $\mathbb{R}_+$ and ϕ_A has an extension $\phi'_A : \mathit{LTIME}_\infty \rightarrow (\Sigma_A, E_A)$, of the sets of functions $f_i^l : \mathit{Valuation}\ \mathit{Time} \rightarrow \mathit{Valuation}$ and $\mathit{max\_stay}_{f_i^l} : \mathit{Valuation} \rightarrow \mathit{Time}_\infty$, and of the function $\mathit{Inv} : \mathit{Location}\ \mathit{Valuation} \rightarrow \mathit{Bool}$. These functions and data types can be given such finitary equational axiomatizations since they are assumed computable [BT87].

- The set L_A of labels is $(\mathit{Lab} \setminus \{\tau\}) \cup \{\mathit{tick}_{f_i^l} \mid l \in \mathit{Loc} \wedge f_i^l \in \mathit{ActGen}_l\}$, where $(\mathit{Lab} \setminus \{\tau\}) \cap \{\mathit{tick}_{f_i^l} \mid l \in \mathit{Loc} \wedge f_i^l \in \mathit{ActGen}_l\} = \emptyset$, and $\mathit{tick}_{f_i^l} \neq \mathit{tick}_{f_j^{l'}}$ if $f_i^l \neq f_j^{l'}$.

- Our last assumption about the hybrid automaton A is that its transitions can be specified by rewrite rules. That is, we require that each transition $(l, \mu, l', a) \in \mathit{Edg}$ can be expressed by a finite set of rewrite rules of the form $[a] : \{l, \langle t_1, \dots, t_n \rangle\} \longrightarrow \{l', \langle t'_1, \dots, t'_n \rangle\}$ **if** C, so that we have $(\{x_1 \mapsto u_1, \dots, x_n \mapsto u_n\}, \{x_1 \mapsto u'_1, \dots, x_n \mapsto u'_n\}) \in \mu$ if and only if $a(\dots) : \{l, \langle u_1, \dots, u_n \rangle\} \longrightarrow \{l', \langle u'_1, \dots, u'_n \rangle\}$ is a ground one-step rewrite using these rules. Then, the set R_A of rewrite rules in $\Psi_{HA}(A)$ and its associated duration assignment function τ_A contains the corresponding rule(s)

$$[a] : \ \{l, \langle t_1, \dots, t_n \rangle\} \longrightarrow \{l', \langle t'_1, \dots, t'_n \rangle\} \ \textbf{if} \ C \wedge \mathit{Inv}(l', \langle t'_1, \dots, t'_n \rangle) = \mathit{true}$$

  for each transition (l, μ, l', a) in Edg with $a \neq \tau$, where the last conjunct in the condition must be added to the translation of the (underlying "untimed") transition to ensure that the resulting state satisfies the invariant of location l'. The tick rules of the system associate to each location l and each activity generator f_i^l a rewrite rule of the form

$$[\mathit{tick}_{f_i^l}] : \{l, V\} \xrightarrow{x_r} \{l, f_i^l(V, x_r)\} \ \textbf{if} \ x_r \leq \mathit{max\_stay}_{f_i^l}(V) \text{ and } f_i^l(V, 0) = V,$$

  where V is a variable of sort *Valuation*.

A rewriting logic translation of the thermostat in Fig. 2.2 may instantiate this translation schema in the following way. The domains $\mathbb{R}_+$ and $\mathbb{R}$ are interpreted by sorts *Time* and *Temp* with the necessary functions, and the rest of the theory is given by the following declarations:

sorts *Location Valuation State System*
ops $\mathit{on}\ \mathit{off} : \rightarrow \mathit{Location}$
op $\langle \_ \rangle : \mathit{Temp} \longrightarrow \mathit{Valuation}$
op $\_,\_ : \mathit{Location}\ \ \mathit{Valuation} \rightarrow \mathit{State}$
op $\{\_\} : \mathit{State} \rightarrow \mathit{System}$
ops $g\ h : \mathit{Valuation}\ \ \mathit{Time} \rightarrow \mathit{Valuation}$
op $\mathit{Inv} : \mathit{Location}\ \ \mathit{Valuation} \rightarrow \mathit{Bool}$
ops $\mathit{max\_stay}_{\mathit{on}}\ \ \mathit{max\_stay}_{\mathit{off}} : \mathit{Valuation} \rightarrow \mathit{Time}_\infty$
var $x : \mathit{Temp}$ **var** $y_r : \mathit{Time}$ **var** $v : \mathit{Valuation}$
eq $\mathit{Inv}(\mathit{on}, \langle x \rangle) = x \leq M$
eq $\mathit{Inv}(\mathit{off}, \langle x \rangle) = x \geq m$
eq $\mathit{max\_stay}_{\mathit{on}}(\langle x \rangle) = (M - x)/2$

$$
\begin{array}{ll}
\textbf{eq} & \mathit{max\_stay}_{\mathit{off}}(\langle x\rangle) = x - m \\
\textbf{eq} & g(\langle x\rangle, y_r) = \langle x + 2 * y_r\rangle \\
\textbf{eq} & h(\langle x\rangle, y_r) = \langle x - y_r\rangle \\
\textbf{crl} & [\mathit{turn\_on}] :\ \{\mathit{off}, \langle m\rangle\} \longrightarrow \{\mathit{on}, \langle m\rangle\} \textbf{ if } \mathit{Inv}(\mathit{on}, \langle m\rangle) \\
\textbf{crl} & [\mathit{turn\_off}] :\ \{\mathit{on}, \langle M\rangle\} \longrightarrow \{\mathit{off}, \langle M\rangle\} \textbf{ if } \mathit{Inv}(\mathit{on}, \langle M\rangle) \\
\textbf{crl} & [\mathit{tick}_{\mathit{on}}] :\ \{\mathit{on}, v\} \xrightarrow{y_r} \{\mathit{on}, g(v, y_r)\} \\
 & \qquad\qquad\qquad \textbf{if } y_r \leq \mathit{max\_stay}_{\mathit{on}}(v) \text{ and } g(v,0) == v \\
\textbf{crl} & [\mathit{tick}_{\mathit{off}}] :\ \{\mathit{off}, v\} \xrightarrow{y_r} \{\mathit{off}, h(v, y_r)\} \\
 & \qquad\qquad\qquad \textbf{if } y_r \leq \mathit{max\_stay}_{\mathit{off}}(v) \text{ and } h(v,0) == v
\end{array}
$$

An *HA-step* in $\Psi_{HA}(A)$ is intended to model a step in A and has one of the following forms:

- a ground rewrite of the form $\mathit{tick}_{f^l_k}(v, r); a(\overline{w}) : \{l, v\} \longrightarrow \{l', v'\}$, with $a \in \mathit{Lab} \setminus \tau$, which models a step $(l, v) \overset{a\ r}{\mapsto}_{(\lambda r'.f^l_k(v,r'))} (l', v')$ in A;
- a ground rewrite of the form $a(\dots) : t \longrightarrow t'$, which models a step $t \overset{a\ 0}{\mapsto}_{id} t'$ in A;
- a ground rewrite of the form $\mathit{tick}_{f^l_k}(t, r) : \{l, t\} \longrightarrow \{l, t'\}$, which models a step $(l, t) \overset{\tau\ r}{\mapsto}_{(\lambda r'.f^l_k(t,r'))} (l', t')$; or
- an identity rewrite $t : t \longrightarrow t$ for t a ground term of sort *System*, which models a step $t \overset{\tau\ 0}{\mapsto}_{id} t$ in A.

Since an activity may be generated by different activity generators, we *identify* the HA-steps $\mathit{tick}_{f^l_k}(u, r) : \{l, u\} \longrightarrow \{l, u'\}$ and $\mathit{tick}_{f^l_j}(u, r) : \{l, u\} \longrightarrow \{l, u'\}$ whenever $f^l_k(u, r') = f^l_j(u, r')$ for all r'. Furthermore, a transition a in A could be modeled by more than one rule in $\Psi_{HA}(A)$, although the choice of *which* a-rule to apply should not make two different a-rewrites from some t to t' into different steps. We therefore identify the HA-steps of the form $a(\overline{w}) : t \longrightarrow t'$ and $a(\overline{w'}) : t \longrightarrow t'$. These two identifications of HA-steps extend to composite HA-steps, making the HA-steps $\mathit{tick}_{f^l_j}; a(\overline{w}) : \{l, u\} \longrightarrow \{l', u'\}$ and $\mathit{tick}_{f^l_k}; a(\overline{w'}) : \{l, u\} \longrightarrow \{l', u'\}$ *equivalent* HA-steps if and only if $f^l_j(u, r') = f^l_k(u, r')$ for all r'.

An *HA-computation* in $\Psi_{HA}(A)$ is a finite or infinite sequence

$$\tilde{\varrho} :\ \{l_0, \langle t_{0_1}, \dots, t_{0_n}\rangle\} \xrightarrow{\alpha_0} \{l_1, \langle t_{1_1}, \dots, t_{1_n}\rangle\} \xrightarrow{\alpha_1} \{l_2, \langle t_{2_1}, \dots, t_{2_n}\rangle\} \xrightarrow{\alpha_2} \cdots$$

of HA-steps $\alpha_i : \{l_i, \langle t_{i_1}, \dots, t_{i_n}\rangle\} \longrightarrow \{l_{i+1}, \langle t_{i+1_1}, \dots, t_{i+1_n}\rangle\}$ in $\Psi_{HA}(A)$, where $\mathit{Inv}(l_0, \langle t_{0_1}, \dots, t_{0_n}\rangle) = \mathit{true}$. The set of all HA-computations in $\Psi_{HA}(A)$ (modulo HA-step equivalence) is denoted $\mathit{HA}\text{-}\mathcal{C}(\Psi_{HA}(A))$.

Theorem 14 *Let $A = ((x_1, \dots, x_n), \mathit{Loc}, \mathit{Lab}, \mathit{Act}, \mathit{Inv}, \mathit{Edg})$ be a hybrid automaton satisfying our additional requirements. Then, there exists a bijective function $\tilde{(\_)} : \mathit{Runs}_{\mathit{Inv}}(A) \to \mathit{HA}\text{-}\mathcal{C}(\Psi_{HA}(A))$ which takes each run of the form ϱ in $\mathit{Runs}_{\mathit{Inv}}(A)$ into an HA-computation of the form $\tilde{\varrho}$ where, for each appropriate i and each $j \in \{1, \dots, n\}$,*

- $t_{i_j} = v_i(x_j)$,
- $\tau_A(\alpha_i) = r_i$,

- $\alpha_i = tick_{f_k^{l_i}}(\langle t_{i_1}, \ldots, t_{i_n}\rangle, r_i); a_i(\overline{w})$ *if* $a_i \neq \tau$ *and* $f_i \neq id$*, for some* $\overline{w}$ *and some* $f_k^{l_i} \in ActGen_{l_i}$ *satisfying* $f_k^{l_i}(v_i, r') = f_i(r')$ *for all* r'*,*
- $\alpha_i = a_i(\overline{w})$ *for some* $\overline{w}$ *if* $a_i \neq \tau$ *and* $f_i = id$*,*
- $\alpha_i = tick_{f_k^{l_i}}(\langle t_{i_1}, \ldots, t_{i_n}\rangle, r_i)$ *for some* $f_k^{l_i} \in ActGen_{l_i}$ *found as above, if* $a_i = \tau$ *and* $f_i \neq id$*,*
- α_i *is the identity step if* $a_i = \tau$ *and* $f_i = id$*.*

Furthermore, each computation in $\Psi_{HA}(A)$ *starting in a state satisfying the invariant of its location, and each prefix of such a computation, corresponds to (at least) one HA-computation in* $HA\text{-}\mathcal{C}(\Psi_{HA}(A))$*.*

Proof sketch: It is easy to see that $\tilde{(\_)}$ induces a bijection between the set of initial states in $Runs_{Inv}(A)$ and the set of initial states of HA-computations. The theorem and definition of HA-steps indicate how each step in the run ϱ is in one-to-one-correspondence with an HA-step in $\tilde{\varrho}$. That, say, $(l, v) \overset{a\ r}{\mapsto}_f (l', v')$ is mapped into a *unique* HA-step $tick_{f_k^l}(v, r); a(\overline{w}) : \{l, v\} \longrightarrow \{l', v'\}$ modulo the equivalence on HA-steps follows from the key property that $f \in Act_l$ implies that there is an activity generator f_k^l such that $f(r) = f_k^l(f(0), r)$ for all r. Conversely, an application $tick^l_{f_k^l}(v, r)$ is the image under $\tilde{(\_)}$ of a step $(l, v) \overset{\tau\ r}{\mapsto}_f (l, f(r))$ in A. The key property here is that the condition of the tick rule implies $f_k^l(v, 0) = v$ for the state $\{l, v\}$ being rewritten, and by the definition of the relationships between activities and activity generators, there is an activity $f \in Act_l$ defined by $f(r') = f_k^l(v, r')$ for all r'. Uniqueness follows directly from the definitions.

Each computation (see Section 2.1.2), or prefix thereof, in $\Psi_{HA}(A)$, which starts with an initial state of sort *System* and where the initial valuation satisfies the invariant of the state's location, is trivially an HA-computation, since each one-step rewrite is also an HA-step.

2.2.3 Timed Transition Systems

A *timed transition system* (TTS) [MP93] is a transition system where each transition is equipped with a lower bound, denoting the shortest time the transition must be continuously enabled before being taken, and an upper bound, denoting the longest time the transition can be continuously enabled without being taken. Formally, a TTS is a tuple (X, T, l, u) where X is a finite set $\{x_1, \ldots, x_n\}$ of data variables (with each x_i ranging over a data sort s_i) defining the state space Σ_D of all sort-consistent valuations v of the variables, T a finite set of transitions $a : \Sigma_D \to 2^{\Sigma_D}$, and l and u are functions $l : T \to \mathbb{R}_+$ and $u : T \to \mathbb{R}_+ \cup \{\infty\}$, which assign to each transition a a "lower bound" l_a and an "upper bound" u_a where $0 \leq l_a \leq u_a$.[7] All transitions are assumed to be *self-disabling*, i.e., $v' \in a(v)$ implies $a(v') = \emptyset$ for all $a \in T$ and $v, v' \in \Sigma_D$.

For example, a system where a variable x ranges over $\{a, b\}$, and changes from a to b at some time $r \in [5, 10]$ after changing from b to a, and then changes back to a at some time $r' \in [4, 8]$ thereafter, can be modeled by a TTS $\mathcal{T}_{ab}$ with transitions $\{\{x \mapsto a\} \mapsto \{\{x \mapsto b\}\}, \{x \mapsto b\} \mapsto \emptyset\}$, having lower bound 5 and upper bound 10, and $\{\{x \mapsto b\} \mapsto \{\{x \mapsto a\}\}, \{x \mapsto a\} \mapsto \emptyset\}$, having lower bound 4 and upper bound 8.

[7] We ignore that a TTS is also equipped with a set of *initial* states.

A *computation* of a TTS $\mathcal{T} = (X, T, l, u)$ is an infinite sequence

$$\sigma : (v_0, r_0) \to (v_1, r_1) \to (v_2, r_2) \to \cdots$$

of pairs of valuations v_i and time values r_i (with $r_0 = 0$) such that for each $i \geq 0$, either $v_i = v_{i+1} \wedge r_i < r_{i+1}$, or $r_i = r_{i+1} \wedge v_{i+1} \in a(v_i)$ holds for some transition a in T which has been enabled uninterruptedly for at least time l_a, that is, there is a $j \leq i$ such that $r_i - r_j \geq l_a$ and a is enabled on all the states $v_j, v_{j+1}, \ldots, v_i$. Furthermore, if a transition b is enabled on a state v_i, it must be disabled on some state v_{i+k} with $r_{i+k} \leq r_i + u_b$. Finally, it is required that r_j grows beyond any bound as j increases. The set of all computations of a TTS $\mathcal{T}$ is denoted by $\mathit{Comps}(\mathcal{T})$.

Again, we let the time domain be a computable subfield of the nonnegative reals and assume that the underlying untimed transition system can be finitely specified in rewriting logic in such a way that all data types are computable, and each transition can be specified by a finite set of rewrite rules in a way that will be made more precise after giving some necessary information about the translation. We also assume that, for each transition a, it is possible to define a computable predicate $\mathit{enabled}_a$ such that $\mathit{enabled}_a(v)$ is true if and only if transition a is enabled on state v, i.e., $\mathit{enabled}_a(v) \Leftrightarrow \exists v' \, (v' \in a(v))$.

We model a TTS in rewriting logic by representing the system state by a term $\{\langle t_1, \ldots, t_n\rangle, c_1, \ldots, c_m\}$, where the term $\langle t_1, \ldots, t_n\rangle$ represents a valuation $v = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ of the transition system variables, and where each c_i is a "clock" value which is nil if transition a_i is not enabled on the state, and is r_i if the transition a_i has been enabled continuously for time r_i. More precisely, let $\mathcal{T}$ be a timed transition system $(\{x_1, \ldots, x_n\}, T, l, u)$ satisfying the above requirements and where the set T of timed transitions is ordered as a sequence $a_1, \ldots, a_m$. Then, its rewriting logic translation $\Psi_{TTS}(\mathcal{T}) = ((\Sigma_\mathcal{T}, E_\mathcal{T}, L_\mathcal{T}, R_\mathcal{T}), \phi_\mathcal{T}, \tau_\mathcal{T})$ is the following real-time rewrite theory:

- $(\Sigma_\mathcal{T}, E_\mathcal{T})$ contains the specification of the sorts $s_1, \ldots, s_n$ of the variables in $\mathcal{T}$ and of the time domain such that the equational theory morphism $\phi_\mathcal{T} : \mathit{TIME} \to (\Sigma_\mathcal{T}, E_\mathcal{T})$ can be extended to an equational theory morphism $\phi'_\mathcal{T} : \mathit{TIME}_\infty \to (\Sigma_\mathcal{T}, E_\mathcal{T})$. In addition, $\Sigma_\mathcal{T}$ contains the sorts *Valuation*, *State*, *System*, and *Time*?, a subsort declaration $\mathit{Time} \leq \mathit{Time}?$, a constant $\mathit{nil} : \to \mathit{Time}?$, an n-ary operator $\langle \_, \ldots, \_\rangle : s_1 \ \ldots \ s_n \to \mathit{Valuation}$, an $(m+1)$-ary operator $\_, \ldots, \_ : \mathit{Valuation} \ \mathit{Time}? \ \ldots \ \mathit{Time}? \to \mathit{State}$, an operator $\{\_\} : \mathit{State} \to \mathit{System}$, and declarations $\_ \leq \_ : \mathit{Time}? \ \mathit{Time}_\infty \to \mathit{Bool}$ and $\_ + \_ : \mathit{Time}? \ \mathit{Time} \to \mathit{Time}?$. We also add to $E_\mathcal{T}$ the axioms $\mathit{nil} + x_r = \mathit{nil}$ for $x_r : \mathit{Time}$, and $\mathit{nil} \leq y_r = \mathit{true}$ for $y_r : \mathit{Time}_\infty$. Finally, $(\Sigma_\mathcal{T}, E_\mathcal{T})$ axiomatizes the predicate $\mathit{enabled}_a : \mathit{Valuation} \to \mathit{Bool}$ for each transition a in T.

- For the set $L_\mathcal{T}$ of labels, it is enough to choose the set $\{\mathit{trans}, \mathit{tick}\}$, since computations in a TTS are independent of the choice of transitions taken.

- Before giving the rules $R_\mathcal{T}$, we make more precise our assumption about being able to represent the transitions in T as rewrite rules by assuming that each (untimed) transition $a_i \in T$ can be modeled by a finite set of rewrite rules of the form $\langle t_1, \ldots, t_n\rangle \longrightarrow \langle t'_1, \ldots, t'_n\rangle$ **if** C, such that $\{x_1 \mapsto w'_1, \ldots, x_n \mapsto w'_n\} \in a_i(\{x_1 \mapsto w_1, \ldots, x_n \mapsto w_n\})$ holds if and only if $\langle w_1, \ldots, w_n\rangle \longrightarrow \langle w'_1, \ldots, w'_n\rangle$ is a one-step ground rewrite using one of these rules. The set $R_\mathcal{T}$ of rules together with its duration assignment $\tau_\mathcal{T}$ can then be given as follows. Each transition a_i is modeled in the *timed* system $\Psi_{TTS}(\mathcal{T})$ by the corresponding instantaneous rewrite rule(s)

$$[\mathit{trans}] : \ \{\langle t_1, \ldots, t_n\rangle, c_1, \ldots, c_m\} \longrightarrow \ \{\langle t'_1, \ldots, t'_n\rangle, c'_1, \ldots, c'_m\} \ \textbf{if } C \wedge (l_{a_i} \leq c_i)$$

where $\{c_1, \ldots, c_m\}$ is a set of m distinct variables of sort *Time?* not occurring in any t_i or t'_i, and each c'_j is an abbreviation for the expression

$$\textbf{if } \mathrm{not}(enabled_{a_j}(\langle t'_1, \ldots, t'_n\rangle)) \textbf{ then } nil \textbf{ else if } c_j == nil \textbf{ then } 0 \textbf{ else } c_j \textbf{ fi fi}.$$

In addition, $R_{\mathcal{T}}$ contains the following tick rule, which ensures, for each transition a_i, that time will not elapse past the moment when a_i would have been enabled for time u_{a_i}:

$$\begin{array}{rcl} [tick] : \{V, c_1, \ldots, c_m\} & \xrightarrow{x_r} & \{V, c_1 + x_r, \ldots, c_m + x_r\} \\ & & \textbf{if } 0 < x_r \wedge \bigwedge_i (c_i + x_r \leq u_{a_i}), \end{array}$$

for V a variable of sort *Valuation* and x_r a variable of sort *Time*.

The real-time rewrite theory $\Psi_{TTS}(\mathcal{T}_{ab})$ of the example system $\mathcal{T}_{ab}$ consists of the following rules, which are simplified in that we do not use the *enabled*-predicate explicitly, and where the transition from a to b is the first transition in the ordering used in the construction of $\Psi_{TTS}(\mathcal{T}_{ab})$:

$$\begin{array}{lrcl} [trans] : & \{\langle a\rangle, x, y\} & \longrightarrow & \{\langle b\rangle, nil, 0\} \textbf{ if } 5 \leq x \\ [trans] : & \{\langle b\rangle, x, y\} & \longrightarrow & \{\langle a\rangle, 0, nil\} \textbf{ if } 4 \leq y \\ [tick] : & \{v, x, y\} & \xrightarrow{x_r} & \{v, x + x_r, y + x_r\} \\ & & & \textbf{if } 0 < x_r \wedge x + x_r \leq 10 \wedge y + x_r \leq 8 \end{array}$$

for variables x, y of sort *Time?*, v of sort *Valuation*, and x_r of sort *Time*.

A *TTS-computation* in $\Psi_{TTS}(\mathcal{T})$ is an infinite sequence

$$\tilde{\sigma} : \{\langle t_{0_1}, \ldots, t_{0_n}\rangle, c_{0_1}, \ldots, c_{0_m}\} \xrightarrow{\alpha_1} \{\langle t_{1_1}, \ldots, t_{1_n}\rangle, c_{1_1}, \ldots, c_{1_m}\} \xrightarrow{\alpha_1} \cdots$$

of one-step ground rewrites in $\Psi_{TTS}(\mathcal{T})$ where the sum $\Sigma_{i=1}^{j} \tau_{\mathcal{T}}(\alpha_i)$ grows beyond any bound as j increases, and where c_{0_i} is 0 if transition a_i is enabled on the valuation $\{x_1 \mapsto t_{0_1}, \ldots, x_n \mapsto t_{0_n}\}$ and *nil* otherwise. The set of TTS-computations in $\Psi_{TTS}(\mathcal{T})$ is denoted by $TTS\text{-}\mathcal{C}(\Psi_{TTS}(\mathcal{T}))$.

Theorem 15 *Let $\mathcal{T} = (\{x_1, \ldots, x_n\}, \{a_1, \ldots, a_m\}, l, u)$ be a timed transition system satisfying the additional requirements in this section. Then, there is a bijective function $\tilde{(\_)} : Comps(\mathcal{T}) \rightarrow TTS\text{-}\mathcal{C}(\Psi_{TTS}(\mathcal{T}))$ taking a computation of $\mathcal{T}$ of the form σ defined above into a TTS-computation of the form $\tilde{\sigma}$ with $\tau_{\mathcal{T}}(\alpha_i) = r_i - r_{i-1}$ for each proof term α_i, and $t_{i_j} = v_i(x_j)$ for each $i \geq 0$ and $j \in \{1, \ldots, n\}$.*

Proof sketch: Clearly, $\tilde{(\_)}$ induces a bijection between the initial states of $Comps(\mathcal{T})$ and the initial states of $TTS\text{-}\mathcal{C}(\Psi_{TTS}(\mathcal{T}))$. To prove that $\tilde{(\_)}$ gives a one-to-one correspondence between prefixes of computations and TTS-computations having $k+1$ steps, given such a correspondence for the first k steps, we observe that, if $c_{i_j} = nil$ in $\tilde{\sigma}$, then a_j is not enabled on v_i in the computation σ, and that if $c_{i_j} \neq nil$, then there is an $l \leq i$ with $r_l + c_{i_j} = r_i$ such that a_j is enabled on $v_l, v_{l+1}, \ldots, v_i$, and either $l = 0$ or a_j is not enabled on v_{l-1}.

2.2.4 Phase Transition Systems

Phase transition systems (PTSs) [MP93] generalize timed transition systems to a model suitable for specifying hybrid systems[8]. We give here only a brief overview of a representation of PTSs in rewriting logic. The reference [ÖM96] gives more details about the translation. Intuitively, the PTS model extends the TTS model by letting time act on each valuation according to a function

$$\delta : \Sigma_D \times \mathbb{R}_+ \to \Sigma_D$$

where $\delta(v, r)$ denotes the state of the PTS after time has acted on a system in state v for time r. Furthermore, time cannot elapse past a moment when the enabling condition of a transition changes. The special case of TTS can then be characterized as those PTSs such that δ is the projection function π_{Σ_D}, that is, when time acts trivially on the states. Since the action of time can change the enabling of transitions, we assume that there is a computable function

$$\mathit{enabling\_change} : \Sigma_D \to \mathbb{R}_+ \cup \{\infty\}$$

which takes a state as argument and gives the maximum time the system can proceed without changing the enabling of a transition.

The global state and the (instantaneous) transitions of a PTS are modeled in rewriting logic as for the TTS case. The functions δ and *enabling_change* are defined on terms of the sort *Valuation*. The following tick rule ensures, in addition to the TTS requirement, that time cannot elapse beyond the latest moment when a transition must be taken, that all state components are updated according to their continuous behavior, and that the corresponding clocks are updated when an enabling condition changes:

$$\begin{array}{ll} [\mathit{tick}]: & \{V, c_1, \ldots, c_m\} \xrightarrow{x_r} \{\delta(V, x_r), c'_1, \ldots, c'_m\} \\ & \textbf{if } \bigwedge_i (c_i + x_r \leq u_{a_i}) \wedge (0 < x_r \leq \mathit{enabling\_change}(V)) \end{array}$$

for V, $c_1, \ldots, c_n$, and x_r distinct variables of appropriate sorts, and where each c'_j abbreviates

$$\begin{array}{l} \textbf{if } \mathit{not}(\mathit{enabled}_{a_j}(\delta(V, x_r)) \textbf{ then } \mathit{nil} \\ \textbf{else if } c_j == \mathit{nil} \textbf{ then } 0 \textbf{ else } c_j + x_r \textbf{ fi fi}. \end{array}$$

2.2.5 Object-Oriented Real-Time Systems

In this section we specialize to real-time systems the representation of concurrent objects in rewriting logic proposed in [Mes93]. In a concurrent object-oriented system, the concurrent state, which is usually called a *configuration*, has typically the structure of a *multiset* made up of objects and messages. Therefore, we can view configurations as built up by a binary multiset union operator, which we can represent with empty syntax as

$$\_\_ : \mathit{Configuration}\ \mathit{Configuration} \to \mathit{Configuration}\ \ [\mathit{assoc}\ \ \mathit{comm}\ \ \mathit{id}:\ \mathit{none}]$$

[8]Note that the expression *phase transition system* is sometimes used for the hybrid systems extension of the *clocked transition system* [MP96] model. We do not discuss clocked transition systems in this thesis.

where the multiset union operator $\_\_$ is declared to satisfy the structural laws of associativity and commutativity, and to have identity *none*. The subsort declaration

$$\mathit{Object}\ \ \mathit{Msg} \leq \mathit{Configuration}$$

states that objects and messages are singleton multiset configurations, so that more complex configurations are generated from them by multiset union. A sort *ObjConfiguration* denoting configurations without messages can be obtained by adding the subsort declaration

$$\mathit{Object} \leq \mathit{ObjConfiguration} \leq \mathit{Configuration}$$

and the operator declaration

$$\begin{array}{lll} \_\_ : \mathit{ObjConfiguration}\ \mathit{ObjConfiguration} & \to & \mathit{ObjConfiguration} \\ & & [\mathit{assoc}\ \ \mathit{comm}\ \ \mathit{id}\colon\ \mathit{none}]. \end{array}$$

Objects are terms (of sort *Object*) of the form

$$\langle O : C \mid \mathit{att}_1 : \mathit{val}_1, \ldots, \mathit{att}_n : \mathit{val}_n \rangle$$

denoting an object named O, where O belongs to a sort *Oid* of *object identifiers*, of class C in a state having values $\mathit{val}_1, \ldots, \mathit{val}_n$ for the attributes $\mathit{att}_1, \ldots, \mathit{att}_n$, respectively.

The dynamic behavior of concurrent object systems is axiomatized by specifying each of its concurrent transition patterns by a rewrite rule. For example, the rule

$$\begin{array}{lll} [l] : & m(O, w) & \langle O : C \mid \mathit{att}_1 : x,\ \mathit{att}_2 : y,\ \mathit{att}_3 : z \rangle \longrightarrow \\ & & \langle O : C \mid \mathit{att}_1 : x + w,\ \mathit{att}_2 : y,\ \mathit{att}_3 : z \rangle\ m'(y, x + w) \end{array}$$

defines a (family of) transition(s) in which a message m having arguments O and w is consumed by an object O of class C, with the effect of altering the attribute att_1 of the object and of generating a new message $m'(y, x + w)$. By convention, attributes, such as att_3 in our example, whose values do not change and do not affect the next state of other attributes need not be mentioned in a rule. Attributes like att_2 whose values influence the next state of other attributes or the values in messages, but are themselves unchanged, may be omitted from righthand sides of the rules. Thus the above rule could also be written

$$[l] :\ m(O, w)\,\langle O : C \mid \mathit{att}_1 : x,\ \mathit{att}_2 : y \rangle \longrightarrow \langle O : C \mid \mathit{att}_1 : x + w \rangle\ m'(y, x + w).$$

Real-time object-oriented systems can be specified by means of real-time rewrite theories by extending this setting with a sort *System* and an operator

$$\{\_\} : \mathit{Configuration} \to \mathit{System}.$$

Therefore, tick rules are *global* transitions that simultaneously advance the time for all objects in the configuration. However, the system may still exhibit concurrency in its *local* transitions, which may occur between tick applications. We illustrate this style of specification of real-time object-oriented systems with a simple example.

Example: A Single-Thermostat System

A *single-thermostat system* consists of a thermostat object and zero or more "user" objects. A user may request a new desired temperature at any time by sending a message to the thermostat, which has to provide a temperature which is within 5 degrees of the user's desire, whenever this is possible. The thermostat otherwise works as the one modeled in Fig. 2.2, in that the temperature increases by 2 degrees per time unit when the heater is turned on, and decreases by 1 degree per time unit when the heater is turned off.

We assume that the specification includes a specification of *Time*, which satisfies the theory *TIME*, and a sort *Temp* denoting possible temperature values together with all the necessary operations. A sort *OnOff* contains the constants *on* and *off*, describing the state of the heater associated with the thermostat. A thermostat object has attributes *curr_temp* and *desired_temp* of sort *Temp*, denoting the current and desired temperatures, as well as an attribute *heater*, denoting the state of its heater. A user object is an object with an empty set of attributes.

In the following, let U and TS be variables of the sort *Oid* of object names, let x_r be a variable of sort *Time*, let y and z be variables of sort *Temp*, and let OC be a variable of the sort *ObjConfiguration* of message-less configurations.

At any time, a user may request a new desired temperature:

$$[\mathit{new\_temp}]: \ \langle U : \mathit{User}\rangle \longrightarrow \langle U : \mathit{User}\rangle(\mathit{set\_temp}(y)).$$

The thermostat object should treat such a message by recording the new desired temperature (followed by the changing of the heater state if necessary):

$$\begin{array}{ll}[\mathit{treat\_request}]: & (\mathit{set\_temp}(y))\langle TS : \mathit{Thermostat} \mid \mathit{desired\_temp}: z\rangle \longrightarrow \\ & \langle TS : \mathit{Thermostat} \mid \mathit{desired\_temp}: y\rangle.\end{array}$$

The thermostat must turn off/on the heater, either when time has acted on a system such that the current temperature is exactly the desired temperature plus/minus 5 degrees, or when the system must change due to the adjustment of the desired temperature, in which case the current temperature may be more than 5 degrees off the desired temperature:

$$\begin{array}{ll}[\mathit{on}]: & \langle TS : \mathit{Thermostat} \mid \mathit{curr\_temp}: y, \mathit{desired\_temp}: z, \mathit{heater}: \mathit{off}\rangle \longrightarrow \\ & \langle TS : \mathit{Thermostat} \mid \mathit{heater}: \mathit{on}\rangle \textbf{ if } y \leq z - 5 \\ [\mathit{off}]: & \langle TS : \mathit{Thermostat} \mid \mathit{curr\_temp}: y, \mathit{desired\_temp}: z, \mathit{heater}: \mathit{on}\rangle \longrightarrow \\ & \langle TS : \mathit{Thermostat} \mid \mathit{heater}: \mathit{off}\rangle \textbf{ if } y \geq z + 5.\end{array}$$

The following tick rules model the *effect* of time elapse on a system and ensure that:

1. time elapse can "stop" at any moment, reflecting the fact that the rule *new_temp* could be applied at any time,
2. time does not elapse past the moments when the heater state should be changed, and
3. time does not elapse while there are any messages in the system (i.e., the requested temperature should be recorded at the time when it is requested by a user); this is accomplished by the use of the variable OC of sort *ObjConfiguration*.

$[tick_{on}]$:
$\{\langle TS : Thermostat \mid curr\_temp\colon y,\ desired\_temp\colon z, heater\colon on\rangle\ OC\} \xrightarrow{x_r}$
$\{\langle TS : Thermostat \mid curr\_temp\colon y + x_r + x_r\rangle\ OC\}$ **if** $y + x_r + x_r \leq z + 5$

$[tick_{off}]$:
$\{\langle TS : Thermostat \mid curr\_temp\colon y,\ desired\_temp\colon z, heater\colon off\rangle\ OC\} \xrightarrow{x_r}$
$\{\langle TS : Thermostat \mid curr\_temp\colon y - x_r\rangle\ OC\}$ **if** $y - x_r \geq z - 5$.

The specification will work as expected, provided that the initial state contains exactly one thermostat object. A specification of a many-thermostat system is given in Section 2.2.5.

Object-Oriented δ-Systems

The elapse of time affects one (functional) attribute in the single-thermostat system described above. The tick rules could therefore be given by specifying directly the effect of time on that attribute. However, in more general object-oriented systems there can be an unbounded number of objects in a configuration which are affected by the elapse of time, in which case a finite number of tick rules cannot specify the elapse of time directly on the functional attributes. A simple solution is to use a function δ denoting the action of time on a configuration. The important class of real-time object-oriented systems where the objects have only functional attributes provides an instance of multiset distributed systems. Therefore, we can use the techniques described in Section 2.1.5 to circumvent lack of coherence problems w.r.t. δ without sacrificing concurrency. The following declarations should be added to the general framework for specifying object-oriented real-time systems given above, with $\{\_\}$ redefined as stated below.

sorts $\quad$ *Tokens* $\ *\ $*Configuration*
subsorts *Tokens Configuration* $\leq *$*Configuration*
op $\quad * : \to$ *Tokens*
op $\quad \_\_ :$ *Tokens Tokens* $\to$ *Tokens* [*assoc comm id*: *none*]
op $\quad \_\_ : *$*Configuration* $*$*Configuration* $\to *$*Configuration*
[*assoc comm id*: *none*]
op $\quad \{\_\} : *$*Configuration* $\to$ *System*
op $\quad \delta :$ *Configuration Time* $\to$ *Configuration*.

The tick rules are of the form (‡) in Section 2.1.5 with the sort *State* specialized to *Configuration*. Each instantaneous rule should have the form[9]

$$[l] : *\, t \longrightarrow * \cdots *\ t',$$

for t, t' terms of sort *Configuration*, and where the number of tokens $*$ in the righthand side should equal one plus the number of objects created by the rule, minus the number of objects deleted by the rule. The initial state of a system should be of the form $\{* \cdots *\ u\}$, where the number of tokens equals the number of objects in the term u of sort *Configuration*.

[9]In systems where the number of objects created by a rule application depends on the state, the condition on the form of the rules could be relaxed so that $* \cdots *$ can be given by a term of sort *Tokens*, computing the number of tokens as a function of the state.

Distribution over configurations. An operator δ acting on configurations provides, as we have just seen, a natural way of expressing the action of time on object systems where the number of objects in a configuration upon which time has an effect is unbounded. In these cases, δ should typically distribute over the elements in the configuration as modeled by the following axioms:

$$\begin{array}{l} \delta(none, x_r) = none \\ \delta(C\ C', x_r) = \delta(C, x_r)\ \delta(C', x_r) \textbf{ if } C \neq none \text{ and } C' \neq none \end{array}$$

(for C, C' variables of sort *Configuration*), to which the definition of the specific effect of time on single objects and on messages must be added to completely specify δ. The condition that C and C' be different from `none` ensures that the two equations above define a terminating rewrite system modulo associativity, commutativity, and identity (*none*) of the configuration union operator, when oriented from left to right.

In systems parameterized by $LTIME_\infty$ theories, a function *mte* giving the maximum time elapse for objects and messages can be extended to configurations by the axioms

$$\begin{array}{l} mte(none) = \infty \\ mte(C\ C') = \min(mte(C), mte(C')) \textbf{ if } C \neq none \text{ and } and C' \neq none. \end{array}$$

Example: A Multi-Thermostat System

A multi-thermostat system can have an arbitrary number of rooms, each equipped with a thermostat that works as in the single-thermostat system. Each user object is extended to contain a list of the thermostats to which it has access.

Let the specification be parameterized by the theory $LTIME_\infty$. Furthermore, let TS and U be variables of sort *Oid*, let S be a variable of a sort of sets of *Oid*s (where set union is denoted by juxtaposition), let C be a variable of sort *Configuration*, let T be a variable of sort *Tokens*, let x_r be a variable of sort *Time*, and let y and z be variables of sort *Temp*. Then, the function $\delta : Configuration\ Time \to Configuration$ denoting the action of time, and the function $mte : Configuration \to Time_\infty$ computing the maximum time elapse in a tick both distribute over configurations according to the equations described above, and are defined for singleton configurations as follows:

$$\begin{array}{l} \delta(set\_temp(A, y), x_r) = set\_temp(A, y) \\ \delta(\langle U : User \rangle, x_r) = \langle U : User \rangle \\ \delta(\langle TS : Thermostat \mid curr\_temp : y,\ heater : on \rangle, x_r) = \\ \quad \langle TS : Thermostat \mid curr\_temp : y + x_r + x_r,\ heater : on \rangle \\ \delta(\langle TS : Thermostat \mid curr\_temp : y,\ heater : off \rangle, x_r) = \\ \quad \langle TS : Thermostat \mid curr\_temp : y - x_r,\ heater : off \rangle \end{array}$$

$$\begin{array}{l} mte(set\_temp(A, y)) = 0 \\ mte(\langle U : User \rangle) = \infty \\ mte(\langle TS : Thermostat \mid curr\_temp : y,\ desired\_temp : z,\ heater : on \rangle) = \\ \quad ((z + 5) \dot{-} y)/2 \\ mte(\langle TS : Thermostat \mid curr\_temp : y,\ desired\_temp : z,\ heater : off \rangle) = \\ \quad y \dot{-} (z - 5). \end{array}$$

The rules of the system can then be given as follows:

$$
\begin{array}{ll}
[\mathit{new\_temp}]: & *\langle U : \mathit{User} \mid \mathit{thermostats}: \mathit{TS}\ S\rangle \longrightarrow \\
& *\langle U : \mathit{User}\rangle\ (\mathit{set\_temp}(\mathit{TS}, y)) \\
[\mathit{set\_temp}]: & *(\mathit{set\_temp}(\mathit{TS}, y))\ \langle \mathit{TS} : \mathit{Thermostat} \mid \mathit{desired\_temp}: z\rangle \longrightarrow \\
& *\langle \mathit{TS} : \mathit{Thermostat} \mid \mathit{desired\_temp}: y\rangle \\
[\mathit{turn\_on}]: & *\langle \mathit{TS}: \mathit{Thermostat} \mid \mathit{curr\_temp}: y,\ \mathit{desired\_temp}: z,\ \mathit{heater}: \mathit{off}\rangle \\
& \longrightarrow *\ \langle \mathit{TS} : \mathit{Thermostat} \mid \mathit{heater}: \mathit{on}\rangle\ \mathbf{if}\ y \leq z - 5 \\
[\mathit{turn\_off}]: & *\langle \mathit{TS}: \mathit{Thermostat} \mid \mathit{curr\_temp}: y,\ \mathit{desired\_temp}: z,\ \mathit{heater}: \mathit{on}\rangle \\
& \longrightarrow *\ \langle \mathit{TS} : \mathit{Thermostat} \mid \mathit{heater}: \mathit{off}\rangle\ \mathbf{if}\ y \geq z + 5 \\
[\mathit{tick}]: & \{T\ C\} \xrightarrow{x_r} \{T\ \delta(C, x_r)\}\ \mathbf{if}\ x_r \leq \mathit{mte}(C).
\end{array}
$$

2.2.6 Timed Petri Nets

A Petri net [Rei85] is usually presented as a set of *places* (each place representing a certain kind of *resource*), a disjoint set of *transitions*, and a relation of causality between them that associates to each transition the set of resources consumed and produced by its firing. Meseguer and Montanari recast this idea in an algebraic framework in [MM90], viewing the distributed states of the net, called *markings*, as multisets of places, and viewing the transitions as the arrows of an ordinary graph whose nodes are markings. In [Mes92, MÖS00] it has been shown how Petri net computations can be expressed by rewriting of markings.

Petri nets have been extended to model real-time systems in different ways (see e.g. [Aal93, MPT91, Han93]). Three of the most frequently used ways of adding time to Petri nets are the following [MPT91], from which other timed versions of Petri nets can be obtained either as special cases or by combining the extensions:

1. Each *transition* t has an associated time interval $[l_t, u_t]$. A transition fires as soon as it can, but the resulting tokens are delayed, that is, when a transition t fires, the resulting tokens are not visible in the system until after some time $r \in [l_t, u_t]$.

2. Each *place* p has a duration d_p. A token at place p cannot participate in a transition until it has been at p for at least time d_p.

3. Each transition t is associated with a time interval $[l_t, u_t]$, and the transition t cannot fire before it has been continuously enabled for at least time l_t. Also, the transition t cannot have been enabled continuously for more than time u_t without being taken.

We will cover the first two cases as special cases of the *interval timed colored Petri net* (ITCPN) model proposed by van der Aalst [Aal93]. The third case can be given a treatment similar to that of timed and phase transition systems in Sections 2.2.3 and 2.2.4.

ITCPNs appear in the context of *colored* Petri nets, where instead of having *atomic* places one has structured data. In this exposition, we abstract away from the colors of the tokens to concentrate on real-time features (see e.g. [MÖS00] for a treatment of colored nets in rewriting logic).

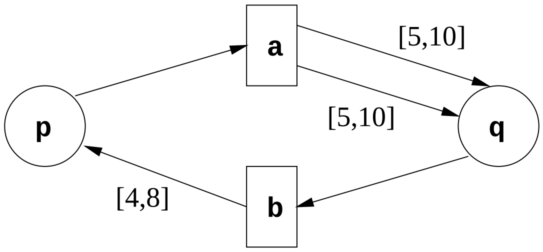

Figure 2.3: An interval timed Petri net.

Interval Timed Petri Nets

We give a definition of *interval timed Petri nets* (ITPNs) similar to that of interval timed colored Petri nets given in [Aal93]. The only differences between these two models are that ITPNs ignore the coloring of the tokens, and have a notion of concurrent firing of multisets of transitions.

In the ITPN model, as in the ITCPN model, we attach a *timestamp* to each token. This timestamp indicates the time when a token becomes available. The *enabling time* of a transition is the maximum timestamp of the tokens to be consumed. Transitions are *eager* to fire (i.e., they fire as soon as possible). Therefore the transition with the smallest enabling time will fire first. Firing is an atomic action, producing tokens with a timestamp equal to the firing time plus some *firing delay*, which is specified by a lower and upper bound of its possible values for each token produced by the transition.

The ITPN in Fig. 2.3 models a setting where each process performs transition a, followed by performing transition b within time 5 to 10, again followed by performing transition a time 4 to 8 thereafter, and so on. Furthermore, each process *forks* when performing transition a (this is modeled by having two arcs from a to q), and the created process has the same behavior as its parent process.

A *finite multiset* over a set S is a function m from S to the set $\mathbb{N}$ of natural numbers such that its support $\mathcal{S}(m) = \{s \in S \mid m(s) > 0\}$ is finite. We denote by $S^{\oplus}$ the set of finite multisets over S, and by $\emptyset_S$ the empty multiset over S, and define membership by $s \in m \Leftrightarrow m(s) > 0$, inclusion by $m \sqsubseteq m' \Leftrightarrow \forall s \in S\,(m(s) \leq m'(s))$, multiset union by $m \oplus m' = \lambda s \in S.m(s) + m'(s)$, and subtraction by $(m - s)(s) = \max(0, m(s) - 1)$ and $(m - s)(s') = m(s')$ for $s' \neq s$.

The time domain of ITCPNs is the set of nonnegative real numbers, but for executability purposes, we assume that the time domain of an ITPN is the nonnegative part $\mathbb{R}_+$ of a computable subfield of the reals. The set TI of all *time intervals* is the set $TI = \{\,[r_1, r_2] \mid r_1, r_2 \in \mathbb{R}_+ \wedge r_1 \leq r_2\}$.

Definition 16 *An* interval timed Petri net *(ITPN) is a tuple* $N = (\mathsf{P}, \mathsf{T}, \partial_0, \partial_1)$, *where* P *and* T *are finite sets of* places *and* transitions, *and* $\partial_0 : \mathsf{T} \to \mathsf{P}^{\oplus}$ *and* $\partial_1 : \mathsf{T} \to (\mathsf{P} \times TI)^{\oplus}$ *are functions (denoting, respectively, the tokens consumed and produced by a transition, together with the delay intervals of the produced tokens) such that* $\partial_0(t) \neq \emptyset_{\mathsf{P}}$ *for each* $t \in \mathsf{T}$.

In the following, let $N = (\mathsf{P}, \mathsf{T}, \partial_0, \partial_1)$ be an ITPN. The set of *markings* $\mathsf{MRK} = (\mathsf{P} \times \mathbb{R}_+)^{\oplus}$ is the set of all finite multisets of pairs (p, r) representing the presence of a token at place p with timestamp r. The function *places* : $\mathsf{MRK} \to \mathsf{P}^{\oplus}$ which removes the timestamps from a marking is defined by $places(m)(p) = \Sigma_{(p,r)\in\mathcal{S}(m)}\, m(p,r)$. The function $max : \mathsf{MRK} \setminus \{\emptyset_{(\mathsf{P}\times\mathbb{R}_+)}\} \to \mathbb{R}_+$ which finds the maximal timestamp in a non-empty marking is given by $max(m) = \max\{r \in \mathbb{R}_+ \mid \exists p\,((p,r) \in m)\}$. The earliest enabling time of any transition in a marking is given by a function $EET : \mathsf{MRK} \to \mathbb{R}_+ \cup \{\infty\}$ defined by

$$EET(m) = \min\{max(m') \mid \exists t \in \mathsf{T}, m' \in \mathsf{MRK}\,(m' \sqsubseteq m \wedge places(m') = \partial_0(t))\}$$

with $\min(\emptyset) = \infty$. The function $addToInt : (\mathsf{P} \times TI)^{\oplus} \times \mathbb{R}_+ \to (\mathsf{P} \times TI)^{\oplus}$ adds the second argument to all the intervals of the tokens, and is defined by $addToInt(m,r)(p,[r_1+r, r_2+r]) = \Sigma_{(p,[r_1,r_2])\in\mathcal{S}(m)}\, m(p,[r_1,r_2])$, and by $addToInt(m,r)(p,[r',r'']) = 0$ when $r' < r$. Finally, to relate multisets of tokens with timestamps with multisets of tokens with time intervals, we define the specialization relation $\lhd \subseteq \mathsf{MRK} \times (\mathsf{P} \times TI)^{\oplus}$, where $m \lhd m'$ holds if and only if each token in m corresponds to one token in m', such that they are in the same place and the timestamp of the token in m is in the interval of the corresponding token in m'. That is, $m \lhd m'$ if and only if either $(m = \emptyset_{\mathsf{P}\times\mathbb{R}_+} \wedge m' = \emptyset_{\mathsf{P}\times TI})$ or $\exists (p,r) \in m\ \ (\exists (p,[r_1,r_2]) \in m'\ \ (r_1 \leq r \leq r_2 \wedge (m - (p,r)) \lhd (m' - (p,[r_1,r_2]))))$.

An ITPN makes computational progress by applying transitions, thereby consuming and producing multisets of timestamped tokens. A finite number of transition firings occurring at the same time can be composed in parallel to form a *step*, which, therefore, is a nonempty finite multiset of transitions. Given an ITPN N, we denote by $m \xrightarrow{e} m'$ a step from marking m to marking m' by the concurrent firing of the transitions e, and define the step relation $\to\ \subseteq \mathsf{MRK} \times (\mathsf{T}^{\oplus} \setminus \{\emptyset_{\mathsf{T}}\}) \times \mathsf{MRK}$ to be all triples $m \xrightarrow{e} m'$ which can be generated by the following rules:

$$\frac{places(m) = \partial_0(t) \qquad max(m) = EET(m) \qquad m' \lhd addToInt(\partial_1(t), EET(m))}{m \xrightarrow{t} m'}$$

$$\frac{m \xrightarrow{e} m' \qquad m'' \in \mathsf{MRK} \qquad EET(m \oplus m'') = EET(m)}{(m \oplus m'') \xrightarrow{e} (m' \oplus m'')}$$

$$\frac{m_1 \xrightarrow{e_1} m_1' \qquad m_2 \xrightarrow{e_2} m_2' \qquad EET(m_1 \oplus m_2) = EET(m_1) = EET(m_2)}{(m_1 \oplus m_2) \xrightarrow{e_1 \oplus e_2} (m_1' \oplus m_2')}.$$

A *step sequence* in N is a (finite or infinite) sequence

$$\varsigma :\ \ m_0 \xrightarrow{e_1} m_1 \xrightarrow{e_2} m_2 \xrightarrow{e_3} \cdots$$

of steps $m_{i-1} \xrightarrow{e_i} m_i$, where each such step represents the simultaneous firing of the transitions e_i at time $EET(m_{i-1})$. The set of all step sequences of an ITPN N is denoted $\mathcal{C}_N^{\infty}$.

Timed Petri nets of type (1), where transitions have durations, that is, where all tokens produced by a transition have the same firing delay, can be seen as a special case of ITPNs as follows. A transition t with time interval $[l_t, u_t]$ which consumes the tokens m and produces the tokens m' can be simulated in an ITPN by adding a new place p_t, and having a transition t_1 which consumes the tokens m and produces one token at place p_t in some time in the interval $[l_t, u_t]$, and another transition t_2 which consumes one token from p_t and produces the tokens m' in zero time. Timed Petri nets of type (2), where each place has a duration, can be seen as a special case of ITPNs where each token produced at place p has firing delay d_p.

Specifying ITPNs as Real-Time Rewrite Theories.

The translation into rewriting logic of timed Petri nets is based on the rewriting logic representation of untimed Petri nets given in [Mes92], where the state of a Petri net is represented by a multiset of places called a *marking*—where if place p has multiplicity n we interpret this as the presence of n tokens in that place—and where the transitions correspond to rewrite rules on the corresponding multisets of pre- and post-places.

For the sake of simplicity of the rewriting logic representation of ITPNs, we choose not to carry the timestamps in the tokens at all times. Instead, a term $[p]$ of sort *VisibleMarking* represents a "visible" occurrence of a token at place p. A token that will become visible at place p in time r is represented by the term $dly(p, r)$ which has the sort *DelayedMarking* whenever $r \neq 0$.[10] A token with delay 0 is visible, i.e., $dly(p, 0) = [p]$. A *marking* is a multiset of these two forms of tokens, where multiset union is represented by juxtaposition. The sort *Marking* denotes the set of all markings. The function *mte* takes a term of sort *DelayedMarking* and returns the time that can pass until the next delayed token becomes available. The function δ models the effect of the passage of time on delayed tokens by decreasing their delays according to the time elapsed.

The rewriting logic translation of an ITPN $N = (\mathsf{P}, \mathsf{T}, \partial_0, \partial_1)$, with $\mathsf{P} = \{p_1, \ldots, p_n\}$, is a real-time rewrite theory $\Psi_{ITPN}(N) = ((\Sigma_N, E_N, L_N, R_N), \phi_N, \tau_N)$ where $\phi_N(\mathit{TIME})$ is an axiomatization of the time domain $\mathbb{R}_+$ which can be extended to $\phi'_N : \mathit{LTIME}_\infty \to (\Sigma_N, E_N)$, together with the following declarations and axioms:

sorts	*Place EmptyMarking VisibleMarking DelayedMarking Marking System*
subsorts	$\mathit{EmptyMarking} \leq \mathit{VisibleMarking}\ \ \mathit{DelayedMarking} \leq \mathit{Marking}$
ops	$p_1\ \ldots p_n \to \mathit{Place}$
op	$dly : \mathit{Place}\ \ \mathit{Time} \to \mathit{Marking}$
op	$\mathit{empty} :\ \to \mathit{EmptyMarking}$
op	$\_\,\_ : \mathit{Marking}\ \mathit{Marking} \to \mathit{Marking}\ \ [\mathit{assoc\ comm\ id} : \mathit{empty}]$
op	$\_\,\_ : \mathit{DelayedMarking}\ \mathit{DelayedMarking} \to \mathit{DelayedMarking}\ \ [\mathit{assoc\ comm\ id} : \mathit{empty}]$
op	$\_\,\_ : \mathit{VisibleMarking}\ \mathit{VisibleMarking} \to \mathit{VisibleMarking}\ \ [\mathit{assoc\ comm\ id} : \mathit{empty}]$
op	$\_\,\_ : \mathit{EmptyMarking}\ \mathit{EmptyMarking} \to \mathit{EmptyMarking}\ \ [\mathit{assoc\ comm\ id} : \mathit{empty}]$
op	$\{\_\} : \mathit{Marking} \to \mathit{System}$
op	$\delta : \mathit{DelayedMarking}\ \ \mathit{Time} \to \mathit{Marking}$
op	$\mathit{mte} : \mathit{DelayedMarking} \to \mathit{Time}_\infty$
vars	$DM\ DM' : \mathit{DelayedMarking}$ **var** $VM : \mathit{VisibleMarking}$
var	$P : \mathit{Place}$ **vars** $x_r\ y_r : \mathit{Time}$
cmb	$dly(P, x_r) : \mathit{DelayedMarking}$ **if** $x_r \neq 0$
eq	$dly(P, 0) = [P]$
eq	$\delta(\mathit{empty}, x_r) = \mathit{empty}$
eq	$\delta(dly(P, x_r), y_r) = dly(P, x_r \dot{-} y_r)$
ceq	$\delta(DM\ DM', x_r) = \delta(DM, x_r)\ \ \delta(DM', x_r)$

[10]We will see later that no interesting information about time is lost by this simplification, since the time when a firing of a transition occurs can always be extracted from the proof term.

if $DM \neq \mathit{empty}$ and $DM' \neq \mathit{empty}$

eq $\mathit{mte}(\mathit{empty}) = \infty$

ceq $\mathit{mte}(\mathit{dly}(P, x_r)) = x_r$ **if** $x_r \neq 0$

ceq $\mathit{mte}(DM\ DM') = \min(\mathit{mte}(DM), \mathit{mte}(DM'))$
if $DM \neq \mathit{empty}$ and $DM' \neq \mathit{empty}$

The set R_N of rules in $\Psi_{ITPN}(N)$ consists of a *lazy* tick rule modeling time elapse and, for each transition in N, a corresponding *eager* rule. A transition t is modeled by an instantaneous rule

$$\textbf{eager}\ [t] :\ m \longrightarrow \mathit{dly}(p_1, x_1) \ldots \mathit{dly}(p_n, x_n)\ \textbf{if}\ l_1 \leq x_1 \leq u_1 \wedge \ldots \wedge l_n \leq x_n \leq u_n$$

where m is a marking consisting of, for each p in P, $\partial_0(t)(p)$ occurrences of the term $[p]$, and where $\mathit{dly}(p_1, x_1) \ldots \mathit{dly}(p_n, x_n)$, with $\{x_1, \ldots, x_n\}$ a set of n distinct variables of the sort of the time domain, is such that for each occurrence of $(p, [l, u])$ in $\partial_1(t)$, there is a distinct $i \in \{1, \ldots, n\}$ with $p_i = p$, $l_i = l$, and $u_i = u$. The tick rule advances time until the first delayed token becomes visible[11]:

$$\textbf{lazy}\ [\mathit{tick}] :\ \{VM\ \ DM\} \stackrel{\mathit{mte}(DM)}{\longrightarrow} \{VM\ \ \delta(DM, \mathit{mte}(DM))\}\ \textbf{if}\ \ \mathit{mte}(DM) \neq \infty.$$

For example, the translation of the ITPN in Fig. 2.3 contains the above tick rule and the following two instantaneous rules:

$$\textbf{eager}\ [a] :\ [p] \longrightarrow \mathit{dly}(q, x_r)\ \ \mathit{dly}(q, y_r)\ \textbf{if}\ 5 \leq x_r \leq 10 \wedge 5 \leq y_r \leq 10$$
$$\textbf{eager}\ [b] :\ [q] \longrightarrow \mathit{dly}(p, x_r)\ \textbf{if}\ 4 \leq x_r \leq 8$$

Transitions are supposed to fire as soon as possible in timed Petri nets. This is accomplished by the strategy described in Section 2.1.5 that triggers all eager transition rules until none of these can be applied, followed by one application of the tick rule. The use of eager and lazy rules allows a simple condition in the tick rule, which would otherwise have to take into account the enabledness of every transition in the system, together with the *mte* part of the tick rule. Here, the tick rule only needs to compute the time until the next delayed token(s) become visible and advance time by that amount. After such a tick, the tick rule is again enabled but, due to its being lazy, it will not be applied if the introduction of the new token(s) enables an (eager) transition (which in turn could trigger more transitions in zero-time).

Although ITPN markings are multisets, it is not necessary to apply the techniques described in Section 2.1.5 to specify ITPNs in rewriting logic. This is because terms of sort *DelayedMarking* cannot be rewritten, which implies that there are no "aged" rewrites, and that the introduction of the function $\mathit{mte} : \mathit{DelayedMarking} \to \mathit{Time}_\infty$ does not lead to undesired nontrivial rewrites in the time domain.

Since a step $m \stackrel{e}{\to} m'$ of an ITPN does not depend on the firing delays of the individual transitions taken in the step, two *one-step* rewrites $\{\alpha\} : u \longrightarrow v$ and $\{\beta\} : u \longrightarrow v$ should be considered equal—in the sense that we add this equivalence as a further equality identifying rewrite proofs—if the multisets of rule labels in α and β are the same. That is, α and β are equivalent iff $\alpha = t(r_1, \ldots, r_n)\ \alpha'$ and $\beta = t(r_1', \ldots, r_n')\ \beta'$ for $t \in \mathsf{T}$ and α' and β' are (recursively) equivalent. An *ITPN-computation* in $\Psi_{ITPN}(N)$ is a finite or infinite sequence

$$\tilde{\varsigma} :\ \ u_0 \stackrel{\alpha_1;\beta_1}{\longrightarrow} u_1 \stackrel{\alpha_2;\beta_2}{\longrightarrow} u_2 \stackrel{\alpha_3;\beta_3}{\longrightarrow} \cdots$$

[11] To have the possibility of stopping at any moment in the time domain, a non-deterministic time advance $x_r \leq \mathit{mte}(DM)$ could be used instead if desired.

of admissible rewrite proofs $\alpha_i; \beta_i : u_{i-1} \longrightarrow u_i$ in $\Psi_{ITPN}(N)$ where $u_{i-1}, u_i \in \mathcal{T}_{\Psi_{ITPN}(N)}$, modulo the equivalence on proof terms given above, such that each α_i corresponds either to the identity proof or to a sequence of tick applications, each β_i corresponds to a one-step concurrent rewrite using instantaneous rules, and u_0 is a term $\{m_0\}$ with m_0 a term of sort *Marking*. The set of ITPN-computations in $\Psi_{ITPN}(N)$ is denoted $ITPN\text{-}\mathcal{C}(\Psi_{ITPN}(N))$. The following proposition follows from the factorization property of proofs in rewriting logic [Mes92].

Proposition 17

- *Each non-identity admissible ground rewrite $\epsilon : \{m\} \longrightarrow \{m'\}$ in $\Psi_{ITPN}(N)$ is equivalent to a rewrite $\alpha; \gamma$, such that α can be rearranged as a finite ITPN computation, and γ corresponds to the identity proof or to a sequence of tick applications.*
- *Each (infinite) computation of $\Psi_{ITPN}(N)$, consisting of admissible rewrites involving ground terms of sort System, which contains an infinite number of applications of instantaneous rules, can be rearranged as an ITPN-computation by an appropriate composition of the arrows.*

The fact that ITPNs are faithfully represented in their rewriting logic translation is made precise in the following theorem.

Theorem 18 *Let N be an ITPN. Then, there is a bijective function $\tilde{(\_)} : \mathcal{C}_N^\infty \to ITPN\text{-}\mathcal{C}(\Psi_{ITPN}(N))$ taking a step sequence of the form ς to an ITPN-computation of the form $\tilde{\varsigma}$ such that:*

- *Each u_i is a term equivalent to a term of the form $\{w_i\}$, which consists of $m_i(p, r)$ occurrences of the term $dly(p, r \dot{-} \tau(\alpha_1; \beta_1; \ldots; \alpha_i; \beta_i))$, for all p and r (recall that $dly(p, 0)$ is equivalent to $[p]$).*
- *The transitions fire at the same time in ς and $\tilde{\varsigma}$, that is, $\tau(\alpha_1; \beta_1; \ldots; \alpha_{i+1}) = EET(m_i)$.*
- *The transitions taken (concurrently) in each step are the same. That is, each β_i is equivalent to a proof term of the form $\{w_i\, \varepsilon_i\}$, where w_i is a term, and where ε_i is a multiset consisting of, for each $t \in \mathsf{T}$, exactly $e_i(t)$ occurrences of proof terms of the form $t(r_1, \ldots, r_n)$.*

Proof sketch Given a step sequence ς, we can inductively define a unique ITPN-computation $\tilde{\varsigma}$, having the desired properties, as follows. The term $u_0 = \{w_0\} = \tilde{m_0}$ in $\tilde{\varsigma}$ is obtained from m_0 by letting w_0 consist of $m_0(p, r)$ terms of the form $\mathtt{dly}(p, r)$ for each $(p, r) \in m_0$. Let $m_i \stackrel{e_{i+1}}{\to} m_{i+1}$ be a step in ς. Then, by induction we already have a unique prefix of $\tilde{\varsigma}$ of i steps. In particular, we can assume that $u_i = \tilde{m_i}$ is a term of the form $\{w_i\}$, where w_i consists of $m_i(p, r)$ occurrences of the term $\mathtt{dly}(p, r \dot{-} r_i)$ for each $(p, r) \in m_i$, where $r_i = \Sigma_{j=1}^{i}(\tau_N(\alpha_j)) = EET(m_{i-1})$ for $i > 0$, and $r_0 = 0$. Furthermore, for each state $m \in \mathsf{MRK}$ and $r \in \mathbb{R}_+$, let $m^{\leq r}$ denote the submarking of m containing only the pairs (p, r') where $r' \leq r$, that is, those elements of m which could be consumed by firings taking place at time r. Then, the multiset of undelayed places in w_i corresponds to $m_i^{\leq r_i}$. Since no transition is enabled on the state m_i before time $EET(m_i)$, no eager rule will be enabled on u_i until time has elapsed $EET(m_i) - r^i$. Therefore, since the tick rule exhibits deterministic behavior in time, a unique sequence α_{i+1} of tick applications with $\tau_N(\alpha_{i+1}) = EET(m_i) - r_i$ can be applied to u_i, giving a term u_i'. The term u_i' contains $m_i(p, r)$ occurrences of the term $\mathtt{dly}(p, r \dot{-} EET(m_i))$ for each $(p, r) \in m_i$. That is, the visible marking in u_i' is $places(m_i^{\leq EET(m_i)})$.

Therefore, we can associate to e_{i+1} a unique (modulo the equivalence on proofs described above) rewrite $\beta_{i+1} : u'_i \longrightarrow u_{i+1}$ where β_{i+1} is a proof term of the form $\{v_{i+1}\ \gamma_{i+1}\}$ for some term v_{i+1}, or of the form $\{\gamma_{i+1}\}$, with γ_{i+1} a nonempty multiset of proofs consisting of an occurrence of t for each transition t in e_{i+1}. The term u_{i+1} is a term $\{w_{i+1}\}$ where w_{i+1} contains $m_{i+1}(p, r)$ occurrences of the term $dly(p, r \dot{-} EET(m_i))$ for each $(p, r) \in m_{i+1}$.

Conversely, given an ITPN-computation of the form $\tilde{\varsigma}$, we can inductively define its uniquely associated step sequence ς, satisfying the desired properties, as follows. For each $p \in \mathsf{P}$ and $r \in \mathbb{R}_+$, $m_0(p, r)$ equals the number of occurrences of the term $dly(p, r)$ in u_0 (again, with $[p] = dly(p, 0)$). Let now $u_i \stackrel{\alpha_{i+1};\beta_{i+1}}{\longrightarrow} u_{i+1}$ be the ith step in $\tilde{\varsigma}$ with $u_i \stackrel{\alpha_{i+1}}{\longrightarrow} u'_i$. By induction, the state m_i corresponding to u_i consists of an occurrence of (p, r'), for some $r' \leq r_i = \Sigma_{j=1}^{i} \tau_N(\alpha_j)$, for each occurrence of the term $[p]$ in u_i, and of an occurrence of $(p, r_i + r'')$ for each occurrence of the term $dly(p, r'')$ in u_i. It can be shown that $r_{i+1} = \Sigma_{j=1}^{i+1} \tau_N(\alpha_j) = EET(m_i)$, since no instantaneous rule is enabled on u_i before time reaches $EET(m_i)$, and that, due to eagerness of instantaneous transitions, the tick rule cannot be applied, since an instantaneous rule would be enabled on u'_i when $r_{i+1} = EET(m_i)$. The multiset of "visible" tokens p in u'_i is then equivalent to $places(m_i^{\leq EET(m_i)})$, and, therefore, β_{i+1} maps to a unique e_{i+1} which contains an occurrence of the transition t for each occurrence of the proof term $t(\dots)$ (for $t \in \mathsf{T}$) in β_{i+1}. A new token $dly(p, r)$ created by β_{i+1} corresponds to the presence of a new pair $(p, EET(m_i) + r)$ in m_{i+1}, ensuring that the resulting states satisfy the desired relationship between u_{i+1} and m_{i+1}.

2.3 Relationship to Timed Rewriting Logic

In this section we investigate the relationship between Kosiuczenko and Wirsing's *timed rewriting logic* (TRL) [KW97] and the framework we have presented for specifying real-time systems directly in rewriting logic. After briefly introducing TRL in Section 2.3.1, we propose in Section 2.3.2 a translation from TRL into rewriting logic such that the translation of any TRL-sequent derivable in a TRL theory is also derivable in the corresponding rewriting logic theory. The converse is in general not true, due to some conceptual differences between TRL and our method of specifying real-time systems in rewriting logic, as explained in Section 2.3.3.

2.3.1 Timed Rewriting Logic

Rewriting logic has been extended by Kosiuczenko and Wirsing to handle real-time systems in their timed rewriting logic (TRL) [KW97]. TRL has been shown well-suited for giving object-oriented specifications of complex hybrid systems such as the steam-boiler [ÖKW96] and has been illustrated by a number of specifications of simpler real-time systems. A translation into ordinary rewriting logic can illuminate the conceptual relationships between both formalisms.

A TRL theory (Σ, E, L, TR) consists of an equational specification (Σ, E) satisfying the theory *TIME*[12], a set L of labels, and a set TR of *timed* rewrite rules of the form $[l] : t \stackrel{r}{\longrightarrow} t'$, where r is a ground term of sort *Time* denoting the duration of the rewrite. A TRL sequent has the form $t \stackrel{r}{\longrightarrow} t'$ and its intuitive meaning is that t evolves to t' in time r. More specifically, the set of sequents derivable from a TRL theory consists of all rules in the theory, and all sequents which

[12]They impose in some cases further requirements, such as *TIME* being an Archimedean monoid. This could of course be easily accommodated.

Timed transitivity:

$$\frac{t_1 \xrightarrow{r_1} t_2 \qquad t_2 \xrightarrow{r_2} t_3}{t_1 \xrightarrow{r_1+r_2} t_3}$$

Synchronous replacement:

$$\frac{t_0 \xrightarrow{r} t_0', \quad t_{i_1} \xrightarrow{r} t_{i_1}', \ldots , t_{i_k} \xrightarrow{r} t_{i_k}'}{t_0(t_1/x_1, \ldots , t_n/x_n) \xrightarrow{r} t_0'(t_1'/x_1, \ldots , t_n'/x_n)}$$

where $\{x_{i_1}, \ldots , x_{i_k}\} = \mathcal{V}(t_0) \cap \mathcal{V}(t_0')$.

Compatibility with equality:

$$\frac{t_1 = u_1, \; r_1 = r_2, \; t_2 = u_2, \; t_1 \xrightarrow{r_1} t_2}{u_1 \xrightarrow{r_2} u_2}$$

Renaming of variables:

$$\frac{t_1 \xrightarrow{r} t_2}{\rho(t_1) \xrightarrow{r} \rho(t_2)}$$

for ρ a renaming of the variables $\mathcal{V}(t_1) \cup \mathcal{V}(t_2)$.

Figure 2.4: Deduction rules in timed rewriting logic.

can be derived by equational reasoning and by using the deduction rules in Figure 2.4, where $\mathcal{V}(t)$ denotes the set of variables in t. This deduction system extends and modifies the rules of deduction in rewriting logic with time stamps as follows:

- Reflexivity is dropped as a general axiom, since it would allow (parts of) the system to stay idle for arbitrarily long periods of time.
- Transitivity yields the addition of the time stamps. If t_1 evolves to t_2 in time r_1 and t_2 evolves to t_3 in time r_2, then t_1 evolves to t_3 in time $r_1 + r_2$.
- The synchronous replacement rule enforces uniform time elapse in all components of a system: a system rewrites in time r iff all its components do so.

2.3.2 Timed Rewriting Logic in Rewriting Logic

In this section we define a mapping $\mathcal{M}$ which takes any timed rewriting logic theory $\mathcal{T}$ to a real-time rewrite theory $\mathcal{M}(\mathcal{T})$ such that $\mathcal{T} \vdash t \xrightarrow{r} t'$ implies that $\mathcal{M}(\mathcal{T}) \vdash \alpha : \{t\} \longrightarrow \{t'\}$, for some α with $\tau(\alpha) = r$, for all ground ($\mathcal{T}$-) terms t, t' of the designated sort *State*.

The idea is to introduce for each sort s an operator $\delta : s \; Time \to s$ corresponding to the effect of time elapse. Then, a TRL sequent $t \xrightarrow{r} t'$ ("t evolves in time r to t'") can be mapped to a rewriting logic sequent $\delta(t, r) \longrightarrow t'$ ("if time has acted on t for time r, then it rewrites to t'") for ground terms t, t'. Rewrite rules must be used to define δ, since the effect of time on a TRL state is not necessarily functional.

Sort information is used to separate terms containing the symbol δ from terms of the original signature, and a tick rule is added to the rules defining δ such that for ground $\mathcal{T}$-terms t, t' of sort *State*, $\mathcal{M}(\mathcal{T}) \vdash \alpha : \{t\} \longrightarrow \{t'\}$ holds for some α with $\tau(\alpha) = r$ if and only if $\mathcal{M}(\mathcal{T}) \vdash \delta(t, r) \longrightarrow t'$, which in turn holds whenever $\mathcal{T} \vdash t \xrightarrow{r} t'$ holds. The resulting real-time rewrite theory $\mathcal{M}(\mathcal{T})$ is not easily executable, since the tick rule introduces two variables in its righthand side. This reflects the fact that in TRL it is in general undecidable whether a term rewrites in time r $(r > 0)$, and, even if it is known that t rewrites in time r, it is also in general undecidable whether t rewrites to a given term t' in time r.

We assume that the time domain is functional, that is, that no rewrites of the form $t \xrightarrow{r} t'$, with $t \neq t'$ terms of sort *Time*, can be inferred from the TRL theory $\mathcal{T}$, and restrict our treatment to TRL theories where no extra variables are introduced in the righthand side of a rule. The reason for the latter restriction is that if $[l_1] : f(x) \xrightarrow{2} g(x, y)$ and $[l_2] : g(x, y) \xrightarrow{2} h(y)$ are two rules, any system t' that appears in $h(t)$ as a result of the second rule, must have evolved for 2 time units from a system t in $g(u, t)$. However, by transitivity of the rules, the sequent $f(x) \xrightarrow{4} h(y)$ is derivable, which means that *any* system t could replace y in $h(y)$, including the systems which have not evolved for 2 time units.

The Mapping from TRL to Real-Time Rewrite Theories

The mapping $\mathcal{M}$ sends an order-sorted TRL theory $\mathcal{T} = (\Sigma, E, L, TR)$ to a real-time rewrite theory $\mathcal{M}(\mathcal{T}) = ((\mathcal{M}(\Sigma), \mathcal{M}(E), \mathcal{M}(L), \mathcal{M}(TR)), \phi(\mathcal{T}), \tau(\mathcal{T}))$ and sends a $\mathcal{T}$-sequent $t \xrightarrow{r} t'$ to an $\mathcal{M}(\mathcal{T})$-sequent $\mathcal{M}(t \xrightarrow{r} t')$. It is defined as follows:

- The signature morphism $\phi(\mathcal{T})$ in $\mathcal{M}(\mathcal{T})$ takes *Time* to the sort *Time* in $\mathcal{T}$ denoting the time domain, and takes the functions in *TIME* to the corresponding functions in $\mathcal{T}$.
- The set of sorts in $\mathcal{M}(\Sigma)$ consists of all the sorts in Σ, plus a new sort s^δ for each sort s in Σ, as well as a new sort *System*. For each sort s in Σ, $s \leq s^\delta$, and if $s \leq s'$ in Σ, then $s^\delta \leq s'^\delta$ in $\mathcal{M}(\Sigma)$.
- $\mathcal{M}(\Sigma)$ contains function declarations $f : s_1 \ldots s_n \to s$ and $f : s_1^\delta \ldots s_n^\delta \to s^\delta$ for each function $f : s_1 \ldots s_n \to s$ in Σ, a constructor $\{\_\} : State \to System$ for the designated sort *State* in Σ, and a function $\delta : s^\delta\ Time \to s^\delta$ for each sort s in Σ.
- $\mathcal{M}(E)$ contains an axiom e^δ for each axion e in E, where e^δ is the axiom e where each variable $x\!:\!s$ is replaced by $x\!:\!s^\delta$. The set $\mathcal{M}(E)$ must also define δ to be a monoid action, that is, it contains the axioms

  $$\delta(x_{s^\delta}, 0_\phi) = x_{s^\delta}$$
  $$\delta(\delta(x_{s^\delta}, y_r), z_r) = \delta(x_{s^\delta}, y_r +_\phi z_r)$$

  for each sort s^δ in $\mathcal{M}(\Sigma)$, variable x_{s^δ} of sort s^δ, and variables y_r, z_r of sort *Time*.
- The mapping $\mathcal{M}$ from TRL-sequents to rewriting logic sequents is given by

  $$\begin{array}{l}\mathcal{M}(t(x_1\!:\!s_1, \ldots, x_n\!:\!s_n) \xrightarrow{r} t'(x_1\!:\!s_1, \ldots, x_n\!:\!s_n)) = \\ \delta(t(x_1\!:\!s_1^\delta, \ldots, x_n\!:\!s_n^\delta), r) \longrightarrow t'(\delta(x_1\!:\!s_1^\delta, r)/x_1, \ldots, \delta(x_n\!:\!s_n^\delta, r)/x_n)\end{array}$$

where the free variables in t are $x_1, \ldots, x_n$ and contain those of t'.

The set of rules $\mathcal{M}(TR)$ consists of a rule $[l] : \mathcal{M}(t \xrightarrow{r} t')$ **if** C^δ for each timed rule $[l]$: $t \xrightarrow{r} t$ **if** C in TR, and a tick rule

$$[tick]: \ \{x\} \xrightarrow{y_r} \{x'\} \ \textbf{if} \ \delta(x, y_r) \longrightarrow x'$$

for variables x, x' of sort *State* and y_r of sort *Time*.

The theorem below shows that $\mathcal{M}$ can be naturally understood as a map of logics. Specifically, as a map $\mathcal{M} : TRL \longrightarrow RWL$ from the entailment system [Mes89] of TRL to that of rewriting logic.

Theorem 19 *Let $\mathcal{T}$ be a TRL specification and let $\mathcal{M}$ be defined as above. Then, for all terms $t, t', r \in \mathcal{T}_\Sigma(X)$,*

$$\mathcal{T} \vdash t \xrightarrow{r} t' \ \ \textit{implies} \ \ \mathcal{M}(\mathcal{T}) \vdash \mathcal{M}(t \xrightarrow{r} t').$$

As a corollary to this theorem, which can be easily proved by induction on the size of the proof $t \xrightarrow{r} t'$, we obtain that $\mathcal{T} \vdash t \xrightarrow{r} t'$ implies $\mathcal{M}(\mathcal{T}) \vdash \delta(t, r) \longrightarrow t'$ for all ground terms t, t', and r, which in turn gives a rewrite $\mathcal{M}(\mathcal{T}) \vdash \alpha : \{t\} \longrightarrow \{t'\}$ with $\tau(\alpha) = r$ when t and t' are of sort *State* by applying the tick rule. It is also easy to see that $\mathcal{M}(\mathcal{T}) \vdash \alpha : \{t\} \longrightarrow \{t'\}$ implies $\mathcal{M}(\mathcal{T}) \vdash \delta(t, \tau(\alpha)) \longrightarrow t'$ for ground $\mathcal{T}$-terms t, t' of sort *State*.

2.3.3 Differences Between TRL and its Rewriting Logic Translation

Even though $t \xrightarrow{r} t'$ implies $\delta(t, r) \longrightarrow t'$ for ground terms, the converse is not necessarily true. In this section we discuss the differences between deduction in TRL and in its translation into rewriting logic.

Zero-Time Idling

In the rewriting logic translation, a TRL sequent $t \xrightarrow{0} t$ translates to

$$\delta(t, 0) \longrightarrow t(\delta(x_1, 0)/x_1, \ldots, \delta(x_n, 0)/x_n),$$

which, due to the axiom $\delta(x, 0) = x$, is equal to $t \longrightarrow t$, which is always deducible in rewriting logic. However, in TRL, $t \xrightarrow{0} t$ is not necessarily valid. This obviously indicates a difference between both systems, since the notion of "zero-time idling" is always available in our approach but not in TRL.

Non-Right-Linear Rules

Given the TRL theory

$$\{\, [l_1] : \, f(x) \xrightarrow{2} g(x, x), \ [l_2] : \ a \xrightarrow{2} b, \ [l_3] : \ a \xrightarrow{2} c \,\},$$

the term $f(a)$ rewrites to either $g(b,b)$ or $g(c,c)$ in time two, but will *not* rewrite to $g(b,c)$. In the rewriting logic translation

$$\{\, [l_1] : \delta(f(x),2) \longrightarrow g(\delta(x,2),\delta(x,2)),\ [l_2] : \delta(a,2) \longrightarrow b,$$
$$[l_3] : \delta(a,2) \longrightarrow c\,,\ [tick] : \{y\} \xrightarrow{x_r} \{y'\} \textbf{ if } \delta(y,x_r) \longrightarrow y'\,\},$$

where y and y' are variables of the designated state sort and range over δ-free terms, there is a rewrite

$$\delta(f(a),2) \longrightarrow g(\delta(a,2),\delta(a,2)) \longrightarrow g(b,c),$$

and therefore also a rewrite

$$\alpha : \{f(a)\} \longrightarrow \{g(b,c)\}$$

with $\tau(\alpha) = 2$.

The difference depends on how the fork of a process is modeled. The rule $[l] : f(x) \xrightarrow{r} g(x,x)$ can be understood as a fork of the (sub)process t in the system $f(t)$. In the TRL setting, the actual "fork" (the point in time when the two instances of the process x can behave independently of each other) is taking place at the end of the time period of length r in the rule. In the rewriting logic setting, the "forking" took place at the beginning of the time period of duration r[13].

Problems Related to Synchrony in TRL

Another aspect in which TRL and our rewriting logic translation are different is illustrated by the following TRL specification:

$$\{\, [l_1] : f(a,y) \xrightarrow{2} g(a,y),\ [l_2] : g(x,y) \xrightarrow{2} h(x,y),$$
$$[l_3] : h(x,c) \xrightarrow{2} k(x,c),\ [l_4] : a \xrightarrow{4} d,\ [l_5] : b \xrightarrow{4} c\,\}.$$

Due to the strong synchrony requirements in TRL, $f(a,b)$ cannot be rewritten, even though the b (in the place of y), and a (for x), could be rewritten in time 4. In many cases, it would however be natural to assume that the system represented by $f(a,b)$ rewrites to $k(d,c)$ in time 6. In the rewriting logic translation, $\delta(f(a,b),6)$ rewrites indeed to $k(d,c)$.

2.3.4 Aging in TRL

To overcome the strong requirements of synchrony in TRL, which caused the differences in Sections 2.3.3 and 2.3.3, the special symbol *age* is introduced in [KW97]. It aims at making a term t, which rewrites in time r', "accessible" to synchronous rewrites in time r with $r' \geq r$, by making it visible as $age(t,r)$.

Formally, with aging the following two deduction rules are added to the TRL deduction rules given in Figure 2.4. In both deduction rules, $[l] : t \xrightarrow{r+r'} t'$ is assumed to be a timed rewrite *rule* in the specification.

$$age_1 : \quad \frac{}{t \xrightarrow{r} age(t,r)} \qquad\qquad age_2 : \quad \frac{}{age(t,r) \xrightarrow{r'} t'}$$

[13]Note that in the rewriting logic setting, adding a rule $[l_4] : \delta(k(x),2) \longrightarrow f(\delta(x,2))$ to the system above gives $\delta(k(x),4) \longrightarrow g(\delta(x,4),\delta(x,4))$, hence a "fork" which took place too early. Such behavior can be avoided by requiring that the variable x in the rule $[l_1] : \delta(f(x),2) \longrightarrow g(\delta(x,2),\delta(x,2))$ has a "non-δ-sort".

The *age* operator also satisfies the axiom $age(age(t,r),r') = age(t, r+r')$ for all terms t and time values r, r'.

With aging, the "fork" differences disappear, since (assuming $g(x,y) \xrightarrow{0} g(x,y)$) we have

$$f(a) \xrightarrow{2} g(age(a,2), age(a,2)) \xrightarrow{0} g(b,c)$$

for the system in the first example of Section 2.3.3, and the strong synchrony is loosened, as illustrated by the fact that in the example of Section 2.3.3, $f(a,b) \xrightarrow{6} k(d,c)$ is derivable, since $f(a,b) \xrightarrow{2} g(a, age(b,2))$, $g(a, age(b,2)) \xrightarrow{2} h(age(a,2),c)$, and $h(age(a,2),c) \xrightarrow{2} k(d,c)$ are derivable.

Unfortunately, the deduction rules for aging lead to counterintuitive results, as illustrated by the following example:

Example 20 *In the TRL theory*

$$\{\, [l_1] : f(x) \xrightarrow{2} g(x),\ [l_2] : f(b) \xrightarrow{2} g(c),\ [l_3] : a \xrightarrow{2} b \,\},$$

one would expect $f(a) \xrightarrow{2} g(c)$ not *to be derivable. However, the sequents* $f(x) \xrightarrow{2} age(f(x),2)$ *and* $age(f(x),2) \xrightarrow{0} g(x)$ *are both derivable, and so are* $f(b) \xrightarrow{2} age(f(b),2)$ *and* $age(f(b),2) \xrightarrow{0} g(c)$. *The sequents* $f(x) \xrightarrow{2} age(f(x),2)$ *and* $a \xrightarrow{2} b$ *give* $f(a) \xrightarrow{2} age(f(b),2)$ *by synchronous replacement, which in turn rewrites to* $g(c)$ *using* $age(f(b),2) \xrightarrow{0} g(c)$. *Transitivity gives the undesired sequent* $f(a) \xrightarrow{2} g(c)$.

We can summarize the situation as follows. We have seen that the rewriting translation of a TRL theory $\mathcal{T}$ is looser than $\mathcal{T}$ itself, in some cases with some pleasant consequences. If we attempt to tighten the correspondence between both systems by adding aging rules to TRL, we get indeed closer, but we unfortunately encounter paradoxical examples in the reformulation of TRL.

2.4 Concluding Remarks

We have presented a general method for specifying real-time and hybrid systems in rewriting logic in an executable way, have shown how a wide range of real-time and hybrid system models can be naturally expressed in rewriting logic, and have illustrated the ideas with several examples. This work should be further extended in several directions.

The systems that we have considered can be distributed and can exhibit concurrent computations, in which several components of the state can change simultaneously and independently. However, time is still in some sense *global*, since time acts on the global state, even though its effects can be local and distributed—for example, by advancing the local clocks of different distributed objects. The situation is entirely similar to that in some real-time models for distributed systems such as Lynch's *general timed automata* [Lyn96], where time also acts uniformly on all the distributed components. In fact, although we have not discussed general timed automata in this chapter, they can also be specified within our general framework. Although the current framework can already be used for specifying and reasoning about a range of distributed time-based systems, it would be worth investigating how the assumption of global time action could be relaxed to local or distributed time actions.

We have explored what we think is a representative range of real-time and hybrid system models. However, the general timed (I/O) automata model mentioned above, real-time dataflow models such as Lustre's [HCRP91], and a variety of other models, including timed process algebra models, should also be specified in detail in rewriting logic. The interest is not merely conceptual: by using a formal meta-tool such as Maude [CDE$^+$99b], one can turn the rewriting logic specification of a model into a tool for executing and analyzing formal specifications in that model, as in fact we have done in the Real-Time Maude language and tool presented in this thesis. Since at present some formalisms lack execution and analysis environments, this offers a way of developing new formal tools with considerably less effort than what would be required for conventional implementations.

Chapter 3

Real-Time Maude: A Language and Tool for Simulating and Analyzing Real-Time and Hybrid Systems

The previous chapter showed how real-time and hybrid systems can naturally be specified by *real-time rewrite theories*. This chapter describes how the *Real-Time Maude* language and tool tool can be used to specify and analyze such theories. Real-Time Maude capitalizes on the executability property of rewriting logic by using the Maude engine as its underlying execution mechanism. The specification language of the tool directly supports real-time aspects with special syntactic features in *timed modules* and *timed object-oriented modules*. At the theoretical level, those syntactic features support the specification of real-time rewrite theories, which can be transformed into ordinary rewrite theories that can then be executed in Maude. This transformation is performed internally by the Real-Time Maude tool.

Executability and formal analysis of real-time specifications poses new problems that the tool must address. To begin with, time advances are often *nondeterministic*. This is reflected in the tick rewrite rules themselves, which often have one or more new variables in their righthand side and/or their condition. A second, commonly occurring situation is that some time-critical rules are *eager* and must be executed instantaneously and without delay, whereas other rules are *lazy* and should only be executed when no eager rules are enabled. The Real-Time Maude tool supports execution and analysis of real-time specifications by providing the user with a library of *execution strategies* for different forms of default execution and formal analysis, that directly address the nondeterminism and eagerness/laziness requirements just mentioned, and support model-checking of temporal logic properties. Furthermore, because of the ease with which new strategies can be defined in Maude by reflection, the tool is easily extensible by the user, who can define new application-specific strategies to perform other forms of analysis. This is illustrated in Section 3.3 with a scheduling case study, in which an application-specific strategy finds all the solutions of a scheduling problem.

The rest of the chapter is organized as follows. Section 3.1 explains the Real-Time Maude specification language and the theory transformation process into Maude, Section 3.2 describes the Real-Time Maude tool's execution and search commands, and Section 3.3 gives an example of how the tool can be used to specify and analyze a class of scheduling problems.

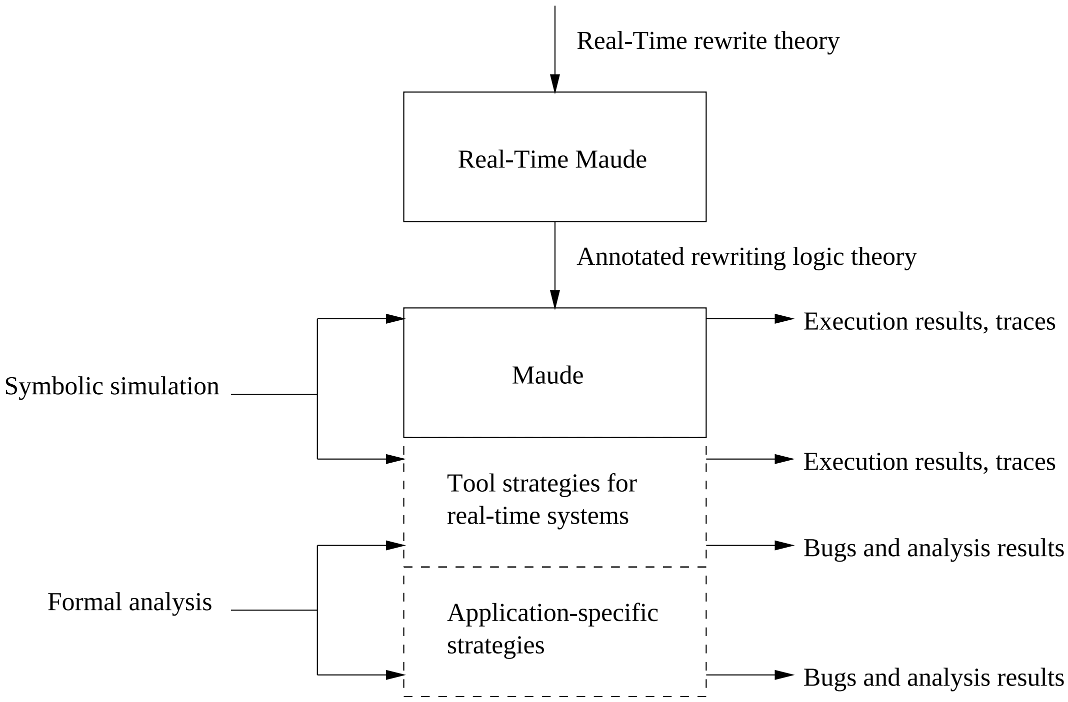

Figure 3.1: Architecture of the Real-Time Maude Tool.

3.1 Real-Time Maude Specifications

The Real-Time Maude language and tool extend Full Maude to support the specification, execution, and formal analysis of real-time rewrite theories. Real-time rewrite theories are specified by *timed modules* and *object-oriented timed modules*. The tool transforms timed modules into a pair consisting of an (untimed) Maude module and a set of labels for the lazy rules in the module. As shown in Figure 1, the tool can be used to execute and analyze real-time rewrite theories in three different ways, namely: (i) by executing specifications in the underlying Maude interpreter; (ii) by using a library of tool-specific strategies; and (iii) with application-specific user-defined strategies.

The first use (i) of Maude's default interpreter is unsatisfactory for most real-time specifications because Maude's general default execution strategy does not take account of time aspects. Therefore, Real-Time Maude provides a library of rewriting strategies specially designed for execution and analysis of real-time specifications. Strategies in such a library are executed in a reflective way as extensions of Maude's `META-LEVEL` module and include default execution strategies, and search strategies. In addition, users of the Real-Time Maude tool can write their own application-specific strategies in other modules extending `META-LEVEL` (or `TIMED-META-LEVEL`, which extends `META-LEVEL` with useful functions for real-time specifications) and can input strategies in the tool's library as reusable components.

3.1.1 Specifying Real-Time Rewrite Theories in Real-Time Maude

In the tool, a real-time rewrite theory $(\mathcal{R}, \phi, \tau)$ is specified as a *timed module* at the user-level by enclosing the module body between the keywords `tmod` and `endtm` for timed system modules, and between the keywords `tomod` and `endtom` for *object-oriented timed modules.* To state non-executable properties, Real-Time Maude also allows the user to specify real-time extensions of abstract Full Maude theories and object-oriented theories by using, respectively, the keywords `tth` and `endtth` for *timed theories*, and `toth` and `endtoth` for *object-oriented timed theories.*

A timed rule $l : t \longrightarrow t'$ **if** C, with t, t' terms of sort `System` and $\tau_l \neq 0$, is written with syntax

`crl [`l`] : `t` => `t'` in time `τ_l` if `C` .`

and with similar syntax for the case of unconditional rules. We can also specify (instantaneous or tick) rules to be *lazy*, by placing the keyword `lazy` in front of `rl` or `crl`.

The equational theory morphism ϕ from the theory *TIME* is not given explicitly at the specification level. Instead, by default, any timed module automatically imports the functional module `TIMED-PRELUDE`, which includes the following declarations:

```
sort Time .
op zero : -> Time .
op _plus_ : Time Time -> Time [assoc comm prec 33 gather (E e)] .
op _monus_ : Time Time -> Time [prec 33 gather (E e)] .
ops _le_ _lt_ _ge_ _gt_ : Time Time -> Bool [prec 37] .
vars R R' : Time .
eq R + zero = R .
eq R le R' = R lt R' or R == R' .
eq R ge R' = R' le R .
eq R gt R' = R' lt R .
```

The morphism ϕ then implicitly maps *Time* to `Time`, 0 to `zero`, $\_+\_$ to `_plus_`, $\_\leq\_$ to `_le_`, etc. Even though Real-Time Maude assumes a fixed syntax for time operations, the tool does not build in a fixed model of time. In fact, the user has complete freedom to specify the data type of time values—which can be either discrete or dense—by:

1. importing a specific data type of time values satisfying the *TIME* theory, with a sort, say `TimeValues` for such values, and
2. declaring a subsort inclusion `TimeValues < Time`, and
3. giving appropriate equations interpreting the constant `zero` and the operators `_plus_`, `_monus_`, and `_lt_` in `TimeValues`.

For example, if we want a discrete time domain, the sort `TimeValues` can be chosen to be the positive machine integers with the declarations

```
subsort PositiveMachineInt < Time .
vars N N' : PositiveMachineInt .
eq zero = 0 .
```

```
eq N plus N' = N + N' .
eq N lt N' = N < N' .
eq N monus N' = if N' <= N then N - N' else 0 fi .
```

In the applications we have developed so far, we have used three data types for time, namely, the positive machine integers, the positive rational numbers fractions on machine integers, and the positive reals with fixed point arithmetic. Their Real-Time Maude specifications are given in Appendix A.

Although acceptable for experimental purposes, these data types are unsatisfactory in that, due to machine overflow, the axioms of the theory *TIME* are strictly speaking not satisfied. The situation can of course be remedied by defining a functional module for Peano arithmetic and defining the rationals in terms of it, but this is inefficient; instead, Maude 2.0 will have a built-in data type `NAT` of naturals of unbounded size allowing easy definition of a module `RAT` of rationals, so that efficient models of discrete and dense time, strictly satisfying the *TIME* theory, become available.

Finally, timed modules always contain by default the sorts `State`, for the state component, and `System`, and the operator `{_} : State -> System`. Therefore, the sort of the state should always be declared as a subsort of the sort `State`.

Module Operations

Since Real-Time Maude extends Full Maude [CDE+99a], all the usual module composition operations provided by Full Maude are also supported in Real-Time Maude, with the following constraints:

- A module importing a timed module or theory, or having a timed theory as parameter must itself be declared to be a timed module or theory.[1]
- No module should contain both eager and lazy rules having the same label. However, the database of all modules may contain eager and lazy rules having the same label, but belonging to different modules.
- An actual parameter should not introduce lazy rules not mentioned in the corresponding formal parameter theory.[2]

Timed Object-Oriented Modules

Timed object-oriented modules extend both object-oriented and timed modules to provide support for object-oriented real-time systems. In particular, as explained in Section 2.2, object-oriented timed modules include the specification of the sorts `Tokens` and `TokenConf`. Furthermore, since, in contrast to untimed object-oriented systems, functions such as δ and *mte*, and the tick rules, involve the global configuration, it is useful to have a richer sort structure for configurations. Therefore, timed object-oriented modules also include the sorts `MsgConf`, `NEMsgConf`, `ObjConf`, `NEObjConf`,

[1] The exception is module inclusion in `META-LEVEL` and `TIMED-META-LEVEL`, whose parameters may be timed modules.

[2] This restriction may be omitted in later versions of the tool.

and `NEConfiguration`, denoting, respectively, multisets of messages, non-empty multisets of messages, multisets of objects, non-empty multisets of objects, and non-empty configurations. The subsort declaration `TokenConf < State` is automatically added to object-oriented specifications.

Extensions of the Time Domain

The module library in Real-Time Maude includes the functional modules `TIME-INF`, `LTIME`, and `LTIME-INF`, which extend the time domain with, respectively, an "infinity" time value `INF` of a supersort `TimeInf` of `Time`, operations `max` and `min`, and the combination of these two extensions.

3.1.2 Processing Specifications of Real-Time Rewrite Theories

In the current executable specification of Real-Time Maude, which is written entirely in Maude and extends the Full Maude specification, described in Chapter 4; we do not introduce a new data type for representing timed modules in the Full Maude database. Instead, the tool reads a timed system module as a term `tmod` T `is` T' `endtm`, and transforms it into a term `mod` T `is` T'' `endm`, which is then evaluated and represented as an untimed module in the Full Maude database using existing Full Maude functions. Other kinds of timed modules and theories are treated in the same way.

However, to keep around important execution information stated in the timed modules, the Full Maude module database object is extended so that it contains an additional attribute `lazyLabels` which maps each declaration of a real-time rewrite theory into the set of labels of lazy rules declared in either the module itself, or in its parameters and included modules.

This early preprocessing of declarations of timed modules has allowed us to extend Full Maude to Real-Time Maude with (almost) no changes to the Full Maude code. In particular, future changes in Full Maude can easily be extended to Real-Time Maude by just adding the current real-time addition, and it will be easy to modify or extend the syntax of timed modules when more experience is gained about real-time rewrite theories. Finally, given a module M in the Full Maude database and the table of lazy labels, it is easy to determine whether M was originally declared as a timed module, and the corresponding timed module can then be easily recovered.

The transformation of a term representing a timed module into a term representing a Full Maude module which can then be processed by Full Maude, consists of the following steps:

1. Lazy rule declarations are substituted by ordinary rule declarations, and the labels of the lazy rules declared either in the timed module in any of its parameters or submodules are stored in the database.

2. A declaration importing the module `TIMED-PRELUDE` is added. In addition to the declaration of the time operations mentioned above, this module contains the following declarations:

```
sort ClockedSystem .
subsort System < ClockedSystem .
op _in time_ : ClockedSystem Time -> ClockedSystem .
vars R R' : Time .
var CS : ClockedSystem .
eq CS in time R in time R' = CS in time (R plus R') .
```

The translation leaves the rules unchanged, but of course the `in time` r syntax in the original rule is now interpreted as a time action on the sort `System`. The translation can be proved equivalent to the one given in Chapter 2, in the sense that t `in time` r rewrites to t' `in time` r' if and only if $\langle t, r\rangle$ rewrites to $\langle t', r'\rangle$ in the original translation. We find the current translation more convenient, since it is simpler (in fact the identity). Furthermore, in this way we do not need to introduce fresh variables for the translation of the timed rules.

3. The module `TIMED-META-LEVEL` extends the module `META-LEVEL` with a library of strategies for real-time specifications described in Section 3.2, including the timed versions of the `meta-apply` and `meta-rewrite` functions, and search strategies.

   In Full Maude, importing `META-LEVEL[`$M_1, \ldots, M_n$`]`, for module names $M_1, \ldots, M_n$, results in Full Maude adding to `META-LEVEL` constants $M_1, \ldots, M_n$, which are then defined to be equal to the meta-representations of the (flattened versions of the) modules $M_1, \ldots, M_n$. In the same way, a module may import `TIMED-META-LEVEL[`$M_1, \ldots, M_n$`]` in Real-Time Maude, which imports the module `TIMED-META-LEVEL` and equates the meta-representations of the *modules* $M_1, \ldots, M_n$ with the *constants* $M_1, \ldots, M_n$. To completely define a timed module, we also need access to the set of labels of the lazy rules of a module. Therefore, for each module name M which occurs as a parameter to `TIMED-META-LEVEL`, Real-Time Maude introduces a constant `lazyLabels-`M', where M' is the "base" of M (that is, the name M without the parameter list in case M has parameters), and defines its value to be the set of labels of the lazy rules in M. To summarize, a declaration

   ```
   including TIMED-META-LEVEL[M1, ..., Mn] .
   ```

   is expanded into:

   ```
   including META-LEVEL[M1, ..., Mn] .
   including TIMED-META-LEVEL .
   ops lazyLabels-M1' ... lazyLabels-Mn' : -> QidSet .
   eq lazyLabels-M1' = "lazy labels of module M1" .
   ...
   eq lazyLabels-Mn' = "lazy labels of module Mn" .
   ```

The transformation steps 1 and 2 are only performed on timed modules; the last one on all modules.

3.2 Executing and Analyzing Real-Time Maude Specifications

Our tool transforms real-time specifications into ordinary Full Maude modules, which can then be executed using Maude's default execution strategy. However, for many of these transformed modules, Maude's default execution strategy may not be very useful, for the following reasons:

- For simulating a system having a continuous time domain, the tick rules will often have the form

  ```
  crl [tick] : {t} => {t'(x_r)} in time x_r if x_r le t'' and C(t) .
  ```

where the variables of t'' are included in those of t, and the variable x_r does not occur in t. Intuitively, t'' computes the *maximum time elapse* permissible to ensure timeliness of time-critical actions, and the condition x_r `le` t'' ensures that time elapse *may* halt temporarily for the *possible* application of a non-time-critical rule (i.e., a rule modeling an action which could occur somewhat "arbitrarily" in time). The current Maude default interpreter cannot execute such tick rules because of the nondeterminism introduced by the extra variable x_r, and therefore time would not advance in a default Maude execution.

- The default interpreter fails to take into account the eager and lazy rules.
- Instead of measuring and controlling the execution by the number of rewrites performed, it is often more useful to control it by the total time elapsed $\tau(\alpha)$ in the rewrite proof $\alpha : t_0 \longrightarrow t$ from the initial term t_0.

To address these issues, the tool executes the tick rules in different *basic modes*, which determine how to apply a tick rule whose duration term is a variable not occurring in the lefthand side of the rule. These, and some additional modes used to optimize the application of rules, are described in Section 3.2.1. The standard execution strategies and some search strategies based on these modes are given in Sections 3.2.2 and 3.2.3. The strategies in this section will work as described when only tick rules may introduce a new variable in its righthand side or its condition.

3.2.1 Rewrite Modes

The *mode* of a one-step rewrite is used to indicate how to instantiate the time increase in case it is represented by a new variable in the rule. The tool currently supports the following three different basic modes for determining the time increase in a tick rule application: *default*, *maximal* possible time increase w.r.t. *each match* of the lefthand side of the tick rule, and *maximal* possible time increase w.r.t. *all matches* of the rule.

All these modes are qualified by a *time limit* and by a user-given *default time increase value*. For optimization purposes, we describe below two optimization modes which do not concern the time increase in nondeterministic tick rules.

In our tool, a tick rule application which increases time by 0 is considered equivalent to not being able to perform the rewrite. The basic modes describe the following strategies for applying a tick rule whose duration term is a new variable:

- In the *default time increase* mode, the new time variable x_r is instantiated with a user-given *default time increase value*, which must be a ground term different from 0.
- In the *maximal possible time increase w.r.t. each match* mode, the tool finds, for each match, a representation for the set of all possible values of the new variable which make the condition hold. To have a possible unique maximal time value, the time domain is assumed to be *linear*, i.e., it satisfies the abstract time theory *LTIME* given Section 2.1.1. The set of values of x_r can then be represented by a set of intervals of finite length (t *op* x_r *op* t') or infinite length (t *op* x_r), where *op* is either `le` or `lt`. When the greatest of these intervals is right-closed, i.e., of the form t *op* x_r `le` t', then t' is chosen as the maximal time value with which to instantiate the new variable; otherwise, some given default time increase value must be used. The tool only handles conditions with one new variable x_r, and where the condition

is a boolean combination of x_r-free terms and equalities and inequalities of the form t `==` t', t `le` t', t `lt` t', t `gt` t', and t `ge` t', where either t or t' does not contain the new variable x_r, and if t (or t') contains x_r, then it has only one occurrence of x_r, which, in addition, is not embedded in function symbols other than `_plus_` and `_monus_`.

- Instead of finding the maximal possible time increase Δ_k w.r.t. match number k, the *maximal time increase w.r.t. all matches* mode instantiates the new variable with the greatest of the maximal time increases $\Delta_1, \ldots, \Delta_n$ of each of the n successful matches for the rule's lefthand side, and is otherwise similar to the previous mode.

These three basic strategies can be modified in different ways. For example, it is sometimes (such as in model-checking of real-time temporal properties) useful to let time advance only up to a certain time r, even if the basic strategies would allow time to advance past time r without "stopping" at time r. Therefore, the basic rewrite modes may be qualified by a *time limit* r, which modifies the application of the tick rule to stop at time r, *if possible*, in the following way: Let Δ be the difference $r \dot{-} \tau(\alpha)$ between the time limit r and the total time elapse $\tau(\alpha)$ in the rewrite proof $\alpha : t_0 \longrightarrow t$ leading to the current state t. Then, if Δ is smaller than the time increase Δ' found by applying the basic mode, then the time duration variable is instantiated by Δ. If the application with the value Δ fails—either because Δ is 0, or because instantiating the time variable to Δ would not satisfy the condition of the tick rule—the duration variable is instantiated with Δ'.

The timed rewrite modes presented above may be generalized to allow user-definable functions (over time intervals, total time elapse, and, possibly, the number of times a nondeterministic tick rule has been applied) in future versions of the tool.

The application of tick rules with *deterministic* duration is not affected by the modes above, but the rewrite is not performed if the time increase is 0.

The following two *optimization modes* do not influence the time increase:

- The *global* mode allows the user to specify that all lefthand sides of rules in the module are terms of sort `System`. Adding this mode substantially speeds up search procedures, because the tool does not need to search for rewrites in all subterms of the state being rewritten.

- The *tokenOO* mode concerns object-oriented systems with tokens '`*`', where tick rules of the form `{`$TOK \quad CF$`}` $\xrightarrow{r}$ `{`$TOK \quad \delta(CF, r)$`}` (for TOK and CF variables of sort `Tokens` and `Configuration`) lead to inefficient specifications. This is because sort information cannot be injected early in the associative-commutative matching process, so that the matching algorithm first generates a possibly exponential number of potential matches, only one of which is well-sorted. To overcome this problem, in the tokenOO mode the variable of sort `Tokens` in tick rules of the form (‡) is instantiated with the number of tokens '`*`' in the (outer-most level) of the state before the application of the tick rule when the tokenOO mode is used.

3.2.2 Basic Timed Rewrite Strategies

The basic timed rewrite strategies are given by the Real-Time Maude commands

`(trew [`n`]` t `mode` *mode* `dti` Δ `in time` *limit* `.)`

and

(trew-in [n] *moduleName* : t mode *mode* dti Δ in time *limit* .)

where:

- n is the maximal number of (one-step) rewrites (there is no bound on the number of rewrites if $n =$ 0);
- t is the term (of sort `System`) to be rewritten;
- *mode* is the rewrite mode which is one of `d`, `m`, `ma`, `gd`, `gm`, `gma`, `tokend`, `tokenm`, or `tokenma`, where `d` denotes the basic default time increase mode, `m` the maximal time increase w.r.t. each match, `ma` the maximal time increase w.r.t. all matches, and where a `g` in front of either of these indicates the use of the global mode, where all rules rewrite terms of sort `System`, and where `token` in front of the basic modes indicates that the tokenOO mode, which instantiates the `Tokens` variable in rules of the form (‡) is used;
- Δ is the default time increase value;
- *limit* denotes the limit for the total time elapse $\tau(\alpha)$ in the rewrite proof $\alpha : t \longrightarrow t'$ to be performed by the strategy and may be `le` r, `lt` r, or `no time limit`, for some ground term r;
- *moduleName* is the name of the module in which the rewrite should take place.

These commands apply the rules in a fair, top-down way. If *limit* is of the form `le` r, then the given mode is qualified by this time limit. The application of rewrites ends if the next application would bring the total time elapse in the rewrite proof up to (in case the limit is of the form `lt` r) or beyond the time limit r (in case the limit is `le` r).

3.2.3 Search Strategies

The tool provides some commands to further analyze real-time specifications by exploring not just one, but many possible behaviors from the initial state. For dense time domains, in practice it may be impossible to explore individually all possible behaviors in the presence of tick rules with nondeterministic time increase, because of the existence of an infinite number of possible matches for the time variable. Instead, the tool provides strategies to explore all possible behaviors—up to a certain depth of the rewrite sequence, duration, or satisfaction of other conditions—when the new variable in the tick rule(s) is instantiated using one of the rewrite modes described earlier. Therefore, the search strategies mentioned in this chapter are implicitly understood to analyze all possible rewrite paths allowed within the chosen restrictions on the application of tick rules.

Search expressions can be *nested*, so that the set of rewrite paths—each path represented by a state with a timestamp—resulting from a search serves as input to other searches. In such cases, the given time limits apply to the total time elapse from the initial state, not just to the duration of the computation from the end of the previous search. (In contrast, when the search is bounded by the number of one-step sequential rewrites performed, each new subsearch starts with depth 0.)

All search commands are of the form (strat *SE* .) where *SE* is a nested search expression as described below.

Basic Search Expressions

The basic search expressions include:

- The simplest search expression, which has the form `term` t, where t is a ground term of sort `System`, denoting the initial state in a nested search. The command `(strat term` t `.)` returns the reduced form of the term t.

- The search expression

  `find [`n`]` n' `maximal terms reachable from` SE `mode` m `dti` Δ
  `within time` *limit*

  where:

  - n is a number denoting the maximal number of one-step sequential rewrites performed in each path in the current (sub)search, and where $n = 0$ means that there is no such limit;
  - n' is the upper bound on the number of solutions returned, where $n' = 0$ means "unlimited";
  - SE is a search expression;
  - m is one of the modes `d`, `m`, `ma`, `gd`, `gm`, `gma`, `tokend`, `tokenm`, or `tokenma` described in Section 3.2.1;
  - Δ is a ground term of sort `Time` denoting the default time increase value;
  - *limit* is, as before, either `le` r, `lt` r, or `no time limit` for r a ground term of sort `Time`;

  returns all terms reachable from the states returned by SE which cannot be further rewritten within the given time and/or depth limits.

  More precisely, let t_0 be the initial term of the whole search, and let $\alpha : t_0 \longrightarrow t$ be one solution of the (sub)search SE (represented by the term t `in time` $\tau(\alpha)$). Then, $\alpha;\beta : t_0 \longrightarrow t'$ is a result of the search only if:

  - $\beta : t \longrightarrow t'$ is a rewrite proof composed of admissible one-step sequential rewrites using the mode m with default time increase Δ, and, in case *limit* is `le` r, time limit $r \dot{-} \tau(\alpha)$; and
  - there is no non-trivial rewrite (using the modes) $\gamma : t' \longrightarrow t''$ such that $\tau(\alpha;\beta;\gamma)$ is within the given time limit, or the proof $\beta : t \longrightarrow t'$ consists of $n' \neq 0$ one-step sequential rewrites.

  A resulting rewrite $\alpha;\beta : t_0 \longrightarrow t'$ is represented in the multiset of solutions as t `in time` $\tau(\alpha;\beta)$ when $\tau(\alpha;\beta) \neq 0$, and by t otherwise.

  The number n' limits the size of the *multiset* of solutions returned, which may not be a *set* of (time-stamped) terms, since u `in time` r may represent two different rewrite paths leading to u in time r.

- The search expression of the form

find [n] n' deadlocks from SE mode m dti Δ within time *limit*

returns all *deadlocks* reachable within the given time and depth starting from the results of the search expression SE.

- The search expression keep n from SE returns no more than n *distinct* time-stamped terms from the results of the search expression SE.

Patterns

More advanced search strategies are defined in terms of whether a state, possibly together with the total time elapse in the path leading to the state, matches a *pattern*. A pattern is either the constant noTerm (which is not matched by any term), the constant anyPattern (which is matched by any term), or has the form $t(\overline{x})$, or $t(\overline{x})$ where $C(\overline{x})$. The term $t(\overline{x})$ should either be a term of sort System or a term of the form $u(\overline{x})$ in time $v(\overline{x})$, allowing us to state properties about the total time elapse. Furthermore, to efficiently being able to check the matching property, the term $t(\overline{x})$ is required to be *ground irreducible* w.r.t. the equations of the theory. That is, if for some substitution σ of the variables $\overline{x}$ in t, the term $t(\overline{x}\sigma)$ can be simplified by an equation, then so can $x_i\sigma$ for some x_i in $\overline{x}$.

A state w reached in a rewrite $\alpha : t_0 \longrightarrow w$ matches a pattern of sort System whenever there is a match from the pattern to w (which satisfies the where-condition if the pattern contains one); it matches a pattern of the form $u(\overline{x})$ in time $v(\overline{x})$ iff there is a match σ s. t. $u(\overline{x}\sigma) = w$ and $v(\overline{x}\sigma) = \tau(\alpha)$ (and the condition $C(\overline{x}\sigma)$ holds if the pattern contains one). Patterns may be *negated*, and a state "matches" a negated pattern P if and only if it does not match P. For example, a state w matches the negation of $t(\overline{x})$ where $C(\overline{x})$ is there does not exist any match from $t(\overline{x})$ to w which satisfies $C(\overline{x})$.

A *matching expression* is either of the form

[not] matching pattern $u(\overline{x})$ [in time $v(\overline{x})$]

or of the form

[not] matching extPattern $u(\overline{x})$ [in time $v(\overline{x})$] where $C(\overline{x})$

where square brackets denote optional parts and not negates the following pattern, or is either matching anyPattern, or matching noPattern, with the expected meaning.

The search expression remove ME from SE removes from SE the solutions which satisfy the matching expression ME.

Temporal Logic Properties

The search strategies described below are motivated by the desire to model-check real-time specifications w.r.t. properties stated in a more abstract logic, such as a suitable linear-time timed temporal logic (see e.g. [AH92]).

The set of *timed computations* starting from an initial term t_0 of sort `System` consists of all the infinite sequences and the non-extensible finite sequences

$$\sigma : \langle t_0, 0\rangle \longrightarrow \langle t_1, r_1\rangle \longrightarrow \langle t_2, r_2\rangle \longrightarrow \cdots$$

where for each $i \geq 1$, either the length of σ is less than $i + 1$, or there is an admissible one-step sequential ground rewrite $\alpha_i : t_{i-1} \longrightarrow t_i$ with $r_i = r_{i-1} + \tau(\alpha_i)$. The suffix $\langle t_i, r_i\rangle \longrightarrow \cdots$ of σ denotes a *timed path* and is written σ_i.

The tool currently supports model-checking of the following kinds of temporal properties:

- *Until properties*: A timed path σ_i satisfies the "until" property

  $$me\ \mathbf{U}_{\leq r}\ me'$$

  iff there exists a $j \geq i$ with $r_j \leq r$ s.t. $\langle t_j, r_j\rangle$ satisfies the matching expression me', and for all $i \leq k < j$, $\langle t_k, r_k\rangle$ satisfies the matching expression me. The temporal operators $\mathbf{U}_{<r}$, and $\mathbf{U}$ when there is no time limit, can be defined similarly.

- *Until/stable properties*: A timed path σ_i satisfies the "until/stable" property

  $$me\ \mathbf{UStable}_{\leq r}\ me'$$

  iff σ_j satisfies $me\ \mathbf{U}_{\leq r}\ me'$, and, in addition, if $\langle t_j, r_j\rangle$ satisfies the matching expression me', so must each $\langle t_k, r_k\rangle$ with $k > l$ and $r_k \leq r$. The operators $\mathbf{UStable}_{<r}$ and $\mathbf{UStable}$ can be defined in a similar way.

A pair $\langle t, r\rangle$ satisfies a temporal property φ iff each timed path σ_i starting with $\langle t, r\rangle$ satisfies φ. A set of such pairs satisfies φ if each pair in the set does so.

The following useful properties are special cases of the above temporal properties, but nevertheless have their own commands, given in Section 3.2.5, in Real-Time Maude:

- *Invariance properties*: All terms reachable within the given depth and time limits should satisfy the given state property. Invariance of a matching expression me is written $\Box_{\leq r}\ me$, and is defined by the formula

  $$\texttt{matching noPattern}\ \mathbf{UStable}_{\leq r}\ me.$$

  The operators $\Box_{<r}$ and $\Box$ can be defined similarly.

- *Liveness properties*: In each path, there is a state satisfying the matching expression me within the given depth and time limits. This liveness property, written $\Diamond_{\leq r}\ me$, is given by the formula

  $$\texttt{matching anyPattern}\ \mathbf{U}_{\leq r}\ me,$$

  with similar definitions of $\Diamond_{<r}$ and $\Diamond$.

A search expression of the form

`find [`n`]` n' `not satisfying` ME `Until` ME' `from` SE `mode` m `dti` Δ
`within time` $limit$

(where n, n', m, Δ are as in the previous search strategies, and where *limit* is either `le` r, `lt` r, or `no time limit`) searches for timed paths *not satisfying* the property $ME\ \mathbf{U}_{limit}\ ME'$ (where $\mathbf{U}_{limit}$ is, respectively, $\mathbf{U}_{\leq r}$,$\mathbf{U}_{<r}$, or $\mathbf{U}$) starting from the pairs returned by the search expression SE. The search strategy returns the pair where non-satisfaction of the property was established from the timed paths which do not satisfy the property. In case the time limit is of the form `le` r, the rewrite mode contains this time limit, so that, if possible, time r is visited.

Since for dense time models we cannot explore *all* rewrites in the presence of tick rules with a new variable, the result of the search may be only partially correct, in that a counterexample found by the search invalidates the property, while the absence of a counterexample does not guarantee that the property holds.

The positive version of the above search has the form

`find [`n`]` n' `satisfying` ME `Until` ME' `from` SE `mode` m `dti` Δ
`within time` $limit$

and returns the first term satisfying the matching expression ME' in timed paths satisfying the property. The existence of a positive path does not prove the satisfaction of the formula w.r.t. the initial terms, since other paths may fail to satisfy the property.[3]

A search expression of the form

`find [`n`]` n' `not satisfying` ME `UntilStable` ME' `from` SE `mode` m
`dti` Δ `within time` $limit$

returns some paths *not satisfying* the property $ME\ \mathbf{UStable}_{limit}\ ME'$, represented by the first pair where non-satisfaction is established.

The search functions used to execute these search expressions are implemented both by depth-first-search functions and breadth-first-search functions, which are not yet optimized. We currently use the depth-first-search functions to interpret the Real-Time Maude search commands.

3.2.4 Extending the Tool's Strategy Library

The data types and functions used to implement the various Real-Time Maude commands are accessible to the user of Real-Time Maude in the module `TIMED-META-LEVEL` described in Chapter 4. A user can extend the strategy library of Real-Time Maude by importing the module `TIMED-META-LEVEL` in a Real-Time Maude module and defining new strategies, as illustrated in Sections 3.3, 5.2.6, and 5.2.7.

[3]However, a positive path would prove an existentially quantified version of the formula in branching-time temporal logics.

3.2.5 The Real-Time Maude Command Language

The Real-Time Maude commands that we have described can be summarized by giving the commands which are not also Full Maude commands in the following extended BNF-grammar, where optional parts are enclosed by a pair of square brackets '[' and ']'; the symbols '{' and '}' are used as meta-parentheses; and '*' and '+' denote, respectively, zero or more, and one or more, repetitions of the preceding unit.

⟨RealTimeMaudeCommand⟩ ::= { `(` { ⟨FullMaudeCommand⟩ | ⟨TimedModule⟩
  | ⟨TimedRewriteCommand⟩ | ⟨ShowLazyCommand⟩ | ⟨SearchCommand⟩ } `)` }+
⟨TimedModule⟩ ::= `tmod` ⟨ModuleHeading⟩ `is` ⟨TimedModDecl⟩* `endtm`
  | `tomod` ⟨ModuleHeading⟩ `is` ⟨TimedModDecl⟩* `endtom`
  | `tth` ⟨ModuleHeading⟩ `is` ⟨TimedModDecl⟩* `endtth`
  | `toth` ⟨ModuleHeading⟩ `is` ⟨TimedModDecl⟩* `endtoth`
⟨TimedModDecl⟩ ::= ⟨ModDecl⟩
  | `lazy rl [`⟨LabelId⟩`] :` ⟨Term⟩ `=>` ⟨Term⟩ [`in time` ⟨Term⟩] `.`
  | `lazy crl [`⟨LabelId⟩`] :` ⟨Term⟩ `=>` ⟨Term⟩ [`in time` ⟨Term⟩] `if` ⟨Condition⟩ `.`
  | `rl [`⟨LabelId⟩`] :` ⟨Term⟩ `=>` ⟨Term⟩ `in time` ⟨Term⟩ `.`
  | `crl [`⟨LabelId⟩`] :` ⟨Term⟩ `=>` ⟨Term⟩ `in time` ⟨Term⟩ `if` ⟨Condition⟩ `.`
⟨TimedRewriteCommand⟩ ::= `trew [`⟨Nat⟩`]` ⟨Term⟩ `mode` ⟨Mode⟩ `dti` ⟨Term⟩ `in time`
  ⟨TimeLimit⟩ `.`
  | `trew-in [`⟨Nat⟩`]` ⟨ModuleName⟩ `:` ⟨Term⟩ `mode` ⟨Mode⟩ `dti` ⟨Term⟩ `in time`
  ⟨TimeLimit⟩ `.`
⟨ShowLazyCommand⟩ ::= `show lazy .` %%% Show the labels of lazy rules
  | `show lazy` ⟨ModuleName⟩ `.`
⟨SearchCommand⟩ ::= `strat` ⟨SearchExpr⟩ `.`
⟨SearchExpr⟩ ::= `term` ⟨Term⟩ %%% Initial term.
  | `keep` ⟨Nat⟩ `from` ⟨SearchExpr⟩ %%% Keep at most Nat distinct terms
  | `remove` ⟨MatchExpr⟩ `from` ⟨SearchExpr⟩
  | `find [`⟨Nat⟩`]` ⟨Nat⟩ `maximal terms reachable from` ⟨SearchExpr⟩ `mode` ⟨Mode⟩
  `dti` ⟨Term⟩ `within time` ⟨TimeLimit⟩
  | `find [`⟨Nat⟩`]` ⟨Nat⟩ `deadlocks from` ⟨SearchExpr⟩ `mode` ⟨Mode⟩ `dti` ⟨Term⟩
  `within time` ⟨TimeLimit⟩ %%% Deadlocks
  | `find [`⟨Nat⟩`]` ⟨Nat⟩ `not satisfying []` ⟨MatchExpr⟩ `from` ⟨SearchExpr⟩ `mode`
  ⟨Mode⟩ `dti` ⟨Term⟩ `within time` ⟨TimeLimit⟩ %%% Invariance
  | `find [`⟨Nat⟩`]` ⟨Nat⟩ `satisfying` ⟨MatchExpr⟩ `from` ⟨SearchExpr⟩ `mode` ⟨Mode⟩
  `dti` ⟨Term⟩ `within time` ⟨TimeLimit⟩ %%% "Positive" liveness
  | `find [`⟨Nat⟩`]` ⟨Nat⟩ `not satisfying <>` ⟨MatchExpr⟩ `from` ⟨SearchExpr⟩ `mode`
  ⟨Mode⟩ `dti` ⟨Term⟩ `within time` ⟨TimeLimit⟩ %%% "Negative" liveness
  | `find [`⟨Nat⟩`]` ⟨Nat⟩ [`not`] `satisfying` ⟨MatchExpr⟩ `Until` ⟨MatchExpr⟩ `from`
  ⟨SearchExpr⟩ `mode` ⟨Mode⟩ `dti` ⟨Term⟩ `within time` ⟨TimeLimit⟩
  | `find [`⟨Nat⟩`]` ⟨Nat⟩ `not satisfying` ⟨MatchExpr⟩ `UntilStable` ⟨MatchExpr⟩ `from`
  ⟨SearchExpr⟩ `mode` ⟨Mode⟩ `dti` ⟨Term⟩ `within time` ⟨TimeLimit⟩
⟨MatchExpr⟩ ::= [`not`] `matching pattern` ⟨Term⟩
  | [`not`] `matching extPattern` ⟨Term⟩ `where` ⟨Term⟩
  | `matching anyPattern` | `matching noPattern`
⟨TimeLimit⟩ ::= `no time limit` | `le` ⟨Term⟩ | `lt` ⟨Term⟩

⟨Mode⟩ ::= d | m | ma | gd | gm | gma | tokend | tokenm | tokenma

3.2.6 A Sample Real-Time Maude Specification and Execution

This section illustrates the use of the Real-Time Maude tool on a very simple example. More interesting examples case studies will be given in the next section and in Chapter 5.

The following object-oriented timed module includes the module NATURALS given in Appendix A defining the natural numbers and defines a class C with two "counter" attributes a1 and a2, and a clock attribute. The time domain is specified to be the nonnegative machine integers using the techniques described in Section 3.1[4]:

```
(tomod TEST is
   including QID .
   including NATURALS .
   subsort Nat < Time .
   subsort Qid < Oid .
   class C | a1 : Nat, a2 : Nat, clock : Time .
   op initTerm : -> System .
   vars N N' : Nat .
   eq zero = 0 .
   eq N lt N' = N < N' .
   eq N plus N' = N + N' .
   eq initTerm = {< 'q : C | a1 : 0, a2 : 0, clock : 0 >} .
endtom)
```

The above module TEST does not satisfy the conditions on real-time rewrite theories given in Def. 2, since it contains an operator of sort System other than {_}, namely the constant initTerm. For convenience, Real-Time Maude supports such definitions of initial terms—that is, we allow function symbols $f : s_1 \ldots s_n \rightarrow$ System where $s_i \neq$ System for each i, if each ground term of sort System reduces (by the equations in the theory) to a term of the form {t}, and no term {u} reduces to a term not of the form {t}.

The following module adds three rules, two of which are *lazy*, one of which is a tick rule, to the timed module TEST:

```
(tomod TEST1 is
   including TEST .
   var Q : Oid .
   vars N N' : Nat .

   crl [r1] : < Q : C | a1 : N > => < Q : C | a1 : N + 1 > if N <= 5 .
   lazy crl [r2] : < Q : C | a2 : N > => < Q : C | a2 : N + 1 > if N <= 4 .
   lazy crl [tick1] : {< Q : C | clock : N >} => {< Q : C | clock : N + N' >}
                      in time N' if N' le 100 .
endtom)
```

[4]Note that the operation monus is already defined in the module NATURALS.

Due to the presence of lazy and eager rules, and nondeterministic time elapse in the tick rule, Maude's default interpreter cannot be used to execute this specification. Instead, we use Real-Time Maude's default timed rewrite strategy with maximal time increase mode to execute a run[5]:

```
Maude> (trew [0] initTerm mode m dti 1000 in time le 466 .)
rewrites: 26657 in 120ms cpu (130ms real) (222141 rewrites/second)
[ ... ]
Result: { < 'q : C | clock : 466, a2 : 5, a1 : 6 > } in time 466 .

Maude> (trew [8] initTerm mode m dti 1000 in time no time limit .)
rewrites: 3831 in 10ms cpu (30ms real) (383100 rewrites/second)
[ ... ]
Result: { < 'q : C | clock : 100, a2 : 1, a1 : 6 > } in time 100 .
```

The following execution returns five terms, together with the duration of the rewrites leading to them, reachable from `initTerm` which can not be further rewritten in paths of length less than or equal to 10, or with duration less than or equal to `447`:

```
Maude> (strat find [10] 5 maximal terms reachable from term initTerm mode m dti 1000
        within time le 477 .)
rewrites: 136128 in 480ms cpu (560ms real) (283600 rewrites/second)
[ ... ]
Result:
{< 'q : C | clock : 0, a2 : 4, a1 : 6 >}
{< 'q : C | clock : 100, a2 : 3, a1 : 6 >} in time 100
{< 'q : C | clock : 100, a2 : 3, a1 : 6 >} in time 100
{< 'q : C | clock : 100, a2 : 3, a1 : 6 >} in time 100
{< 'q : C | clock : 200, a2 : 2, a1 : 6 >} in time 200
```

The state `{< 'q : C | clock : 100, a2 : 3, a1 : 6 >}` reachable in time `100` appears three times, each occurrence representing a distinct rewrite from the initial state.

Similarly, we can find all deadlocked terms, that is, terms which cannot be further rewritten using the strategies for applying tick rules, reachable in no more than time 300:

```
Maude> (strat find [0] 1 deadlocks from term initTerm mode m dti 1000
       within time le 300 .)
rewrites: 2252260 in 8210ms cpu (16140ms real) (274331 rewrites/second)
[ ... ]
Result:
No terms!
```

To make the execution of the search commands less trivial, we make all rules eager:

```
Maude>
(tomod TEST2 is
   including TEST .
   var Q : Oid .
```

[5] We often write `[ ... ]` instead of Real-Time Maude's "echo" of the command given.

```
   vars N N' : Nat .

   crl [r1] : < Q : C | a1 : N > => < Q : C | a1 : N + 1 > if N <= 5 .
   crl [r2] : < Q : C | a2 : N > => < Q : C | a2 : N + 1 > if N <= 4 .
   crl [tick1] : {< Q : C | clock : N >} => {< Q : C | clock : N + N' >}
                       in time N' if N' le 100 .
endtom)
```

The following search searches for five states, reachable in no more than 10 one-step sequential rewrites and with duration less than `300`, where both counters have value `4`:

```
Maude> (strat find [10] 5 satisfying matching pattern
                   ({< Q : C | a1 : 4, a2 : 4, clock : N >})
       from term initTerm mode m dti 1000 within time lt 300 .)
rewrites: 2285518 in 7040ms cpu (13870ms real) (324647 rewrites/second)
[ ... ]
Result:
{< 'q : C | clock : 0 , a2 : 4, a1 : 4 >}
{< 'q : C | clock : 100, a2 : 4, a1 : 4 >} in time 100
{< 'q : C | clock : 100, a2 : 4, a1 : 4 >} in time 100
{< 'q : C | clock : 200, a2 : 4, a1 : 4 >} in time 200
{< 'q : C | clock : 200, a2 : 4, a1 : 4 >} in time 200
```

The following nested search expression first finds five states where both counters have `4`, and, starting with these terms and their respective duration, finds one term in no more than five additional rewrite steps, with duration greater than or equal to `300` and where the counter `a1` has value `6`:

```
Maude> (strat find [5] 1 satisfying matching extPattern
       ({< Q : C | a1 : 6, a2 : N', clock : N >}) where N ge 300
       from
          find [10] 5 satisfying matching pattern
          ({< Q : C | a1 : 4, a2 : 4, clock : N >}) from term initTerm
          mode m dti 1000 within time no time limit
       mode m dti 1000 within time lt 400 .)
rewrites: 2942565 in 9050ms cpu (17880ms real) (325145 rewrites/second)
[ ... ]
Result:
{ < 'q : C | clock : 300, a2 : 4, a1 : 6 > } in time 300
```

We may need variables in search patterns which are not present in the current module. In these cases, a new module with the appropriate variables can be defined:

```
Maude> (tomod varTEST2 is
          including TEST2 .
          var ATTS : AttributeSet .
        endtom)
rewrites: 23373 in 140ms cpu (150ms real) (166950 rewrites/second)

Introduced timed module: varTEST2
Maude> (strat find [10] 5 satisfying matching pattern
```

```
                    ({< 'q : C | a1 : 4, ATTS >})
       from term initTerm mode m dti 1000 within time lt 300 .)
rewrites: 165725 in 510ms cpu (710ms real) (324950 rewrites/second)
[ ... ]
Result:
{< 'q : C | clock : 0 , a2 : 0 , a1 : 4 >}
{< 'q : C | clock : 0 , a2 : 1 , a1 : 4 >}
{< 'q : C | clock : 0 , a2 : 2 , a1 : 4 >}
{< 'q : C | clock : 0 , a2 : 5 , a1 : 4 >}
{< 'q : C | clock : 100 , a2 : 5 , a1 : 4 >} in time 100
```

We can find the lazy rules in the current module `TEST2` and the module `TEST1` as follows:

```
Maude> (show lazy .)
rewrites: 25 in 30ms cpu (40ms real) (833 rewrites/second)
The lazy rules of module TEST2 ( including imported rules ) are: .

Maude> (show lazy TEST1 .)
rewrites: 23 in -10ms cpu (0ms real) (~ rewrites/second)
The lazy rules of module TEST1 ( including imported rules ) are: r2 tick1 .
```

Finally, we illustrate the transformation of declarations importing `TIMED-META-LEVEL` with a parameter:

```
Maude> (mod META-TEST1 is
          protecting TIMED-META-LEVEL[TEST1] .
        endm)
rewrites: 2250 in 620ms cpu (1230ms real) (3629 rewrites/second)

Introduced module: META-TEST1
Maude> (show module .)
rewrites: 146 in 550ms cpu (840ms real) (265 rewrites/second)
mod META-TEST1 is
        including META-LEVEL[TEST1] .
        including TIMED-META-LEVEL .
        including BOOL .
        including UP .
        op lazyLabels-TEST1 : -> QidSet .
        eq lazyLabels-TEST1 = 'r2 ; 'tick1 .
endm
```

3.2.7 Known Problems

- The parser have some problems parsing terms (in the timed modules) in strategy expressions. Therefore, terms representing initial states, patterns in searches, time limits, or *dti*-values may need to be enclosed in parentheses `( )`.
- The tool cannot handle multiple parameters of the module `TIMED-META-LEVEL`, since the tool transforms such an importation to the corresponding importation of the module `META-LEVEL`, and version 1.0.5 of Full Maude cannot handle multiple parameters of the module `META-LEVEL`.

3.3 A Scheduling Case Study

We show in this section how Real-Time Maude can be used to specify and analyze scheduling problems. We consider scheduling problems in which all periods and execution times are fixed rational numbers, and where no preemption is allowed. Our specification method allows us to analyze scheduling problems with an arbitrary number of processes and processors without requiring any changes to the specification.

3.3.1 The Scheduling Problem

We are given a number of identical *processors* and a set $\{p_i \mid i \in I\}$ of *periodic processes*, where each p_i has a period $\pi(p_i) \in \mathbf{Q}^+$ and an execution time $\epsilon(p_i) \in \mathbf{Q}^+$. A *schedule* is an allocation of the processors to the processes as a function of time such that:

1. No processor serves more than one process simultaneously.
2. Each process p_i must have been served continuously by one processor for exactly time $\epsilon(p_i)$ in its last completed period.
3. Each process is served an unbounded number of periods.

The scheduling problem is usually addressed by specifying all behaviors of the system which satisfy the first two requirements (the "mutual exclusion" specification) and then finding behaviors which satisfy the last requirement.

3.3.2 Specifying Mutual Exclusion

A state of the system can be seen as an object-oriented system consisting of a (multi-) set of objects representing the processes, and of a number of special objects `PRCSSR`, where each occurrence of `PRCSSR` denotes the availability of a processor. We do not want to restrict the initial state to a certain number of processes (as must be done in less flexible formalisms like automata or Petri nets, where the specification must be changed—often done by "composing" automata or nets—for each combination of processes and processors). Since time acts on each process object, and the maximum time elapse depends on each process object, we should use the specification techniques described in Section 2.2 for such object-oriented systems, with the use of $*$'s, *mte*, and δ, the latter two distributing over the objects in the configuration.

A process is represented by an object of class `Process` with the following attributes:

- `state` denotes the state of the process: `sleep` indicates that it has done everything in the current period and just waits for the period to end, at which time it goes to state `wait` waiting to be served by a processor, and is in state `usesPRCSSR` while being served by a processor;
- `period`, denotes the (length of the) period of the process;
- `exTime`, denotes the (length of the) execution time of the process;
- `timeInPeriod`, denotes the amount of time spent in the current period;

- `timeExInPeriod`, denotes the amount of processor time used in the current execution.

Note that, since the initial state can have an arbitrary number of processes *and* processors, our `MUTEX` specification below can deal with a very general class of scheduling problems.

It is also important to clarify how the system should evolve if some process cannot be served in one of its periods. To let the execution continue could be misunderstood as meaning that everything has gone well. In contrast to some models of real-time systems (where time elapse is not modeled explicitly and where time divergence therefore must be required in order for time to advance at all), our specification formalism does not require time divergence. Therefore, when time cannot advance without some process being served, the system just timelocks. This avoids the need to treat these cases using "error-states."

Our specification of mutual exclusion uses a function `mte` to compute the maximum possible time elapse in a configuration. Time can elapse until the end of the period of a process in state `sleep`, when it should begin a new period. A process p cannot stay in `wait` mode longer than $\pi(p) - \epsilon(p)$, and a process in state `usesPRCSSR` must change its state when its execution time is over. The specification uses also a function `delta` which defines the effect of time elapse on the configuration by updating the clocks which measure current time in the period and current time "under execution."

The complete Real-Time Maude specification of mutual exclusion given below has four rules: `endOfRound`, `execute`, `release`, and `tick`. The rule `endOfRound` updates the necessary attributes when a new round is started, that is, when the time in the current period equals the length of the period. The rule `execute` grabs an available processor if there is enough time left in its period to execute the task. The rule `release` releases a processor when it has been used for exactly the required amount of time, and the `tick` rule is as described in Section 2.2. We think that it is relatively easy, compared to mutual exclusion specification in other formalisms, to see—or to prove, using invariance reasoning—that the specification satisfies the mutual exclusion property.

The module also declares two initial states: `initTerm1`, which has one processor and two processes, with periods 5 and 7 and execution times 2 and 4, and `initTerm2`, which has two processors and three processes. The specification imports the module `POSITIVE-RATIONALS` that defines the data type of positive machine rational numbers by extending the machine integers, and contains definitions of the functions `_plus_`, `_monus_`, and `_lt_` on the positive rationals.

```
(tomod MUTEX is
  including QID .
  including POSITIVE-RATIONALS .     *** Include the rationals
  including LTIME-INF .              *** Include INF, min, and max
  subsort Qid < Oid .

  *** The nonnegative rationals (PosRat) is the time domain:
  subsort PosRat < Time .
  eq zero = 0 .

  op delta : Configuration Time -> Configuration .
  op mte : Configuration -> TimeInf .

  class Process | state : PrState, period : Time, exTime : Time,
                  timeInPeriod : Time, timeExInPeriod : Time .
```

```
op PRCSSR : -> Configuration .  *** Special object denoting an available processor

sort PrState .                  *** The states of a process
ops sleep wait usesPRCSSR : -> PrState .

var TOKENS : Tokens .  vars C C' : Configuration .
var Q : Qid .  vars R R' R'' : Time .

eq mte(< Q : Process | state : sleep, period : R, timeInPeriod : R' >)
       = R monus R' .
eq mte(< Q : Process | state : wait, period : R, exTime : R',
                      timeInPeriod : R'' >) = (R monus R') monus R'' .
eq mte(< Q : Process | state :  usesPRCSSR,  exTime : R,
                       timeExInPeriod : R' >) = R monus R' .
eq mte(none) = INF .
eq mte(PRCSSR) = INF .
ceq mte(C C') = min(mte(C), mte(C')) if  C =/= none and C' =/= none .
eq delta(none, R) = none .
eq delta(PRCSSR, R) = PRCSSR .
ceq delta(C C', R) = delta(C, R) delta(C', R) if C =/= none and C' =/= none .
eq delta(< Q : Process | state : sleep, timeInPeriod : R >, R') =
      < Q : Process | state : sleep, timeInPeriod : R plus R' > .
eq delta(< Q : Process | state : wait, timeInPeriod : R >, R') =
      < Q : Process | state : wait, timeInPeriod : R plus R' > .
eq delta(< Q : Process | state : usesPRCSSR, timeInPeriod : R ,
                         timeExInPeriod : R' >, R'') =
      < Q : Process | state : usesPRCSSR, timeInPeriod : R plus R'',
                      timeExInPeriod : R' plus R'' > .

rl [endOfRound] :
   * < Q : Process | state : sleep, period : R, timeInPeriod : R > =>
   * < Q : Process | state : wait,  timeInPeriod : 0, timeExInPeriod : 0 > .

crl [execute] :
   * < Q : Process | state : wait, period : R, exTime : R',
                     timeInPeriod : R'' > PRCSSR =>
   * < Q : Process | state : usesPRCSSR, timeExInPeriod : 0 >
     if R'' le R monus R' .

rl [release] :
   * < Q : Process | state : usesPRCSSR, exTime : R, timeExInPeriod : R > =>
   * < Q : Process | state : sleep, timeExInPeriod : 0 > PRCSSR .

crl [tick] : { TOKENS C } => { TOKENS delta(C, R) }  in time R
             if R le mte(C) .

*** Initial states:
ops initTerm1 initTerm2 : -> System .
eq initTerm1 =
   { * < 'p1 : Process | state : wait, period : 5, exTime : 2,
                         timeInPeriod : 0, timeExInPeriod : 0 >
     * < 'p2 : Process | state : wait, period : 7, exTime : 4,
                         timeInPeriod : 0, timeExInPeriod : 0 > PRCSSR } .
```

```
  eq initTerm2 =
     { * < 'p1 : Process | state : wait, period : 6, exTime : 2,
                           timeInPeriod : 0, timeExInPeriod : 0 >
       * < 'p2 : Process | state : wait, period : 7, exTime : 3,
                           timeInPeriod : 0, timeExInPeriod : 0 >
       * < 'p3 : Process | state : wait, period : 5, exTime : 2,
                           timeInPeriod : 0, timeExInPeriod : 0 >
       PRCSSR PRCSSR } .
endtom)
```

3.3.3 Analyzing the Mutual Exclusion Specification

Having the rationals as time domain combined with the need to be able to stop time elapse at any time to model all possible behaviors, implies that the time elapse in the tick rule is nondeterministic. Therefore, Maude's default interpreter cannot be used. The following shows the results of executing the specification using the maximal time increase and default time increase strategies, starting in a state with one processor and two processes, with periods 5 and 7, and execution times 2 and 4.

Executing the term `initTerm1` using the "default" timed rewrite strategy, in this case using maximal time increase mode with default time increase 100, and without depth or time limits, gives the following result:

```
Maude> (trew [0] initTerm1 mode m dti 100 in time no time limit .)
[ ... ]
Result: { * * < 'p1 : Process | timeExInPeriod : 0,
   timeInPeriod : 3, exTime : 2, period : 5, state : usesPRCSSR >
 < 'p2 : Process | timeExInPeriod : 0, timeInPeriod : 3, exTime : 4,
   period : 7, state : wait > } in time 3 .
```

Analyzing this result shows a deadlock: time advances by three, then `'p1` grabs the processor, and time cannot advance further, since `'p2` cannot advance in time.

Deciding Whether a Schedule Exists

For deciding the scheduling problem and finding all schedules we need the following observations about properties of our mutual exclusion specification and of scheduling:

1. Starting from an initial state where all processes are at the beginning of their periods in state `wait`, there is a schedule if and only if there is a time $r^* > 0$ where each process is at the beginning of its period. This property depends on the lengths of the periods being rational numbers, and one such r^*-value is $\Pi_{i \in I}\, m_{p_i}$ if, for each $i \in I$, $\pi(p_i) = \frac{m_{p_i}}{n_{p_i}}$ for natural numbers m_{p_i} and n_{p_i}.

2. In the specification of mutual exclusion, if the system reaches such a time r^* or a greater time, then all processes have been served in each of their rounds until time r^*. Conversely, if there is no rewrite from the intial state with a duration of at least time r^*, then there is no way of serving all the processes all the way to time r^*, and hence there is no schedule. Therefore, reachability of a state in time r^* decides whether there is schedule.

That is, there is a schedule if and only if the initial state rewrites to this same initial state in time greater than 0, or equivalently, if time can elapse more than time r^* for r^* given above. However, this is not sufficient to search for schedules in Real-Time Maude, since the time advance in tick rule is nondeterministic. We therefore need to use the following additional property:

3. Assume that there is a schedule where a process p is served at starting time $r+\Delta$ ($\Delta < \pi(p)$), where the process was ready to perform the task at time r, and where the processor serving the process was not serving any process in (parts of) the time interval $[r, r+\Delta]$. Then, there is a also a schedule where the task is performed starting at time r instead of at time $r+\Delta$, and where the schedule otherwise is the same.

This last property implies that if there is a schedule, then there is another schedule equivalent up to "time padding" in the sense that the processes execute at the earliest possible times for the given order of process execution. In our specification, this means that if there is a schedule, then there is also an equivalent schedule up to time padding, where all applications of the tick rule advance time by the maximum possible amount.

A schedule therefore exists for `initTerm1` if and only if there is a rewrite with total time elapse at least 35, when the maximal time elapse rewrite mode is used when applying the tick rule. (Note that $\frac{4}{7}+\frac{2}{5}=\frac{34}{35}$, which implies that it is far from obvious that a schedule exists.) We can check whether there is a reachable state with time elapse 35 by ascertaining whether the pattern `SYS in time 35`, where `SYS` is a variable of sort `System`, can be reached. First, we need to introduce the variable `SYS`:

```
Maude> (tomod varMUTEX is including MUTEX .
          var SYS : System .
        endtom)
```

Then, we can check the liveness property by looking for a term which matches the pattern, and where the time limit is 35, and there is no limit on the depth of the derivations:

```
Maude> (strat find [0] 1 satisfying matching pattern (SYS in time 35)
                   from term initTerm1 mode m dti 100 within time le 35 .)
[ ... ]
Result:
{ * * PRCSSR < 'p1 : Process | timeExInPeriod : 0,
   timeInPeriod : 5, exTime : 2, period : 5, state : sleep >
 < 'p2 : Process | timeExInPeriod : 0, timeInPeriod : 7, exTime : 4,
   period : 7, state : sleep > } in time 35
```

Similarly, we can find out whether there is a schedule for `initTerm2` without computing the product of the numerators of the periods, with the following liveness query (where `R` has previously been introduced as a variable of sort `Time`):

```
Maude> (strat find [0] 1 satisfying matching extPattern (SYS in time R)
                   where (SYS == initTerm2 and R =/= 0)
                   from term initTerm2 mode m dti 100 within time le 300 .)
[ ... ]
```

```
Result:
{ * * * PRCSSR PRCSSR < 'p1 : Process | timeExInPeriod : 0, timeInPeriod : 0,
   exTime : 2, period : 6, state : wait >
 < 'p2 : Process | timeExInPeriod : 0, timeInPeriod : 0, exTime : 3,
   period : 7, state : wait >
 < 'p3 : Process | timeExInPeriod : 0, timeInPeriod : 0, exTime : 2,
   period : 5, state : wait > } in time 210
```

3.3.4 Finding All Schedules

We exemplify how Real-Time Maude's strategy library presented in Section 4.2 can be extended to exhibit all schedules (up to time padding) of the scheduling problems presented above.

In the following module, the function `findSchedule` returns all executions, that is, all sequences of one-step sequential rewrites, where the initial state is reached again. Such executions are therefore sequences of the form

$$t_0 \texttt{ ==> } t_1 \texttt{ ==> } \cdots \texttt{ ==> } t_n = t_0$$

for $t_0, \ldots, t_n$ (meta-representations) of terms, where t_0 is the initial state, and where $t_0 \neq t_j$ for $0 < j < n$ with $t_n = t_0$ reached with duration different from 0. The constant `emptyExecutionSet` is returned when no schedule exists.

The function is implemented by a (non-optimized) breadth-first-search strategy in (Real-Time) Maude's metalevel. It uses the `TIMED-META-LEVEL` function `allOneStepRewrites` (which finds all terms reachable in one step using the given rewrite mode) to find the potential next states in an execution. To ensure that the function is terminating (while returning *all* schedules), it uses the `TIMED-META-LEVEL` function `notTooOld` (which returns those states that have been reached in some time less than or equal to a given time) to cut off executions whose total time elapse is greater than the total time elapse in (the first iteration of) a schedule already found.

```
(fmod FIND-SCHEDULES is
 including TIMED-META-LEVEL[MUTEX] .

 *** Datatypes for executions.
 sort Execution .
 subsort Term < Execution .
 op _==>_ : Execution Term -> Execution .

 sort ExecutionSet .
 subsort Execution < ExecutionSet .
 op emptyExecutionSet : -> ExecutionSet .
 op executionSet : ExecutionSet ExecutionSet -> ExecutionSet
                                   [assoc comm id: emptyExecutionSet] .
 vars EXECUTION EXECUTION' : Execution .
 vars EXSET EXSET' : ExecutionSet .
 vars T T' T'' : Term .
 vars Q Q' : Qid .
 vars TS TS' : TermSet .
 var TL : TermList .
 var DT : DefTerm .
```

```
op findSchedule : Term -> ExecutionSet .
*** Usage: findSchedule(initialState)

eq findSchedule(T) =
      findSchedule(meta-reduce(MUTEX, T),
                   emptyExecutionSet, noTerm, meta-reduce(MUTEX, T)) .
op findSchedule : ExecutionSet ExecutionSet
                      DefTerm Term -> ExecutionSet .
*** Usage: findSchedule(executionsToCheckInRound,
***                     executionsToCheckInNextRound,
***                     durationOfScheduleFound, initialState)

*** This round is over:
eq findSchedule(emptyExecutionSet, executionSet(EXECUTION, EXSET), DT, T) =
     executionSet(executionEndsWithInNonZeroTime(
                                executionSet(EXECUTION, EXSET), T),
                  findSchedule(executionSet(EXECUTION, EXSET),
                               emptyExecutionSet,
                               if DT == noTerm then
                               getExTime(executionEndsWithInNonZeroTime(
                                         executionSet(EXECUTION, EXSET), T))
                               else DT fi, T))  .

eq findSchedule(emptyExecutionSet, emptyExecutionSet, DT, T) =
      emptyExecutionSet .

*** Now, compute reachable terms in this round:
eq findSchedule(executionSet(EXECUTION, EXSET), EXSET', DT, T) =
      findSchedule(EXSET, executionSet(extendWithAllNextRewrites(EXECUTION,
                                          notTooOld(MUTEX, DT,
                                          allOneStepTimedRewrites(MUTEX,
                                             rightTerm(EXECUTION), none,
                                             maxTime('1000'Time)))),
                                       EXSET'), DT, T) .

*** Extract those executions which ends with the initial state in time > 0:
op executionEndsWithInNonZeroTime : ExecutionSet Term -> ExecutionSet .
eq executionEndsWithInNonZeroTime(emptyExecutionSet, T) =
        emptyExecutionSet .
ceq executionEndsWithInNonZeroTime(executionSet(EXSET, EXSET'), T) =
       executionSet(executionEndsWithInNonZeroTime(EXSET, T),
                    executionEndsWithInNonZeroTime(EXSET', T))
    if EXSET =/= emptyExecutionSet and EXSET' =/= emptyExecutionSet .
eq executionEndsWithInNonZeroTime(EXECUTION, T) =
      if findTimeTermNonZero(T, rightTerm(EXECUTION))
      then EXECUTION
      else emptyExecutionSet
      fi .

op rightTerm : Execution -> Term .
eq rightTerm(T) = T .
eq rightTerm(EXECUTION ==> T) = T .
```

```
 op extendWithAllNextRewrites : Execution TermSet -> ExecutionSet .
 eq extendWithAllNextRewrites(EXECUTION, emptyTermSet) = emptyExecutionSet .
 ceq extendWithAllNextRewrites(EXECUTION, termSet(TS, TS')) =
        executionSet(extendWithAllNextRewrites(EXECUTION, TS),
                     extendWithAllNextRewrites(EXECUTION, TS'))
           if TS =/= emptyTermSet and TS' =/= emptyTermSet .
 eq extendWithAllNextRewrites(EXECUTION, T) = EXECUTION ==> T .

 op  findTimeTermNonZero : Term Term -> Bool .
 *** findTimeTermNonZero(T, T') holds iff T' is T in time R with R
 *** different from 0
 eq findTimeTermNonZero(T, '_in`time_[T', T'']) =
       meta-reduce(MUTEX, '_==_[T, T']) == 'true'Bool and
       meta-reduce(MUTEX, '_=/=_[T'', '0'Time]) == 'true'Bool .
 ceq findTimeTermNonZero(T, Q[TL]) = false if Q =/= '_in`time_ .
 eq findTimeTermNonZero(T, Q) = false .
 eq findTimeTermNonZero(T, QQ') = false .

 *** Extend getTime to the last term of executions:
 op getExTime : ExecutionSet -> DefTerm .
 eq getExTime(emptyExecutionSet) = noTerm .
 eq getExTime(executionSet(EXECUTION, EXSET)) = getTime(rightTerm(EXECUTION)) .
endfm)
```

The following command gives the set of executions defining all the schedules from initial state `initTerm1` up to time padding:

```
Maude> (red findSchedule({'initTerm1}'System) .)
```

This call returns a set of execution sentences, the first of which represents the following schedule:

time	0	2	6	8	12	14	18	20	22	26	28	32
process	p_1	p_2	p_1	p_2	p_1	p_2	p_1	p_1	p_2	p_1	p_2	p_1.

If instead, the process `'p2` has execution time 5, then there is no schedule for the problem:

```
Maude> (red findSchedule(up(MUTEX, { * < 'p1 : Process | state : wait,
                                     period : 5, exTime : 2,
                                     timeInPeriod : 0, timeExInPeriod : 0 >
                                     * < 'p2 : Process | state : wait,
                                     period : 7, exTime : 5, timeInPeriod : 0,
                                     timeExInPeriod : 0 > PRCSSR })) .)
[ ... ]
result ExecutionSet : emptyExecutionSet
```

The function `findSchedule` defines an algorithm for finding all schedules up to time padding when all processes start in the beginning of their periods, that is, when the value of the `timeInPeriod` is `0` for each process. One could also think of starting in a more arbitrary state, starting in the middle of some periods, with some tasks already performed, etc. The solution presented above can be modified to handle such initial states by first checking whether it is possible to arrive at a state where all "period clocks" are `0`, and then start with the normal algorithm for finding schedules. Since our strategy uses comparisons between states, it can easily be accommodated to handle this more general setting.

3.4 Other Case Studies

In addition to the scheduling problem and some simple examples, the tool has been used in the following case studies:

- The generalized railroad crossing controller (see, e.g. [HL94]), which is a benchmark example of real-time systems.
- Development of an execution environment for Sifakis' *timed automata with deadlines* (TAD) model [BST98]. The *deadline* of a transition in a TAD is given as a boolean combination of linear inequalities over the clocks in the system. Our TAD tool computes the maximum time elapse until some deadline becomes true. The TAD tool can be used to execute one run of a TAD using either the default time increase strategy or the maximal time increase strategy, and can execute all runs in a TAD, up to a certain length, using the maximal time increase strategy.
- In collaboration with M. Keaton and S. Zabele at TASC, S. Bhattacharyya at Sprint, and C. Talcott at Stanford, we have specified and formally analyzed the AER/NCA [KBK$^+$99] suite of active network protocol components for reliable multicast. This is perhaps the most complex suite of protocols studied so far using Maude, because of their time-sensitive behavior and the need to analyze both correctness and performance. The Real-Time Maude tool has allowed us to find important errors and inconsistencies in the Maude specification derived from the original use-case informal specification. Chapter 5 describes the AER/NCA protocol and our specification and analysis effort of this protocol suite in detail.

Chapter 4

The Maude Specification of Real-Time Maude

This chapter shows how Maude's reflective features, together with its `LOOP-MODE` and parsing facilities, have been used to specify the Real-Time Maude tool in Maude. Section 4.2 presents the specification of Real-Time Maude's strategy library, which is used to execute Real-Time Maude's search and model checking commands, and to which the Real-Time Maude user has access through the module `TIMED-META-LEVEL`. Section 4.3 shows how the specification of Full Maude [DM99] has been extended to a specification of the Real-Time Maude tool.

4.1 Overview of the Real-Time Maude Specification

The Real-Time Maude specification extends Full Maude to allow it to reuse Full Maude's facilities for using parameterized and object-oriented modules, its operations for parsing modules and commands, its other useful operations such as the functions `up` and `down`, and its facilities for letting a user-defined module have easy access to the meta-representation of other user-defined modules.

Real-Time Maude extends Full Maude in the following ways:

- The `GRAMMAR` module in Full Maude which is used to parse the input (the first time) is extended to a module `TIMED-GRAMMAR`, which describes the syntax of timed modules and timed execution and search commands.
- The persistent state is extended by having a table, where each timed module is assigned to the set of the labels of the lazy rules in the module and its submodules.
- The module `TIMED-PRELUDE` described in Sections 3.1 and 4.3.7, which is imported by timed modules in the module transformation to untimed modules, is added as a "built-in" module to the database. We also add modules `LTIME`, `TIME-INF`, and `LTIME-INF` extending the time domain, and the module `TIMED-OO-PRELUDE`, which is imported implicitly by all object-oriented timed modules and theories.
- Input to the Real-Time Maude tool is first parsed w.r.t. the extended module `TIMED-GRAMMAR`. If the resulting term indicates that the input is a timed module or theory, then, *without*

extracting the signature of the module being defined, the term is transformed into a term which can be parsed by Full Maude as an untimed module. The only changes made are:

  – The terms corresponding to lazy rules are changed into normal rule declarations.
  – A declaration importing the module `TIMED-PRELUDE`, and `TIMED-OO-PRELUDE` in case of object-oriented modules, is added to the term.
  – If the module `TIMED-META-LEVEL` is imported with a parameter, then this is treated by adding a declaration importing `META-LEVEL` with the same parameter (that is, we let Full Maude do the job of assigning the meta-representation of the module to the corresponding constant), by importing the module `TIMED-META-LEVEL` without parameters, and by adding the declaration of the constant denoting the lazy labels in the module given as parameter and defining its value by adding an equation.

  In addition, the table of labels of lazy rules in the persistent state is updated with the current module and its lazy labels.

  The resulting term is a term of sort `PreModule` and is passed to the Full Maude part of Real-Time Maude. Full Maude then does the real job of first extracting the signature of the module, then parsing all bubbles in the extracted signature to arrive at a complete meta-representation of the module being defined, and, finally, adding the module to the module database and flattening it to get an "executable" module when needed.

- If the input is an untimed module, the treatment of a possible `TIMED-META-LEVEL` importation is handled as for timed modules, and the term is passed to Full Maude without further modifications.
- An untimed command is treated by the Full Maude part of the specification.
- Timed commands are interpreted by calling the appropriate strategy functions defined in the Real-Time Maude part of the tool.

In the next section we give an overview of the library of strategies for analyzing real-time specifications available in the module `TIMED-META-LEVEL`. Besides providing the precise semantics of the search commands in Real-Time Maude, these strategy functions can also be used to define application-specific strategies.

4.2 Real-Time Maude's Strategy Library

This section presents the strategy library needed to execute the rewrite and the search commands of Real-Time Maude. Although Real-Time Maude comes equipped with a powerful search strategy language, we have seen in Section 3.3 that we may want to extend this strategy language for application-specific purposes, and we will see in Chapter 5 that we want to optimize some strategies for certain kinds of rewrite theories. We have therefore also made Real-Time Maude's strategy library available at the user level in the module `TIMED-META-LEVEL`, which is added to Full Maude's database as a built-in module.

The strategies in Real-Time Maude are implemented by functions taking as parameters the representation $\overline{M}$ of a Maude module M and the representation $\overline{t}$ of a term t in M, and some some other parameters, and use the function `meta-apply` as the basic building block to simulate rewrite steps in M. There is therefore no need to define a data type for the representation of timed modules. Instead, a timed module is given by its flattened untimed version, which includes the module `TIMED-PRELUDE`, together with its set of lazy rules (represented as a set of quoted identifiers).

The strategies of course assume that the module given to them as a parameter is indeed the result of transforming a timed module. For instance, we can assume that the global state is of the form `{` t `}` (since `{_}` is the operator of sort `System` in the automatically imported module `TIMED-PRELUDE`, and the definition of real-time rewrite theories (Def. 2) prohibits other operators of sort `System`), and that the global state equipped with a timestamp denoting the total time elapse in the path leading to the state is a term of the form `{` t `} in time` r. Furthermore, the strategies may assume the other declarations in the module `TIMED-PRELUDE`, such as the operators `zero`, `_plus_`, etc. These operators of sort `Time` should be further defined by the user, for example, by letting `zero` be interpreted by `0`. The strategies may also assume that the time domain satisfies the theory *TIME* in Section 2.1.1. Strategies comparing time values with time limits, and finding maximal time increases, will assume that the time domain is linear; that is, that it satisfies the theory *LTIME* in Section 2.1.1, with the interpretations of the sorts and operators in *TIME* given by the standard interpretation described in Section 3.1. Nothing else is assumed by the time domain, allowing the user the flexibility to define his own time domain.

Further assumptions about the modules that can be treated by the strategies were described in Chapter 3. For example, the tick rule should have the required form that allows the strategies to extract the possible time values by which time can increase. To decide enabledness of eager rules (and, hence, whether an enabled *lazy* rule can be applied), an eager rule should be either a tick rule of the assumed form, or should not have new variables in its righthand side or its condition. The patterns in the search expressions are, as already mentioned, assumed to be ground irreducible.

Why should we have specific strategies for real-time systems? We have indicated some reasons for having strategies specific to real-time theories, such as the presence of eager and lazy rules, nondeterministic tick rules, the need to relate an execution to the length of the simulated time (the usefulness of which is clearly illustrated by the scheduling example in Section 3.3 and the case study in Chapter 5).

Similarly, model checking and search in timed systems involve more time-specific features than just naïvely increasing time maximally in nondeterministic tick rules. To illustrate some subtle aspects in model checking of real-time specifications w.r.t. timed temporal logic properties, we look at the problem of model checking, say, the liveness property that in each execution from t_0, a state satisfying a matching property P will be found in time less than or equal to 20. If no P-states are found in a path until time 19, and a maximal time increase strategy would lead us to a state with total time elapse 21, then taking this path would give a path not satisfying the property. However, time could have elapsed to a state with total time elapse 20 with the property P holding in that state. Should the liveness property be considered invalidated by this path? If we view the system as implicitly visiting all moments of time in the presence of nondeterministic tick rules, then we may consider the property valid, as the desired state was passed at time 20 on the way from time 19 to time 21. This viewpoint, trying to adjust the the time increase to stop time advance at the deadline of a temporal formula, is used by the model checking strategies of Real-Time Maude, when possible, since

- in some cases may seem to be a more natural interpretation of correctness of such a property, and
- the other setting, where the strategy does not bother to try to stop at the time deadline of a reachability property, can be simulated by searching for, in the example case, the pattern

  P `in time` x_r `where` x_r `le 20`

  instead, and without deadline 20[1].

Real-Time Maude's strategy library consists of about 2500 lines of Maude code, and will not be given in its entirety here. Instead we present here a high-level summary of the library to indicate

- what strategies are available, and where these, and similar, strategies can be found;
- how the strategies are defined, so that the user can more easily write new strategies of his/her; and
- to give the actual semantics of nontrivial strategies, such as the one in the example above.

The strategy library consists of the following parts:

- Useful data types and operations on untimed systems. For example, we define data types for sets of terms, and various operations on modules, such as deciding whether a term has a certain sort, whether a new variable is introduced in the righthand side or condition of a rule, and finding all terms reachable in one step from a given term.
- Data types and operations for finding and representing all time values by which time may advance in an application of a tick rule. This includes a data type representing sets of time intervals, an operation for deciding whether a tick rule satisfies the syntactic conditions given in Section 3.2.1, and, finally, a function which finds the values by which time can advance in a state using a given tick rule. We briefly discuss the soundness and completeness of the axioms finding this set of possible time advances.
- Operations for executing real-time specifications, corresponding to timed extensions `meta-apply` and `meta-rewrite`, and the data type defining the rewrite modes.
- The data type for patterns, used to define temporal logic properties, functions deciding whether a given state matches a pattern, functions finding the set of terms that match (or do not match) a pattern, and functions finding the set of terms whose associated timestamp is greater/smaller than a given time value.
- Functions used to model check real-time specifications w.r.t. a given initial term. That is, functions which find—according to the given mode, and within given time and depth limits—the set of terms representing rewrite paths satisfying, or not satisfying, temporal properties. Each model checking search strategy is defined in two different ways, using breadth-first search and depth-first search techniques. The search methods are extended to allow also a *set* of initial states.

[1]The searches are not the same, since the latter search will not stop necessary after time 20, but could go on searching forever for a state satisfying the property.

4.2.1 Some Variables Used in the Strategy Library

The following variables will be used frequently in the strategy library, and will not be declared elsewhere in this exposition:

```
vars N N' N'' N''' : MachineInt .
vars NZ NZ' NZ'' NZ''' : NzMachineInt .
vars Q Q' Q'' Q''' Q'''' Q''''' F F' : Qid .
vars QL QL' QL'' QL''' QL'''' QL''''' QL'''''' : QidList .
vars QS QS' QS'' QS''' QS'''' : QidSet .
vars T T' T'' T''' T'''' : Term .
vars TL TL' TL'' TL''' TL'''' : TermList .
vars M M' MOD : Module .        vars SB SUBST : Substitution .
var IL : ImportList .           var SD : SortDecl .
var SSDS : SubsortDeclSet .     var ODS : OpDeclSet .
var VDS : VarDeclSet .          var MAS : MembAxSet .
var ES : EquationSet .          var RS : RuleSet .
vars RULE RULE' : Rule .        var FMOD : FModule .
vars GT GT' GT'' : GroundTerm . vars DT DT' : DefTerm .
var DGT : DefGroundTerm .       var TRM : TimedRewriteMode .
vars TS TS' : TermSet .
```

4.2.2 Some Standard Data Types and Operations

The module `STANDARD-METAFUNCTIONS` extends the module `META-LEVEL` with data types for ground terms and sets of terms, and by various useful functions on terms and modules used later.

Sets[2] of terms are defined as in Full Maude:

```
sort TermSet .
subsort Term < TermSet .
op emptyTermSet : -> TermSet .
op termSet : TermSet TermSet -> TermSet [assoc comm id: emptyTermSet] .
```

We will sometimes need to distinguish ground terms from terms containing variables:

```
sorts NonGroundTerm GroundTerm NonGroundTermList GroundTermList .
subsorts Qid < NonGroundTerm < NonGroundTermList Term .
subsort GroundTerm < Term GroundTermList .
subsorts NonGroundTermList GroundTermList < TermList .
op _[_] : Qid GroundTermList -> GroundTerm .
op _[_] : Qid NonGroundTermList -> NonGroundTerm .
op {_}_ : Qid Qid -> GroundTerm .
op _,_ : GroundTermList GroundTermList -> GroundTermList
                                        [assoc gather (e E) prec 120] .
op _,_ : NonGroundTermList TermList -> NonGroundTermList
                                        [assoc gather (e E) prec 120] .
op _,_ : TermList NonGroundTermList -> NonGroundTermList
                                        [assoc gather (e E) prec 120] .
```

[2]In the absence of an idempotency axiom, the following data type indeed defines *multisets* of terms. These declarations are the same as in the Full Maude module `O-O-TO-SYSTEM-TRANSF`

We add a "default" term value `noTerm`:

```
sorts DefTerm DefGroundTerm .
subsorts Term DefGroundTerm < DefTerm .
subsort GroundTerm < DefGroundTerm .
op noTerm : -> DefGroundTerm .
```

The following are some useful functions for extracting the rules, the rule labels, and the variables from a module, and the lefthand and righthand side of a rule:

```
ops lhs rhs : Rule -> Term .
op rulename : Rule -> Qid .
op variables : Module -> VarDeclSet .
op rules : Module -> RuleSet .
op allRuleLabels : Module -> QidSet .
op allRuleLabels : RuleSet -> QidSet .
op getRule : Module Qid -> Rule .
op getRule : RuleSet Qid -> Rule .
op var : TermList -> QidSet .
op var : Module -> QidSet .
op var : VarDeclSet -> QidSet .
```

The following functions extract the term and substitution from a `ResultPair`:

```
op extrTerm : ResultPair -> Term .
op extrSubst : ResultPair -> Substitution .
```

Some operations on sets and lists of quoted identifiers:

```
op qidSetMinus : QidSet QidSet -> QidSet .
op _subsetof_ : QidSet QidSet -> Bool .
op _in_ : Qid QidSet -> Bool .
op _union_ : QidSet QidSet -> QidSet [assoc comm] .
op qidSetIntersection : QidSet QidSet -> QidSet .
op removeDuplicates : QidSet -> QidSet .
op _inQidList_ : Qid QidList -> Bool .
op qidSetToQidList : QidSet -> QidList .
```

Apply a substitution on a rule, on a term, or on another substitution:

```
op use_on_ : Substitution Rule -> Rule .
op use_on_ : Substitution TermList -> TermList .
op use_on_ : Substitution Substitution -> Substitution .
```

Find whether a term has a given sort:

```
op hasSort : Module Term Qid -> Bool .
```

The following function returns the given module with only one rule, namely, the rule given as the second parameter. It is mostly used for checking whether a term matches a pattern. Removing the other rules has the benefit of avoiding the possibility of having more than one rule with the introduced label:

```
op singleRule : Module Rule -> Module .
```

The following functions return the cardinality of a multiset of terms:

```
op size : TermSet -> MachineInt
```

If the module is added as an additional parameter, then the function `size` can check whether two terms are semantically equivalent, and return the size of the *set* of terms:

```
op size : Module TermSet -> MachineInt .
```

To give a flavor for the equational definitions of the above functions, we show the definition of some of them. First, the function returning the lefthand side of a rule:

```
eq lhs(rl [Q] : T => T' .) = T .
eq lhs(crl [Q] : T => T' if T'' = T''' .) = T .
```

The set of rules can be extracted from a module as follows:

```
eq rules(mod Q is IL SD SSDS ODS VDS MAS ES RS endm) = RS .
eq rules(FMOD) = none .
```

The following function assumes that the module contains at least one rule with the given label, and returns one such rule:

```
eq getRule(M, Q) = getRule(rules(M), Q) .
eq getRule((rl [ Q ] : T => T' .) RS, Q) = (rl [ Q ] : T => T' .) .
eq getRule((crl [ Q ] : T => T' if T'' = T''' .) RS, Q) =
      (crl [ Q ] : T => T' if T'' = T''' .) .
```

It is somewhat problematic to check whether a term has a given sort, since the descent function `leastSort` in `META-LEVEL` does not take membership axioms into considerations, and since the test $t : s$ only works if the sort of t is in the same connected component as s:

```
eq hasSort(M, T, Q) = if sameComponent(M, leastSort(M, T), Q)
                      then meta-reduce(M, _:_(T, Q)) == {'true}'Bool
                      else false fi .
```

The module `STRATEGIES` imports the previous module `STANDARD-METAFUNCTIONS` and uses the descent function `meta-apply` to find, respectively, the result of applying a rule somewhere in a term, to find all terms reachable in one step by applying a rule at the top of the term, and, finally, finding all terms reachable in one rewrite step from a given term using a *set* of rules:

```
op applySomewhere : Module Term Qid -> Term .
op applySomewhereSubst : Module Term Qid Substitution -> Term .
op allOneStepRewritesAtTop : Module Term Qid -> TermSet .
op allOneStepRewritesAtTop : Module Term Qid MachineInt -> TermSet .
op allOneStepRewrites : Module Term QidSet -> TermSet .
```

To find all terms reachable in one step by rewriting at the top, we apply `meta-apply` with all applicable matches:

```
eq allOneStepRewritesAtTop(M, T, Q) = allOneStepRewritesAtTop(M, T, Q, 0) .
eq allOneStepRewritesAtTop(M, T, Q, N) =
      if extrTerm(meta-apply(M, T, Q, none, N)) == error*
      then emptyTermSet
      else termSet(extrTerm(meta-apply(M, T, Q, none, N)),
                   allOneStepRewritesAtTop(M, T, Q, N + 1)) fi .
```

4.2.3 Finding the Possible Time Increase Values of Tick Rule Applications

The following module `LINEARTIME-SEQUENCES` defines sequences of time intervals, which will be used to represent possible time values in a linear time domain. The operations in this, and in all the following modules, assume that the modules under consideration are timed specifications. In addition, `LINEARTIME-SEQUENCES` assumes that the modules treated satisfy the theory *LTIME*.

A time interval, e.g., $[r, r')$, is represented by a term `interval(lE(closed,` $\overline{r}$`), rE(open,` $\overline{r'}$`))`, and denotes all time values r'' where r `le` r'' and r'' `lt` r'. If the interval is unbounded, then the righ endpoint has the value `INF` instead[3]:

```
sorts BracketType LeftEndPoint RightEndPoint EndPoint Interval NEInterval .
subsort NEInterval < Interval .
subsorts LeftEndPoint RightEndPoint < EndPoint .
ops open closed : -> BracketType .   *** left/right-open or left/right-closed?
op lE : BracketType Term -> LeftEndPoint .
op rE : BracketType Term -> RightEndPoint .
op INF : -> RightEndPoint .
op interval : LeftEndPoint RightEndPoint -> NEInterval . *** interval constr.
op noInterval : -> Interval .
```

Intervals $I_1, \ldots, I_n$ can be combined by a "union" operator `+` to form an interval sequence I_1`+`$\cdots$`+`I_n. The operations in this module assume, and maintain, a strictly increasing ordering of the intervals, such that, for all $i \in 1, \ldots, n-1$, the upper bound of I_i is less than or equal to the lower bound of I_{i+1}, and, if the bounds are equal, then I_i is right-open and I_{i+1} is left-open.

```
sorts NEIntervalSequence IntervalSequence .
subsort NEInterval < NEIntervalSequence .
subsort NEIntervalSequence < IntervalSequence .
subsort Interval < IntervalSequence .
op _+_ : IntervalSequence IntervalSequence -> IntervalSequence
```

[3]This constant should not be confused with the value `INF` of sort `TimeInf` in the module `TIME-INF`.

```
                                             [assoc id: noInterval] .
op _+_ : NEIntervalSequence NEIntervalSequence -> NEIntervalSequence
                                             [assoc id: noInterval] .
```

The following are the important operations on interval sequences:

```
op merge : Module IntervalSequence IntervalSequence -> IntervalSequence .
op intersect : Module IntervalSequence IntervalSequence -> IntervalSequence .
op diff : Module IntervalSequence IntervalSequence -> IntervalSequence .
```

The following module OK-TICK-RULE checks whether a tick rule satisfies the requirements given in Section 3.2.1 on the form of tick rules w.r.t. introduced variables:

```
fmod OK-TICK-RULE is
  including STANDARD-METAFUNCTIONS .

  op isOKTickRule : Module Qid Substitution -> Bool .
  op isOKTickRule : Module Qid -> Bool .
  op isOKTickRule : Module Rule Substitution -> Bool .
  op isOKTickRule : Rule Substitution -> Bool .
  op isOKTickRule : Rule -> Bool .

  eq isOKTickRule(M, Q, SUBST) = isOKTickRule(use SUBST on getRule(M, Q)) .
  eq isOKTickRule(M, Q) = isOKTickRule(getRule(M, Q)) .
  eq isOKTickRule(M, RULE, SUBST) = isOKTickRule(RULE, SUBST) .
  eq isOKTickRule(RULE, SUBST) = isOKTickRule(use SUBST on RULE) .
```

An unconditional tick rule satisfies the requirements if either no new variable is introduced in the righthand side of the rule, or the introduced variable is the duration term of the rule:

```
  eq isOKTickRule(rl [Q] : '`{_`}[T] => '_in`time_[T', T''] .) =
        ((var(T') union var(T'')) subsetof var(T)) or
        (T'' : Qid and (var(T') subsetof (var(T'') union var(T)))) .
```

A conditional rule may introduce one new variable in its righthand side and condition, in which case the duration term must be this variable, and, furthermore, the condition must have the desired form:

```
  eq isOKTickRule(crl [Q] : '`{_`}[T] => '_in`time_[T', T'']
                            if T''' = {'true}'Bool .) =
        ((var(T') union var(T'') union var(T''')) subsetof var(T))
        or
        (T'' : Qid and ((var(T') union var(T''')) subsetof
                        (var(T'') union var(T)))
         and isOKCondition(T''', T'')) .
```

The new variable (Q in the equation below) in the tick rule should not be embedded by functions other than _and_, _or_, not_, _==_, _lt_, _le_, _gt_, _ge_, _plus_, and _monus_:

```
eq isOKCondition(F[TL], Q) =
      if (not Q in var(F[TL])) then true
      else
         (if ((F  == '_and_) or (F == '_or_) or (F == 'not_))
          then  isOKCondition(TL, Q)
          else (if ((F == '_lt_) or (F == '_le_) or (F == '_ge_)
                    or (F == '_gt_))
                then isOKTimeVal(TL, Q)
                else (if F == '_==_ then
                         (isOKCondition(TL, Q) or isOKTimeVal(TL, Q))
                      else false
                      fi)
                fi)
          fi)
       fi .
eq isOKCondition({Q}Q', Q'') = true .
eq isOKCondition(Q, Q') = Q =/= Q' .
eq isOKCondition((T , TL), Q) =
      isOKCondition(T, Q) and isOKCondition(TL, Q) .
```

The new variable should occur at most once in terms of sort Time:

```
op isOKTimeVal : TermList Qid -> Bool .
eq isOKTimeVal(F[TL], Q) =
       if (not Q in var(F[TL])) then true
       else (if (F == '_monus_ or F == '_plus_) then isOKTimeVal(TL, Q)
             else false
             fi)
       fi .
eq isOKTimeVal(Q, Q') = true .
eq isOKTimeVal({Q}Q', Q'') = true .
eq isOKTimeVal((T , TL), Q) = if (Q in var(T)) then not (Q in var(TL))
                              else isOKTimeVal(TL, Q) fi .
```

The following module FIND-SOLUTIONS defines the function

```
op findSol : Module Term -> IntervalSequence .
```

which takes a boolean condition, satisfying the requirements defined in the previous command, with all variables occurring in the lefthand side of the corresponding rule already instantiated by the matching ground terms, and returns all possible time values of the single variable which make the (instantiated) condition true. We will sketch a soundness and correctness proof of parts of the specification, assuming correctness of the operations in LINEARTIME-SEQUENCES.

The following equations are used to find the time values for boolean subterms of the condition. (Strictly speaking, the boolean operators could be overloaded, in which case it may be prudent to add a condition asserting that the terms involved are of sort Bool.) Since _and_ and _or_ are associative, they may take more than two arguments:

```
vars GT GT' : GroundTerm .          var GTL : GroundTermList .
var NGT : NonGroundTerm .           var NGTL : NonGroundTermList .

ceq findSol(M, '_==_[T, T']) =
       merge(M, intersect(M, findSol(M, T), findSol(M, T')),  *** Both true
                intersect(M, findSol(M, 'not_[T]), findSol(M, 'not_[T'])))
if hasSort(M, T, 'Bool) .

eq findSol(M, '_and_[T, T']) =
      intersect(M, findSol(M, T), findSol(M, T')) .

eq findSol(M, '_and_[T, T', TL]) =
      intersect(M, findSol(M, T), findSol(M, '_and_[T', TL])) .

eq findSol(M, '_or_[T, T']) =
      merge(M, findSol(M, T), findSol(M, T')) .

eq findSol(M, '_or_[T, T', TL]) =
      merge(M, findSol(M, T), findSol(M, '_or_[T', TL])) .

eq findSol(M, 'not_[NGT]) =
      diff(M, interval(lE(closed, {'zero}'Time), INF), findSol(M, NGT)) .

*** For boolean ground terms:
ceq findSol(M, GT) =
      (if meta-reduce(M, GT) == {'true}'Bool
       then interval(lE(closed, {'zero}'Time), INF)
       else noInterval
       fi)
if hasSort(M, GT, 'Bool) .
```

The single uninstantiated variable has the sort of the time domain, and the values making conditions of the form x_r `==` t and x_r `le` t true are given as follows:

```
eq findSol(M, '_==_[Q, GT]) = interval(lE(closed, GT), rE(closed, GT)) .

eq findSol(M, '_le_[Q, GT]) =
     interval(lE(closed, {'zero}'Time), rE(closed, GT)) .
```

The following equation puts the term with the variable on the left side of `==`:

```
eq findSol(M, '_==_[GT, NGT]) = findSol(M, '_==_[NGT, GT]) .
```

We treat more closely the case where the condition is of the form $t(x_r)$ `plus` r `==` r' for a term t with a variable x_r, and ground terms r and r':

```
eq findSol(M, '_==_['_plus_[NGT, GTL], GT]) =
      if meta-reduce(M, '_le_['_plus_[{'zero}'Time, GTL], GT])  == {'true}'Bool
      then findSol(M, '_==_[NGT, meta-reduce(M, '_monus_[GT,
                                       '_plus_[{'zero}'Time, GTL]])])
      else noInterval   *** r' < r
      fi .
```

We indicate here how the axioms of the theory *LTIME* can be used to prove soundness and completeness of the function `findSol` to find all time values which make the condition of a tick true.

The equation says that if r `le` r', then the values of x_r which make

$$t(x_r)\ \texttt{plus}\ r\ \texttt{==}\ r'$$

hold[4], are the same values of x_r for which

$$t(x_r)\ \texttt{le}\ r'\ \texttt{monus}\ r$$

hold. (The addition by `zero` in the axiom is just a technical detail needed because `_plus_` may have more than two parameters.) The soundness proof of this part goes as follows: Let r'' be a solution of

$$t(x_r)\ \texttt{le}\ r'\ \texttt{monus}\ r.$$

By the "congruence" axiom of $\leq$ w.r.t. $+$ in *TIME*, we have

$$t(r'')\ \texttt{plus}\ r\ \texttt{le}\ (r'\ \texttt{monus}\ r)\ \texttt{plus}\ r,$$

and since r `le` r', the next to last axiom in *TIME* gives

$$(r'\ \texttt{monus}\ r)\ \texttt{plus}\ r = r',$$

which gives the desired

$$t(r'')\ \texttt{plus}\ r\ \texttt{==}\ r'.$$

For completeness, assume that r'' is a solution of

$$t(x_r)\ \texttt{plus}\ r\ \texttt{==}\ r',$$

that is,

$$t(r'')\ \texttt{plus}\ r\ \texttt{==}\ r'$$

holds. Furthermore, assume contrapositively that

$$t(r'')\ \texttt{le}\ r'\ \texttt{monus}\ r$$

does *not* hold. Then, by linearity,

$$r'\ \texttt{monus}\ r\ \texttt{le}\ t(r'')$$

holds. The "congruence" axiom of $\leq$ w.r.t. $+$ gives

$$(r'\ \texttt{monus}\ r)\ \texttt{plus}\ r\ \texttt{le}\ t(r'')\ \texttt{plus}\ r,$$

which reduces to

$$r'\ \texttt{le}\ t(r'')\ \texttt{plus}\ r$$

[4]We sometimes write t for $t = \texttt{true}$.

since r `le` r'. We then have

$$t(r'') \texttt{ plus } r \texttt{ le } r'$$

and

$$r' \texttt{ le } t(r'') \texttt{ plus } r,$$

which, by anti-symmetry of `le` implies

$$r' = t(r'') \texttt{ plus } r.$$

Congruence of equational logic gives

$$r' \texttt{ monus } r = (t(r'') \texttt{ plus } r) \texttt{ monus } r,$$

which by the derived property $(x_r + y_r) \dot{-} y_r = x_r$ in *TIME* gives

$$r' \texttt{ monus } r = t(r''),$$

which by the definition of $\leq$ in *TIME* gives

$$t(r'') \texttt{ le } r' \texttt{ monus } r,$$

contradicting the undesired assumption. The unproved step

$$(x_r + y_r) \dot{-} y_r = x_r$$

in *TIME* follows the axiom

$$(x_r \dot{-} y_r) + y_r \textbf{ if } y_r \leq x_r,$$

which, with x_r instantiated by $x_r + y_r$, gives

$$((x_r + y_r) \dot{-} y_r) + y_r = x_r + y_r$$

(since $y_r \leq x_r + y_r$ follows trivially from the axioms for $\leq$), which by the first axiom in *TIME* gives the desired lemma

$$(x_r + y_r) \dot{-} y_r = x_r.$$

(Other assumptions used in the proof, such as anti-symmetry of $\leq$, can be shown easily as well.)

The other part of the equation above says that there is no r'' such that

$$t(r'') \texttt{ plus } r \texttt{ le } r'$$

holds if it is not the case that r `le` r' holds. Soundness is trivial. For completeness, assume that there actually exists a r'' making

$$t(r'') \texttt{ plus } r \texttt{ le } r'$$

true. The side condition that r `le` r' does not hold, together with assumed linearity, gives

$$r' \texttt{ lt } r$$

By the "congruence" axiom of $\leq$ w.r.t. $+$, 0 being an identity element of $+$, and $0 \leq x_r$, we get $x_r \leq x_r + y_r$, which in our case gives

$$r \texttt{ le } t(r'') \texttt{ plus } r.$$

This latter property, together with the assumption

$$t(r'') \texttt{ plus } r \texttt{ le } r'$$

give, by transitivity of $\leq$,

$$r \texttt{ le } r',$$

contradicting the assumption that r `le` r' does *not* hold.

We have elsewhere proved soundness and completeness in more detail for all the axioms for finding the time values. The following axioms complete the specification `FIND-SOLUTIONS`, where the next axiom pushes the term with a variable one step to the left in the list of parameters to `_plus_`:

```
ceq findSol(M, F['_plus_[GT, NGTL], GT']) =
       findSol(M, F['_plus_[NGTL, GT], GT'])
    if (F == '_==_) or (F == '_le_) .

*** r monus t(x) = r':
eq findSol(M, '_==_['_monus_[GT, NGT], GT']) =
      if meta-reduce(M, '_==_[GT', {'zero}'Time]) == {'true}'Bool
      then findSol(M, '_le_[GT, NGT])
      else (if meta-reduce(M, '_le_[GT', GT]) == {'true}'Bool
            then findSol(M, '_==_[NGT, '_monus_[GT, GT']])
            else noInterval
            fi)
      fi .

*** t(x) monus r <= r':
eq  findSol(M, '_==_['_monus_[NGT, GT], GT']) =
      if meta-reduce(M, '_==_[GT', {'zero}'Time]) == {'true}'Bool
      then findSol(M, '_le_[NGT, GT])
      else findSol(M, '_==_[NGT, '_plus_[GT, GT']])
      fi .

*** t(x) monus r <= r' and r monus t(x) <= r':
eq findSol(M, '_le_['_monus_[NGT, GT], GT']) =
      findSol(M, '_le_[NGT, meta-reduce(M, '_plus_[GT, GT'])]) .

eq findSol(M, '_le_['_monus_[GT, NGT], GT']) =
      findSol(M, '_le_[meta-reduce(M, '_monus_[GT, GT']), NGT]) .

*** t < t' :
eq findSol(M, '_lt_[T, T']) =
      findSol(M, '_and_['_le_[T, T'], 'not_['_==_[T, T']]]) .

*** r <= t(x):
eq findSol(M, '_le_[GT, NGT]) =
```

```
      findSol(M, '_or_['not_['_le_[NGT, GT]], '_==_[GT, NGT]]) .

  *** For greater than, etc., we just use their definitions:
  eq findSol(M, '_gt_[T, T']) = findSol(M, '_lt_[T', T]) .

  eq findSol(M, '_ge_[T, T']) = findSol(M, '_le_[T', T]) .
```

The module `FIND-TIME-INCREASE-INTERVALS` uses the above function `findSol` to find the possible time increase values of a tick rule. This is done by the function

```
op findAllTimeIncreases : Module Rule GroundTerm Substitution ->
                          IntervalSequence .
op findAllTimeIncreases : Module Rule GroundTerm -> IntervalSequence .
op findAllTimeIncreases : Module Qid GroundTerm Substitution ->
                          IntervalSequence .
op findAllTimeIncreases : Module Qid GroundTerm -> IntervalSequence .
```

by taking the union of the possible time increase values for each match of the global state w.r.t. the *lefthand side* of the rule[5]:

```
eq findAllTimeIncreases(M, RULE, '_in`time_[GT, GT'], SUBST) =
      findAllTimeIncreases(M, RULE, GT, SUBST) .

eq findAllTimeIncreases(M, RULE, '`{_`}[GT], SUBST) =
      findAllTimeIncreasesHelp(M, use SUBST on RULE, '`{_`}[GT], 0) .

op findAllTimeIncreasesHelp : Module Rule GroundTerm MachineInt
                              -> IntervalSequence .
eq findAllTimeIncreasesHelp(M, RULE, GT, N) =
      if matches(M, GT, lhs(RULE), N)
      then merge(M, findTimeIncreaseIntervals(M, RULE, GT, N),
                 findAllTimeIncreasesHelp(M, RULE, GT, N + 1))
      else noInterval fi .
```

Observe that, given a timestamped term t `in time` r, only the `System` part t is treated. This will be the case in many of the following functions, since many strategies apply in the same way to the state independently of the total time elapse in the path leading to the state. The function `findIncreaseIntervals` finds all time increase values of the nth match w.r.t. the lefthand side of the rule:

```
op findTimeIncreaseIntervals : Module Qid GroundTerm Substitution MachineInt ->
                               IntervalSequence .
op findTimeIncreaseIntervals : Module Rule GroundTerm Substitution
                               MachineInt -> IntervalSequence .
op findTimeIncreaseIntervals : Module Qid GroundTerm MachineInt ->
                               IntervalSequence .
op findTimeIncreaseIntervals : Module Rule GroundTerm MachineInt ->
```

[5]We cannot match directly w.r.t. the whole rule, since `meta-apply` would indicate that no match exists in the presence of a new variable in the condition of the rule.

```
                              IntervalSequence .

eq findTimeIncreaseIntervals(M, RULE, GT, N) =
      if isOKTickRule(M, RULE) and matches(M, GT, lhs(RULE), N)
      then findTickSol(M, use findMatch(M, lhs(RULE), GT, N) on RULE)
      else noInterval fi .
```

The function findTickSol above calls findSol to find the appropriate interval, if the duration is a variable also after the rule is instantiated w.r.t. the Nth match. Otherwise, it computes the possible time increases itself:

```
op findTickSol : Module Rule -> IntervalSequence .

eq findTickSol(M, crl [Q] : GT => '_in`time_[T, Q'] if T' = {'true}'Bool .) =
      findSol(M, T') .

eq findTickSol(M, rl [Q] : GT => '_in`time_[T, GT'] .) =
      interval(lE(closed, meta-reduce(M, GT')),
               rE(closed, meta-reduce(M, GT'))) .

eq findTickSol(M, crl [Q] : GT => '_in`time_[T, GT'] if GT'' = GT''' .) =
      if meta-reduce(M, '_==_[GT'', GT''']) == {'true}'Bool
      then interval(lE(closed, meta-reduce(M, GT')),
                    rE(closed, meta-reduce(M, GT')))
      else noInterval fi .

eq findTickSol(M, rl [Q] : GT => '_in`time_[T, Q'] .) =
      interval(lE(closed, {'zero}'Time), INF) .
```

4.2.4 The Functions timed-meta-apply and timed-meta-rewrite

The module TIMED-STRATEGIES defines the basic timed strategy timed-meta-apply and the "default rewrite strategy" timed-meta-rewrite, which extend to the timed context the descent functions meta-apply and meta-rewrite for untimed systems.

The function timed-meta-apply takes as additional parameter the mode of the rewrite step. The mode dictates how much time is increased in an application of a nondeterministic tick rule, and is one of the modes described in Section 3.2.1, with the additional possibility to stop time advance at a given time, an option particularly useful in model checking applications. There is also a mode for efficiently executing object-oriented specification with tokens '*' when the lefthand side of the tick rule has the form { T c }, for T a variable of sort Tokens, and c a term of sort Configuration, in which case T is instantiated with the set of tokens * in the (outermost level of the) state, before the rule is applied. If the time advanced by the application of a tick rule would be zero, then the application is considered unsuccessful and timed-meta-apply returns the value error*.

The behavior of timed-meta-apply should mimic that of meta-apply. This leads to some complications when the timed rewrite mode asks for maximal time elapse with respect to each match, since for some matches the maximal time elapse value cannot be found, or is 0. Therefore, the nth successful match of timed-meta-apply may be the $(n+m)$th successful match w.r.t. the lefthand side of the tick rule. timed-meta-apply may therefore not be very efficient when there any many matches w.r.t. the lefthand side of the tick rule. The function `timed-meta-apply` behaves as `meta-apply` on instantaneous rules, with the exception that the rule is applied to the `System` part of a term of sort `ClockedSystem`. The function `apply` applies the rule somewhere in a term, and corresponds to the function `apply-somewhere` in the module `STRATEGIES`.

The function `timed-meta-rewrite` behaves as the standard interpreter function `meta-rewrite` with the additional features that nondeterministic tick rules are applied using the given rewrite mode, that the execution can be related to the length of a rewrite path, and that the eagerness and the lazyness of the rewrite rules are taken into consideration.

These functions are declared as follows:

```
op timed-meta-apply : Module GroundTerm Qid Substitution MachineInt
                      TimedRewriteMode -> Term .
op apply : Module GroundTerm Qid TimedRewriteMode -> Term .
op timed-meta-rewrite : Module GroundTerm MachineInt
                        QidSet TimedRewriteMode DefGroundTerm -> Term .
*** usage: timed-meta-rewrite(Module, TermToRewrite, noOfRewrites,
***                           lazyLables, rewriteMode, deadline)
```

The timed rewrite modes described in Section 3.2.1 are represented by the following data type:

```
sorts BasicTimedRewriteMode LimitTimedRewriteMode .
sorts GlobalTimedRewriteMode TimedRewriteMode .
subsort BasicTimedRewriteMode < LimitTimedRewriteMode .
subsorts LimitTimedRewriteMode GlobalTimedRewriteMode < TimedRewriteMode .

ops default maxTime maxTimeAll : GroundTerm -> BasicTimedRewriteMode .
op timeLimit : BasicTimedRewriteMode GroundTerm -> LimitTimedRewriteMode .
op tokenOO : LimitTimedRewriteMode -> TimedRewriteMode .
op global : LimitTimedRewriteMode -> GlobalTimedRewriteMode .
```

The meaning of these operators should be clear. The parameter to `default`, `maxTime`, and `maxTimeAll` is the default time increase value. The `timeLimit` imposes a time limit saying when time advance should stop, if possible.

The function `adjustedMte` takes as parameters an interval sequence representing the possible time increase values, a default time increase value, and, possibly, a deadline, and computes the maximal time increase in the following way:

```
op adjustedMte : IntervalSequence GroundTerm -> GroundTerm .
op adjustedMte : RightEndPoint GroundTerm -> GroundTerm .
op adjustedMte : IntervalSequence GroundTerm GroundTerm -> GroundTerm .
*** The third argument is the time limit.

var NEIS : NEIntervalSequence .
var IS : IntervalSequence .

eq adjustedMte(noInterval, GT) = {'zero}'Time .      *** Error value.
eq adjustedMte(NEIS, GT) = adjustedMte(rightEndPoint(NEIS), GT) .
eq adjustedMte(INF, GT) = GT .
```

```
eq adjustedMte(rE(closed, GT), GT') = GT .
eq adjustedMte(rE(open, GT), GT') = GT' .
eq adjustedMte(IS, GT, GT') =
      'if_then_else_fi['_or_['_==_[adjustedMte(IS, GT), {'zero}'Time],
                              '_and_['_=/=_[GT', {'zero}'Time],
                                     '_le_[GT', adjustedMte(IS, GT)]]],
                       GT',
                       adjustedMte(IS, GT)] .
```

The following axiom ensures that `timed-meta-apply` rewrites the state part of a state with a timestamp. Observe that the time limit, if there is one, must be modified to reflect the time already elapsed in the state:

```
eq timed-meta-apply(M, '_in`time_[GT, GT'], Q, SUBST, N, TRM) =
     if timed-meta-apply(M, GT, Q, SUBST, N, adjustLimit(TRM, GT')) == error*
     then error* else
     meta-reduce(M, '_in`time_[timed-meta-apply(M, GT, Q, SUBST, N,
                                              adjustLimit(TRM, GT')), GT']) fi .
```

We show how to apply a rule in default time increase mode. The default value is `GT'`. If the rule under consideration—as seen from the specification of the rule, only one rule should have the given label—is a tick rule, and its duration term is a new variable, then this variable is instantiated with `GT'` before calling `meta-apply`. If the rule is not a tick rule, or is a deterministic tick rule, then it is executed directly by `meta-apply`. The "filter" `noZeroTimeStamp` is used to return `error*` if the resulting state has a timestamp `zero`:

```
ceq timed-meta-apply(M, GT, Q, SUBST, N, default(GT')) =
     noZeroTimeStamp(M, extrTerm(meta-apply(M, GT, Q,
                          if tick-rule(getRule(M, Q)) and
                             timeIncreaseIsIntroducedVariable(getRule(M, Q))
                          then (SUBST ; (time-increase(getRule(M, Q)) <- GT'))
                          else SUBST fi, N)))
    if not clockedSystemTerm(GT) .

op noZeroTimeStamp : Module Term -> Term .
eq noZeroTimeStamp(M, '_in`time_[T, T']) =
     if meta-reduce(M, '_==_[T', {'zero}'Time]) == {'true}'Bool
     then error* else '_in`time_[T, T'] fi .
eq noZeroTimeStamp(M, '`{_`}[T]) = '`{_`}[T] .
```

If in default time increase mode there is a time limit `GT''`, then `timed-meta-apply` first tries to instantiate the new variable with the smallest of the default time increase value and the time limit value, and if this fails (the value does not satisfy a condition, or the resulting time increase is `zero`), then time is increased by `GT'`. The time domain is assumed to be linear when a time limit is used, since the current time should always be comparable with the time limit:

```
ceq timed-meta-apply(M, GT, Q, SUBST, N, timeLimit(default(GT'), GT'')) =
      (if timed-meta-apply(M, GT, Q, SUBST, N, default('min[GT', GT'']))
          =/= error*
```

```
      then timed-meta-apply(M, GT, Q, SUBST, N, default('min[GT', GT'']))
      else timed-meta-apply(M, GT, Q, SUBST, N, default(GT')) fi)
    if not clockedSystemTerm(GT) .
```

The rewrite mode `maxTime`, where time is increased maximally w.r.t. each match of the lefthand side of the rule, is less trivial, as mentioned above. Increasing time by the maximal time increase w.r.t. *all* matches is easier:

```
ceq timed-meta-apply(M, GT, Q, SUBST, N, maxTimeAll(GT')) =
      noZeroTimeStamp(M, extrTerm(meta-apply(M, GT, Q,
          if (tick-rule(getRule(M, Q)) and
              timeIncreaseIsIntroducedVariable(getRule(M, Q))) then
               (SUBST ; (time-increase(getRule(M, Q)) <-
                   adjustedMte(findAllTimeIncreases(M, Q, GT, SUBST), GT')))
          else SUBST fi, N)))
      if not clockedSystemTerm(GT) .
```

It follows from the above equation that the application of a *deterministic* tick rule may not yield the maximum time increase w.r.t. *all* matches, even in `maxTimeAll` mode.

The `tokenOO` mode is handled by testing whether the tick rule is of the form described above, with one variable of sort `Tokens`, and no tokens in (the outermost level of) the lefthand side of the rule. If that is the case, then the variable of sort `Tokens`, given by the function `tokenVar` is instantiated with the set of tokens in the state, given by the function `tokens`, before `timed-meta-apply` is called:

```
ceq timed-meta-apply(M, GT, Q, SUBST, N, tokenOO(TRM)) =
      (if isTypicalTokenOOTickRule(M, use SUBST on Q)
       then timed-meta-apply(M, GT, Q,
                             use(tokenVar(M, use SUBST on getRule(M, Q)) <-
                                 tokens(M, GT), SUBST),
                             N, TRM)
       else  timed-meta-apply(M, GT, Q, SUBST, N, TRM)
       fi)
    if not clockedSystemTerm(GT) .
```

The function `timed-meta-rewrite` takes, in addition to the given module, the initial state, and the maximum number of rewrite steps to perform, also a time limit denoting the maximal (simulated) duration of the rewrite, the timed rewrite mode, and the set of labels of the lazy rules. It calls a function

```
op timed-meta-rewriteAux : Module GroundTerm MachineInt MachineInt
                           QidList QidList QidList QidList TimedRewriteMode
                           DefGroundTerm -> Term .
```

where in

$$\texttt{timed-meta-rewriteAux}(M, t, n, n', ql, ql', ql'', ql''', mode, timeLimit),$$

M is the module in which the rewrite is to take place, t is the current state, n is the number of rewrite steps performed, n' is the maximal number of rewrite steps to perform, ql is a *list* of quoted identifiers denoting the labels of the eager rules which have been attempted applied in the current round, ql' is the list of labels of the eager rules which have not yet been tried in the current round, ql'' denotes the list of labels of lazy rules tried in the current state, ql''' is the list of labels of the lazy rules not yet tried in the current state, *mode* is the timed rewrite mode, and *timeLimit* is the desired maximal duration of the rewrite path. There is no such upper time limit if *timeLimit* is `noTerm`. The following axiom starts the process of rewriting:

```
eq timed-meta-rewrite(M, GT, N, QS, TRM, DGT) =
    timed-meta-rewriteAux(M, meta-reduce(M, GT), 0, N, nil,
                          qidSetToQidList(qidSetMinus(allRuleLabels(M), QS)),
                          nil, qidSetToQidList(QS),
                          addLimit(TRM, DGT), DGT) .
```

The time limit was added to the rewrite mode to try to stop time elapse at exactly the desired time, and the function `qidSetToQidList` is used to transform *sets* of quoted identifiers into *lists* of such identifiers. If the rules were represented by sets of labels, then the same rule would be tried first in each round. Using lists allows for a certain fairness by trying to apply all rules equally often using a "round-robin" strategy. The function `qidSetToQidList` has a nondeterministic flavor, and should perhaps have been defined by rewrite rules. However, we keep the simpler, equational, definition, as it is no more nondeterministic than, say, the functions `meta-apply` and `meta-rewrite`.

The following equation models the case where all eager and lazy rules have been tried unsuccessfully in the current round:

```
eq timed-meta-rewriteAux(M, GT, N, N', QL, nil, QL', nil, TRM, DGT) = GT .
```

In the next equation, the application of the eager rule `Q` is attempted, but not before it has been checked whether the current state has reached the desired time limit, or whether the maximum number of rewrites have been performed. The function `apply` then tries to apply the rule somewhere in the current state. If the application is successful, that is, if the rewrite can be performed and the resulting state stays within the time limit, then we get ready for the next round by updating the state, and by appending the rules tried in this round to the list of rules not tried in the round:

```
eq timed-meta-rewriteAux(M, GT, N, N', QL, nil, QL', (Q  QL''), TRM, DGT) =
       if reachedTimeLimit(M, GT, DGT) or (N' > 0 and N == N')
       then GT
       else (if ensureTimeLimit(M, apply(M, GT, Q, TRM), TRM) =/= error*
             then timed-meta-rewriteAux(M, apply(M, GT, Q, TRM), N + 1,
                                  N', nil, QL, nil, (QL''  QL'  Q), TRM, DGT)
             else timed-meta-rewriteAux(M, GT, N, N', QL, nil,
                                        (QL'  Q), QL'', TRM, DGT)
             fi)
       fi .

op apply : Module GroundTerm Qid TimedRewriteMode -> Term .
eq apply(M, GT, Q, TRM) =
     if tick-rule(getRule(M, Q))
```

```
    then timed-meta-apply(M, GT, Q, none, 0, TRM)
    else applySomewhere(M, GT, Q) fi .
```

Finally, if all eager rules have been tried in the current round, then a lazy rule (denoted by `Q` in the following equation) is tried:

```
eq timed-meta-rewriteAux(M, GT, N, N', QL, nil, QL', (Q  QL''), TRM, DGT) =
      if reachedTimeLimit(M, GT, DGT) or (N' > 0 and N == N')
      then GT
      else (if ensureTimeLimit(M, apply(M, GT, Q, TRM), TRM) =/= error*
            then timed-meta-rewriteAux(M, apply(M, GT, Q, TRM), N + 1,
                                N', nil, QL, nil, (QL''  QL'  Q), TRM, DGT)
            else timed-meta-rewriteAux(M, GT, N, N', QL, nil,
                                           (QL'  Q), QL'', TRM, DGT) fi) fi .
```

This axiom concludes the specification of `timed-meta-rewrite`. Before concluding the description of the module `TIMED-STRATEGIES`, we also mention the useful function

```
op getTime : Term -> Term .
```

which returns the timestamp if the term has one, and `{'zero}'Time` otherwise.

4.2.5 Patterns and Some Search Preliminaries

The module `PATTERNS` defines patterns and some other useful functions for real-time specifications, such as finding all terms reachable in one step from a state, and finding those terms from a set of terms whose timestamp is smaller/greater than a given time value.

The functions

```
op allOneStepTickRewrites : Module Term Qid TimedRewriteMode -> TermSet .
op allOneStepTimedRewrites : Module Term QidSet
                             TimedRewriteMode -> TermSet .
```

find, respectively, all terms reachable from a given state using a tick rule, and using all rules in the module, respectively, with the `QidSet` parameter denoting the lazy rules.

The function `ruleApplicable` checks whether any of a set of (eager) rules is applicable, and assumes that no instantaneous rule introduces a new variable in its righthand side or its condition.

The following functions return from a set of terms those terms whose timestamps are, respectively, greater than or equal to, strictly greater that, strictly smaller than, and smaller than or equal to, a given time value. The value `noTerm` is interpreted in these cases as an infinity value:

```
ops tooOld strictlyTooOld strictlyYounger notTooOld : Module DefTerm
                                                      TermSet -> TermSet .
eq tooOld(M, noTerm, TS) = emptyTermSet .
eq tooOld(M, DT, emptyTermSet) = emptyTermSet .
eq tooOld(M, T, termSet(T', TS)) =
   if meta-reduce(M, '_le_[T, getTime(T')]) == {'true}'Bool
   then termSet(T', tooOld(M, T, TS)) else tooOld(M, T, TS) fi .
```

A *pattern* is a (meta-representation of a) term, possibly with variables, of sort `System` or `ClockedSystem`, and may have a condition on the variables in a `where`-part. The default term `noTerm` is not matched by any term, and the pattern `anyPattern` is matched by any term. The data type for patterns can be given as follows:

```
sort Pattern .
subsort Term < Pattern .
op _where_ : Term Term -> Pattern .

sort DefPattern .                  *** Pattern with default values "noTerm"
subsort DefTerm < DefPattern .     *** and "anyPattern".
subsort Pattern < DefPattern .

op anyPattern : -> DefPattern .
op negatedPattern  : DefPattern -> DefPattern .
eq negatedPattern(noTerm) = anyPattern .
eq negatedPattern(anyPattern) = noTerm .
```

A term matches a term of sort `System` if its state part matches the term, and matches a term of sort `ClockedSystem` only if, in addition, the associated time elapse of the state (given by its timestamp, or `zero` when the state does not have a timestamp) matches the timestamp part of the pattern. Matching is decided by checking whether the state can be applied to a rule having the pattern as its lefthand side. A `where`-condition corresponds to a condition on such a rule:

```
var DPATTERN : DefPattern .

op matchesPattern : Module DefPattern Term -> Bool .

eq matchesPattern(M, T'', '_in`time_[T, T']) =
      extrTerm(meta-apply(singleRule(M, (rl ['r1] : T'' => T'' .)),
                          T, 'r1, none, 0)) =/= error* or
      extrTerm(meta-apply(singleRule(M, (rl ['r1] : T'' => T'' .)),
                          '_in`time_[T, T'], 'r1, none, 0)) =/= error* .

eq matchesPattern(M, T', '`{_`}[T]) =
      extrTerm(meta-apply(singleRule(M, (rl ['r1] : T' => T' .)),
                          '`{_`}[T], 'r1, none, 0)) =/= error* or
      extrTerm(meta-apply(singleRule(M, (rl ['r1] : T' => T' .)),
                          '_in`time_['`{_`}[T], {'zero}'Time], 'r1, none, 0))
                          =/= error* .

eq matchesPattern(M, T'' where T''', '_in`time_[T, T']) =
      extrTerm(meta-apply(singleRule(M, (crl ['r1] : T'' => T''
                                         if T'''  = {'true}'Bool .)),
                          T, 'r1, none, 0)) =/= error* or
      extrTerm(meta-apply(singleRule(M, (crl ['r1] : T'' => T''
                                         if T'''  = {'true}'Bool .)),
                          '_in`time_[T, T'], 'r1, none, 0)) =/= error* .

eq matchesPattern(M, T' where T'', '`{_`}[T]) =
      extrTerm(meta-apply(singleRule(M, (crl ['r1] : T' => T'
```

```
                              if T'' = {'true}'Bool .)),
                   ''{_'}[T], 'r1, none, 0)) =/= error* or
    extrTerm(meta-apply(singleRule(M, (crl ['r1] : T' => T'
                              if T'' = {'true}'Bool .)),
                   '_in'time_[''{_'}[T], {'zero}'Time], 'r1, none, 0))
                   =/= error* .

eq matchesPattern(M, noTerm, T) = false .
eq matchesPattern(M, anyPattern, T) = true .
eq matchesPattern(M, negatedPattern(PATTERN), T) =
      not matchesPattern(M, PATTERN, T) .
```

The functions `keepMatches` and `removeMatches` gives from a set of terms those that match, resp. do not match, a pattern:

```
op keepMatches : Module TermSet DefPattern -> TermSet .
op removeMatches : Module TermSet DefPattern -> TermSet .

eq keepMatches(M, emptyTermSet, DPATTERN) = emptyTermSet .
eq keepMatches(M, termSet(T, TS), DPATTERN) =
      if matchesPattern(M, DPATTERN, T)
      then termSet(T, keepMatches(M, TS, DPATTERN))
      else keepMatches(M, TS, DPATTERN) fi .
```

4.2.6 Model Checking and Search Functions

The module `DEPTH-FIRST-SEARCH` imports the module `PATTERNS` and defines some search functions which explore all rewrite paths—in a given mode, and up to certain length and duration limits—from a given set of initial states, where the application of nondeterministic tick rules is performed using the given rewrite mode. The search functions use depth-first search techniques and can be used to analyze and model check specifications w.r.t. properties expressed in the real-time temporal logic defined in Section 3.2. The functions return a set of states which represent rewrite paths from the initial states which do, or do not, satisfy the given property. When a property should hold for all paths reachable in time less than or equal to a time r, then this time limit is added to the rewrite mode to ensure that time advance stops at such time, if possible. This adjustment is not performed if the formula applies to paths of duration strictly less than a time r. The search functions have options to restrict the number of resulting states, and have optimized versions when at most one state should be returned.

The function

```
op dfsAllLe : Module GroundTerm QidSet DefTerm TimedRewriteMode
              MachineInt MachineInt -> TermSet .
```

searches for terms reachable from the initial state which cannot be further rewritten within the given time and depth limits. That is,

$$\texttt{dfsAllLe}(M, t, \mathit{lazyLabels}, \mathit{timeLimit}, \mathit{mode}, \mathit{maxDepth}, \mathit{maxNoOfSols})$$

returns at most *maxNoOfSols* terms reachable from t, which cannot be further rewritten to some state with associated timestamp less than or equal to *timeLimit*, or which is reached in *maxDepth* number of rewrite steps if $maxDepth \neq 0$. *lazyLabels* denotes the labels of the lazy rules, and *mode* is the rewrite mode in which to perform the rewrites. There is no limit on the duration, the number of solutions, or the number of rewrite steps in a path if *timeLimit* is `noTerm`, or *maxNoOfSols* or *maxDepth* is 0, respectively. (Note that the result of this function is not necessarily the same as the results not satisfying the property

$$\texttt{matching anyPattern}\ \mathbf{U}_{\leq timeLimit}\ \texttt{matching noPattern}$$

since, if t' rewrites to t'' inside the time limit, and to t'' beyond the time limit, then t' is returned as representing a counterexample to the temporal formula, while t' is not returned by `dfsAllLe`.)

The function `dfsAllLt` only differs from `dfsAllLe` in that it searches for states reached in time strictly less than the time limit. There is no attempt to adjust the rewrite mode to stop time advance at a given time.

It is sometimes useful to find deadlocked states. The function

```
op dfsDeadLockLe : Module GroundTerm QidSet DefTerm TimedRewriteMode
                   MachineInt MachineInt -> TermSet .
```

corresponds to `dfsAllLe` except that only states within the limits, and which cannot be further rewritten at all, are returned. The function

```
op dfsDeadLockLe1 : Module GroundTerm QidSet DefTerm TimedRewriteMode
                   MachineInt -> DefTerm .
```

is a somewhat optimized version of `dfsDeadLockLe` which returns at most one deadlocked state (and returns `noTerm` when no desired deadlocked state exists), and which is called by `dfsDeadLockLe` if the maximal number of solutions desired is one. The function `dfsDeadLockLt` and `dfsDeadLockLt1` are the corresponding functions when the deadlocked state must be reached in time less than the time limit.

The function

```
op dfsULeNeg : Module GroundTerm QidSet DefPattern DefPattern DefTerm
               TimedRewriteMode MachineInt MachineInt -> TermSet .
```

is used to model check properties of the form

$$me_1\ \mathbf{U}_{\leq timeLimit}\ me_2$$

for pattern matching expression me_1 and me_2.

`dfsULeNeg`($M, t, lazyLabels, pattern_1, pattern_2, timeLimit, mode, maxDepth, maxNoOfSols$) returns at most *maxNoOfSols* states representing paths not satisfying the temporal property. That is, either a state not matching $pattern_1$ is found in a path before a state matching $pattern_2$ is found, or there is a path in which a state matching $pattern_2$ does not exist within the depth and time limits. Observe again that if a term t in a path where a state matching $pattern_2$ has not yet been reached rewrites to a state t' beyond the time time limit, then t represents a bad path even though it may have another successors t'' within the time limit. The time limit is adjusted to stop at the time limit, if possible. The function

```
op dfsULeNeg1 : Module GroundTerm QidSet DefPattern DefPattern DefTerm
                TimedRewriteMode MachineInt -> DefTerm .
```

finds at most one state representing a path not satisfying the property, and returns `noTerm` if it cannot find a counterexample. We give the specification of this function in detail. It calls the function

```
op dfsULeNeg1 : Module TermSet QidSet DefPattern DefPattern DefTerm
                TimedRewriteMode MachineInt MachineInt -> DefTerm .
```

which performs the depth-first search. The second parameter (call it *ts*) denotes the set of terms to analyze. It is an invariant that all states in the paths leading from the initial terms to *ts* match the first pattern $pattern_1$, that *ts* is a nonempty set, and that the timestamps of the terms in *ts* are smaller than, or equal to, the time limit. The last parameter denotes the number of rewrite steps in the paths leading to the terms *ts*.

The search function is initialized by letting it analyze the initial state, unless the timestamp of the state is greater than the time limit. The time limit is added to the rewrite mode to ensure that time advance stops at the time limit, if possible:

```
eq dfsULeNeg1(M, GT, QS, DPATTERN, DPATTERN', DT, TRM, N) =
     if strictlyOlderThan(M, GT, DT) then GT else
     dfsULeNeg1(M, GT, QS, DPATTERN, DPATTERN', DT, addLimit(TRM, DT), N, 0) fi .
```

The term `GT` is analyzed in the following equation. If it matches $pattern_2$ (denoted below by `DPATTERN'`), then it ends a path satisfying the formula, and `noTerm` is returned. Otherwise, if `GT` does not match $pattern_1$ (denoted below by `DPATTERN`), then the path ending with `GT` is a counterexample and `GT` is returned. Furthermore, if `GT` does not match $pattern_2$, and it has no successor states, then $pattern_2$ will never be matched in the path, and `GT` is returned. This is also the case if the upper bound on the number of rewrite steps has been reached without encountering a $pattern_2$-state. Finally, if `GT` has an immediate successor state beyond the time limit, then there is a path where a $pattern_2$-state is not reached within the time limit, and `GT` is returned. If neither of these cases apply, the search function is called recursively to analyze the successor states of `GT`. Due to the `strictlyTooOld`-test in the condition, the successor states in the recursive call will satisfy the invariant:

```
eq dfsULeNeg1(M, GT, QS, DPATTERN, DPATTERN', DT, TRM, N, N') =
      if matchesPattern(M, DPATTERN', GT)
      then noTerm
      else
         (if (allOneStepTimedRewrites(M, GT, QS, TRM) == emptyTermSet)
             *** Solutions needed and t is a deadlocked term
             or (N > 0 and N' == N)        *** Enough rewrites in path
             or not(matchesPattern(M, DPATTERN, GT))
             or (strictlyTooOld(M, DT, allOneStepTimedRewrites(M, GT, QS, TRM))
                 =/= emptyTermSet)
          then GT
          else dfsULeNeg1(M, allOneStepTimedRewrites(M, GT, QS, TRM), QS,
```

```
                    DPATTERN, DPATTERN', DT, TRM, N, N' + 1)
          fi)
      fi .
```

Finally, when more than one term is being analyzed, if there is a solution for the first term GT, then that solution is returned, otherwise the result of analyzing the remaining terms is returned:

```
ceq dfsULeNeg1(M, termSet(GT, TS), QS, DPATTERN, DPATTERN', DT, TRM, N, N') =
       (if dfsULeNeg1(M, GT, QS, DPATTERN, DPATTERN', DT, TRM, N, N')
           =/= noTerm
        then dfsULeNeg1(M, GT, QS, DPATTERN, DPATTERN', DT, TRM, N, N')
        else dfsULeNeg1(M, TS, QS, DPATTERN, DPATTERN', DT, TRM, N, N')
        fi)
    if TS =/= emptyTermSet .
```

Instead of returning only terms of paths *not* satisfying the property, the functions `dfsULePos` and `dfsULePos1` find the first term matching $pattern_2$ in paths satisfying

$$\texttt{matching}\ pattern_1\ \mathbf{U}_{\leq timeLimit}\ \texttt{matching}\ pattern_2.$$

Similarly, the functions `dfsULtNeg`, `dfsULtNeg1`, `dfsULtPos`, and `dfsULtPos1` find paths not satisfying, and satisfying, respectively, temporal formulas of the form

$$me_1\ \mathbf{U}_{<timeLimit}\ me_2.$$

The last class of properties are the until/stable properties, describing paths where the first matching expression is satisfied until the second one is satisfied, after which the second matching property holds throughout the path. For an example of the usefulness of such a property, we are interested in finding some values in the protocol in Chapter 5, and once they are found, they should not change. We are only interested in returning states representing paths which do *not* satisfy the property. Although the until/stable operator cannot be expressed by the other operators in our restricted logic, a search function model checking such properties can be defined in terms of the negative and positive until-functions. However, to avoid multiple searches, the function

```
op dfsUStableLeNeg : Module GroundTerm QidSet DefPattern DefPattern DefTerm
                     TimedRewriteMode MachineInt MachineInt -> TermSet .
```

and its variations `dfsUStableLeNeg1`, `dfsUStableLtNeg`, and `dfsUStableLtNeg1` are defined explicitly in the module `DEPTH-FIRST-SEARCH`. The strategy library does not contain any functions to model check invariance properties and simple liveness properties, since these properties can be model checked without significant additional cost by the functions model checking, respectively, until/stable properties and until properties.

The module `DEPTH-FIRST-SEARCH-TERMSETS` extends `DEPTH-FIRST-SEARCH` by allowing the search functions to take a *set* of initial states as parameter.

Breadth-First Search

The module BREADTH-FIRST-SEARCH specifies the above model checking and search function using breadth-first search techniques[6]. In addition to the parameters of the depth-first search functions, the breadth-first functions take an additional parameter, namely, the maximal breadth on each level. If the number of terms at a certain depth exceeds this maximum breadth, then some of the terms are omitted from further analysis. For example, the ninth parameter to the function

```
op bfsULeNeg : Module GroundTerm QidSet DefPattern DefPattern DefTerm
               TimedRewriteMode MachineInt MachineInt MachineInt -> TermSet .
```

denotes the maximal number of terms in each level of the search. Otherwise, the parameters and the semantics of the breadth-first search functions correspond to their depth-first search counterpart. We will in the future compare the efficiency of the two search strategies more carefully. Our initial results seem to indicate that, although breadth-first search methods have advantages since they provide semi-decision procedures for the various model checking problems when the depth bound is not specified, they suffer from having huge sets of states which can negatively impact the performance.

4.3 The Real-Time Maude Extension of Full Maude

This section describes how Full Maude is extended to Real-Time Maude. The description is intended to illustrate what changes are necessary to reuse Full Maude in a real-time context, how well they can be localized, and *how* these changes are made.

Real-Time Maude is an extension of Full Maude that leaves the treatment of Full Maude modules and execution commands unchanged. The current specification of Real-Time Maude extends version 1.0.5 of Full Maude, and is available at URL `www.csl.sri.com/~peter/RTM`.

Full Maude 1.0.5 itself contains an "additional" part, illustrating how it can be extended to handle two new module composition operators. We have removed this part of the Full Maude specification. Otherwise, only minor changes are made to the specification of Full Maude, as described below.

Real-Time Maude adds the specification of the following tasks to the specification of Full Maude:

- The strategy library defined in the previous section must be included in the specification of Real-Time Maude, which uses it to execute Real-Time Maude commands, and must be made available at the Real-Time Maude application-level in the module TIMED-META-LEVEL.
- The syntax of Real-Time Maude modules and commands (which include Full Maude modules and commands) must be defined in a module TIMED-GRAMMAR, which can be used by meta-parse to parse input from the LOOP-MODE input facilities.
- A premodule, that is, an input which has been parsed w.r.t. the module TIMED-GRAMMAR and determined to be an (attempt at a) declaration of a module, must be treated before it is passed onto the Full Maude part of the specification. Timed premodules are transformed

[6]There is no special breadth-first function when at most one solution is requested.

into untimed premodules by adding declarations importing module the TIMED-PRELUDE or the module TIMED-OO-PRELUDE, and by transforming lazy rule declarations to declarations of "ordinary" rules. The set of labels of lazy rules must therefore be updated at this stage, and the data type of tables of labels is defined. Furthermore, all premodules are checked for importation of the TIMED-META-LEVEL module, which is treated as described above.

- Executing timed commands.
- The Full Maude database object storing all user-defined (and some predefined) modules must be extended to also store the table of labels of lazy rules. We define a new class TimedDatabase as a subclass of Full Maude's Database class, to store also the table of labels of lazy rules in the persistent loop state.
- Certain modules such as TIMED-PRELUDE, LTIME, and TIMED-OO-PRELUDE must be defined and added to Real-Time Maude's database.
- Finally, the Full Maude extension of LOOP-MODE to treat input must be changed to instantiate the database with the additional modules listed above, and to parse input in the module defining the grammar of Real-Time Maude input.

This section first outlines the changes made in the Full Maude part of the specification, and then presents the Real-Time Maude additions to the Full Maude specification.

4.3.1 Changes to Full Maude

The specification of Real-Time Maude modifies the specification of version 1.0.5 of Full Maude in the following ways:

- To avoid unnecessary confusion in the specification, we have omitted the part of the Full Maude specification illustrating how Full Maude can be extended with new module composition operations. That is, the module FULL-MAUDE-SIGN and all subsequent modules are omitted from the specification.
- The strategy library defined in the previous section and the Real-Time Maude extension defined in this section are appended to the specification of Full Maude.
- The module PRE-FULL-MAUDE handling input and output is removed.
- The module TIMED-META-LEVEL is fairly large, and we do not want Real-Time Maude to store it in its database, and to treat its importation by flattening it. Instead, we let it be a Maude module whose importation is treated directly by the Maude engine. This should save a lot of space, and some time. To avoid that Real-Time Maude tries to treat its inclusion as the inclusion of a module in the module database, it has to be declared a "built-in" module by adding the module name TIMED-META-LEVEL to the constant builtIns in the module DATABASE, modifying the definition to the following:

```
eq builtIns
  = 'TRUTH-VALUE & 'TRUTH & 'BOOL & 'IDENTICAL & 'MACHINE-INT &
    'QID & 'QID-LIST & 'META-LEVEL & 'LOOP-MODE & 'TIMED-META-LEVEL .
```

Since TIMED-META-LEVEL will not be stored in the Real-Time Maude database, Real-Time Maude does not have access to its meta-representation, and TIMED-META-LEVEL therefore cannot be renamed or otherwise manipulated by Real-Time Maude or user-defined modules, which we believe should not cause any incoveniences.

- The module DATABASE-HANDLING handles the treatment of Full Maude commands and premodules. The rules module and theory treat (Full Maude) premodules. If timed modules have been introduced, then the database object is an instance of the class TimedDatabase, and the premodules should first be treated by Real-Time Maude for possible handling of inclusions of TIMED-META-LEVEL. Therefore, we include already in the module DATABASE-HANDLING the declaration of the sort TimedDatabaseClass, and ensure that the current database is not a timed database. Premodules are dealt with in the module TIMED-DATABASE-HANDLING when timed modules have been introduced. The declarations

```
sort TimedDatabaseClass .
subsort TimedDatabaseClass < DatabaseClass .
```

are added to the module DATABSE-HANDLING to be able to test whether the current database object is a timed database object, and the test not (X@Database : TimedDatabaseClass) itself is added to the rule module which is, therefore, changed to

```
crl [module] :
   < O : X@Database | db : DB, input : (F[T, T']),
      output : nil, default : MN, Atts >
   => < O : X@Database |
         db : processUnit(F[T, T'], DB), input : nilTermList,
         output :
           ('\n 'Introduced 'module: modNameToQid(parseModName(T))),
         default : parseModName(T), Atts >
      if ((F == 'fmod_is_endfm) or (F == 'obj_is_endo) or
         (F == 'obj_is_jbo) or (F == 'mod_is_endm)
            or (F == 'omod_is_endom)) and
         not (X@Database : TimedDatabaseClass) .
```

The rule theory is modified in the same way. (We have chosen not to omit these rules to keep all treatment of untimed specifications as unchanged as possible.)

- In the Full Maude GRAMMAR module defining the syntax of Full Maude input, the operator

```
op '__ : 'SDeclList 'SDeclList -> 'SDeclList [assoc] .
```

defines the syntax of concatenation of declarations in a premodule. When transforming a term of sort PreModule to a "real" module, Full Maude assumes that the term returned by meta-parse when parsing a list of declarations has the form

```
'__[T, '__[T', '__[T'', '__[ ... ]]]],
```

where the top symbol of T, T', ... is different from '__, even though the '__ operator is declared associative. Real-Time Maude's transformation of premodules may add new declarations to a premodule, which may therefore have subterms of the form

```
'__['__[T', T'], T''].
```

To treat also such premodules, the equation

```
eq processUnit3('__[T, T'], PU, U, DB)
  = processUnit3(T', preUnit(parseDecl(T, PU, U)),
      unit(parseDecl(T, PU, U)), DB) .
```

in the module `UNIT-PROCESSING` is replaced by the equations

```
ceq processUnit3('__[F[TL], T'], PU, U, DB)
  = processUnit3(T', preUnit(parseDecl(F[TL], PU, U)),
      unit(parseDecl(F[TL], PU, U)), DB)
if F =/= '__ .

eq processUnit3('__['__[T, T''], T'], PU, U, DB) =
     processUnit3('__[T, '__[T'', T']], PU, U, DB) .
```

4.3.2 Adding the Strategy Language

The Real-Time Maude extension of the Full Maude specification starts by including the strategy library defined in Section 4.2, and by making the library available in `TIMED-META-LEVEL` by the following declaration:

```
fmod TIMED-META-LEVEL is
  protecting DEPTH-FIRST-SEARCH-TERMSETS .
  protecting BREADTH-FIRST-SEARCH .
endfm
```

The two modules included contain the full strategy library already described in Section 4.2.

4.3.3 Defining the Syntax of Real-Time Maude Modules and Commands

We define the syntax of Real-Time Maude modules, theories, and commands. The descent function `meta-parse` takes as first parameter the *meta-representation* of the module in which a list of quoted identifiers is parsed. To define the syntax in more readable "object level" Maude modules, we follow the technique used in the Full Maude specification, where the syntax of Full Maude modules and commands is given by modules at Maude's object level, and where their meta-representations are obtained by importing them in a "meta-module."

The module `VIEWS` (in the specification of Full Maude) defines at Maude's object level the syntax of Full Maude modules, theories, and views. It is extended in the following module to define also the signature of timed modules and timed theories:

```
fmod TIMED-MODULE-SYNTAX is
  including VIEWS .
  sorts LazyRuleDecl TDeclList .           *** Timed DeclList.
  subsorts RuleDecl < LazyRuleDecl < TDeclList .
  subsort SDeclList < TDeclList .

  *** Kazy rules:
  op lazy rl[_]:_=>_. : Token Bubble Bubble -> LazyRuleDecl .
  op lazy crl[_]:_=>_if_. : Token Bubble Bubble Bubble -> LazyRuleDecl .
  op __ : TDeclList TDeclList -> TDeclList [assoc] .

  *** Timed modules and theories:
  op tmod_is_endtm : ModuleName TDeclList -> PreModule .
  op tth_is_endtth : ModuleName TDeclList -> PreModule .

  *** Declarations for object-oriented modules:
  sort TODeclList .
  subsorts ODeclList TDeclList < TODeclList .
  op __ : TODeclList TODeclList -> TODeclList [assoc] .
  op tomod_is_endtom : ModuleName TODeclList -> PreModule .
  op toth_is_endtoth : ModuleName TODeclList -> PreModule .
endfm
```

The module `COMMANDS` defines the syntax of Full Maude commands, and is used in the following definition of the syntax of Real-Time Maude commands. (The module `META-PRE-FULL-MAUDE-SIGN` defines the sort `Bubble` to denote bubbles of arbitrary length, and the sort `Token` to denote bubbles of length one, with the given constructors. To get an intuition of the syntax below, terms of sort `Bubble` are intended to develop into terms in the user-defined modules, and terms of sort `Tokens` are intended to represent a single symbol, such as a number.) The extended BNF-syntax of the Real-Time Maude modules and commands is given in Section 3.2.5:

```
fmod TIMED-COMMANDS is
  including COMMANDS .

  sorts ModeExpr TimeLimitExpr TermSetExpr MatchExpr .

  *** Timed meta-rewrite:
  op trew[_]_mode_dti_in time_. : Token Bubble ModeExpr Bubble
                                TimeLimitExpr -> PreCommand .
  op trew-in[_]_:_mode_dti_in time_. : Token ModExp Bubble ModeExpr Bubble
                         TimeLimitExpr -> PreCommand .

  *** ModeExpr:
  ops d m ma gd gm gma tokend tokenm tokenma : -> ModeExpr .

  *** Time limit expresions:
  op no time limit : -> TimeLimitExpr .
  op le_ : Bubble -> TimeLimitExpr .
  op lt_ : Bubble -> TimeLimitExpr .

  *** Shows lazy rules in current module:
  op show lazy . : -> PreCommand .
```

```
  op show lazy_. : ModExp -> PreCommand .

  *** Exhaustive search strategies.
  op term_ : Bubble -> TermSetExpr .
  op find[_]_maximal terms reachable from_mode_dti_within time_ : Token Token
             TermSetExpr ModeExpr Bubble TimeLimitExpr -> TermSetExpr .
  op find[_]_deadlocks from_mode_dti_within time_ : Token Token TermSetExpr
                            ModeExpr Bubble TimeLimitExpr -> TermSetExpr .
  op find[_]_not satisfying[]_from_mode_dti_within time_ : Token Token
        MatchExpr TermSetExpr ModeExpr Bubble TimeLimitExpr -> TermSetExpr .
  op find[_]_satisfying_from_mode_dti_within time_ : Token Token
        MatchExpr TermSetExpr ModeExpr Bubble TimeLimitExpr -> TermSetExpr .
  op find[_]_not satisfying <>_from_mode_dti_within time_ : Token Token
        MatchExpr TermSetExpr ModeExpr Bubble TimeLimitExpr -> TermSetExpr .
  op find[_]_satisfying_Until_from_mode_dti_within time_ : Token Token
                              MatchExpr MatchExpr TermSetExpr ModeExpr
                              Bubble TimeLimitExpr -> TermSetExpr .
  op find[_]_not satisfying_Until_from_mode_dti_within time_. : Token Token
                              MatchExpr MatchExpr TermSetExpr ModeExpr
                              Bubble TimeLimitExpr -> TermSetExpr .
  op find[_]_not satisfying_UntilStable_from_mode_dti_within time_ :
                             Token Token  MatchExpr MatchExpr TermSetExpr
                             ModeExpr Bubble TimeLimitExpr -> TermSetExpr .
  op keep_from_ : Token TermSetExpr  -> TermSetExpr .
  op remove_from_ : MatchExpr TermSetExpr -> TermSetExpr .

  *** Matches
  op matching pattern_ : Bubble -> MatchExpr .
  op not matching pattern_ : Bubble -> MatchExpr .
  op matching extPattern_where_ : Bubble Bubble -> MatchExpr .
  op not matching extPattern_where_ : Bubble Bubble -> MatchExpr .
  op matching anyPattern : -> MatchExpr .
  op matching noPattern : -> MatchExpr .

  *** Wrapper:
  op strat_. : TermSetExpr -> PreCommand .
endfm
```

The syntax of Real-Time Maude modules and commands is then given in the module

```
fmod REAL-TIME-MAUDE-SIGN is
  including TIMED-MODULE-SYNTAX .
  including TIMED-COMMANDS .
endfm
```

whose meta-representation is equated to the constant `TIMED-GRAMMAR` of sort `Module`:

```
fmod META-REAL-TIME-MAUDE-SIGN is
  including META-LEVEL .
  protecting META-PRE-FULL-MAUDE-SIGN .
  protecting UNIT .
```

```
  op TIMED-GRAMMAR : -> FModule .
  eq TIMED-GRAMMAR = addImportList((including 'REAL-TIME-MAUDE-SIGN .),
                                   GRAMMAR) .
endfm
```

(The UNIT function addImportList adds the inclusion of REAL-TIME-MAUDE-SIGN to the term GRAMMAR of sort Module, which defines the operators of the sorts Token and Bubble.)

4.3.4 Preprocessing Premodules

Untimed premodules are transformed into the meta-representation of Full Maude modules by the function processUnit in the module UNIT-PROCESSING. This function is called directly in the module TIMED-DATABASE-HANDLING presented later, but only after the premodule has been slightly "preprocessed." The function timedPreModuleToPreModule transforms a timed premodule to an untimed premodule by adding the appropriate inclusion declaration, and by transforming lazy rule declaration to ordinary rule declarations (this is done by the function removeLazy). Elements of the data type LazyLabelTable store, for each timed module, its name and the labels of the lazy rules in the module and its submodules and parameters. The function updateLazyLabels takes a timed premodule and the global table of module names with their labels of lazy rules, and returns the table of labels of lazy rules after the labels of the lazy rules of the given premodule has been added to it. The function processTimedMetaLevel takes care of possible importations of TIMED-META-LEVEL.

```
fmod TIMED-UNIT-PROCESSING is
  including STANDARD-METAFUNCTIONS .

  vars F Q Q' : Qid .
  vars T T' T'' T''' T'''' : Term .
  var TL : TermList .
  var LLT : LazyLabelTable .
  var QS : QidSet .

  op timedPreModuleToPreModule : Term -> Term .
```

A timed premodule becomes an untimed premodule by changing the wrapper tmod_is_endtm to mod_is_endm, by importing TIMED-PRELUDE, and by transforming lazy rules to ordinary rules:

```
  eq timedPreModuleToPreModule('tmod_is_endtm[T, T']) =
       'mod_is_endm[T, '__['including_.['token[{''TIMED-PRELUDE}'Qid]],
                          removeLazy(T')]] .
```

Similarly, an object-oriented timed premodule imports the module TIMED-OO-PRELUDE:

```
  eq timedPreModuleToPreModule('tomod_is_endtom[T, T']) =
       'omod_is_endom[T, '__['including_.['token[{''TIMED-OO-PRELUDE}'Qid]],
                            removeLazy(T')]] .
```

Theories are treated in the same way:

```
eq timedPreModuleToPreModule('tth_is_endtth[T, T']) =
      'th_is_endth[T, '__['including_.['token[{''TIMED-PRELUDE}'Qid]],
                         removeLazy(T')]] .
eq timedPreModuleToPreModule('toth_is_endtoth[T, T']) =
      'oth_is_endoth[T, '__['including_.['token[{''TIMED-OO-PRELUDE}'Qid]],
                          removeLazy(T')]] .
```

removeLazy takes the meta-representation of a TODeclList and transforms declarations of lazy rules to declarations of ordinary rules, and leaves the other declarations unchanged:

```
op removeLazy : Term -> Term .
eq removeLazy('__[T, T']) = '__[removeLazy(T), removeLazy(T')] .
eq removeLazy(F[T]) = F[T] .
ceq removeLazy(F[T, T']) = F[T, T']  if (F =/= '__) .
ceq removeLazy(F[T, T', T'']) =
       F[T, T', T''] if (F =/= 'lazy`rl`[_`]:_=>_.) .
ceq removeLazy(F[T, T', T'', T''']) = F[T, T', T'', T''']
       if (F =/= 'lazy`crl`[_`]:_=>_if_.) .
eq removeLazy(F[T, T', T'', T''', T'''']) = F[T, T', T'', T''', T''''] .
eq removeLazy('lazy`rl`[_`]:_=>_.[T, T', T'']) = 'rl`[_`]:_=>_.[T, T', T''] .
eq removeLazy('lazy`crl`[_`]:_=>_if_.[T, T', T'', T''']) =
      'crl`[_`]:_=>_if_.[T, T', T'', T'''] .
```

We define the data type of labels of lazy rules ("lazy labels"). The term lazyLabelPair(M, qs) represents an entry in the table of lazy labels, where the module with base name M (that is, the module name without parameters) has associated the set qs as its lazy labels:

```
sort LazyLabelTable .
op emptyLazyLabelTable : -> LazyLabelTable .
op lazyLabelPair : Qid QidSet -> LazyLabelTable .
op __ : LazyLabelTable LazyLabelTable -> LazyLabelTable
                                    [assoc comm id: emptyLazyLabelTable] .
```

The function lazyLabels returns the set of lazy labels associated with the given module name:

```
op lazyLabels : Qid LazyLabelTable -> QidSet .
eq lazyLabels(Q, lazyLabelPair(Q, QS) LLT) = QS .
ceq lazyLabels(Q, LLT) = none if not Q inLazyLabelTable LLT .

op _inLazyLabelTable_ : Qid LazyLabelTable -> Bool .
eq Q inLazyLabelTable emptyLazyLabelTable = false .
eq Q inLazyLabelTable (lazyLabelPair(Q', QS) LLT) =
      Q == Q' or Q inLazyLabelTable LLT .
```

The function updateLazyLabels updates the table by adding an entry for a given a timed premodule. The variable F in the equation below will only be instantiated with 'tmod_is_endtm or a similar constructor (for object-oriented timed modules, or for timed theories), T is the module name with parameters, and T' denotes the "body" of the premodule:

```
op updateLazyLabels : Term LazyLabelTable -> LazyLabelTable .

eq updateLazyLabels(F[T, T'], LLT) =
     (LLT lazyLabelPair(getName(T),
                        removeDuplicates((getLazyLabelsTitle(T, LLT) ;
                                          getLazyLabels(T', LLT))))) .
```

getName returns the module name without possible parameters:

```
op getName : Term -> Qid .
eq getName('token[{Q}'Qid]) = strip(Q) .
eq getName('_`[_`] ['token[{Q}'Qid], T]) = strip(Q) .
```

The following function getLazyLabelsTitle extracts the lazy labels from the module name. That is, it extracts the lazy labels from the parameter theories. A parameterlist in a premodule is a list of pairs T :: ME separated by commas, where T is the formal parameter, and ME is a module expression, possibly involving renaming or instantiations:

```
op getLazyLabelsTitle : Term LazyLabelTable -> QidSet .
eq getLazyLabelsTitle('token[T], LLT) = none .
eq getLazyLabelsTitle('_`[_`][T, T'], LLT) =  *** Parameters may have lazy rules
      getLazyLabelsParameterList(T', LLT) .

op getLazyLabelsParameterList : Term LazyLabelTable -> QidSet .
eq getLazyLabelsParameterList('_`,_[T, TL], LLT) =
     (getLazyLabelsParameterList(T, LLT) ;
      getLazyLabelsParameterList(TL, LLT)) .
eq getLazyLabelsParameterList((T , TL), LLT) =
     (getLazyLabelsParameterList(T, LLT) ;
      getLazyLabelsParameterList(TL, LLT)) .
eq getLazyLabelsParameterList('_::_[T, T'], LLT) =
      getLazyLabelsParameterList(T', LLT) .
eq getLazyLabelsParameterList('token[{Q}'Qid], LLT) =
      lazyLabels(strip(Q), LLT) .    *** A single theory
*** Renaming:
eq getLazyLabelsParameterList('_*`(_`) [T, T'], LLT) =
      renameLazy(getLazyLabelsParameterList(T, LLT), T') .
eq getLazyLabelsParameterList('_`[_`] [T, T'], LLT) =
      getLazyLabelsParameterList(T, LLT) . *** No renaming of views.
```

The function renameLazy takes a set of quoted identifiers (representing lazy labels) and a renaming map, and applies the label renaming (label l to l') to the lazy labels:

```
op renameLazy : QidSet Term -> QidSet .
op renameLazy : QidSet Qid Qid -> QidSet .
eq renameLazy(QS, '_`,_[T, T']) =
      renameLazy(renameLazy(QS, T), T') .
ceq renameLazy(QS, F[TL]) = QS
    if (F == 'op_to) or (F == 'op_:_to_) or (F == 'op_to_`[_`]) or
       (F == 'op_:_to_`[_`]) or (F == 'sort_to_) or (F == 'class_to_) or
```

```
       (F == 'attr_._to_) or (F == 'msg_to_) or (F == 'msg_:_->_to_) or
       (F == 'msg_:`->_to_) .
  eq renameLazy(QS, 'label_to_['token[{Q}'Qid], 'token[{Q'}'Qid]]) =
       renameLazy(QS, strip(Q), strip(Q')) .
  eq renameLazy((Q ; QS), Q, Q') = qidSetMinus((Q' ; QS), Q) .
  ceq renameLazy(QS, Q, Q') = QS if not (Q in QS) .
```

The function getLazyLabels finds the labels of the lazy rules in the body of a timed premodule, namely the labels of lazy rules in imported modules together with the labels of the lazy rules declared in the premodule itself:

```
  op getLazyLabels : Term LazyLabelTable -> QidSet .
  eq getLazyLabels('__[T, TL], LLT) =
       (getLazyLabels(T, LLT) ; getLazyLabels(TL, LLT)) .
  eq getLazyLabels((T , TL), LLT) =
       (getLazyLabels(T, LLT) ; getLazyLabels(TL, LLT)) .
  eq getLazyLabels('lazy`rl`[_`]:_=>_.['token[{Q}'Qid], T, T'], LLT) =
       strip(Q) .
  eq getLazyLabels('lazy`crl`[_`]:_=>_if_.['token[{Q}'Qid], T, T', T''], LLT)
       = strip(Q) .
  ceq getLazyLabels(F[T], LLT) = getLazyLabelsParameterList(T, LLT)
      if (F == 'including_.) or (F == 'inc_.) or
         (F == 'protecting_.) or (F == 'pr_.) .
  ceq getLazyLabels(F[TL], LLT) = none
      if  not ((F == 'including_.) or (F == 'inc_.) or
               (F == 'protecting_.) or (F == 'pr_.) or
               (F == '__) or (F == 'lazy`rl`[_`]:_=>_.) or
               (F == 'lazy`crl`[_`]:_=>_if_.)) .
```

An importation of TIMED-META-LEVEL[$M_1, \ldots, M_n$] is processed as described in Section 3.1.2. Observe that the premodule declaration of an equation `eq lazyLabels-M = 'a ; 'b .` is written

```
'eq_=_.['bubble[{''lazyLabels-M}'Qid],
        'bubble['__[{'''a}'Qid, {'';}'Qid, {'''b}'Qid]]]
```

The function processTimedMetaLevel handles the importation of TIMED-META-LEVEL:

```
  op processTimedMetaLevel : Term LazyLabelTable -> Term .
  ceq processTimedMetaLevel(Q[T, T'], LLT) =
       Q[T, processTimedMetaLevel(T', LLT)]
     if (Q == 'fmod_is_endfm) or (Q == 'mod_is_endm) or
        (Q == 'omod_is_endom) or (Q == 'tmod_is_endtm) or
        (Q == 'tomod_is_endtom) or (Q == 'fth_is_endfth) or
        (Q == 'th_is_endth) or (Q == 'oth_is_endoth) or
        (Q == 'obj_is_jbo) or (Q == 'obj_is_endo) or
        (Q == 'view_from_to_is_endv) or (Q == 'tth_is_endtth) or
        (Q == 'toth_is_endtoth) .

  eq processTimedMetaLevel('__[T, T'], LLT) =
       '__[processTimedMetaLevel(T, LLT), processTimedMetaLevel(T', LLT)] .
```

The following equation treats an importation of `TIMED-META-APPLY` with a list of parameters `T`. `META-LEVEL` is imported with the same list of parameters, and `processMetaTimedModules` adds the appropriate `lazyLabels-` ... declarations and definitions:

```
ceq processTimedMetaLevel(Q['_`[_`]['token[{''TIMED-META-LEVEL}'Qid], T]],
                          LLT) =
      '__[Q['_`[_`]['token[{''META-LEVEL}'Qid], T]],
         '__['protecting_.['token[{''TIMED-META-LEVEL}'Qid]],
             processMetaTimedModules(T, LLT)]]
    if (Q == 'including_.) or (Q == 'inc_.) or
       (Q == 'protecting_.) or (Q == 'pr_.) .

ceq processTimedMetaLevel(Q['_`[_`]['token[{Q'}'Qid], T]], LLT) =
      Q['_`[_`]['token[{Q'}'Qid], T]]
    if (strip(Q') =/= 'TIMED-META-LEVEL) and ((Q == 'including_.) or
       (Q == 'inc_.) or
       (Q == 'protecting_.) or (Q == 'pr_.)) .

ceq processTimedMetaLevel(Q[Q'[TL]], LLT) = Q[Q'[TL]]
    if (Q' =/= '_`[_`]) and ((Q == 'including_.) or
       (Q == 'inc_.) or
       (Q == 'protecting_.) or (Q == 'pr_.)) .

ceq processTimedMetaLevel(Q['token[T]], LLT) = Q['token[T]]
    if (Q == 'including_.) or
       (Q == 'inc_.) or
       (Q == 'protecting_.) or (Q == 'pr_.) .

ceq processTimedMetaLevel(Q[TL], LLT) = Q[TL]
    if not((Q == 'including_.) or (Q == 'inc_.) or
       (Q == 'protecting_.) or (Q == 'pr_.) or
       (Q == 'fmod_is_endfm) or (Q == 'mod_is_endm) or
       (Q == 'omod_is_endom) or (Q == 'tmod_is_endtm) or
       (Q == 'tomod_is_endtom) or (Q == 'fth_is_endfth) or
       (Q == 'th_is_endth) or (Q == 'othd_is_endoth) or
       (Q == 'obj_is_jbo) or (Q == 'obj_is_endo) or
       (Q == 'view_from_to_is_endv) or (Q == 'tth_is_endtth) or
       (Q == 'toth_is_endtoth) or (Q == '__)) .

op processMetaTimedModules : Term LazyLabelTable -> Term .

eq processMetaTimedModules('_`,_[T, T'], LLT) =
     '__[processMetaTimedModules(T, LLT),
         processMetaTimedModules(T', LLT)] .

eq processMetaTimedModules('viewToken[{Q}'Qid], LLT) =
     '__['op_:`->_.['token[{conc('', conc('lazyLabels-, strip(Q)))}'Qid],
                    'sortToken[{''QidSet}'Qid]],
         'eq_=_.['bubble[{conc('', conc('lazyLabels-, strip(Q)))}'Qid],
        'bubble[qidSetToBubbleBody(lazyLabels(strip(Q), LLT))]]] .

op qidSetToBubbleBody : QidSet -> Term .
eq qidSetToBubbleBody(none) = {''none}'Qid .
```

```
  eq qidSetToBubbleBody(Q) = {conc(''', Q)}'Qid .
  ceq qidSetToBubbleBody(Q ; QS) =
        '__[qidSetToBubbleBodyList(Q ; QS)]
      if QS =/= none .
  op qidSetToBubbleBodyList : QidSet -> TermList .
  ceq qidSetToBubbleBodyList(Q ; QS) =
        ({conc(''', Q)}'Qid ,  {'';}'Qid , qidSetToBubbleBodyList(QS))
  if QS =/= none .
  eq qidSetToBubbleBodyList(Q) =
        {conc(''', Q)}'Qid .
endfm
```

4.3.5 Executing Timed Commands

This section treats the execution of the timed rewrite and the search commands. The module `TIMED-DATABASE-HANDLING` calls the functions `processTimedCommand` and `processStrategyCommand`. The command is given as a "precommand" term resulting from parsing the user input w.r.t. to the grammar `TIMED-GRAMMAR`. The states to be rewritten, the patterns to search for, etc., are all given as "bubbles." Before calling the strategy functions in the strategy library, the "executable" module must be computed and the bubbles must be parsed as terms. We reuse Full Maude's facilities for parsing bubbles and computing the modules. The function `solveBubbles` takes a bubble, the module in which the bubble should be parsed, and some other parameters, and returns the term resulting from parsing the bubble. The function `evalModExp` updates the database after evaluating a module expression. The function `parseModExp` returns a module expression from a module name, and the function `getFlatUnit` produces an "executable" module.

First, we define some functions which are used to process bubbles and other parts of commands:

```
fmod TIMED-COMMAND-PROCESSING is
  pr DATABASE .
  pr DEPTH-FIRST-SEARCH-TERMSETS .
  pr UNIT-DECL-PARSING .
  pr EVALUATION .
  pr UNIT-META-PRETTY-PRINT .
  pr TIMED-UNIT-PROCESSING .     *** Data type of lazy labels

  op getTerm : Term Module Database -> Term .
  op getInt : Term -> MachineInt .
  op getMode : Term Term Module Database -> TimedRewriteMode .
  *** Usage: getMode(mode, dti-term, M, DB).
  op getPattern : Term Module Database -> DefPattern .
  op getTimeLimit : Term Module Database -> DefTerm .

  var MN : ModName .               var DB : Database .
  vars T T' T'' T''' T'''' T''''' T'''''' T''''''' T'''''''' : Term .
  var QS : QidSet .                var M : Module .
  var ME : ModExp .                var VE : ViewExp .
  var MS : MapSet .                vars Q Q' : Qid .
  var LLT : LazyLabelTable .       var TL : TermList .
  vars TS TS' : TermSet .
```

```
eq getTerm(T, M, DB) = meta-reduce(M, solveBubbles(T, M, true, DB)) .

eq getInt('token[T]) = downMachineInt(downMetaMachineInt(T)) .

eq getPattern('matching`pattern_[T], M, DB) = solveBubbles(T, M, true, DB) .
eq getPattern('not`matching`pattern_[T], M, DB) =
     negatedPattern(solveBubbles(T, M, true, DB)) .
eq getPattern('matching`extPattern_where_[T, T'], M, DB) =
     (solveBubbles(T, M, true, DB) where solveBubbles(T', M, true, DB)) .
eq getPattern('not`matching`extPattern_where_[T, T'], M, DB) =
     negatedPattern(solveBubbles(T, M, true, DB) where
                    solveBubbles(T', M, true, DB)) .
eq getPattern({'matching`noPattern}'MatchExpr, M, DB) = noTerm .
eq getPattern({'matching`anyPattern}'MatchExpr, M, DB) = anyPattern .

eq getMode({'d}'ModeExpr, T, M, DB) = default(getTerm(T, M, DB)) .
eq getMode({'m}'ModeExpr, T, M, DB) = maxTime(getTerm(T, M, DB)) .
eq getMode({'ma}'ModeExpr, T, M, DB) = maxTimeAll(getTerm(T, M, DB)) .
eq getMode({'gd}'ModeExpr, T, M, DB) = global(default(getTerm(T, M, DB))) .
[ ... ]

eq getTimeLimit({'no`time`limit}'TimeLimitExpr, M, DB) = noTerm .
eq getTimeLimit('le_[T], M, DB) = getTerm(T, M, DB) .
eq getTimeLimit('lt_[T], M, DB) = getTerm(T, M, DB) .
```

The function

```
op processTimedCommand : Term ModName Database LazyLabelTable -> QidList .
```

executes a timed rewrite by computing the executable module (and its name) and then calling the function

```
op processTimedCommand1 : Term ModName Module Database
                          LazyLabelTable -> QidList .
```

as follows:

```
eq processTimedCommand('trew-in`[_`]_:_mode_dti_in`time_.[T, T', TL],
                       MN, DB, LLT) =
      if unitInDatabase(parseModExp(T'), evalModExp(parseModExp(T'), DB))
      then processTimedCommand1('trew`[_`]_mode_dti_in`time_.[T, TL],
                                parseModExp(T'),
                                getFlatUnit(parseModExp(T'),
                                            evalModExp(parseModExp(T'), DB)),
                                DB, LLT)
      else ('ERROR: 'Module my-meta-pretty-print(parseModExp(T'))
            'not 'in 'database. '\n )
      fi .

eq processTimedCommand('trew`[_`]_mode_dti_in`time_.[TL], MN, DB, LLT) =
     processTimedCommand1('trew`[_`]_mode_dti_in`time_.[TL], MN,
                          getFlatUnit(MN, DB), DB, LLT) .
```

processTimedCommand1 then checks whether the state to rewrite, the default time increase value, and the time limit expression are parsable in the executable module. If either of these is not parsable, then an error message is returned. Otherwise, processTimedCommand1 parses these parts, finds the rewrite mode, the maximal number of rewrite steps, and extracts the set of lazy labels from the table, and executes the function timed-meta-rewrite defined in Section 4.2.4. The output should be a set of quoted identifiers, and the function my-meta-pretty-print transforms the resulting term into such a list. An echo of the command is also written before the resulting term:

```
eq processTimedCommand1('trew`[_`]_mode_dti_in`time_.[T,T',T'',T''',T''''],
                        MN, M, DB, LLT) =
      if getTerm(T', M, DB) =/= error*
      then (if getTerm(T''', M, DB) =/= error*
            then (if simpleTimeLimitOK(T'''', M, DB)
                  then ('Timed 'rewrite
                        my-meta-pretty-print(M, getTerm(T', M, DB))
                        'in my-meta-pretty-print(MN)
                        'with getModeText(T'') 'time 'increase  'with
                        'dti
                        my-meta-pretty-print(M, getTerm(T''', M, DB)) 'and
                        timeLimitText(T'''', M, DB) '. '\n
                        'Result:
                         my-meta-pretty-print(M,
                             timed-meta-rewrite(M,
                                        getTerm(T', M, DB),
                                        getInt(T),
                                        lazyLabels(firstPartOfModName(MN),
                                                   LLT),
                                        getMode(T'', T''', M, DB),
                                        getTimeLimit(T'''', M, DB))) '.
                         '\n)
                    else ('ERROR: 'Unparsable 'time 'limit!)
                    fi)
              else ('ERROR: 'Unparsable 'default 'time 'increase 'term!)
              fi)
         else ('ERROR! 'Unparsable 'term 'to 'rewrite!)
         fi .

op simpleTimeLimitOK : Term Module Database -> Bool .
eq simpleTimeLimitOK({'no`time`limit}'TimeLimitExpr, M, DB) = true .
eq simpleTimeLimitOK('le_[T], M, DB) =
      getTerm(T, M, DB) =/= error* .
eq simpleTimeLimitOK('lt_[T], M, DB) = false .

op firstPartOfModName : ModName -> Qid .
eq firstPartOfModName(nullModName) = 'ERROR .
eq firstPartOfModName(Q) = Q .
eq firstPartOfModName(par Q :: ME) = 'ERROR .
eq firstPartOfModName(_<_>(ME, VE)) = firstPartOfModName(ME) .
eq firstPartOfModName(_*<_>(ME, MS)) = firstPartOfModName(ME) .

op timeLimitText : Term Module Database -> QidList .
eq timeLimitText({'no`time`limit}'TimeLimitExpr, M, DB) =
```

```
      ('no 'time 'limit) .
  eq timeLimitText('le_[T], M, DB) =
      ('in 'time '<= meta-pretty-print(M, getTerm(T, M, DB))) .
  eq timeLimitText('lt_[T], M, DB) =
      ('in 'time '< meta-pretty-print(M, getTerm(T, M, DB))) .

  op getModeText : Term -> QidList .
  eq getModeText({'d}'ModeExpr) = ('default) .
  eq getModeText({'m}'ModeExpr) = ('local 'maximal) .
  eq getModeText({'ma}'ModeExpr) = ('global 'maximal) .
  eq getModeText({'gd}'ModeExpr) = ('global 'strategy 'and 'default) .
  [ ... ]
```

The function

```
op processStrategyCommand : Term ModName Database LazyLabelTable -> QidList .
```

handles a search command by computing the executable module in which the search is to take place, and its set of lazy labels, and calls the function

```
op processStrategyCommand : Term Module ModName Database QidSet -> QidList .
```

as follows:

```
eq processStrategyCommand(T, MN, DB, LLT) =
      processStrategyCommand(T, getFlatUnit(MN, DB), MN, DB,
                             lazyLabels(firstPartOfModName(MN), LLT)) .
```

This latter function handles the search command by calling the function

```
op solveStrat : Term Module Database QidSet -> TermSet .
```

which finds the set of terms resulting from the nested search expression, as follows:

```
eq processStrategyCommand('strat_.[T], M, MN, DB, QS) =
      ('Strategy: printStrategy(T, M, DB) '. '\n
       'Result: '\n
       printTermSet(M, solveStrat(T, M, DB, QS)) '\n) .
```

The functions

```
op printTermSet : Module TermSet -> QidList .
op printStrategy : Term Module Database -> QidList .
```

respectively, transforms a set of term into a list of quoted identifiers, and "prints" an echo of the search command. The function `solveStrat` must then parse the search command and call the appropriate strategy function.

The simplest case is when the term to find is the initial term:

```
eq solveStrat('term_[T], M, DB, QS) = meta-reduce(M, getTerm(T, M, DB)) .
```

The search functions were specified in two different versions, depending on whether the time limit of the search is strictly less than a certain time value, or less than or equal to that value. For example, to find all reachable terms within certain time and depth limits, we call the function `dfsAllLe` or the function `dfsAllLt`, depending on whether the time limit is strict or not:

```
eq solveStrat('find`[_`]_maximal`terms`reachable`from_mode_dti_within`time_[T, T',
                    T'', T''', T'''', T''''', M, DB, QS) =
      if strictTimeLimit(T''''') then
      dfsAllLt(M, solveStrat(T'', M, DB, QS), QS,
                  getTimeLimit(T''''', M, DB),
                  getMode(T''', T'''', M, DB),
                  getInt(T), getInt(T'))
      else
      dfsAllLe(M, solveStrat(T'', M, DB, QS), QS,
                  getTimeLimit(T''''', M, DB),
                  getMode(T''', T'''', M, DB),
                  getInt(T), getInt(T'))
      fi .

op strictTimeLimit : Term -> Bool .
eq strictTimeLimit('lt_[T]) = true .
eq strictTimeLimit('le_[T]) = false .
eq strictTimeLimit({'no`time`limit}'TimeLimitExpr) = false .
```

The treatment of the other search expressions is entirely similar:

```
eq solveStrat('find`[_`]_deadlocks`from_mode_dti_within`time_[T, T',
                    T'', T''', T'''', T'''''], M, DB, QS) = [ ... ]
```

Finding terms not satisfying an invariant $\Box_{\leq r} P$ is done by calling `dfsUStableLeNeg` to find states representing paths not satisfying (`matches noTerm`) $\textbf{UStable}_{\leq r}\ P$:

```
eq solveStrat('find`[_`]_not`satisfying`[`]_from_mode_dti_within`time_[T, T',
                T'', T''', T'''', T''''', T''''''], M, DB, QS) =
      if strictTimeLimit(T'''''') then
         dfsUStableLtNeg(M, solveStrat(T''', M, DB, QS), QS,
                           noTerm, getPattern(T'', M, DB),
                           getTimeLimit(T'''''', M, DB),
                           getMode(T'''', T''''', M, DB),
                           getInt(T), getInt(T'))
      else
         dfsUStableLeNeg(M, solveStrat(T''', M, DB, QS), QS,
                           noTerm, getPattern(T'', M, DB),
                           getTimeLimit(T'''''', M, DB),
                           getMode(T'''', T''''', M, DB),
                           getInt(T), getInt(T'))
      fi .
```

```
  eq solveStrat('find`[_`]_satisfying_from_mode_dti_within`time_[T, T',
                  T'', T''', T'''', T''''', T''''''], M, DB, QS) = [ ... ]

  eq solveStrat('find`[_`]_not`satisfying`<>_from_mode_dti_within`time_[T, T',
                  T'', T''', T'''', T''''', T''''''], M, DB, QS) = [ ... ]

  eq solveStrat('find`[_`]_satisfying_Until_from_mode_dti_within`time_[T, T',
          T'', T''', T'''', T''''', T'''''', T'''''''], M, DB, QS) = [ ... ]

  eq solveStrat('find`[_`]_not`satisfying_Until_from_mode_dti_within`time_[T,
      T', T'', T''', T'''', T''''', T'''''', T'''''''], M, DB, QS) = [ ... ]

  eq solveStrat('find`[_`]_not`satisfying_UntilStable_from_mode_dti_within`time_[T,
      T', T'', T''', T'''', T''''', T'''''', T'''''''], M, DB, QS) = [ ... ]

  eq solveStrat('keep_from_[T, T'], M, DB, QS) =
      restrictSize(M, solveStrat(T', M, DB, QS), getInt(T)) .

  eq solveStrat('remove_from_[T, T'], M, DB, QS) =
      removeMatches(M, solveStrat(T', M, DB, QS), getPattern(T, M, DB)) .
```

Sets of terms and the strategy echoes are given as quoted identifiers as follows:

```
  eq printTermSet(M, emptyTermSet) = ('No 'terms!) .
  ceq printTermSet(M, termSet(TS, TS')) =
       (printTermSet(M, TS) '\n printTermSet(M, TS'))
     if TS =/= emptyTermSet and TS' =/= emptyTermSet .
  eq printTermSet(M, T) =
       my-meta-pretty-print(M, T) .

  op printInt : Term -> Qid .
  eq printInt('token[{Q}Q']) = strip(Q) .

  eq printStrategy('term_[T], M, DB) =
      my-meta-pretty-print(M, getTerm(T, M, DB)) .

  eq printStrategy('find`[_`]_maximal`terms`reachable`from_mode_dti_within`time_[T,
                    T', T'', T''', T'''', T'''''], M, DB) =
      ( 'find printInt(T') 'terms 'reachable 'from
        '`( printStrategy(T'', M, DB) '`)
        printModeAndTimeLimit(T''', T'''', T''''', M, DB) ) .

[ ... ]

  op printModeAndTimeLimit : Term Term Term Module Database -> QidList .
  eq printModeAndTimeLimit(T, T', T'', M, DB) =
       ( 'with getModeText(T) 'time 'increase  'with
         'dti  my-meta-pretty-print(M, getTerm(T', M, DB)) 'and
                          timeLimitText(T'', M, DB) ) .
[ ... ]
endfm
```

4.3.6 Database Handling

The database object in Full Maude's persistent state stores the "application-level" modules entered in the Full Maude session, and is responsible for treating parsed input. The database class has the following attributes: `db`, which contains the database of modules read by Full Maude; `input`, which contains the parsed input term representing a premodule or a precommand to be treated; `output`, which contains a list of quoted identifiers resulting from treating the input term; and `default`, whose value is the name of the "current" module.

The persistent state for timed specifications should also store the table of lazy labels for each module. We do not modify the class `Database`, but instead define a subclass `TimedDatabase` of `Database` with an attribute `lazyLabels`. (The specification of Real-Time Maude is written in (Core) Maude, where there is currently no syntactic support for object-oriented specification. The module `CONFIGURATION` defines some features allowing us to define the database in an object-oriented way, albeit without Full Maude's convenient syntactic sugar for defining classes.)

Recalling that the declarations

```
  sort TimedDatabaseClass .
  subsort TimedDatabaseClass < DatabaseClass .
```

are already added to the module `DATABASE-HANDLING`, the class `TIMED-DATABASE-HANDLING` is declared as follows:

```
mod TIMED-DATABASE-HANDLING is
  including DATABASE-HANDLING .
  including TIMED-UNIT-PROCESSING . *** Importing the data type for lazy labels
  including TIMED-COMMAND-PROCESSING .

  op TimedDatabase : -> TimedDatabaseClass .
  op lazyLabels :_ : LazyLabelTable -> Attribute .

  var ATTS : AttributeSet .       var DATABASE : DatabaseClass .
  var TIMEDDATABASE : TimedDatabaseClass .
  var DB : Database .             var F : Qid .
  vars T T' T'' T''' : Term .     var TL : TermList .
  var O : Oid .                   var MN : ModName .
  var LLT : LazyLabelTable .
```

The database object is initially an instance of the class `Database`, and not of the class `TimedDatabase`, and is transformed into an object of the latter class when a timed module or timed theory is about to be treated:

```
  crl [databaseToTimeddatabase] :
      < O : DATABASE | input : (F[T, T']), ATTS >
      =>
      < O : TimedDatabase | input : (F[T, T']), ATTS,
                               lazyLabels : emptyLazyLabelTable >
      if ((F == 'tmod_is_endtm) or (F == 'tomod_is_endtom) or
          (F == 'tth_is_endtth) or (F == 'toth_is_endtoth)) and
         not (DATABASE : TimedDatabaseClass) .
```

A timed premodule is treated by performing the preprocessing described in Section 3.1.2 before calling the function `processUnit`:

```
crl [timedModule] :
    < O : TIMEDDATABASE | db : DB, input : (F[T, T']), output : nil,
                          default : MN, lazyLabels : LLT, ATTS >
    =>
    < O : TIMEDDATABASE | db : processUnit(
                                  timedPreModuleToPreModule(
                                      processTimedMetaLevel(F[T, T'], LLT)),
                                                          DB),
                          input : nilTermList,
                          output : ('\n 'Introduced 'timed 'module:
                                    modNameToQid(parseModName(T))),
                          default : parseModName(T),
                          lazyLabels : updateLazyLabels(F[T, T'], LLT),
                          ATTS >
    if (F == 'tmod_is_endtm) or (F == 'tomod_is_endtom) .

crl [timedTheory] :
    < O : TIMEDDATABASE | db : DB, input : (F[T, T']), output : nil,
                          default : MN, lazyLabels : LLT, ATTS >
    =>
    < O : TIMEDDATABASE | db : processUnit(
                                  timedPreModuleToPreModule(
                                    processTimedMetaLevel(F[T, T'], LLT)),
                                                          DB),
                          input : nilTermList,
                          output : ('\n 'Introduced 'timed 'theory:
                                    modNameToQid(parseModName(T))),
                          default : parseModName(T),
                          lazyLabels : updateLazyLabels(F[T, T'], LLT),
                          ATTS >
    if (F == 'tth_is_endtth) or (F == 'toth_is_endtoth) .
```

If an untimed premodule is read when the database object is an instance of the class `TimedDatabase`, then the `TIMED-META-LEVEL` preprocessing is performed before the premodule is treated as in Full Maude:

```
crl [module2] :
   < O : TIMEDDATABASE | db : DB, input : (F[T, T']),
      output : nil, default : MN, lazyLabels : LLT, ATTS >
   => < O : TIMEDDATABASE |
         db : processUnit(
                  processTimedMetaLevel(F[T, T'], LLT), DB),
         input : nilTermList,
         output :
           ('\n 'Introduced 'module: modNameToQid(parseModName(T))),
         default : parseModName(T), lazyLabels : LLT, ATTS >
      if ((F == 'fmod_is_endfm) or (F == 'obj_is_endo) or
         (F == 'obj_is_jbo) or (F == 'mod_is_endm)
            or (F == 'omod_is_endom)) .
```

```
crl [theory2] :
   < O : TIMEDDATABASE | db : DB, input : (F[T, T']),
      output : nil, default : MN, lazyLabels : LLT, ATTS  >
   => < O : TIMEDDATABASE |
         db : processUnit(processTimedMetaLevel(F[T, T'], LLT),  DB),
         input : nilTermList,
         output :
           ('\n 'Introduced 'theory: modNameToQid(parseModName(T))),
         default : parseModName(T), lazyLabels : LLT, ATTS >
      if ((F == 'fth_is_endfth) or (F == 'th_is_endth)
            or (F == 'oth_is_endoth)) .
```

The `show lazy` commands are treated directly in this module:

```
rl [showLazyLabels1] :
   < O : TIMEDDATABASE | db : DB,
                         input : ({'show`lazy`.}'PreCommand),
                         output : nil,
                         default : MN,
                         lazyLabels : LLT, ATTS >
   =>
   < O : TIMEDDATABASE | db : DB,
                         input : nilTermList,
                         output : ('The 'lazy 'rules 'of 'module
                                    modNameToQid(MN) '`( 'including
                                    'imported 'rules '`) 'are:
                                    qidSetToQidList(lazyLabels(MN, LLT)) '.),
                         default : MN,
                         lazyLabels : LLT, ATTS > .

rl [showLazyLabels2] :
   < O : TIMEDDATABASE | db : DB, input : ('show`lazy_.[T]), output : nil,
                         default : MN, lazyLabels : LLT, ATTS >
   =>
   < O : TIMEDDATABASE | db : DB,
                         input : nilTermList,
                         output : ('The 'lazy 'rules 'of 'module
                                    getName(T) '`( 'including
                                    'imported 'rules '`) 'are:
                           qidSetToQidList(lazyLabels(getName(T), LLT)) '.),
                         default : MN,
                         lazyLabels : LLT, ATTS > .
```

Timed rewrite and search strategy commands are treated by the functions `processTimedCommand` and `processStrategyCommand`, respectively:

```
crl [trew] :
   < O : TIMEDDATABASE | db : DB,  input : (F[TL]), output : nil,
                         default : MN, lazyLabels : LLT, ATTS >
   =>
```

```
    < O : TIMEDDATABASE | db : DB, input : nilTermList,
                          output : processTimedCommand(F[TL], MN, DB, LLT),
                          default : MN, lazyLabels : LLT, ATTS >
    if (F == 'trew-in`[_`]_:_mode_dti_in`time_.) or
       (F == 'trew`[_`]_mode_dti_in`time_.) .

  rl [strat] :
    < O : TIMEDDATABASE | db : DB, input : ('strat_.[T]), output : nil,
                          default : MN, lazyLabels : LLT, ATTS >
    =>
    < O : TIMEDDATABASE | db : DB, input : nilTermList,
                          output :
                             processStrategyCommand('strat_.[T], MN, DB, LLT),
                          default : MN, lazyLabels : LLT, ATTS > .
```

Finally, some error messages should be printed when timed commands are given before timed modules have been read:

```
  crl [errorTimedCommand1] :
      < O : DATABASE | input : (F[TL]), output : nil, ATTS >
      =>
      < O : DATABASE | input : nilTermList,
                       output : ('Error: 'no 'timed 'module 'introduced), ATTS >
      if ((F == 'trew-in`[_`]_:_mode_dti_in`time_.) or (F == 'show`lazy_.) or
          (F == 'trew`[_`]_mode_dti_in`time_.) or (F == 'strat_.)) and
         not (DATABASE : TimedDatabaseClass) .

  crl [errorTimedCommand2] :
      < O : DATABASE | input : ({'show`lazy`.}'PreCommand),
                       output : nil, ATTS >
      =>
      < O : DATABASE | input : nilTermList,
                       output : ('Error: 'no 'timed 'module 'introduced), ATTS >
      if not (DATABASE : TimedDatabaseClass) .
endm
```

4.3.7 Predefined Modules

The module TIMED-PREDEF-UNITS defines some modules which are placed in the persistent database, and can be imported by any user-defined Real-Time Maude module. The predefined modules are the module TIMED-PRELUDE, which is imported automatically by all timed modules; the module TIME-INF, which extends the time domain with an infinity value INF; the module LTIME, which assumes a linear time domain and defines the functions min and max; the module LTIME-INF which combines these last two modules; and the module TIMED-OO-PRELUDE, which defines the richer sort structure for object-oriented real-time specifications and which is imported automatically by all object-oriented timed modules:

```
fmod TIMED-PREDEF-UNITS is
  protecting PREDEF-UNITS .
```

```
op TIMED-PRELUDE : -> StrFModule .
eq TIMED-PRELUDE =
     (fmod 'TIMED-PRELUDE is
        nilParameterList
        including 'BOOL .
        sorts 'Time ; 'State ; 'System ; 'ClockedSystem .
        subsort 'System < 'ClockedSystem .
        op 'zero : nil -> 'Time [none] .
        op '_plus_ : 'Time 'Time -> 'Time
                                    [assoc comm prec(33) gather('E 'e)] .
        op '_monus_ : 'Time 'Time -> 'Time [prec(33) gather('E 'e)] .
        op '_le_ : 'Time 'Time -> 'Bool [prec(37)] .
        op '_lt_ : 'Time 'Time -> 'Bool [prec(37)] .
        op '_ge_ : 'Time 'Time -> 'Bool [prec(37)] .
        op '_gt_ : 'Time 'Time -> 'Bool [prec(37)] .
        op '`{_`} : 'State -> 'System [none] .
        op '_in`time_ : 'ClockedSystem 'Time -> 'ClockedSystem [none] .
        var 'CLSYSTEM : 'ClockedSystem .
        var 'R : 'Time .
        var 'R' : 'Time .
        none
        eq '_plus_[{'zero}'Time, 'R] = 'R .            *** Monoid with id 0.
        eq '_in`time_['_in`time_['CLSYSTEM, 'R], 'R'] =
               '_in`time_['CLSYSTEM, '_plus_['R, 'R']] .
        eq '_le_['R, 'R'] = '_or_['_lt_['R, 'R'], '_==_['R, 'R']] .
        *** Also define gt and ge by le and lt:
        eq '_ge_['R, 'R'] = '_le_['R', 'R] .
        eq '_gt_['R, 'R'] = '_lt_['R', 'R] .
      endfm) .

op TIME-INF : -> StrFModule .
eq TIME-INF =
     (fmod 'TIME-INF is
        nilParameterList
        including 'TIMED-PRELUDE .
        sorts 'TimeInf .
        subsort 'Time < 'TimeInf .
        op 'INF : nil -> 'TimeInf [none] .
        op '_plus_ : 'TimeInf 'TimeInf -> 'TimeInf
                                        [assoc comm prec(33) gather('E 'e)] .
        op '_monus_ : 'TimeInf 'Time -> 'TimeInf [prec(33) gather('E 'e)] .
        op '_le_ : 'TimeInf 'TimeInf -> 'Bool [prec(37)] .
        op '_lt_ : 'TimeInf 'TimeInf -> 'Bool [prec(37)] .
        op '_ge_ : 'TimeInf 'TimeInf -> 'Bool [prec(37)] .
        op '_gt_ : 'TimeInf 'TimeInf -> 'Bool [prec(37)] .
        var 'TI : 'TimeInf .
        var 'TI' : 'TimeInf .
        var 'R : 'Time .
        none
        eq '_plus_[{'INF}'TimeInf, 'TI] =  {'INF}'TimeInf .
        eq '_monus_[{'INF}'TimeInf, 'R] =  {'INF}'TimeInf .
        eq '_le_['TI, {'INF}'TimeInf] = {'true}'Bool .
        eq '_le_[{'INF}'TimeInf, 'R] = {'true}'Bool .
```

```
        eq '_lt_[{'INF}'TimeInf, 'TI] = {'false}'Bool .
        eq '_lt_['R, {'INF}'TimeInf] = {'true}'Bool .
        eq '_ge_['TI, 'TI'] = '_le_['TI', 'TI] .
        eq '_gt_['TI, 'TI'] = '_lt_['TI', 'TI] .
      endfm) .

op LTIME : -> StrFModule .
eq LTIME =
     (fmod 'LTIME is
        nilParameterList
        including 'TIMED-PRELUDE .
        sorts none .
        none
        op 'min : 'Time 'Time -> 'Time [assoc comm] .
        op 'max : 'Time 'Time -> 'Time [assoc comm] .
        var 'R : 'Time .
        var 'R' : 'Time .
        none
        ceq 'max['R, 'R'] = 'R' if '_le_['R, 'R'] = {'true}'Bool .
        ceq 'min['R, 'R'] = 'R if '_le_['R, 'R'] = {'true}'Bool .
      endfm) .

op LTIME-INF : -> StrFModule .
eq LTIME-INF =
     (fmod 'LTIME-INF is
        nilParameterList
        including 'TIME-INF .
        including 'LTIME .
        sorts none .
        none
        op 'min : 'TimeInf 'TimeInf -> 'TimeInf [assoc comm] .
        op 'max : 'TimeInf 'TimeInf -> 'TimeInf [assoc comm] .
        var 'TI : 'TimeInf .
        none
        eq 'max['TI, {'INF}'TimeInf] = {'INF}'TimeInf .
        eq 'min['TI, {'INF}'TimeInf] = 'TI .
      endfm) .

op TIMED-OO-PRELUDE : -> StrFModule .
eq TIMED-OO-PRELUDE =
     (fmod 'TIMED-OO-PRELUDE is
        nilParameterList
        including 'CONFIGURATION .
        including 'TIMED-PRELUDE .
        sorts 'EmptyConf ; 'NEConfiguration ; 'MsgConf ; 'NEMsgConf ;
              'ObjConf ; 'NEObjConf ; 'NETokens ; 'Tokens ; 'TokenConf .
        subsort 'EmptyConf < 'MsgConf .
        subsort 'EmptyConf < 'ObjConf .
        subsort 'EmptyConf < 'Tokens .
        subsort 'Msg < 'NEMsgConf .
        subsort 'Object < 'NEObjConf .
        subsort 'NEMsgConf < 'MsgConf .
        subsort 'NEMsgConf < 'NEConfiguration .
```

```
          subsort 'NEObjConf < 'ObjConf .
          subsort 'NEObjConf < 'NEConfiguration .
          subsort 'NETokens < 'Tokens .
          subsort 'MsgConf < 'Configuration .
          subsort 'ObjConf < 'Configuration .
          subsort 'NEConfiguration < 'Configuration .
          subsort 'Configuration < 'TokenConf .
          subsort 'Tokens < 'TokenConf .
          subsort 'TokenConf < 'State .
          op 'none : nil -> 'EmptyConf [none] .
          op '* : nil -> 'NETokens [none] .
          op '__ : 'EmptyConf 'EmptyConf -> 'EmptyConf
                [comm assoc id({'none}'Configuration)] .
          op '__ : 'NEConfiguration 'NEConfiguration -> 'NEConfiguration
                [comm assoc id({'none}'Configuration)] .
          op '__ : 'MsgConf 'MsgConf -> 'MsgConf
                [comm assoc id({'none}'Configuration)] .
          op '__ : 'NEMsgConf 'NEMsgConf -> 'NEMsgConf
                [comm assoc id({'none}'Configuration)] .
          op '__ : 'ObjConf 'ObjConf -> 'ObjConf
                [comm assoc id({'none}'Configuration)] .
          op '__ : 'NEObjConf 'NEObjConf -> 'NEObjConf
                [comm assoc id({'none}'Configuration)] .
          op '__ : 'NETokens 'NETokens -> 'NETokens
                [comm assoc id({'none}'Configuration)] .
          op '__ : 'Tokens 'Tokens -> 'Tokens
                [comm assoc id({'none}'Configuration)] .
          op '__ : 'TokenConf 'TokenConf -> 'TokenConf
                [comm assoc id({'none}'Configuration)] .
       none none none endfm) .
endfm
```

4.3.8 Handling Input/Output and Defining the Persistent Loop State

Finally, the module `REAL-TIME-MAUDE` extends the built-in Maude module `LOOP-MODE` [CDE+99a] to define and initialize the persistent state, and to specify the communication between the loop—the input/output of the system—and the database.

```
mod REAL-TIME-MAUDE is
  protecting META-REAL-TIME-MAUDE-SIGN .
  protecting TIMED-DATABASE-HANDLING .
  protecting TIMED-PREDEF-UNITS .
  including LOOP-MODE .

  op o : -> Oid .
```

The state of the persistent system is represented as a single object:

```
  subsort Object < State .
```

The constant `init` defines the initial loop state:

```
op init : -> System .

rl [init] :
   init
   => [nil,
       < o : Database |
          db : evalUnit(TIMED-OO-PRELUDE,
               evalUnit(LTIME-INF,
               evalUnit(LTIME,
               evalUnit(TIME-INF,
               evalUnit(TIMED-PRELUDE,
               evalUnit(CONFIGURATION,
                 evalUnit(TRIV,
                   evalUnit(UP, emptyDatabase)))))))),
          input : nilTermList,
          output : nil,
          default : nullModName >,
      ('\n '\t 'Real-Time 'Maude 'July '12. '\n)] .
```

When some text has been introduced in the loop, the first argument of the operator `[_,_,_]` is different from `nil`, and we can use this fact to activate the following rule, that enters an input such as a module or a command from the user into the database. The constant `TIMED-GRAMMAR` names the module containing the signature defining the top level syntax of Real-Time Maude. This signature is used by the `meta-parse` function to parse the input. If the input is syntactically valid[7], the parsed input is placed in the `input` attribute of the database object; otherwise, an error message is placed in the output buffer of the loop:

```
var Atts : AttributeSet .          var X@Database : DatabaseClass .
var O : Oid .                      var DB : Database .
var MN : ModName .                 vars QIL QIL' QIL'' : QidList .
var TL : TermList .

crl [in] :
   [QIL,
    < O : X@Database | db : DB, input : nilTermList,
       output : nil, default : MN, Atts >,
    QIL']
   => if meta-parse(TIMED-GRAMMAR, QIL) == error*
      then [nil,
            < O : X@Database | db : DB, input : nilTermList,
               output : ('ERROR: 'incorrect 'input '.),
               default : MN, Atts >,
            QIL']
      else [nil,
            < O : X@Database | db : DB,
               input : meta-parse(TIMED-GRAMMAR, QIL),
               output : nil, default : MN, Atts >,
            QIL']
      fi
      if QIL =/= nil .
```

[7] The input may be syntactically valid, but not semantically valid, since further processing—for example, of bubbles—may reveal a semantic inconsistency.

When the `output` attribute of the persistent object contains a nonempty list of quoted identifiers, the rule `out` moves it to the third argument of the loop. Then the Maude system displays it in the terminal:

```
  crl [out] :
     [QIL,
      < O : X@Database | db : DB, input : TL,
         output : QIL', default : MN, Atts >,
      QIL'']
     => [QIL,
         < O : X@Database | db : DB, input : TL,
            output : nil, default : MN, Atts >,
         (QIL' QIL'')]
        if QIL' =/= nil .
endm
```

Chapter 5

Specifying and Analyzing The AER/NCA Active Network Protocol in Real-Time Maude

This chapter describes our effort to specify and analyze the AER/NCA protocol suite, a new and complex real-time protocol suite for reliable multicast in active networks. The suite consists of a composable collection of protocol components supporting active error recovery (AER) and nominee-based congestion avoidance (NCA) features.

The AER/NCA protocol suite was proposed by S. K. Kasera (Bell Labs), S. Bhattacharyya (Sprint Labs), M. Keaton, D. Kiwior (both TASC Inc.), J. Kurose, D. Towsley (both UMass Amherst), and S. Zabele (TASC Inc.) [KBK^{+}99]. The protocol was originally specified informally as a set of use cases. This informal specification is given in Appendix B.

We have formalized the protocol in Real-Time Maude based on the (test) version 1.0 of the informal protocol given in Appendix B, consulting with Mark Keaton on various issues of the protocol. The Real-Time Maude specification is the first formal specification of the protocol. The formalization phase helped clarifying ambiguities and forced us to make implicit, but essential, knowledge explicit. After specifying the protocol, we executed it using Real-Time Maude's default interpreter, and we found a first set of errors this way. The flaws detected during this prototyping of earlier versions of the protocol were corrected, leading to a better specification, and more features, for example in the form of more accurate modeling of communication, were also added. The specification presented here is the result of many such iterations of the specification–prototyping–correction–feature-addition cycle. Parts of the protocol have been further validated by using Real-Time Maude's strategy library to explore all possible behaviors from a given initial state. Other parts of our final specification were analyzed by executing the specification using Real-Time Maude's default interpreter, which revealed some errors. We have traced the undesired behavior back to the original informal specification. The protocol was revised after our specification and analysis effort. The new version 1.1 [AER00] of the informal specification addresses many of the issues brought up by our work, both in terms of the behavior of the protocol, and in terms of making unstated and fuzzy knowledge explicit and clear.

Section 5.1 motivates the need for the protocol and gives an overview of it. Section 5.2 gives the detailed Real-Time Maude specification of the protocol, and gives some examples of how Real-Time

Maude executions were used to formally analyze various parts of the protocol. Section 5.3 discusses the formal methodology used and the experience gained in using Real-Time Maude to specify and analyze the AER/NCA protocol suite. Finally, Appendix B presents the original informal protocol upon which our specification was based.

5.1 Overview of the Protocol

The AER/NCA protocol is a (suite of) protocol(s) for reliable multicast in active networks, which makes use of the possibility of having some processing capabilities at "active nodes" between the sender and the receivers to achieve scalability and efficiency.

5.1.1 Motivation

The goal of reliable multicast is to send a sequence of data packets from a sender to a group of receivers. Packets may be lost due to congestion in the network, and it must be ensured that each receiver eventually receives each data packet.

Existing multicast protocols are either not scalable or do not guarantee delivery for reasons which include the following [KBK+99]:

- To ensure reliability, the sender must be given feedback from the receivers, either by acknowledging the reception of data packets (ACK), or by signaling the lack of an expected packet (NAK). When there are many receivers, and each one frequently sends (positive or negative) acknowledgments to the single sender, then the sender—and the links closest to it in the network—easily become overwhelmed by this traffic.
- If there are many receivers, then some packet will be lost somewhere almost all the time, keeping the sender busy with retransmissions. Furthermore, the sender has to multicast the repair packet to all the receivers—even though only a small group may have lost the packet—thereby increasing congestion in the network, or the sender must unicast the repair packet to the group, which is not desirable either for efficiency purposes when the losses are high.

In addition, the protocol should find the appropriate sending rate to ensure that there is some bandwidth left for competing unicast TCP sessions.

5.1.2 Repair Servers

To overcome the above problems, Kasera et al. [KBK+99] have suggested the use of *active services* at strategic locations inside the network. These active services can execute application-level programs inside routers, or, equivalently, on servers co-located with routers along the physical multicast distribution tree. By caching packets, these active services can subcast lost packets directly to "its" receivers, thereby localizing error recovery, making error recovery more efficient while solving the problem of retransmission scoping. They call such an active service a *repair server*. If a repair server does not have the missing packet in its cache, it aggregates all the non-acknowledgments (NAKs) it receives, and sends only one request for the lost packet towards the sender, solving the problem of feedback implosion at the sender.

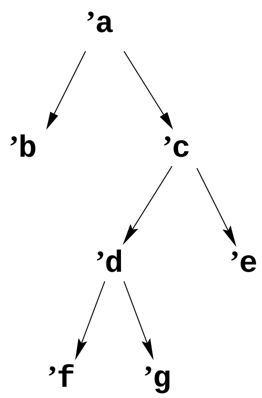

Figure 5.1: A multicast distribution tree.

Terminology: The multicast distribution tree has the *sender* at its root, the *receivers* in the multicast group as its leaf nodes, and the *repair servers* as internal nodes. In this tree, the first node on the path from a node n to the root is called the *parent* of n. The *siblings* and the *children* of a node can be defined analogously. The parent is sometimes also denoted the *upstream* node of a given node, and children are denoted as the *downstream* nodes. We use the expression *the (upstream) repair server of a node n* to denote the parent of node n, which is therefore a repair server or the sender! For example, the multicast tree in Fig. 5.1 has sender `'a`, repair servers `'c` and `'d`, and receivers `'b`, `'e`, `'f`, and `'g`. The sibling of node `'c` is `'b`, and the repair server of node `'c` is `'a`.

5.1.3 Real-Time Aspects

Time becomes a dimension in the protocol due to the fact that congestion is an essentially time-dependent phenomenon addressed by AER/NCA that must be modeled. Packets are sent through links, and the time it takes from a packet is sent from a node until it arrives at the receiver node of a link depends on the size of the packet, the number of packets already in the link, and the link speeds and propagation delays of the link. All these factors will affect the degree of congestion and must be modeled to faithfully analyze the AER/NCA protocols. Further real-time aspects also appear in the protocol through the use of timers to start new queries and to detect the loss of various kinds of packets.

5.1.4 Informal Description of the Protocol

The main tasks of the protocol are the following:

- The *active error recovery* component deals with packet losses and tries to ensure that each packet is eventually received by all receivers in the multicast group. To enhance efficiency, error recovery should happen as close as possible to the nodes where the losses were detected.

- *Rate control:* The loss of a substantial number of packets indicates over-congestion due to a too high frequency in the sending of packets. The rate control part of the protocol aims at dynamically adjusting the rate by which the sender sends new packets, so that the frequency decreases when many packets are lost, and increases in time intervals when no, or few, packet losses are detected.

- *Finding the nominee receiver:* The sender needs feedback about discovered packet losses to adjust its sending rate. However, letting *all* receivers report their loss rates would result in too many messages being sent around just to determine the loss rate. Instead, the protocol tries to find the "worst" receiver, based on the loss rates and the distance to the sender. Then the sender takes only the losses reported from this *nominee* receiver into account when determining the sending rate.

- *Finding round trip time values:* For finding the sending rate and the nominee, and to find how frequently to check for missing packets, some knowledge about the various *round trip times* (the time it takes for a packet to travel from a given node to another given node, and back) in the network is needed.

The (sub)protocols addressing these four tasks are described separately in the informal specification given in Appendix B.

Active Error Recovery

The active error recovery part of the protocol should ensure that all packets are received by all the members of the multicast group, while handling the error recovery as close as possible to the place where the packet was lost.

The *sender* sends a sequence of data packets to the multicast group. It handles messages indicating the loss of a packet by resending the packet (the packets are identified by their sequence numbers), if the desired retransmission has not been done before.

When a *repair server* or a *receiver* discovers that it has not received some data packets(s), it signals this by sending a NAK packet with the sequence number of the missing packet to its upstream repair server. This NAK-sending may be somewhat delayed, and is repeated at certain retransmission intervals until the packet is received. The repair server or receiver needs an estimate of its round trip time to the sender to know how frequently it should resend this NAK packet[1]. In addition, a repair server stores received data packets in its cache, and subcasts original data packets downstream. When it discovers a packet loss, it also subcasts a NAK packet downstream to indicate to its children that it is aware of the missing data packet and has started the error recovery, and that it is no use for the children to complain. Finally, if a repair server receives a NAK from downstream, it subcasts the missing data packet if it has it.

Congestion Control

The frequency at which the sender sends new data packets was not specified in the previous protocol. The aim of the congestion control part is to find a sending rate such that the multicast session does not overly congest any path from the sender to the receiver. That is, the most congested path should be identified, and the sending frequency should be adjusted so that this path is not overly

[1]For example, it is no use recomplaining 0.1 seconds after the first complaint if it takes the complaint 0.2 seconds to reach the sender.

congested, in order to ensure that there is enough bandwidth for competing TCP-sessions, and that the worst receiver can handle the onslaught of packets.

The congestion control consists of two subtasks, each of which is treated by a (sub)protocol:

- Finding the worst receiver.
- Adjusting the sending rate according to the loss rate at the worst receiver.

Rate control. Whenever the worst receiver, the *nominee* receiver, receives an original data packet, it sends a *congestion control message* (CCM) packet to the sender. Simply speaking, the frequency increases when CCM packets are received by sender, and decreases when the sender hasn't seen a CCM packet for a while. The actual computations are non-trivial.

Nominee receiver selection. The *nominee* receiver is the receiver with the highest value of $rtt \times \sqrt{lpe}$, where *rtt* is the estimated *round trip time* to the source, and *lpe* is the *loss probability estimate*.

Each *receiver* uses a sliding window to record the sequence of original packets it has received, based upon which it can compute its estimated loss rate, its *lpe*. Each receiver sends a *congestion status message* (CSM) packet with its current *lpe* and *rtt* estimates to its upstream repair server at regular intervals.

Each *repair server* receives CSM packets from each of its children, containing the nominee the subtree where the child is the root, together with the associated *rtt* and *lpe* values. The repair server computes the nominee in its own subtree based on these received values. Whenever there is a change in the nominee or its data, the repair server sends a CSM packet to its upstream repair server with these new data. It also resends this data at regular intervals, and blanks out its data if its nominee data has not been updated for a while.

The *sender* chooses the nominee receiver based on the CSM packets from its children. When a new nominee receiver is found, the sender sends a *nominee activation message* (NAM) packet to the old and the new receiver to notify them of their change of status (since (only) the new receiver should report back acknowledgments). Finally, the sender resends a NAM packet to the nominee at regular intervals.

5.2 Specifying and Analyzing the AER/NCA Protocol in Real-Time Maude

We specify in this section the AER/NCA protocol suite in Real-Time Maude in an object-oriented style. Although the four protocol components are closely inter-related, it is nevertheless interesting to analyze each part separately, as well as analyzing the combined protocol. In the specification presented in this section, we have tried to specify the four parts separately by identifying an "interface" to the other parts. We have also specified the combined protocol and have used object-oriented inheritance techniques so that transitions which are the same when a subprotocol is executed alone, and when it is executed with the other parts, are reused in both the stand-alone and the combined protocol. The specification follows the structure of the informal specification given in Appendix B, with the label of each rewrite rule corresponding to the name of the use case it models.

The protocol is quite complex and, since it addresses scalability issues and congestion phenomena associated with them, symbolic simulation of the algorithm often requires large states, and the need to efficiently execute a large number of rewrite steps. These considerations, together with the need for performing formal analysis involving exhaustive state space searches, dictated us to remove some potentially time-consuming bottlenecks, without altering the specification too much. In particular, we have made the following choices for execution efficiency purposes:

- We use a discrete time domain, namely, the nonnegative machine integers, instead of a dense time domain like the rationals.
- The tick rule is deterministic.
- Instead of using the techniques described in Section 2.2.5 for specifying object-oriented real-time systems using tokens '*', all rules are instead global, i.e., involving terms of sort `System`. This yields a specification which is essentially equivalent to a system with tokens, in the sense that $\{t * \cdots *\} \longrightarrow \{t' * \cdots *\}$ is a one-step sequential rewrite in the corresponding systems with tokens and local rules if and only if $\{t\} \longrightarrow \{t'\}$ is a one-step sequential rewrite in the tokenless system.
- There are some timers which sometimes should be set to a value between 1 and n. Instead of modeling this by introducing a new variable x_r in the righthand side of the rule, with $1 \texttt{<=} x_r$ `and` $x_r \texttt{<=} n$, we use a "random" number generator to set the value of the timer.

Having the natural numbers as time domain poses no problems. Nothing is lost by having a deterministic tick rule, since it is easy to see that the protocol is time-deterministic, in the sense that no instantaneous action can take place before time advances as much as possible in the tick rule[2]. The situation is slightly inelegant with the use of global rules instead of the use of tokens, since, although the one-step sequential rewrites are equivalent (with the obvious correspondence) in the two kinds of specifications, concurrency is lost when only global rules are used and is instead simulated by interleavings. Similarly, the use of pseudo-random numbers instead of having a truly nondeterministic choice in the timer values amounts to an inconvenient overspecification of the protocol.

In summary, we may view the protocol specification $\mathcal{R}_{\phi,\tau}$ given in this section as an *implementation*, or *refinement*, of a more abstract and general, and conceptually more satisfying specification $\mathcal{R}'_{\phi,\tau}$ of the protocol, where global instantaneous rules of the form $\{t\} \longrightarrow \{t'\}$ in $\mathcal{R}_{\phi,\tau}$ are replaced by corresponding rules $t * \longrightarrow t' *$ in $\mathcal{R}'_{\phi,\tau}$, and where the use of pseudo-random numbers is replaced by the corresponding nondeterministic rules.

5.2.1 The Time Domain

Earlier versions of the protocol used the machine rational numbers as the time domain. However, it turned out that the straightforward extension of the built-in machine integers to the nonnegative rational numbers given in Appendix A made some computations on the rationals very inefficient—up to 300 000 rewrites were sometimes required for relatively simple computations. This made

[2]The possible exception is if sending of new packets can be attempted at "any" time.

our current specification of the rationals infeasible for serious specification efforts. Although we extended the machine integers to an efficient version of the fixed point real numbers, we were told that the set of natural numbers is an adequate time domain for the protocol. However, the fixed point reals are used in the specification of the rate control protocol, where integers are not sufficient.

The following specification of the machine natural numbers is used in the protocol:

```
(fmod NATURALS is
including MACHINE-INT .

sorts Nat NzNat .
subsort Nat < MachineInt .
subsort NzNat < Nat .
subsort NzNat < NzMachineInt .

op abs : MachineInt -> Nat .
op _minus_ : MachineInt MachineInt -> MachineInt [prec 33 gather(E e)] .
op _monus_ : Nat Nat -> Nat  [prec 33 gather(E e)] .
vars I I' : MachineInt .
var NZI : NzMachineInt .
vars N N' : Nat .

mb 0 : Nat .
cmb NZI  : NzNat if NZI > 0 .

eq I minus I' = _-_(I, I') .
eq abs(I) = if I < 0 then - I else I fi .
eq N monus N' = if N < N' then 0 else N minus N' fi .
endfm)
```

The machine naturals can be made into the time domain following the scheme described in Section 3.1. We also add a sort for the non-zero time values:

```
(fmod DISCRETE-TIME-DOMAIN is
  including LTIME-INF .
  including NATURALS .
  sort NzTime .
  subsort Nat < Time .
  subsort NzTime < Time .
  subsort NzNat < NzTime .

  vars N N' : Nat .
  eq zero = 0 .
  eq N plus N' = N + N' .
  eq N lt N' = N < N' .
endfm)
```

The protocol uses many timers. The intended use of a timer is to force an action to take place at a certain time. This is achieved by not letting time elapse beyond the expiration time of the timer, and by turning off or resetting the timer when the action associated to the expiration of the timer is performed. A timer is an attribute whose value is a time value r when the timer expires in time

r, and which has the value `noTimeValue` when the timer is turned off. To handle such a default time value, the following specification extends the time domain with the sort `DefTime`, the constant `noTimeValue` of this sort, and extends some operators on the time domain to this extended time domain. For clarity, we define new operators, such as `default-le`, instead of extending `le` to the new domain by overloading:

```
(fmod DEFAULT-VALUE-TIME-DOMAIN is
including DISCRETE-TIME-DOMAIN .
sorts DefTime DefTimeInf .
subsort Time < DefTime .
subsort TimeInf < DefTimeInf .
subsort DefTime < DefTimeInf .

op noTimeValue : -> DefTime .
op _default-le_ : DefTimeInf DefTimeInf
                  -> Bool [prec 37] .  *** "less-than-or-equal"
op _default-lt_ : DefTimeInf DefTimeInf -> Bool [prec 37] .  *** "less-than"
op _default-monus_ : DefTimeInf Time -> DefTimeInf [prec 33 gather(E e)] .
op default-min : DefTimeInf DefTimeInf -> DefTimeInf [assoc comm] .

vars TI TI' : TimeInf .
var DT : DefTime .
vars R R' : Time .

eq TI default-le TI' = TI le TI' .
eq TI default-le noTimeValue = false .
eq noTimeValue default-le DT = true .
eq TI default-lt TI' = TI lt TI' .
eq DT default-lt noTimeValue = false .
eq noTimeValue default-lt TI = true .
eq noTimeValue default-monus R = noTimeValue .
eq TI default-monus R = TI monus R .
eq noTimeValue default-monus R = noTimeValue .
eq default-min(TI, TI') = min(TI, TI') .
eq default-min(TI, noTimeValue) = TI .
eq default-min(noTimeValue, noTimeValue) = INF .
endfm)
```

5.2.2 Declarations for Object-Oriented Real-Time Systems

The following object-oriented timed module contains some generic declarations for object-oriented real-time systems. It defines the object identifiers to be the set `Qid` of quoted identifiers [CDE+99a], defines the set of such object identifiers, together with a default object identifier `noOid`. The module also declares the functions `mte`, for maximal time elapse, and `delta`, for the action of time on an object configuration, and adds the axioms which show how these functions distribute over objects and messages in a configuration:

```
(tomod TIMED-OO-SYSTEM is
including QID .
including LTIME-INF .
```

```
subsort Qid < Oid .     *** Oid's are just quoted identifiers.
sort DefOid .           *** Oid's with a default value, namely noOid.
subsort Oid < DefOid .
op noOid : -> DefOid .
sort OidSet .
subsort Oid < OidSet .
op none : -> OidSet .
op __ : OidSet OidSet -> OidSet [assoc comm id: none prec 15] .

op delta : Configuration Time -> Configuration .
op mte : Configuration -> TimeInf .

vars NECF NECF' : NEConfiguration .
var R : Time .
eq delta(none, R) = none .
eq delta(NECF NECF', R) = delta(NECF, R) delta(NECF', R) .
eq mte(none) = INF .
eq mte(NECF NECF') = min(mte(NECF), mte(NECF')) .
endtom)
```

The following module defines the tick rule, which, as stated above, is deterministic, which should not restrict the timed behavior of the instantaneous rules in the system at all[3]:

```
(tomod DETERMINISTIC-TICK-RULE is
 including TIMED-OO-SYSTEM .

 var OC : ObjConf .

 crl [tick] : {OC} => {delta(OC, mte(OC))} in time mte(OC)
                        if mte(OC) =/= INF and mte(OC) =/= 0 .
endtom)
```

The use of the variable `OC` of sort `ObjConf` requires that the configuration only consists of objects when the `tick` rule is applied, and therefore forces messages to be treated without delay when they are present at the outermost level in a configuration, because the above rule will not match and therefore time will not advance when there are messages present in the configuration.

5.2.3 Modeling Communication and the Communication Topology

The communication topology is modeled by defining the multicast tree with the sender, the repair servers co-located with, or inside, some routers, and the receivers. The appropriate classes for these objects are defined in the class `COMMON-CLASSES` as follows:

```
 class Sendable | children : OidSet .
 class Receivable | repairserver : DefOid .
```

[3]We are, as mentioned above, not entirely convinced that the sending of new data packets should not have nondeterministic behavior in time—forcing us to have a nondeterministic tick rule—but the latest discussions on this topic with our colleagues at TASC indicate that that is not the case.

```
class Sender .
subclass Sender < Sendable .

class Receiver .
subclass Receiver < Receivable .

class Repairserver .
subclass Repairserver < Sendable .
subclass Repairserver < Receivable .
```

The router table is not given explicitly. It is given implicitly by the multicast tree, which is given by the objects together with the values of their `children` attributes in their initial states. (The `repairserver` attribute values are set in the protocol, and not in the initial state, where they all have the value `noOid`.) A multicast follows this mutlicast tree. That is, the server sends the packet to its children, who then subcast it to their children, and so on.

Links

Packets travel along bidirectional *links*. A link is characterized by its end nodes, its *bound*, its *linkspeed*, and its *propagation delay*. There is one link between each parent-child pair in the topology. The bound defines how many packets can travel along a link at any time. A packet is lost if and only if there is an attempt to send it along a link which is full. The class `Link` can be defined as follows:

```
(tomod LINK is
including TIMED-OO-SYSTEM .
including DEFAULT-VALUE-TIME-DOMAIN .

*** Lists of messages may be the most appropriate way of storing
*** the messages:

sort MsgList .
subsort Msg < MsgList .
op nil : -> MsgList .
op _+_ : MsgList MsgList -> MsgList [assoc id: nil] .
*** The leftmost element in a channel as the first one, the oldest one.

class Link | up : Oid,                  *** "Upnode"
             down : Oid,                *** "DownNode"
             bound : NzNat,             *** Maximal size of buffer in link
             propDelay : NzTime,        *** Propagation delay. Constant.
             linkSpeed : NzNat,         *** in megabits per sec.
             downMsgs : MsgList,        *** Buffer of messages going downstream
             downSize : Nat,            *** size of above
             upMsgs : MsgList,          *** Buffer of messages going upstream
             upSize : Nat .             *** size of above
```

Before defining the behavior of a link, we list the variables used in the axioms specifying such behavior in this module:

```
var M : Msg .                     vars R R' R'' : Time .
vars NZR : NzTime .               vars ML ML' : MsgList .
vars Q Q' Q'' : Oid .             vars N N' : Nat .
var NZN : NzNat .                 vars TI TI' : TimeInf .
var CF : Configuration .          var OS : OidSet .
var SMALLPACKET : Packet .        var LARGEPACKET : LargePacket .
```

The *transmission delay* of a packet in a link is the packet size divided by the linkspeed, and the total delay of a packet entering a link is

$$\max(\text{propagation delay}, \mathit{maxDelayInLink}) + \text{transmission delay},$$

where *maxDelayInLink* is the current delay of the last packet entered in the link, and is 0 if there are no packets in the link. Data packets are usually around 1500 bytes large, and all other kinds of packets are 64 bytes large:

```
sorts Packet LargePacket .
subsort Packet < Msg .
subsort LargePacket < Msg .

*** Transmission delay given message and link speed:
op transDelay : Msg NzNat -> Time .
eq transDelay(SMALLPACKET, NZN) =
     (64 * 8 plus ((NZN * 1000) minus 1)) / (NZN * 1000) .
eq transDelay(LARGEPACKET, NZN) =
     (1500 * 8 plus ((NZN * 1000) minus 1)) / (NZN * 1000) .
*** NZN times 1000 gives bits per millisecond, and we also round off
*** to closest integer.
```

The attempt to enter a packet p into the link from a to b takes place by the method call/message `send`(p, a, b). This message is treated by the link by either discarding the packet if the link is full, or by storing it in the link as `dly`(p, r), where, at any time, r is the current delay of the packet. That is, the time remaining until the packet is delivered at the other end of the link. The packet is delivered by sending the message p `from` a `to` b to the global configuration, where it should then be treated by object b:

```
*** Packet ``wrappers'':
msg send : Msg Oid Oid -> Msg .
msg _from_to_ : Msg Oid Oid -> Msg .
msg dly : Msg Time -> Msg .
```

In the current version of the protocol, the link capacity limit is related only to the number of packets, regardless of the sizes of the packets. Furthermore, there is a possible ambiguity when the link is full and the delay of some packet p in the link is 0, and there is an attempt to send another packet p' along the link. Should p' be lost due to link overflow even though packet p is ripe for delivery, or should such packets always be released before sending another packet is attempted? The current specification always checks first whether a packet is ready for delivery and releases such packets before treating sending messages for new packets:

```
*** The functions leastDly and greatestDly gives the least and greatest
*** delay of a message in the message list:
op leastDly : MsgList -> TimeInf .
eq leastDly(nil) = INF .
eq leastDly(dly(M, R) + ML) = R .
op greatestDly : MsgList -> Time .
eq greatestDly(nil) = 0 .
eq greatestDly(ML + dly(M, R)) = R .
```

The following rules take a packet `M` to be sent from `Q` to `Q'` and put it in the link with appropriate delay if the link is not full; otherwise they drop the packet. The rules do not apply when there is a packet ready for delivery in the link, forcing the link to deliver such a packet before treating `send`-requests:

```
crl [intoLinkDown] :
    { send(M, Q, Q')
    < Q'' : Link | up : Q, down : Q', bound : N, propDelay : NZR,
                   linkSpeed : NZN, downMsgs : ML,  downSize : N' >  CF } =>
    { if N' lt N then
        (< Q'' : Link | downMsgs : ML + dly(M, max(NZR, greatestDly(ML))
                                           plus transDelay(M, NZN)),
                        downSize : N' plus 1 > CF)
      else  (< Q'' : Link | > CF) fi }
    if leastDly(ML) =/= 0 .

crl [intoLinkUp] :
    { send(M, Q, Q')
    < Q'' : Link | down : Q, up : Q', bound : N, propDelay : NZR,
                   linkSpeed : NZN, upMsgs : ML, upSize : N' > CF }  =>
    { if N' lt N then
        (< Q'' : Link | upMsgs : ML +  dly(M, max(NZR, greatestDly(ML))
                                          plus transDelay(M, NZN)),
                        upSize : N' + 1 > CF)
    else (< Q'' : Link | > CF) fi }
    if leastDly(ML) =/= 0 .
```

A message is ready to get out of the link when its delay is 0:

```
rl [outOfUpLink] :
   { < Q : Link | down : Q', up : Q'',
                  upMsgs : dly(M, 0) + ML, upSize : N > CF } =>
   { < Q : Link | upMsgs : ML, upSize : N minus 1 > (M from Q' to Q'') CF } .

rl [outOfDownLink] :
   { < Q : Link |  down : Q', up : Q'', downMsgs : dly(M, 0) + ML,
                   downSize : N > CF } =>
   { < Q : Link | downMsgs : ML, downSize : N minus 1 >  (M from Q'' to Q') CF } .
```

The timed behavior of a link is as follows, assuming that the class `Link` will never have subclasses:

```
eq delta(< Q : Link | downMsgs : ML, upMsgs : ML' >, R) =
        < Q : Link | downMsgs : delta(ML, R), upMsgs : delta(ML', R) > .
op delta : MsgList Time -> MsgList .
op delta : Msg Time -> Msg .              *** Ensures regularity!
eq delta((nil).MsgList, R) = (nil).MsgList .
ceq delta(ML + ML', R) = delta(ML, R) + delta(ML', R)
                         if ML =/= nil and ML' =/= nil .
eq delta(dly(M, R), R') = dly(M, R monus R') .

eq mte(< Q : Link | downMsgs : ML, upMsgs : ML' >) =
      min(leastDly(ML), leastDly(ML')) .
```

Finally, we define how to send packets to a *group* of objects:

```
msg multiSend : Msg Oid OidSet -> Configuration .
eq multiSend(M, Q, (Q' OS)) =
            send(M, Q, Q') multiSend(M, Q, OS - Q') .
eq multiSend(M, Q, none) = none .

op _-_ : OidSet Oid -> OidSet .
eq (Q OS) - Q' = if Q == Q' then OS - Q' else Q (OS - Q') fi .
eq none - Q = none .
eq Q Q = Q .
endtom)
```

The tick rule forces messages at the outermost level of a configuration to be treated before time advances. There is therefore no need to define the functions `mte` and `delta` on such messages, except just to avoid introducing "junk" terms of sort `Time` in the specification.

Representation of the communication topology revisited. The communication topology is represented by the multicast tree defined by the sender, the repair servers, and the receivers, together with the value of the `children` attribute. However, not all the routers have associated repair servers. Therefore, the modeling of the topology may not be accurate, since the path between two active nodes may involve many links instead of just one. A possible way of making the model more precise is to introduce passive "router" nodes whose only job is to forward packets to its children. This would model (downstream) subcast and multicast communication more accurately. However, packets are also *unicast* upstream from receivers to its repair servers. If passive nodes are added, there is no easy way to model upstream unicast, since unicast may not necessarily follow the multicast path. In this version of the protocol, we have followed the advice of the network experts and have kept the model of communication described above.

5.2.4 The Class Hierarchy

The protocol is specified so that the four parts RTT (finding round trip times), NOM (finding the nominee receiver), RC (the rate control algorithm), and RS (repair service/error recovery) can be executed both separately and together in the combined protocol. We use object-oriented inheritance techniques to reuse those parts which are common to a stand-alone protocol and the

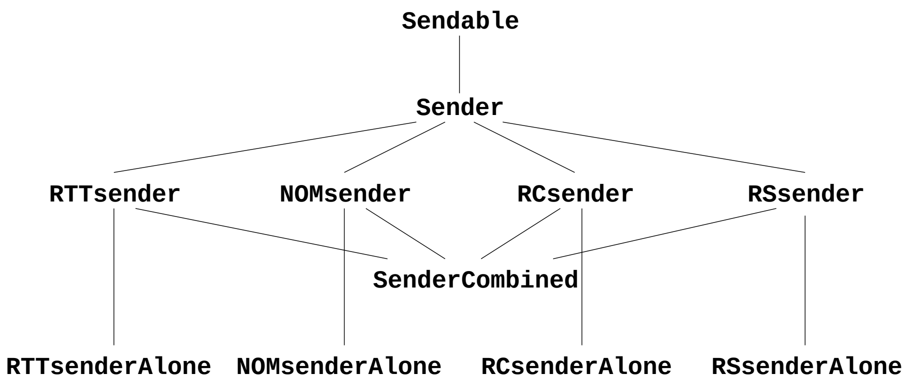

Figure 5.2: The sender class hierarchy.

combined protocol. Fig. 5.2 shows a slightly simplified version of the class hierarchy for sender objects. The class hierarchies for repair servers and receivers are entirely similar.

Objects of the class `RTTsenderAlone` should be used when the RTT part of the protocol is analyzed separately, while the sender object in the combined protocol should be an instance of the class `SenderCombined`. Since `RTTsenderAlone` and `SenderCombined` are subclasses of the class `RTTsender`, rules which apply to both the stand-alone part and the combined protocol should involve objects of class `RTTsender`. Rules which describe transitions that only may occur in the stand-alone RTT protocol should involve objects of class `RTTsenderAlone`, while transitions which only occur in the combined protocol should involve objects of class `SenderCombined`. Although we haven't done it, one could use the same technique to specify the combination of two or three of the subprotocols.

Transitions sometimes differ only slightly between their use in a stand-alone protocol and the combined protocol. Instead of writing distinct rules for each case, we may instead write only one rule for both cases, and test whether the object applied to the rule is an instance of a "combined" class, indicating that the combined protocol is being executed. Examples of such rules are `B5a` and `B5b` in the RTT protocol. For that purpose we define a sort `CombinedObject`, and define it as follows:

```
sort CombinedObject .
subsort CombinedObject < Object .

var Q : Oid .
mb (< Q : SenderCombined | >) : CombinedObject .
mb (< Q : RepairserverCombined | >) : CombinedObject .
mb (< Q : ReceiverCombined | >) : CombinedObject .
```

5.2.5 Other Frequently Used Classes

To adequately prototype a stand-alone protocol, it needs to be given some data which is normally provided by other parts of the (combined) protocol. In a simple first setting, an environment object defining the interface to the other parts of the protocol has one attribute, containing all messages, with given delays using the `dly`-operator, which are to be sent to the protocol. These environment messages will not be generated by rules, but will be given in the initial state:

```
(tomod ENVIRONMENT is
including LINK .

var Q : Oid .                    var MC : MsgConf .
vars NEMC NEMC' : NEMsgConf .    var CF : Configuration .
var M : Msg .                    var R : Time .

class Env | msgsFromEnv : MsgConf .

rl [env1] :
    < Q : Env | msgsFromEnv : dly(M, 0) MC > CF  =>
    < Q : Env | msgsFromEnv : MC > M CF  .

eq mte(< Q : Env | msgsFromEnv : MC >) = leastDly(MC) .

op leastDly : MsgConf -> TimeInf .
eq leastDly(dly(M, R)) = R .
eq leastDly(none) = INF .
eq leastDly(NEMC NEMC') = min(leastDly(NEMC), leastDly(NEMC')) .

eq delta(< Q : Env | msgsFromEnv : MC >, R) =
      < Q : Env | msgsFromEnv : delta(MC, R) > .
endtom)
```

The last commonly used module introduces the pseudo-random number generator, which is used in most protocols. The function `random` generating a pseudo-random sequence of natural numbers of natural numbers is an instance of a class of "good" pseudo-random number generators given in [Knu81]. An object of class `RandomNGen` carries the seed used for the random function, but each object could also have a "local" seed. The seed should be updated by applying the `random` function to it whenever the `random` function is applied with the seed.

```
(tomod RANDOM is
including TIMED-OO-SYSTEM .
including NATURALS .
class RandomNGen | seed : Nat .
op random : Nat -> Nat .      *** random(x) generates the next random number
var N : Nat .                 var NZN : NzNat .
var R : Time .                var Q : Qid .
eq random(N) = ((104 * N) + 7921) % 10609 .

*** The seed may be modified by applying the random function many times:
op repeatRandom : Nat Nat -> Nat .
*** repeatRandom(seed, noOfRepetitions)
eq repeatRandom(N, NZN) = repeatRandom(random(N), NZN minus 1) .
eq repeatRandom(N, 0) = N .

eq delta(< Q : RandomNGen | >, R) = < Q : RandomNGen | > .
eq mte(< Q : RandomNGen | >) = INF .
endtom)
```

Most classes have a clock attribute, and can be defined as subclasses of the class `Clock`:

```
class Clock | clock : Time .
```

A List of Frequently Used Variables

To avoid repeating in this exposition the declarations of variables used frequently in axioms, we list below some variables used frequently in the modules below.

```
var CF : Configuration .
vars DQ DQ' DQ'' : DefOid .
vars DT DT' DT'' DT''' : DefTime .
vars I I' : MachineInt .
vars M M' : Msg .
vars MC MC' : MsgConf .
vars ML ML' : MsgList .
vars N N' N'' N''' : Nat .
vars NEMC NEMC' : NEMsgConf .
vars NZN NZN' NZN'' NZN''' NZN'''' : NzNat .
vars Q Q' Q'' Q''' Q'''' : Oid .
vars R R' R'' R''' R'''' R''''' : Time .
var OS : OidSet .
vars X Y Z X' : Bool .
```

Shared Messages

The following messages and packets are shared by different subprotocols also in stand-alone mode, and are declared in the module `SHARED-MSGS`:

```
 SPMPacket : Nat -> Packet .
 msg newRTT : DefTime DefTime Oid -> Msg .
 msg NAMPacket : Bool -> Packet .
```

`SPMPacket`s are used to let the nodes know their upstream repair servers so as to be able to unicast packets to them for localized error handling, etc. It's parameter gives the number of data packets transmitted by the sender. `newRTT` is used for method calls when new RTT values are found by the RTT protocol, and some additional updates have to be done by the repair service part of the protocol. `NAMPacket`s are unicast to receivers, and the parameter tells the receiver whether it is the nominee receiver.

5.2.6 The RTT Protocol

The task of the RTT part of the protocol is to find, for each repair server and receiver object, the following values:

- `sourceRTT`: The round trip time (RTT) from the sender to the object.

- `maxUpRTT`: The maximal RTT from the upstream repair server to any of its children.

The round trip time values should be the most recent estimated values, which is not a problem for `sourceRTT`. The `maxUpRTT` value cannot be set each time a node receives values from its repair server, since this value is dependent of the values of many of its siblings. Therefore, we are interested in a "fairly recent" value of `maxUpRTT`.

The other auxiliary attributes used to determine the desired attributes, are:

- `maxDownRTT`: The largest RTT estimate from this node to any of its immediate downstream nodes. `maxDownRTTSetTime` tells when this value was last updated, and `maxRecentDownRTT` is the largest `maxDownRTT` value since `maxDownRTTSetTime`, and is used for `maxDownRTT` when too much time has elapsed since the latter was last set.
- `myUpRTT`: The current RTT estimate from the node to its immediate upstream repair server or sender.

In addition, the timer attribute `getRTTResendTimer` is used to initiate an RTT request to an objects's repair server.

The protocol goes roughly as follows:

- A `startRTT` message initiates the sender, which sends *source path message* (SPM) packets downstream. Upon reception of such a packet, a node initiates its timer to a value between 0 and 30 ms. and subcasts the SPM packet downstream.
- When a timer expires, its node sends a `getRTTRequest` packet to its (upstream) repair server with the current time, and the node's current estimate of the RTT to its repair server.
- When the upstream repair server receives this `getRTTRequest` packet, it may need to update its own `maxDownRTT`, and sends a `getRTTResponse` back to the downstream node, with the timestamp unchanged, and its current `sourceRTT` and current `maxDownRTT` values.
- When the originator of this packet exchange receives the response, it can compute the "latest" RTT to its upstream repair server by just taking the current time minus the timestamp. Having this 1-step RTT, it adds this to the received `sourceRTT` value of its upstream repair server and gets its new `sourceRTT` estimate. It also compares the received `maxDownRTT` value with its own `maxUpRTT` estimate. Finally, the timer interval may be updated.

The informal specification does not say anything about whether we are interested in the round trip times of large packets, or small packets, or the time it takes for a small packet to go upstream plus the time it takes for a large packet to go downstream. The protocol presented here computes the round trip times of small packets, and can be changed by declaring the `getRTTRequest` and/or `getRTTResponse` packets to have sort `LargePacket`.

The repair servers perform many of the same transitions as the sender and the receivers in this protocol (only). We therefore find it convenient to define the superclasses `RTTsendable` and `RTTreceivable`, for, respectively, sender and repair servers, and repair servers and receivers. The class hierarchy and the packets involved in the protocol are given as follows:

```
(tomod AER-RTT is
including LINK .          *** Both of these include the crucial module
including RANDOM .        *** "TIMED-OO-SYSTEM"
including DEFAULT-VALUE-TIME-DOMAIN .
including COMMON-CLASSES .
including DETERMINISTIC-TICK-RULE .
including SHARED-MSGS .

*** All RTT objects are subclasses of RTT:
class RTT | sourceRTT : DefTime .

*** Sender and repair server:
class RTTsendable | maxDownRTT : DefTime,
                    maxDownRTTSetTime : Time,
                    maxRecentDownRTT : Time .
subclass RTTsendable < RTT .
subclass RTTsendable < Sendable .
subclass RTTsendable < Clock .

*** Repair server and receiver:
class RTTreceivable | resendInterval : Time,
                      maxUpRTT : DefTime,
                      myUpRTT : DefTime,
                      getRTTResendTimer : DefTime .
subclass RTTreceivable < RTT .
subclass RTTreceivable < Receivable .
subclass RTTreceivable < Clock .

*** Sender both in stand-alone and combined protocols:
class RTTsender .
subclass RTTsender < RTTsendable .

class RTTreceiver .
subclass RTTreceiver < RTTreceivable .

class RTTrepairserver .
subclass RTTrepairserver < RTTreceivable .
subclass RTTrepairserver < RTTsendable .

*** The objects for stand-alone protocol only:
class RTTsenderAlone .
subclass RTTsenderAlone < RTTsender .

class RTTreceivableAlone .
subclass RTTreceivableAlone < RTTreceivable .

class RTTreceiverAlone .
subclass RTTreceiverAlone < RTTreceiver .
subclass RTTreceiverAlone < RTTreceivableAlone .

class RTTrepairserverAlone .
subclass RTTrepairserverAlone < RTTrepairserver .
subclass RTTrepairserverAlone < RTTreceivableAlone .
```

```
msg getRTTRequest : Time DefTime -> Packet .
*** Usage: getRTTRequest(xmitTime, upRTT).

msg getRTTResponse : Time DefTime DefTime  -> Packet .
*** Usage: getRTTResponse(xmitTime, peerGroupRTT, globalRTT)
```

The sender initializes the protocol by multicasting an SPM packet. The handling of SPM packets is quite elaborate in the combined protocol. When only the RTT part is specified, such packets are only used to set the `repairserver` attributes and to start the protocol by initializing the `getRTTResendTimer` to a value between 0 and 30:

```
msg startRTT : Oid -> Msg .
rl [startRTT] :
   { startRTT(Q) < Q : RTTsenderAlone | children : OS >  CF }
   =>
   { < Q : RTTsenderAlone | >  multiSend(SPMPacket(0), Q, OS)  CF } .

rl [R2rs] :
    { (SPMPacket(0) from Q' to Q)  < Q'' : RandomNGen | seed : N >
    < Q : RTTrepairserverAlone | children : OS >   CF }
    =>
    { < Q'' : RandomNGen | seed : random(N) >
    < Q : RTTrepairserverAlone | repairserver : Q',
                                 getRTTResendTimer : random(N) % 31 >
      multiSend(SPMPacket(0), Q, OS)  CF } .

rl [R2rcv] :
    { (SPMPacket(0) from Q' to Q)  < Q'' : RandomNGen | seed : N >
    < Q : RTTreceiverAlone | >  CF }
    =>
    { < Q'' : RandomNGen | seed : random(N) >
    < Q : RTTreceiverAlone | repairserver : Q',
                             getRTTResendTimer : random(N) % 31 > CF } .
```

A node initiates a request/response round when its timer expires:

```
rl [R3] :
   { < Q : RTTreceivable | clock : R, repairserver : Q', resendInterval : R',
                           myUpRTT : DT, getRTTResendTimer : 0 > CF }
   =>
   { < Q : RTTreceivable | getRTTResendTimer : R'  >
   send(getRTTRequest(R, DT), Q, Q') CF } .
```

We split the treatment of `getRTTRequest` packets according to whether the $myUpRTT$ value, that is, the second parameter of the packet, and the value of the attribute `maxDownRTT` are `noTimeValue` or time values (we exploit that the variables `R`, `R'`, ... range over time values, and `DT`, `DT'`, ... range over the sort `DefTime` comprising time values and `noTimeValue`):

```
*** upRTT and maxDownRTT are both time values, and upRTT > maxDownRTT:
crl [R4a] :
    { (getRTTRequest(R, R') from Q to Q')
    < Q' : RTTsendable | clock : R'', sourceRTT : DT, maxDownRTT : R''' > CF }
    =>
    { < Q' : RTTsendable | maxDownRTT : R',
                           maxDownRTTSetTime : R'', maxRecentDownRTT : 0 >
    send(getRTTResponse(R, R', DT), Q', Q)   CF }
    if R''' lt R' .

*** upRTT is still a time value, but maxDownRTT is now noTimeValue:
rl [R4b] :
   { (getRTTRequest(R, R') from Q to Q')
   < Q' : RTTsendable | clock : R'', sourceRTT : DT, maxDownRTT : noTimeValue >
   CF }
   =>
   { < Q' : RTTsendable | maxDownRTT : R',
                          maxDownRTTSetTime : R'', maxRecentDownRTT : 0 >
    send(getRTTResponse(R, R', DT), Q', Q) CF } .

*** upRTT is noTimeValue:
rl [R4c] :
    { (getRTTRequest(R, noTimeValue) from Q to Q')
    < Q' : RTTsendable | clock : R', sourceRTT : DT, maxDownRTT : DT',
                       maxDownRTTSetTime : R'', maxRecentDownRTT : R''' > CF }
    =>
    { if (R'' plus 3500) lt R' then
        (< Q' : RTTsendable | maxDownRTT : R''',
                              maxDownRTTSetTime : R', maxRecentDownRTT : 0 >
        send(getRTTResponse(R, R''', DT), Q', Q) CF)
      else
        (< Q' : RTTsendable | >
    send(getRTTResponse(R, DT', DT), Q', Q) CF) fi } .

*** Both upRTT and maxDownRTT are time values, and upRTT <= maxDownRTT:
crl [R4d] :
    { (getRTTRequest(R, R') from Q to Q')
    < Q' : RTTsendable | clock : R'', sourceRTT : DT, maxDownRTT : R''',
                         maxDownRTTSetTime : R'''', maxRecentDownRTT : R''''' >
    CF }
    =>
    { if (R'''' plus 3500) lt R'' then
        (< Q' : RTTsendable | maxDownRTT : max(R', R'''''),
                              maxDownRTTSetTime : R'', maxRecentDownRTT : 0 >
         send(getRTTResponse(R, max(R', R'''''), DT), Q', Q) CF)
      else
        (< Q' : RTTsendable | maxRecentDownRTT : max(R',R''''') >
         send(getRTTResponse(R, R''', DT), Q', Q) CF) fi }
    if R' le R''' .
```

The treatment of `getRTTResponse` packets is also divided into two cases, depending on whether both the the **`maxUpRTT`** attribute value and the *peerGroupRTT* value (the second parameter) in the received packet are time values. The repair service (part of the combined) protocol needs to update some values whenever the maxUpRTT value changes. Instead of repeating the following quite complex rules and combining them with the corresponding rules in the RS protocol, we found it more convenient to check whether the object involved in the transition is a "combined" object, in which case the method/message newRTT is called/sent to the object itself and is treated by the repair service part:

```
crl [R5a] :
    { (getRTTResponse(R, DT, DT') from Q to Q')
    < Q' : RTTreceivable | clock : R', sourceRTT : DT'', resendInterval : R'',
                           maxUpRTT : DT''' > CF }
    =>
    { < Q' : RTTreceivable | sourceRTT : (if DT' =/= noTimeValue then
                                          DT' plus (R' monus R) else DT'' fi),
                              resendInterval : min(2 * (R' monus R), 3000),
                              maxUpRTT : (if DT =/= noTimeValue then
                                           max(R' monus R, DT) else (R' monus R) fi),
                              myUpRTT : R' monus R,
                              getRTTResendTimer : R' monus R >
    send(getRTTRequest(R', R' monus R), Q', Q)
    (if (< Q' : RTTreceivable | > : CombinedObject) then
     newRTT(if DT' =/= noTimeValue then  DT' plus (R' monus R) else DT'' fi,
            if DT =/= noTimeValue then max(R' monus R, DT)
            else (R' monus R) fi, Q') else none fi)  CF }
    if DT''' == noTimeValue or DT == noTimeValue .

*** Neither peerGroupRTT nor maxUpRTT is noTimeValue:
rl [R5b] :
    { (getRTTResponse(R, R', DT) from Q to Q')
    < Q' : RTTreceivable | clock : R'', sourceRTT : DT', resendInterval : R''',
                           maxUpRTT : R'''' >  CF }
    =>
    { < Q' : RTTreceivable | sourceRTT : (if DT =/= noTimeValue then
                                          DT plus (R'' monus R) else DT' fi),
                             resendInterval : min(2 * R''', 3000),
                             maxUpRTT : max(R'' monus R, R'),
                             myUpRTT : R'' monus R >
    (if (< Q' : RTTreceivable | > : CombinedObject) then
     newRTT(if DT =/= noTimeValue then DT plus (R'' monus R) else DT' fi,
            max(R'' monus R, R'), Q') else none fi)  CF } .
```

Finally, we need to specify how time acts on an RTT object in the stand-alone protocol, and how long time can advance before the timer expires:

```
eq delta(< Q : RTTsenderAlone | clock : R >, R') =
          < Q : RTTsenderAlone | clock : R plus R' > .
eq delta(< Q : RTTreceivableAlone | clock : R, getRTTResendTimer : DT >, R') =
     < Q : RTTreceivableAlone | clock : R plus R',
                         getRTTResendTimer : DT default-monus R' > .

eq mte(< Q : RTTsenderAlone | >) = INF .
eq mte(< Q : RTTreceivableAlone | getRTTResendTimer : R >) = R .
```

```
eq mte(< Q : RTTreceivableAlone | getRTTResendTimer : noTimeValue >) = INF .
endtom)
```

Analyzing the RTT protocol using Real-Time Maude

We define below an initial state, parameterized by the initial value of the seed to be used with calls to `random`, which is used to execute the RTT specification:

```
(tomod AER-RTT1 is
including AER-RTT .

ops RTTstate1 RTTstate2 : Nat -> System .
[ ... ]
eq RTTstate2(N) =
({ startRTT('a)
  < 'a : RTTsenderAlone | clock : 0, sourceRTT : 0, children : 'b 'c,
                          maxDownRTT : noTimeValue, maxDownRTTSetTime : 0,
                          maxRecentDownRTT : 0 >
  < 'b : RTTreceiverAlone | clock : 0, sourceRTT : noTimeValue,
                            repairserver : noOid, resendInterval : 200,
                            maxUpRTT : noTimeValue, myUpRTT : noTimeValue,
                            getRTTResendTimer : noTimeValue >
  < 'c : RTTrepairserverAlone | clock : 0, sourceRTT : noTimeValue,
                                children : 'd 'e,
                                maxDownRTT : noTimeValue, maxDownRTTSetTime : 0,
                                maxRecentDownRTT : 0, repairserver : noOid,
                                resendInterval : 200, maxUpRTT : noTimeValue,
                                myUpRTT : noTimeValue,
                                getRTTResendTimer : noTimeValue >
  < 'd : RTTrepairserverAlone | clock : 0, sourceRTT : noTimeValue,
                                children : 'f 'g,
                                maxDownRTT : noTimeValue, maxDownRTTSetTime : 0,
                                maxRecentDownRTT : 0, repairserver : noOid,
                                resendInterval : 200, maxUpRTT : noTimeValue,
                                myUpRTT : noTimeValue,
                                getRTTResendTimer : noTimeValue >
  < 'e : RTTreceiverAlone | clock : 0, sourceRTT : noTimeValue,
                            repairserver : noOid, resendInterval : 200,
                            maxUpRTT : noTimeValue, myUpRTT : noTimeValue,
                            getRTTResendTimer : noTimeValue >
  < 'f : RTTreceiverAlone | clock : 0, sourceRTT : noTimeValue,
                            repairserver : noOid, resendInterval : 200,
                            maxUpRTT : noTimeValue, myUpRTT : noTimeValue,
                            getRTTResendTimer : noTimeValue >
  < 'g : RTTreceiverAlone | clock : 0, sourceRTT : noTimeValue,
                            repairserver : noOid, resendInterval : 200,
                            maxUpRTT : noTimeValue, myUpRTT : noTimeValue,
                            getRTTResendTimer : noTimeValue >
  < 'random : RandomNGen | seed : N >
  < 'ab : Link | up : 'a, down : 'b, bound : 5, propDelay : 21,
                 linkSpeed : 1, downMsgs : nil, downSize : 0,
                 upMsgs : nil, upSize : 0 >
```

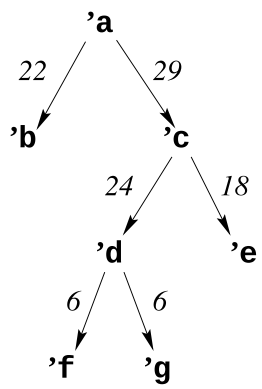

Figure 5.3: A multicast distribution tree decorated with the transmission delay of small packets in otherwise empty links.

```
  < 'ac : Link | up : 'a, down : 'c, bound : 21, propDelay : 28,
                 linkSpeed : 3, downMsgs : nil, downSize : 0,
                 upMsgs : nil, upSize : 0 >
  < 'cd : Link | up : 'c, down : 'd, bound : 9, propDelay : 23,
                 linkSpeed : 3 / 2, downMsgs : nil, downSize : 0,
                 upMsgs : nil, upSize : 0 >
  < 'ce : Link | up : 'c, down : 'e, bound : 4, propDelay : 17,
                 linkSpeed : 1, downMsgs : nil, downSize : 0,
                 upMsgs : nil, upSize : 0 >
  < 'df : Link | up : 'd, down : 'f, bound : 12, propDelay : 5,
                 linkSpeed : 10, downMsgs : nil, downSize : 0,
                 upMsgs : nil, upSize : 0 >
  < 'dg : Link | up : 'd, down : 'g, bound : 12, propDelay : 5,
                 linkSpeed : 10, downMsgs : nil, downSize : 0,
                 upMsgs : nil, upSize : 0 >}) .
endtom)
```

Fig. 5.3 shows shows the multicast distribution tree of `RTTstate2`, where the number associated with each link indicates how much time it takes for a small packet to travel through the link if the link was empty.

For example, in otherwise empty links, the round trip time to the source from the nodes `'c`, `'d`, and `'e` is, respectively, 58, 106, and 94, and the `maxUpRTT` of these nodes is, respectively, 58, 48, and 48. A quick prototyping to check whether these values are obtained can be achieved by using Maude's rewrite command[4]:

```
Maude> (rew- [1000] RTTstate2(1) .)
rewrites: 149925 in 3170ms cpu (6220ms real) (47294 rewrites/second)
rewrite in AER-RTT : RTTstate2 ( 1 ) .
result Error ( ClockedSystem ) : { < 'e : RTTreceiverAlone | clock : 31549,
    repairserver : 'c, sourceRTT : 94, getRTTResendTimer : 2982, myUpRTT :
    36, maxUpRTT : 48, resendInterval : 3000 >
```

[4]The more efficient standard Maude's rewrite command can be used here because the tick rule is deterministic and the system contains no lazy rules.

```
  < 'd : RTTrepairserverAlone | clock : 31549, children : 'f 'g,
    repairserver : 'c, sourceRTT : 106, maxRecentDownRTT : 12,
    maxDownRTTSetTime : 30089, maxDownRTT : 12, getRTTResendTimer : 1402,
    myUpRTT : 48, maxUpRTT : 48, resendInterval : 3000 >
  < 'random : RandomNGen | seed : 868 >
  < 'a : RTTsenderAlone | clock : 31549, children : 'b 'c, sourceRTT : 0,
    maxRecentDownRTT : 58, maxDownRTTSetTime : 27546, maxDownRTT : 58 >
  < 'b : RTTreceiverAlone | clock : 31549, repairserver : 'a,
    sourceRTT : 44, getRTTResendTimer : 956, myUpRTT : 44, maxUpRTT : 58,
    resendInterval : 3000 >
  < 'c : RTTrepairserverAlone | clock : 31549, children : 'e 'd,
    repairserver : 'a, sourceRTT : 58, maxRecentDownRTT : 0,
    maxDownRTTSetTime : 31549, maxDownRTT : 48, getRTTResendTimer : 1968,
    myUpRTT : 58, maxUpRTT : 58, resendInterval : 3000 >
  < 'f : RTTreceiverAlone | clock : 31549, repairserver : 'd,
    sourceRTT : 118, getRTTResendTimer : 1550, myUpRTT : 12,
    maxUpRTT : 12, resendInterval : 3000 >
  < 'ac : Link | upSize : 0, upMsgs : nil, downSize : 0, downMsgs : nil,
    linkSpeed : 3, propDelay : 28, bound : 21, up : 'a, down : 'c >
  < 'ab : Link | upSize : 0, upMsgs : nil, downSize : 0, downMsgs : nil,
    linkSpeed : 1, propDelay : 21, bound : 5, up : 'a, down : 'b >
  < 'g : RTTreceiverAlone | clock : 31549, repairserver : 'd,
    sourceRTT : 118, getRTTResendTimer : 1534, myUpRTT : 12,
    maxUpRTT : 12, resendInterval : 3000 >
  < 'cd : Link | upSize : 0, upMsgs : nil, downSize : 0, downMsgs : nil,
    linkSpeed : 1, propDelay : 23, bound : 9, up : 'c, down : 'd >
  < 'ce : Link | upSize : 0, upMsgs : nil, downSize : 1,
    downMsgs : dly(getRTTResponse(31531, 48, 58), 18), linkSpeed : 1,
    propDelay : 17, bound : 4, up : 'c, down : 'e >
  < 'df : Link | upSize : 0, upMsgs : nil, downSize : 0, downMsgs : nil,
    linkSpeed : 10, propDelay : 5, bound : 12, up : 'd, down : 'f >
  < 'dg : Link | upSize : 0, upMsgs : nil, downSize : 0, downMsgs : nil,
    linkSpeed : 10, propDelay : 5, bound : 12, up : 'd, down : 'g > }
  in time 31549
```

These values seem to be correct. Since we later assume that a nominee is found within time 5000, and because we need the RTT values for finding the nominee receiver, it can be interesting to check whether these values can be found within time 5000[5]:

```
Maude> (trew [0] RTTstate2(1) mode m dti 10000 in time le 5000 .)

rewrites: 7502161 in 135540ms cpu (157290ms real) (55350 rewrites/second)
[ ... ]
Result: { < 'e : RTTreceiverAlone | clock : 4567, repairserver : 'c,
    sourceRTT : 94, getRTTResendTimer : 2964, myUpRTT : 36, maxUpRTT : 48,
    resendInterval : 3000 >
  < 'd : RTTrepairserverAlone | clock : 4567,
    children : 'f 'g, repairserver : 'c, sourceRTT : 106, maxRecentDownRTT :
    12, maxDownRTTSetTime : 77, maxDownRTT : 12, getRTTResendTimer : 1384,
    myUpRTT : 48, maxUpRTT : 48, resendInterval : 3000 >
```

[5]The output of Real-Time Maude executions will be manually tabulated for readability purposes, and parts of the output omitted in the exposition will be replaced by '[...]'.

```
  < 'c : RTTrepairserverAlone | clock : 4567, children : 'e 'd,
    repairserver : 'a, sourceRTT : 58, maxRecentDownRTT : 0,
    maxDownRTTSetTime : 4549, maxDownRTT : 48, getRTTResendTimer : 1950,
    myUpRTT : 58, maxUpRTT : 58, resendInterval : 3000 >
  [ ... ] } in time 4567 .
```

These values are indeed the values we are looking for. To gain further assurance of the correctness of the specification, we would like to explore not just *one* behavior, arbitrarily chosen by Real-Time Maude's default interpreter, but *all* possible behaviors—relative to certain conditions—starting from the initial state. The main property the stand-alone RTT protocol should satisfy is that, as long as no more than one packet travels simultaneously in the same direction in the same link, then:

- each rewrite path will reach a state with the desired `sourceRTT` and `maxUpRTT` values within given time and depth limits (reachability);
- once these desired values have been found, they will not change within the given time limit (stability).

These two properties can be expressed by the formula

$$\texttt{matching anyPattern } \mathbf{UStable}_{\leq timeLimit} \texttt{ matching pattern } \mathit{desiredValues}$$

in the temporal logic defined in Section 3.2, if *desiredValues* is a pattern containing the desired values of the attributes of the objects.

The following module `varRTT` introduces the variables needed to express the pattern of the desired states:

```
(tomod varRTT is
  protecting AER-RTT1 .
  vars ATTS1 ATTS2 ATTS3 : AttributeSet .
  var CF : Configuration .
endtom)
```

The pattern satisfied by the desired states is

```
{< 'c : RTTrepairserverAlone | sourceRTT : 58, maxUpRTT : 58, ATTS1 >
 < 'd : RTTrepairserverAlone | sourceRTT : 106, maxUpRTT : 48, ATTS2 >
 < 'e : RTTreceiverAlone | sourceRTT : 94, maxUpRTT : 48, ATTS3 >  CF }
```

This pattern is matched by all states with correct values of the `sourceRTT` and `maxUpRTT` values of the objects `'c`, `'d`, and `'e`.

The Real-Time Maude search expression which returns all terms reachable from `RTTstate2(1)` which do *not* satisfy the desired reachability and stability property (in rewrite paths with total simulated time elapse less than or equal to 400) can be given as follows:

```
Maude> (strat find [200] 1 not satisfying matching anyPattern UntilStable
          matching pattern
          ({< 'c : RTTrepairserverAlone | sourceRTT : 58, maxUpRTT : 58, ATTS1 >
            < 'd : RTTrepairserverAlone | sourceRTT : 106, maxUpRTT : 48, ATTS2 >
            < 'e : RTTreceiverAlone | sourceRTT : 94, maxUpRTT : 48, ATTS3 >
            CF })
          from term (RTTstate2(1)) mode gm dti 1 within time le 400 .)
```

Unfortunately, the current version of the tool executes this command too slowly for the large states involved, since it performs all kinds of checks on lazy rules, variables introduced in the righthand sides of tick rules, etc. However, using Real-Time Maude's strategy library, we can easily define an optimized strategy corresponding to the search described above. (See Chapter 4 for an explanation of the `TIMED-META-LEVEL` features involved.)

```
(fmod AER-STRATEGIES is
including TIMED-META-LEVEL .

vars T T' : Term .        vars TS TS' TS'' TS''' : TermSet .
var Q : Qid .             vars N N' : MachineInt .
var M : Module .          var QS : QidSet .
var DT : DefTerm .        var DP : DefPattern .

eq termSet(T, T) = T .
```

By including `TIMED-META-LEVEL`, we have access to Real-Time Maude's strategy library. We do not yet add the parameter `varRTT` to the import declaration, since the module is also used for analyzing other protocols.

The function `applyQ` uses the fact that all rewrite rules are global and that the tick rule is deterministic, to define a reasonably efficient function giving the result of applying a rule to a term of the form `{` t `} in time` r, or of the form `{` t `}`:

```
op applyQ : Module Term Qid MachineInt -> Term .

eq applyQ(M, '`{_`}[T], Q, N) =
      extrTerm(meta-apply(M, '`{_`}[T], Q, none, N)) .

eq applyQ(M, '_in`time_['`{_`}[T], T'], Q, N) =
      if extrTerm(meta-apply(M, '`{_`}[T], Q, none, N)) == error* then error*
      else meta-reduce(M, '_in`time_[extrTerm(meta-apply(M, '`{_`}[T], Q,
                                             none, N)), T']) fi .
```

This function can be extended to find all terms reachable in one rewrite step from a given term, using the knowledge that there are no lazy rules in the system:

```
op applyQAll : Module Term -> TermSet .
op applyQAll : Module Term QidSet -> TermSet .
eq applyQAll(M, T) = applyQAll(M, T, allRuleLabels(M)) .
eq applyQAll(M, T, (Q ; QS)) = termSet(applyQAll(M, T, Q, 0),
                                       applyQAll(M, T, QS)) .
```

```
eq applyQAll(M, T, none) = emptyTermSet .

op applyQAll : Module Term Qid MachineInt -> TermSet .
eq applyQAll(M, T, Q, N) =
      if applyQ(M, T, Q, N) =/= error*
      then termSet(applyQ(M, T, Q, N), applyQAll(M, T, Q, N + 1))
      else emptyTermSet fi .
```

Observe the use of the `TIMED-META-LEVEL` functions `extrTerm` and `allRuleLabels` above.

The search property is given by the function `ustable`, where

$$\texttt{ustable}(mod, t_0, n, timeLimit, pattern)$$

gives the set of terms representing paths invalidating the reachability-and-stability property described above w.r.t. the *pattern*, starting from initial term t_0, having maximal bound n on the number of rewrites in the path, and time limit *timeLimit* on the total time elapse in the path. To further enhance efficiency, the following function does not return *all* bad states, but only the first state(s) found which invalidate the property:

```
op ustable : Module Term MachineInt DefTerm DefPattern -> TermSet .
*** Usage: ustable(M, termToAnalyze, NoOfRewrites, TimeLimit, desiredPattern)
```

The search is performed as a breadth-first-search by the function

```
op ustable : Module MachineInt DefTerm DefPattern TermSet TermSet
             TermSet TermSet MachineInt -> TermSet .
*** Usage: ustable(M, MaxNoOfRewrites, TimeLimit, desiredPattern,
***                nonPatternTermsToCheckInThisRound, nonPatternTermsToCheck-
***                InNextRound, patternTermsToCheckInThisRound,
***                patternTermsToCheckNextRound, currentRound)
```

where, in $\texttt{ustable}(mod, n, timeLimit, pattern, ts, ts', ts'', ts''', currentRound)$, n is the limit on the length of the rewrite paths (where 0 stands for "unlimited"), *timeLimit* is the given limit (where `noTerm` stands for the absence of a limit) on the total (simulated) time elapsed in a path, n' is the current "round", *ts* is the set of terms reachable from the initial state in $n' - 1$ steps where no state in the path has matched the pattern, and are to be rewritten in the current round, ts' denotes the terms reached in the current round which do not match the pattern, ts'' denotes the set of terms matching the pattern which are still to be rewritten in this round, and ts''' are the terms matching the pattern reached in this round.

The following equation initializes the search function. If the initial state already matches the given pattern, it should be placed in the ts'' place for terms satisfying *pattern* to be checked in the current round:

```
eq ustable(M, T, N, DT, DP) =
     if matchesPattern(M, DP, meta-reduce(M, T)) then
        ustable(M, N, DT, DP, emptyTermSet, emptyTermSet, meta-reduce(M, T),
                emptyTermSet, 1)
     else ustable(M, N, DT, DP, meta-reduce(M, T), emptyTermSet, emptyTermSet,
                  emptyTermSet, 1)
     fi .
```

In the following equation , we analyze all states reachable in one step from a term T not satisfying the pattern. If T cannot be further rewritten, it represents a path which invalidates the desired property, since it does not lead to a state satisfying the pattern, and T is returned. If T has a successor state beyond the time limit, then there is a rewrite path where the desired state is not reached within the time limit, and T is returned in this case as well. If neither of these cases apply, the set of terms reachable from T in one step is added to the set of terms to be checked in the next round, according to whether they match the pattern:

```
eq ustable(M, N, DT, DP, termSet(T, TS), TS', TS'', TS''', N') =
      if (applyQAll(M, T) == emptyTermSet or
          strictlyTooOld(M, DT, applyQAll(M, T)) =/= emptyTermSet) then T
      else ustable(M, N, DT, DP, TS,
                   termSet(TS', removeMatches(M, applyQAll(M, T), DP)), TS'',
                   termSet(TS''', keepMatches(M, applyQAll(M, T), DP)), N')
      fi .
```

Next, we analyze the states reachable from a state T matching the pattern. If T has successor states within the time limit which do not match the pattern, then the stability-part of the property is invalidated by these successor states, which are returned by the function. Otherwise, the reachable states within the time limit are added to the terms to check in the next round:

```
eq ustable(M, N, DT, DP, TS, TS', termSet(T, TS''), TS''', N') =
      if notTooOld(M, DT, removeMatches(M, applyQAll(M, T), DP)) =/=
         emptyTermSet
      then notTooOld(M, DT, removeMatches(M, applyQAll(M, T), DP))
      else ustable(M, N, DT, DP, TS, TS', TS'', termSet(TS''',
                    notTooOld(M, DT, keepMatches(M, applyQAll(M, T), DP))), N')
      fi .
```

A round ends when there are no more terms to treat in the current round. If the number of rounds completed equals the depth limit, then the states which have not reached the pattern are returned. If there are no terms to check in the next round, then there are no counterexamples and the property is proved for the given initial state, and time and depth limit. Otherwise a new round is started:

```
eq ustable(M, N, DT, DP, emptyTermSet, TS, emptyTermSet, TS', N') =
     if ((N =/= 0 and N == N') or            *** Enough rounds
         (TS == emptyTermSet and TS' == emptyTermSet)) then TS
     else ustable(M, N, DT, DP, TS, emptyTermSet, TS', emptyTermSet, N' + 1)
     fi .
endfm)
```

We were able to develop this application-specific strategy very quickly by making heavy use of TIMED-META-LEVEL's data types for patterns and sets of terms, and functions such as removeMatches, keepMatches, notTooOld, etc., on these data types.

Using the up function to get the meta-representation of a term, we can check whether our desired property holds in all rewrite paths with total time elapse less than or equal to 400 starting from state RTTstate2(1):

```
Maude> (fmod RTT-STRATEGIES is
          protecting AER-STRATEGIES .
          including META-LEVEL[varRTT] .
        endfm)
rewrites: 4084 in 30ms cpu (40ms real) (136133 rewrites/second)

Introduced module: RTT-STRATEGIES
Maude> (red ustable(varRTT, 'RTTstate2['1'Nat], 0, {'400}'Nat,
         up(varRTT,
           {< 'c : RTTrepairserverAlone | sourceRTT : 58, maxUpRTT : 58, ATTS1 >
           < 'd : RTTrepairserverAlone | sourceRTT : 106, maxUpRTT : 48, ATTS2 >
           < 'e : RTTreceiverAlone | sourceRTT : 94, maxUpRTT : 48, ATTS3 >
           CF })) .)
rewrites: 31752555 in 2207800ms cpu (2208080ms real) (14381 rewrites/second)
[ ... ]
result TermSet : emptyTermSet
```

No path *not* satisfying the desired property was found, increasing our confidence in the correctness of the protocol. To gain further assurance, we could analyze executions starting with other states, either by just changing the seed for the random number generator, or by changing the communication topology.

5.2.7 The NOM Protocol

The goal of the NOM part of the protocol is to find the receiver with the highest value of $rtt * \sqrt{lpe}$, where rtt is the round trip time from the receiver to the sender, and lpe is the loss probability estimate of the receiver, and to let the receiver know that it is the chosen *nominee* receiver.

The sender receives messages of type `csmPacket` from its children, with each such packet containing the lpe and rtt values of the receiver with the most congested path in the subtree with the child as root. The sender compares these received values with the values of its current nominee, stored in the attributes `csmNominee`, `csmLPE`, and `csmRTT`. If the received values indicates a more congested receiver path (or, if no nominee has yet been found, or more than 17 seconds have elapsed since the nominee-values were last updated), then the values are updated, and `NAMPacket` messages are sent to the old and new nominee receiver if the nominee has changed. The `NAMPacket` is resent to the nominee at regular intervals, namely, whenever the `NAMTimer` expires.

The sender is declared as follows:

```
(tomod NCA-NOM is
including RANDOM .
including LINK .
including WINDOW .
including COMMON-CLASSES .
including DETERMINISTIC-TICK-RULE .
including SHARED-MSGS .
including ENVIRONMENT .

class NOMsender |
      NAMTimer : DefTime,
```

```
      csmNominee : DefOid,
      csmLPE : DefNat,
      csmRTT : DefTime,
      csmSetTime : Time .
subclass NOMsender < Sender .
subclass NOMsender < Clock .

class NOMsenderAlone .
subclass NOMsenderAlone < NOMsender .
```

The message startNOM is used to initiate the protocol:

```
vars DN DN' DN'' DN''' : DefNat .

msg startNOM : Oid -> Msg .
msg csmPacket : DefNat DefTime DefOid -> Packet .
*** Usage: csmPacket(lpe, RTT, addrLeafNode)

rl [initNOM] :
   { startNOM(Q) < Q : NOMsenderAlone | children : OS >  CF }
   =>
   { < Q : NOMsenderAlone | > multiSend(SPMPacket(0), Q, OS)  CF } .
```

The function newNomValues take the received and the stored nominee values, as well as the current time and the last time the nominee-values were updated, and returns true iff the nominee values should be updated. Instead of comparing $rtt * \sqrt{lpe}$-values, we compare their squares $rtt^2 * lpe$. Furthermore, the NAMPackets to the new and old nominee receivers should be unicast without going through the links, and could in a first abstraction be seen as having no delay. Therefore, the NAMPackets already have the ready-to-read form:

```
op newNomValues : DefNat DefTime DefOid DefOid DefNat DefTime Time Time -> Bool .
*** Usage: newNomValues(lpe, RTT, addr, csmNominee/csmAddress, csmLPE, csmRTT,
***                     currentTime, csmSetTime) .

eq newNomValues(DN, DT, DQ', DQ, DN', DT', R, R') =
       if DN == noNat or DN' == noNat then
          (DQ == noOid or DQ == DQ' or
           (DN == noNat and DN' == noNat and DT' default-lt DT) or
           ((17000 plus R') lt R))
       else
          (DQ == noOid or DQ == DQ' or ((0 le DN) and (DN le 1000) and
                                        (0 le DN') and (DN' le 1000) and
                                        (gSquare(DN', DT') lt gSquare(DN, DT)))
           or ((17000 plus R') lt R)) fi .

op gSquare : Nat DefTime -> Nat .
eq gSquare(R, R') = R * R' * R' .
eq gSquare(R, noTimeValue) = 0 .

rl [D2D3] :
   { (csmPacket(DN, DT, DQ') from Q to Q')
```

```
   < Q' : NOMsenderAlone | clock : R, csmNominee : DQ, csmLPE : DN',
                           csmRTT : DT', csmSetTime : R', NAMTimer : DT'' > CF }
   =>
   { if newNomValues(DN, DT, DQ', DQ, DN', DT', R, R') then
       (< Q' : NOMsenderAlone | csmNominee : DQ', csmLPE : DN,
                                csmRTT : DT, csmSetTime : R,
                                NAMTimer : (if DQ =/= DQ' then 7000 else DT'' fi) >
        (if DQ =/= DQ' and DQ' =/= noOid       *** New nominee
         then (NAMPacket(true) from Q' to DQ') else none fi)
        (if DQ =/= DQ' and DQ =/= noOid        *** Previous nominee
         then (NAMPacket(false) from Q' to DQ) else none fi)
        CF)
   else  (< Q' : NOMsenderAlone | > CF) fi } .
```

A `NAMPacket` should be sent to the nominee every seven seconds:

```
rl [D4] :
   { < Q : NOMsender | csmNominee : DQ, NAMTimer : 0 >  CF }
   =>
   { if DQ =/= noOid then
       (< Q : NOMsender | NAMTimer : 7000 > (NAMPacket(true) from Q to DQ) CF)
     else (< Q : NOMsender | NAMTimer : noTimeValue > CF) fi } .
```

A repair server stores the values of the receiver with the most congested path in its subtree. Whenever there is a change in this worst receiver status, the new values are sent upstream to its repair server. Each time the `csmTimer` expires (rule `F4`), the repair server sends its current nominee values to its repair server, unless more than 17 seconds have elapsed since the values were last updated, in which case all the values are set to their default values:

```
class NOMrepairserver |
      csmLPE : DefNat,
      csmRTT : DefTime,
      csmAddress : DefOid,
      csmSetTime : Time,
      csmTimer : DefTime .
subclass NOMrepairserver < Repairserver .
subclass NOMrepairserver < Clock .

class NOMrepairserverAlone .
subclass NOMrepairserverAlone < NOMrepairserver .

*** Initialize csm timer and repairserver attributes:
rl [F1] :
   { (SPMPacket(0) from Q' to Q)
   < Q : NOMrepairserverAlone | children : OS > CF }
   =>
   { < Q : NOMrepairserverAlone | repairserver : Q', csmTimer : 7000 >
   multiSend(SPMPacket(0), Q, OS) CF } .

rl [F2F3] :
   { (csmPacket(DN, DT, DQ') from Q to Q')
```

```
   < Q' : NOMrepairserver | clock : R, repairserver : Q''', csmAddress : DQ,
                            csmLPE : DN', csmRTT : DT', csmSetTime : R' >  CF }
   =>
   { if newNomValues(DN, DT, DQ', DQ, DN', DT', R, R') then
       (< Q' : NOMrepairserver | csmAddress : DQ', csmLPE : DN, csmRTT : DT,
                                 csmSetTime : R, csmTimer : 7000 >
       (if DQ' =/= noOid then send(csmPacket(DN, DT, DQ'), Q', Q''')
        else none fi)  CF)
     else (< Q' : NOMrepairserver | > CF) fi } .

rl [F4] :
   { < Q : NOMrepairserver | clock : R, repairserver : Q', csmLPE : DN,
                             csmRTT : DT, csmAddress : DQ, csmSetTime : R',
                             csmTimer : 0 > CF }
   =>
   { if (R' plus 17000) lt R then
       (< Q : NOMrepairserver | csmLPE : noNat, csmRTT : 0, csmAddress : noOid,
                                csmSetTime : R, csmTimer : 7000 > CF)
     else (< Q : NOMrepairserver | csmTimer : 7000 >
           (if DQ =/= noOid then send(csmPacket(DN, DT, DQ), Q, Q')
            else none fi) CF) fi } .
```

Each time the `csmTimer` of a receiver expires, the receiver should send its current *lpe* and *rtt* values in a `csmPacket` upstream. (In the stand-alone protocol, we set the `sourceRTT` attribute in the initial state, and leave it unchanged.) The receiver updates a *sliding window* with the sequence number of the data packets it receives to estimate its loss probability. The following declarations define the interface of the sliding window module `WINDOW` given in [Ölv00]:

```
*** Needs default numbers since we may have to "divide" by 0!
sort DefNat .
subsort Nat < DefNat .
op noNat : -> DefNat .

sort Window .
op initWindow : NzNat -> Window .   *** Empty window with given max size
op size : Window -> Nat .
op add : NzNat Window -> Window .   *** adds a sequence number to a window
op windowLPE : Window -> DefNat .   *** estimates lpe in per mille
```

In the stand-alone NOM protocol we use a simpler data packet, and define the receiver as follows:

```
msg dataPacket : NzNat Time -> LargePacket .
*** Usage: dataPacket(seqNo, timeStamp).

class NOMreceiver |
      isNominee : Bool,
      sourceRTT : DefTime,
      msgWindow : Window,   *** Window for storing message numbers
      csmTimer : DefTime .
subclass NOMreceiver < Receiver .
```

```
class NOMreceiverAlone .
subclass NOMreceiverAlone < NOMreceiver .

var W : Window .

*** Timer should be started with an initial value randomly chosen between
*** 0 and 5000:
rl [initNOMreceiver] :
   { (SPMPacket(0) from Q' to Q) < Q'' : RandomNGen | seed : N >
   < Q : NOMreceiverAlone | > CF }
   =>
   { < Q'' : RandomNGen | seed : random(N) >
   < Q : NOMreceiverAlone | repairserver : Q',
                            csmTimer : random(N) % (5001) > CF } .

rl [E2] :
   { (dataPacket(NZN, R) from Q to Q')
   < Q' : NOMreceiverAlone | msgWindow : W > CF }
   =>
   { < Q' : NOMreceiverAlone | msgWindow : add(NZN, W) > CF } .
```

A receiver sends a `csmPacket` with its current `sourceRTT` and `lpe` values to its repair server when the `csmTimer` expires. According to the original protocol, the *lpe* estimate is considered unreliable if the size of the window is less than 150, so the default value `noNat` is sent instead. However, for easier prototyping, we have changed the number 150 to 2 below.

```
rl [E3] :
   { < Q : NOMreceiver | csmTimer : 0, msgWindow : W, repairserver : Q',
                         sourceRTT : DT > CF }
   =>
   { < Q : NOMreceiver | csmTimer : 5000 >
   send(csmPacket(if (size(W) < 2) then noNat else windowLPE(W) fi,
                  DT, Q), Q, Q') CF } .
```

The `isNominee` attribute is updated according to received status in the `NAMPacket`:

```
rl [E4] :
   { (NAMPacket(Z) from Q' to Q) < Q : NOMreceiverAlone | > CF }
   =>
   { < Q : NOMreceiverAlone | isNominee : Z > CF } .
```

Time acts on the NOM objects as in the RTT protocol, namely, by updating the clock and timer attributes, and by letting `mte` ensure that time advance stops when a timer expires:

```
eq delta(< Q : NOMsenderAlone | clock : R', NAMTimer : DT >, R) =
    < Q : NOMsenderAlone | clock : R' plus R, NAMTimer : DT default-monus R > .
eq delta(< Q : NOMrepairserverAlone | clock : R', csmTimer : DT >, R) =
      < Q : NOMrepairserverAlone | clock : R' plus R,
                                   csmTimer : DT default-monus R > .
eq delta(< Q : NOMreceiverAlone | csmTimer : DT >, R) =
```

```
      < Q : NOMreceiverAlone | csmTimer : DT default-monus R > .

eq mte(< Q : NOMsenderAlone | NAMTimer : DT >) =
      if DT =/= noTimeValue then DT else INF fi .
eq mte(< Q : NOMrepairserverAlone | csmTimer : DT >) =
      if DT == noTimeValue then INF else DT fi .
eq mte(< Q : NOMreceiverAlone | csmTimer : DT >) =
      if DT == noTimeValue then INF else DT fi .
endtom)
```

Analyzing the NOM Protocol

The module NCA-NOM1 extends the module NCA-NOM with the declaration of some initial states, having the same topology as the initial states used to analyze the RTT protocol. Since data packets are not generated in the stand-alone protocol, they are entered into the environment object in the initial state:

```
(tomod NCA-NOM1 is
protecting NCA-NOM .
[ ... ]
op NOMstate2 : Nat -> System .
eq NOMstate2(N) =
({ startNOM('a)
  < 'a : NOMsenderAlone | clock : 0, children : 'b 'c,  NAMTimer : noTimeValue,
                          csmLPE : noNat, csmRTT : 0, csmSetTime : 0,
                          csmNominee : noOid >
  < 'b : NOMreceiverAlone | sourceRTT : 44,  repairserver : noOid, isNominee : false,
                            msgWindow : initWindow(4), csmTimer : noTimeValue >
  < 'c : NOMrepairserverAlone | clock : 0, children : 'd 'e, repairserver : noOid,
                                csmLPE : noNat,  csmRTT : 0, csmAddress : noOid,
                                csmSetTime : 0, csmTimer : noTimeValue >
  < 'd : NOMrepairserverAlone | clock : 0, children : 'f 'g, repairserver : noOid,
                                csmLPE : noNat,  csmRTT : 0, csmAddress : noOid,
                                csmSetTime : 0, csmTimer : noTimeValue >
  < 'e : NOMreceiverAlone | sourceRTT : 94,  repairserver : noOid, isNominee : false,
                            msgWindow : initWindow(4), csmTimer : noTimeValue >
  < 'f : NOMreceiverAlone | sourceRTT : 118, repairserver : noOid, isNominee : false,
                            msgWindow : initWindow(4), csmTimer : noTimeValue >
  < 'g : NOMreceiverAlone | sourceRTT : 118, repairserver : noOid, isNominee : false,
                            msgWindow : initWindow(4), csmTimer : noTimeValue >
  < 'random : RandomNGen | seed : N >
  < 'ab : Link | up : 'a, down : 'b, bound : 5, propDelay : 21,
                 linkSpeed : 1, downMsgs : nil, downSize : 0,
                 upMsgs : nil, upSize : 0 >
  < 'ac : Link | up : 'a, down : 'c, bound : 21, propDelay : 28,
                 linkSpeed : 3, downMsgs : nil, downSize : 0,
                 upMsgs : nil, upSize : 0 >
  < 'cd : Link | up : 'c, down : 'd, bound : 9, propDelay : 23,
                 linkSpeed : 3 / 2, downMsgs : nil, downSize : 0,
                 upMsgs : nil, upSize : 0 >
  < 'ce : Link | up : 'c, down : 'e, bound : 4, propDelay : 17,
                 linkSpeed : 1, downMsgs : nil, downSize : 0,
```

```
                upMsgs : nil, upSize : 0 >
  < 'df : Link | up : 'd, down : 'f, bound : 12, propDelay : 5,
                linkSpeed : 10, downMsgs : nil, downSize : 0,
                upMsgs : nil, upSize : 0 >
  < 'dg : Link | up : 'd, down : 'g, bound : 12, propDelay : 5,
                linkSpeed : 10, downMsgs : nil, downSize : 0,
                upMsgs : nil, upSize : 0 >
  < 'env : Env | msgsFromEnv :
                  dly(dataPacket(1, 1) from 'a to 'b, 5001)
                  dly(dataPacket(4, 1) from 'a to 'b, 5004)
                  dly(dataPacket(2, 1) from 'd to 'f, 14996)
                  dly(dataPacket(3, 1) from 'd to 'f, 14999)
                  dly(dataPacket(4, 1) from 'd to 'f, 15031)
                  dly(dataPacket(4, 1) from 'd to 'g, 5002)
                  dly(dataPacket(5, 1) from 'd to 'g, 15000)
                  dly(dataPacket(6, 1) from 'd to 'g, 15004)
                  dly(dataPacket(1, 1) from 'c to 'e, 5003)
                  dly(dataPacket(2, 1) from 'c to 'e, 5018)
                  dly(dataPacket(16, 1) from 'c to 'e, 15001)  > }) .
endtom)
```

There are no *lpe* values before time 5000 when starting from this initial state, since no data packets have been received, and after time 5018 the receiver with the largest *rtt* value, in this case `'f` or `'g`, should be the chosen nominee:

```
Maude> (trew [0] NOMstate2(1) mode m dti 1000 in time le 5000 .)
[ ... ]
Result: { < 'a : NOMsenderAlone | clock : 4764, children : 'b 'c, csmLPE : noNat,
    csmNominee : 'f, csmSetTime : 490, csmRTT : 118, NAMTimer : 2726 >
  < 'f : NOMreceiverAlone | repairserver : 'd, isNominee : true,
    csmTimer : 667, msgWindow : window (nil, 4, 0), sourceRTT : 118 >
 [ ... ] } in time 4764 .
```

After time 5018, and before time 14996, the receiver `'f` still has not received a packet, and will therefore remain the nominee:

```
Maude>  (trew [0] NOMstate2(1) mode m dti 1000 in time le 14500 .)
[ ... ]
Result: { < 'a : NOMsenderAlone | csmLPE : noNat, csmNominee : 'f, [ ... ] >
  < 'f : NOMreceiverAlone | isNominee : true, msgWindow : window(nil, 4, 0), [ ... ] >
  [ ... ] } in time 14490 .
```

After the messages received around time 15000, `'f` has a loss rate of only 25%, so node `'e`, with its loss rate of 75%, should be the nominee:

```
Maude> (trew [0] NOMstate2(1) mode m dti 1000 in time le 20000 .)
[ ... ]
Result: { < 'e : NOMreceiverAlone | isNominee : true,
     msgWindow : window(16, 4, 1), sourceRTT : 94, [ ... ] >
  < 'a : NOMsenderAlone | csmNominee : 'e, csmLPE : 750, [ ... ] >
  < 'f : NOMreceiverAlone | isNominee : false, msgWindow : window(2 3 4, 4, 3), [ ... ] >
  [ ... ] } in time 19764 .
```

The executions above using Real-Time Maude's default strategy gave the desired results. To further analyze the protocol, we should again explore all possible behaviors—relative to appropriate requirements—from a given initial state.

As will be shown later, it is crucial for the combined protocol that a nominee receiver must exist before the sender sends its first data packet. The property that the system always has at least one receiver which is the nominee and knows it, is not stable, because the system may be without a receiver which knows it is the nominee after a `NAMPacket(false)` has been treated but before the corresponding `NAMPacket(true)` has been treated. Therefore, the `ustable` function used to analyze the RTT protocol cannot be used to analyze the NOM protocol. Instead, we have defined in the module `AER-STRATEGIES` given earlier the function

```
op bfsLive : Module Term MachineInt DefTerm DefPattern -> TermSet .
*** Usage: bfsLive(Module, TermToAnalyze, MaxNoOfRewrites, TimeLimit, Pattern).
```

for (efficiently) finding rewrite paths where a given pattern is *not* reached within a certain time or number of rewrites. That is, the function `bfsLive` model check liveness properties of the form

$$\Diamond_{\leq timeLimit}\ \texttt{matching pattern}\ pattern$$

by exploring *all* rewrite paths—up to a given limit on the number of rewrite steps in the path—from the initial state. It returns some state(s) representing paths which invalidate the property if it does not hold, and `emptyTermSet` if the property holds.

As usual, a limit 0 on the length of the rewrite path means that there is no such limit, and a time limit `noTerm` means that the total time elapse allowed in a path is unbounded. The breadth-first search is performed by the following function

```
op bfsLive : Module MachineInt DefTerm DefPattern
             TermSet TermSet MachineInt -> TermSet .
*** Usage: bfsLive(Module, NoOfRewrites, TimeLimit, GoodPattern,
***                TermsToCheckInRound,
***                TermsToCheckInNextRound, currentRoundNo).
```

where the fifth and sixth arguments denote the terms which should still be rewritten in the current round and in the next round, respectively. No term in either of these sets matches the pattern or has an associated time elapse beyond the time limit.

The following equation initiates the search by letting the initial state be the state to analyze in round 1, unless it satisfies the pattern:

```
eq bfsLive(M, T, N, DT, DP) =
      bfsLive(M, N, DT, DP, removeMatches(M, meta-reduce(M, T), DP),
              emptyTermSet, 1) .
```

If a term `T` to be analyzed in the current round does not have a successor state, or has a successor state with a total time elapse beyond the time limit, then the term `T` represents rewrite path(s) where the pattern is not reached (within the time limit). Otherwise, the successor states not matching the pattern are added to the states to be checked in the next round.

```
eq bfsLive(M, N, DT, DP, termSet(T, TS), TS', N') =
      if (applyQAll(M, T) == emptyTermSet or
          strictlyTooOld(M, DT, applyQAll(M, T)) =/= emptyTermSet) then T
      else bfsLive(M, N, DT, DP, TS,
                   termSet(removeMatches(M, applyQAll(M, T), DP), TS'), N')
      fi .
```

Finally, when there are no terms to check in the current round, a new round is started, unless the limit on the length of the rewrite paths is reached or no more states are left to be analyzed:

```
eq bfsLive(M, N, DT, DP, emptyTermSet, TS, N') =
      if ((N == N' and N =/= 0) or TS == emptyTermSet) then TS
      else bfsLive(M, N, DT, DP, TS, emptyTermSet, N' + 1)
      fi .
```

In the module

```
(tomod varNOM is
  protecting NCA-NOM1 .
  var ATTS1 : AttributeSet .
  var CF : Configuration .
  var Q : Qid .
endtom)
```

the pattern `{ Q : NOMreceiverAlone | isNominee : true, ATTS1 > CF }` is matched by receivers with their nominee flag set to true. Therefore, the temporal property

$\Diamond_{\leq 500}$ `matching pattern { Q : NOMreceiverAlone | isNominee : true, ATTS1 > CF }`

holds if a nominee receiver is always found within time 500. This property can be successfully model checked for initial state `NOMstate2(1)` in the module

```
(fmod NOM-STRATEGIES is
  protecting AER-STRATEGIES .
  including META-LEVEL[varNOM] .
endfm)
```

as follows:

```
Maude> (red bfsLive(varNOM, 'NOMstate2[{'1}'Nat], 0, {'500}'Nat,
     up(varNOM, {< Q : NOMreceiverAlone | isNominee : true, ATTS1 > CF })) .)
rewrites: 77952 in 7770ms cpu (7780ms real) (10032 rewrites/second)
[ ... ]
result TermSet : emptyTermSet
```

Again, there is no path not satisfying the property. However, it is not the case that the a nominee is always aware of its status within time 300[6]:

[6]The command `down` makes the output more readable, and can be used under the assumption that the result will be a single term.

```
Maude> (down varNOM : red bfsLive(varNOM, 'NOMstate2[{'1}'Nat], 0, {'300}'Nat,
     up(varNOM, {< Q : NOMreceiverAlone | isNominee : true, ATTS1 > CF })) .)
rewrites: 42089 in 4000ms cpu (4000ms real) (10522 rewrites/second)

result Error(ClockedSystem) : {
  < 'a : NOMsenderAlone | csmNominee : noOid,  [ ... ] >
  < 'b : NOMreceiverAlone | isNominee : false, [ ... ] >
  < 'f : NOMreceiverAlone |  isNominee : false , csmTimer : 372, [ ... ] >
 [ ... ] } in time 59
```

The receiver `'e` should be the nominee in all paths. This property can be defined by the temporal formula

$$\Diamond\,(\texttt{matches \{< 'e : NOMreceiverAlone | isNominee : true, ATTS1 > CF\}}),$$

and is checked successfully as follows:

```
Maude> (red bfsLive(varNOM, 'NOMstate2[{'1}'Nat], 0, noTerm,
     up(varNOM, {< 'e : NOMreceiverAlone | isNominee : true, ATTS1 > CF })) .)
rewrites: 461316 in 34960ms cpu (34970ms real) (13195 rewrites/second)
[ ... ]
result TermSet : emptyTermSet
```

Finally we show that after the nominee is set, there exists a path where, at some point, no receiver knows that it is the nominee receiver. This property can be verified by finding counterexamples to the property

`matching anyPattern` **UStable**
`matching pattern {< Q : NOMreceiverAlone | isNominee : true, ATTS1 > CF}`,

which states that after the value of the `isNominee` attribute of some receiver is `true`, there is always a receiver (not necessarily the same) with its `isNominee` flag set to `true`. As expected, this property does not hold, and is refuted by providing a counterexample:

```
Maude> (down varNOM : red ustable(varNOM, 'NOMstate2[{'1}'Nat], 0, noTerm,
     up(varNOM, {< Q : NOMreceiverAlone | isNominee : true, ATTS1 > CF })) .)
rewrites: 490839 in 45110ms cpu (45130ms real) (10880 rewrites/second)

result Error(ClockedSystem) : {
  < 'e : NOMreceiverAlone | isNominee : false, [ ... ] >
  < 'a : NOMsenderAlone | csmNominee : 'e, csmSetTime : 19504, [ ... ] >
  < 'b : NOMreceiverAlone | isNominee : false, [ ... ] >
  < 'f : NOMreceiverAlone | isNominee : false, [ ... ] >
  < 'g : NOMreceiverAlone | isNominee : false, [ ... ] >
  (NAMPacket(true) from 'a to 'e)   [ ... ] } in time 19504
```

The result is a counterexample showing the expected scenario when `'e` becomes the new nominee instead of `'f`, and `'f` has read its `NAMPacket(false)` while `'e` has yet to read its `NAMPacket(true)`. Such a change of nominee can also occur in the combined protocol due to change in the rtt values, and could cause problems if a data packet is also received at the same time.

To sum up the analysis effort: Both default execution and exhaustive search methods validated the protocol, and, in addition, they confirmed that there are times when no receiver in the systems thinks it is the nominee receiver.

5.2.8 The Rate Control Protocol

The rate control protocol tries to adjust the sending frequency of data packets based on feedback from the nominee receiver. The protocol does not decide the shortest time until the next packet can be sent, since the reception of a message could alter the earliest time the next packet could be sent. Instead, the protocol is used to decide whether a packet can be sent at the current time.

The sender decides its allowed sending rate based on the reception of `ccmPacket` messages from the nominee receiver. Whenever the nominee receives a data packet it has not seen before, it sends a `ccmPacket` with the sequence number of its *lowest outstanding* data packet (together with the timestamp and the transmission flag of the received data packet) to the sender.

The sender part is fairly complex and we just outline the main ideas. The sender has a "window" of sequence numbers, defining the set of data packets which can be sent, starting with the sequence number of the lowest outstanding data packet at the nominee receiver (the attribute `winLowerSeq`), and containing $\lfloor \texttt{winSize} \rfloor$ elements. That is, if the lowest sequence number the nominee receiver hasn't seen is n, and the window size is q, then, the highest sequence number of a data packet which can be sent is $n + \lfloor q \rfloor - 1$. The other limitation on sending is that a new packet should not be sent if the `maxBurstCount` value, denoting the number of original data packets sent since the last `ccmPacket` was received, is greater than a certain limit.

Much of the protocol is concerned with increasing and decreasing the `winSize` value based on the feedback, or lack thereof, from the nominee. Each time a `ccmPacket` is received (in normal recovery mode), the value of the `winSize` attribute increases by 1 until a threshold is reached, after which `winSize` increases by the much smaller amount $\frac{1}{\lfloor \texttt{winSize} \rfloor}$. The sender goes into *fast repair* mode when it has received three `ccmPackets` with the same lowest outstanding sequence number. That is, when its `dupLowerSeq` value is 3. In fast recovery mode, the number of messages that can be sent is lowered by dividing `winSize` by 2. Finally, if too much time has passed from the last sending of data packets until a `ccmPacket` is received, then the rate control window is "reset" to its start state, albeit with a much smaller threshold. The initial value of the `ccmTimer` depends on the round trip time estimate to the nominee, and on the number of times in a row that the `ccmTimer` has expired.

To make an execution of the stand-alone RC protocol more interesting, we added a sending mechanism that attempts to send a new packet every ms. For convenient analysis purposes we also add to the sender a list of the times it has managed to send an original data packet. Although, strictly speaking, the latter part is not a part of the protocol and should not appear in the specification, but could be provided by a strategy which records the times the packet-sending rule was applied, we add it to the protocol to get more information directly by just executing standard Real-Time Maude commands. The data type `TimeList` defining lists of time values, with list concatenation denoted by juxtaposition, is given in a module `NAT-TIME-LIST`.

The value of the `winSize` attribute is sometimes increased by its inverse or divided by two. Instead of using the rationals, which is computationally inefficient, we have extended the machine integers to the fixed point reals where the number $n/1000000$ is denoted by the term `[`n`]` of sort `FixedPointNumber`. The specification of the fixed point reals is given in Appendix A. Here it suffices to say that the term `(fixed)` n gives the fixed point number corresponding to n (namely, `[`n000000`]`), that `(int)` r gives the truncated integer part of the fixed point number r, and that the fixed point numbers are equipped with all the necessary functions like `_plus_`, `_minus_`, `_times_`, `_div_`, `_le_`, `_min_`, `_floor_`, and `abs`.

The (stand-alone) rate control protocol is given by the following specification:

```
(tomod NCA-RC is
including LINK .
including COMMON-CLASSES .
including DETERMINISTIC-TICK-RULE .
including ENVIRONMENT .
including FIXED-POINT-REALS .

msg ncarcDataPacket : NzNat Time Bool -> LargePacket .
*** Data packet used in stand-alone RC protocol.
*** Usage: ncarcDataPacket(seqNo, timestamp, retransmissionflag)

msg ccmPacket : Oid Nat Time Bool -> Packet .
*** Usage : ccmPacket(originator, sequenceNo, timeStamp, retransmissionFlag)

*** Reset the isNominee flag:
msg NAMPacket : Bool -> Packet .
*** Usage: NAMPacket(receiverIsNominee?)

class RCsender |
      csmNominee : DefOid,
      timerNominee : DefOid,  *** nominee when timer last set
      winSize : PosFixedPointNumber,
      winLowerSeq : Nat,
      dupLowerSeq : Nat,      *** no of ccmPackets with seqNo winLowerSeq
      maxSeqSent : Nat,       *** seqNo of last original data packet sent
      smoothedRTT : MachineInt,
      rttVariance : MachineInt,
      ccmTimeout : Time,
      ssThreshold : PosFixedPointNumber,  *** threshold for slower winSize-
                                          *** increase
      backOffFactor : Nat,    *** no of times the timer has expired in a row
      fastRecovFlag : Bool,   *** fast recovery mode?
      fastRecovSeq : Nat,     *** last seqNo for which should stay in fR mode
      maxBurstCount : Nat,    *** number of original data packets sent
                              *** since last ccm packet received
      ccmTimer : DefTime .
subclass RCsender < Clock .
subclass RCsender < Sender .
```

The following attributes are not really part of the protocol and are only used to execute the (stand-alone) RC protocol in an interesting way. They will not appear in the combined protocol, because they are declared in the following subclass `RCsenderAlone`:

```
class RCsenderAlone |
      sendDataTimer : DefTime,  *** Timer to send data. Initialized to 2000
                                *** to let the system find a nominee first.
      nextNoToSend : NzNat,     *** Next message number to send.
      sentTimes : TimeList .    *** Times of sendings of data packets.
subclass RCsenderAlone < RCsender .
```

```
vars FROM TO : Oid .
var SEQNO : NzNat .
vars TIMESTAMP  CCMTO : Time .
vars RTFLAG FRF : Bool .
vars WS SST : PosFixedPointNumber .
vars WLS DLS MSS BOF FRS MBC : Nat .
vars RTTV SRTT : MachineInt .
var CCMT : DefTime .
var TL : TimeList .
```

A `ccmPacket` is ignored if it does not come from the nominee receiver:

```
crl [G2a] :
    { (ccmPacket(FROM, SEQNO, TIMESTAMP, RTFLAG) from Q to TO)
    < TO : RCsender | csmNominee : DQ >  CF }
    =>
    { < TO : RCsender | >  CF }
    if FROM =/= DQ .
```

Reception of a `ccmPacket` from the nominee is somewhat complex:

```
*** Use cases G2 to G10:
rl [G2b] :
   { (ccmPacket(FROM, SEQNO, TIMESTAMP, RTFLAG) from Q to TO)
      < TO : RCsender | clock : R, csmNominee : FROM,
                        winSize : WS, winLowerSeq : WLS,
                        dupLowerSeq : DLS, maxSeqSent : MSS,
                        smoothedRTT : SRTT, rttVariance : RTTV,
                        ccmTimeout : CCMTO, ssThreshold : SST,
                        backOffFactor : BOF, fastRecovFlag : FRF,
                        fastRecovSeq : FRS, maxBurstCount : MBC,
                        ccmTimer : CCMT >  CF }
   =>
   { < TO : RCsender |
       winSize :  (if (SEQNO =/= WLS and FRF and FRS < max(SEQNO, WLS))
                   then max(min(SST, (fixed) 256), (fixed) 1)
                   else (if (SEQNO =/= WLS and (not FRF) and
                              3 < newDupLowerSeq(SEQNO, WLS, DLS))
                         then max(min(max(g7winSize(SEQNO, WLS, FRF, WS,
                                                         SST) div (fixed) 2,
                                      (fixed) 2) plus (fixed) 3,
                                      (fixed) 256),
                                  (fixed) 1)
                         else (if (SEQNO =/= WLS and FRF and
                               max(SEQNO, WLS) <= FRS)
                               then   max(min(g7winSize(SEQNO, WLS, FRF, WS,
                                                         SST) plus (fixed) 1,
                                              (fixed) 256),
                                          (fixed) 1)
                               else g7winSize(SEQNO, WLS, FRF, WS, SST) fi)
                         fi)
                   fi),
```

```
winLowerSeq : max(SEQNO, WLS),
dupLowerSeq : newDupLowerSeq(SEQNO, WLS, DLS),
smoothedRTT : (if RTFLAG then SRTT
               else (if SRTT == 0 then max(1, R minus TIMESTAMP)
                     else (SRTT + (max(1, R minus TIMESTAMP)
                                         minus SRTT) / 8)
                     fi)
               fi),
rttVariance : (if RTFLAG or (SRTT == 0) then RTTV
               else (if RTTV == 0 then
                        abs(max(1, R minus TIMESTAMP) minus SRTT)
                     else RTTV + ((abs(max(1, R minus TIMESTAMP)
                                         minus SRTT) minus RTTV) / 4)
                     fi)
               fi),
ssThreshold : (if (SEQNO =/= WLS and (not FRF) and
                   (3 < newDupLowerSeq(SEQNO, WLS, DLS)))
               then max(g7winSize(SEQNO, WLS, FRF, WS, SST) div
                           (fixed) 2,
                        (fixed) 2)
               else SST
               fi),
ccmTimeout : (if RTFLAG then CCMTO
              else (if SRTT == 0 then max(1, R minus TIMESTAMP)
                    else (if RTTV == 0 then
                             (4 * abs(max(1, R minus TIMESTAMP)
                                          minus SRTT))
                              + (SRTT + (max(1, R minus TIMESTAMP)
                                             minus SRTT) / 8)
                          else (SRTT + (max(1, R minus TIMESTAMP)
                                             minus SRTT) / 8) +
                               4 * (RTTV +
                                     ((abs(max(1, R minus TIMESTAMP)
                                         minus SRTT) minus RTTV) / 4))
                          fi)
                    fi)
                fi),
fastRecovFlag : (if FRF then max(SEQNO, WLS) <= FRS
                 else (3 <= newDupLowerSeq(SEQNO, WLS, DLS)) fi),
fastRecovSeq : (if (SEQNO =/= WLS and FRF and FRS < max(SEQNO, WLS))
                then 0
                else (if (SEQNO =/= WLS and (not FRF) and
                          3 < newDupLowerSeq(SEQNO, WLS, DLS))
                      then (max(WLS, SEQNO) +
                            (int) g7winSize(SEQNO, WLS, FRF, WS, SST)
                            minus 1)
                      else FRS
                      fi)
                fi),
backOffFactor : 1,
maxBurstCount : 0,
ccmTimer : (if MSS < max(WLS, SEQNO) then noTimeValue else CCMT fi) > CF } .
```

```
op newDupLowerSeq : Nat Nat Nat -> Nat .
eq newDupLowerSeq(SEQNO, WLS, DLS) =
      (if SEQNO == WLS then DLS + 1
       else (if WLS < SEQNO then 0 else DLS fi) fi) .
op g7winSize : Nat Nat Bool PosFixedPointNumber PosFixedPointNumber
                            -> PosFixedPointNumber .
*** Gives the winSize at the line "if (fastRecovFlag == TRUE)" in the protocol:
eq g7winSize(SEQNO, WLS, FRF, WS, SST) =
     (if (FRF or (SEQNO <= WLS)) then WS
      else max(min((if (WS le SST) then (WS plus (fixed) 1) else
                   (WS plus ((fixed) 1 div floor(WS)))fi), (fixed) 256),
               (fixed) 1)
      fi) .
```

The following function decides whether a new packet can be sent. It checks whether the next data packet, having sequence number `maxSeqSent + 1` is in the window of sequence numbers that can be sent, and whether the `maxBurstCount` is less than 2 if the sender is in fast recovery mode, and less than 4 otherwise:

```
op sendAllowed : Nat Nat PosFixedPointNumber Bool Nat -> Bool .
*** Usage: sendAllowed(maxSeqSent, winLowerSeq, winSize, fastRecFlag, maxBuCount)
eq sendAllowed(MSS, WLS, WS, FRF, MBC) =
     (MSS < (WLS + (((int) WS) minus 1))) and
     (if FRF then (MBC < 2) else (MBC < 4) fi) .
```

When the `sendDataTimer` expires (which should be every ms after an initial wait of 2000 ms), a new data packet is sent by rule `G12` if possible, and the window is updated by rule `G13` when the `ccmTimer` expires:

```
rl [G12] :
   { < Q : RCsenderAlone | children : OS, clock : R, csmNominee : DQ,
                           sendDataTimer : 0, nextNoToSend : NZN,
                           maxSeqSent : MSS, winLowerSeq : WLS,
                           winSize : WS, fastRecovFlag : FRF,
                           maxBurstCount : MBC, ccmTimer : DT,
                           timerNominee : DQ', backOffFactor : BOF,
                           ccmTimeout : CCMTO, sentTimes : TL > CF }
    =>
    { if (NZN == MSS + 1) and sendAllowed(MSS, WLS, WS, FRF, MBC) then
        (< Q : RCsenderAlone | sendDataTimer : 1,
                               nextNoToSend : NZN + 1,
                               maxSeqSent : MSS + 1,
                               ccmTimer :
                                (if WLS <= MSS then (BOF * CCMTO)
                                 else noTimeValue fi),
                               maxBurstCount : MBC + 1,
                               timerNominee : (if WLS <= MSS then DQ else DQ' fi),
                               sentTimes : TL R >
         multiSend(ncarcDataPacket(NZN, R, false), Q, OS) CF)
      else (< Q : RCsenderAlone | sendDataTimer : 1 > CF) fi } .
```

```
rl [G13] :
   { < Q : RCsender | ccmTimer : 0, timerNominee : DQ,
                      ssThreshold : SST, backOffFactor : BOF,
                      winSize : WS, winLowerSeq : WLS,
                      maxSeqSent : MSS, fastRecovFlag : FRF,
                      fastRecovSeq : FRS, maxBurstCount : MBC >  CF }
   =>
   { < Q : RCsender | ccmTimer : noTimeValue,
                      ssThreshold : (if DQ == noOid then (fixed) 64 else
                                     max(WS div (fixed) 2, (fixed) 2) fi) ,
                      backOffFactor : (if DQ == noOid then 1 else
                                       min(2 * BOF, 64) fi),
                      winSize : (fixed) 1, winLowerSeq : MSS + 1,
                      fastRecovFlag : false,
                      fastRecovSeq : 0, maxBurstCount : 0 > CF } .
```

The `ccmTimeout` value should be changed when the nominee is nulled out. In the stand-alone protocol the change of nominee is modeled by a method `nomineeIs`:

```
msg nomineeIs : DefOid -> Msg .

rl [G14] :
   { nomineeIs(DQ)
     < Q : RCsenderAlone | csmNominee : DQ', ccmTimeout : R > CF }
   =>
   { < Q : RCsenderAlone | csmNominee : DQ,
                           ccmTimeout : (if (DQ == noOid and DQ =/= DQ')
                                         then 1000 else R fi) >  CF } .
```

A repair server only subcasts packets from its repair server downstream (rule `RCrepserv`) and propagates packets from its children upstream:

```
class RCrepairserver .
subclass RCrepairserver < Repairserver .

class RCrepairserverAlone .
subclass RCrepairserverAlone < RCrepairserver .

rl [RCrepserv] :
   { (ncarcDataPacket(NZN, R, X) from Q to Q')
     < Q' : RCrepairserverAlone | children : OS >  CF }
   =>
   { < Q' : RCrepairserverAlone | >
     multiSend(ncarcDataPacket(NZN, R, X), Q', OS)  CF } .

rl [RCrepserv2] :
   { (ccmPacket(Q, NZN, R, X) from Q'' to Q')
     < Q' : RCrepairserver | repairserver : Q''' > CF }
   =>
   {  < Q' : RCrepairserver | >
      send(ccmPacket(Q, NZN, R, X), Q', Q''')  CF } .
```

The receiver stores the sequence numbers of the data packets it has seen in the attribute `msgRcvd` of sort `NzNatList` of ordered lists of nonzero natural numbers, with list concatenation denoted by juxtaposition, to be able to find the lowest outstanding sequence number. This sort also has operations `add` for adding an element to the sorted list, and an operation `_inList_` to decide whether an element is in the list. (In the combined protocol, the lowest outstanding sequence number is the given by the value of the attribute `readNextSeq`.)

```
class RCreceiver | isNominee : Bool .

class RCreceiverAlone | msgRcvd : NzNatList .
subclass RCreceiverAlone < RCreceiver .

vars NNL NNL' : NzNatList .

*** Find the lowest outstanding sequence number:
op lowestOutstanding : NzNatList -> NzNat .
eq lowestOutstanding(nil) = 1 .
eq lowestOutstanding(NZN NNL) =
      if NZN =/= 1 then 1 else firstMissing(NZN NNL) fi .
op firstMissing : NzNatList -> NzNat .
eq firstMissing(NZN NZN' NNL) =
    if (NZN + 1) =/= NZN' then (NZN + 1)
    else firstMissing(NZN' NNL) fi .
eq firstMissing(NZN) = NZN + 1 .
```

The nominee receiver should send a `ccmPacket` upstream (though not unicast!) to the sender by way of the intermediate repair servers when a new data packet is received. The rule `resetIsNominee` updates the nominee status of the receiver.

```
rl [H1] :
   { (ncarcDataPacket(NZN, R, X) from Q to Q')
     < Q' : RCreceiverAlone | isNominee : Y, msgRcvd : NNL >  CF }
   =>
   { < Q' : RCreceiverAlone | msgRcvd : add(NZN, NNL) >
    (if (Y and (not (NZN inList NNL))) then
       send(ccmPacket(Q', lowestOutstanding(add(NZN, NNL)), R, X), Q', Q)
     else none fi) CF } .

rl [resetIsNominee] :
   { (NAMPacket(X) from Q' to Q) < Q : RCreceiverAlone | >  CF }
   =>
   { < Q : RCreceiverAlone | isNominee : X > CF } .
```

Finally, the timed behavior of the objects is entirely as before:

```
eq mte(< Q : RCsenderAlone | ccmTimer : DT, sendDataTimer : DT' >)
     = default-min(DT, DT') .

eq delta(< Q : RCsenderAlone | clock : R', ccmTimer : DT,
                               sendDataTimer : DT' >, R) =
```

```
      < Q : RCsenderAlone | clock : R' plus R,
                              ccmTimer : DT default-monus R,
                              sendDataTimer : DT' default-monus R > .
eq delta(< Q : RCrepairserverAlone | >, R) =
     < Q : RCrepairserverAlone | > .
eq mte(< Q : RCrepairserverAlone | >) = INF .
eq delta(< Q'' : RCreceiverAlone | >, R) =
      < Q'' : RCreceiverAlone | > .
eq mte(< Q'' : RCreceiverAlone | >) = INF .
endtom)
```

We add the following initial state, suggested by Mark Keaton, to prototype the stand-alone RC protocol.

```
(tomod NCA-RC1 is
protecting NCA-RC .
op RCstate1 : -> System .
eq RCstate1 =
 ({< 'a : RCsenderAlone | clock : 0, children : 'b, csmNominee : noOid,
                          timerNominee : noOid,
                          winSize : (fixed) 1, winLowerSeq : 1, dupLowerSeq : 0,
                          maxSeqSent : 0, smoothedRTT : 0, rttVariance : 0,
                          ccmTimeout : 1000, ssThreshold : (fixed) 64,
                          backOffFactor : 1, fastRecovFlag : false,
                          fastRecovSeq : 0, maxBurstCount : 0,
                          ccmTimer : noTimeValue, nextNoToSend : 1,
                          sendDataTimer : 6000, sentTimes : nil >
 < 'b : RCrepairserverAlone | children : 'c, repairserver : 'a >
 < 'c : RCreceiverAlone | isNominee : false, msgRcvd : nil >
 < 'env : Env | msgsFromEnv : dly(nomineeIs('c), 1444)
                              dly(NAMPacket(true) from 'a to 'c, 1444) >
 < 'ab : Link | up : 'a, down : 'b, bound : 6, propDelay : 10,
               linkSpeed : 1, downMsgs : nil, downSize : 0,
               upMsgs : nil,  upSize : 0 >
 < 'bc : Link | up : 'b, down : 'c, bound : 3, propDelay : 10,
               linkSpeed : 1, downMsgs : nil, downSize : 0,
               upMsgs : nil,  upSize : 0 > }) .
endtom)
```

What results do we expect from executing the specification? At the beginning, the sending window is (1), and packet 1 is sent. When the acknowledgment of packet 1 (the `ccmPacket` with lowest outstanding sequence number 2) is received, both the lowest window number, and the size of the window, are increased, that is, the window is now $(2, 3)$, and both these data packets are sent. When the acknowledgment of packet 2 is received, the window is $(3, 4, 5)$, and packets 4 and 5 can be sent. The sequence of sending and responses can be given as

message sent	1		2,3		4,5		6,7		8,9		10,11	$\cdots$
response		1		2		3		4		5		$\cdots$

Furthermore, were it not for congestion delays, the sending rate would grow exponentially. (Message 1 releases messages 2 and 3 in one round trip time, and 2 and 3 release the messages 4, 5, 6, and 7 in the next round trip time cycle, etc.) The exponential increase will flatten out when the window size has reached the threshold, since the window size will then only increase by its inverse. However, increasing the sending frequency will also result in packets getting lost. In the full protocol, these packets would be repaired, but we did not add any repair service to the stand-alone RC protocol. The system will therefore get stuck when the first data packet is lost, as the receiver will require it, and it cannot get it. Eventually, the day is saved by the expiration of the `ccmTimer`, which will reset `winLowerSeq` to next packet to be sent, but will also reset the window size to 1, starting the slow sending once again.

A sample execution of the protocol gives this result:

```
Maude> (rew- [5000] RCstate1 .)
rewrites: 640500 in 5240ms cpu (10220ms real) (122232 rewrites/second)
rewrite in NCA-RC1 : RCstate1 .
result Error(ClockedSystem) : { < 'a : RCsenderAlone | clock : 7882,
    children : 'b, csmNominee : 'c, ccmTimer : 111, maxBurstCount : 1,
    fastRecovSeq : 38, fastRecovFlag : true, backOffFactor : 1, ssThreshold
    : [2000000], ccmTimeout : 113, rttVariance : 10, smoothedRTT : 73,
    maxSeqSent : 114, dupLowerSeq : 9, winLowerSeq : 38, winSize :
    [77000000], timerNominee : 'c, sentTimes : (6000 6066 6067 6132 6133
    6144 6145 6198 6199 6210 6211 6222 6223 6234 6235 6264 6265 6276 6277 6288
    6289 6300 6301 6312 6313 6324 6325 6336 6337 6348 6349 6360 6361 6372 6373
    6384 6385 6548 6614 6615 6680 6681 6692 6693 6746 6758 6770 6782 6812 6824
    6836 6848 6878 6890 6902 6914 6944 6956 6968 6980 7010 7022 7034 7046 7076
    7088 7100 7112 7142 7154 7166 7178 7208 7220 7232 7244 7274 7286 7298 7310
    7340 7352 7364 7376 7406 7418 7430 7442 7472 7484 7496 7508 7538 7550 7562
    7574 7604 7616 7628 7640 7670 7682 7694 7706 7736 7748 7760 7772 7802 7814
    7826 7838 7868 7880), nextNoToSend : 115, sendDataTimer : 1 >
  < 'b : RCrepairserverAlone | children : 'c, repairserver : 'a >
  < 'c : RCreceiverAlone | isNominee : true, msgRcvd : (1 2 3 4 5 6 7 8 9 10 11 12
    13 14 15 16 17 18 20 22 24 26 28 30 32 34 36 38 39 40 41 42 43 44 45 46 47
    48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
    73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97
    98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 ) >
  [ ... ] } in time 7882
```

Indeed, the first packets are sent pairwise with (exponentially?) increasing frequency. The first packet lost is number 19, so packet 38 cannot be sent. The system therefore gets stuck after packet 37 is sent (time 6385), and only after quite a long time (163 ms), when the timer expires, does the system get back to normal (with `winLowerSeq` set to the packet 38 waiting to be sent), with slow sending rate. However, packet 19 is still not repaired, and will cause further problems. If relevant at all due to the lack of repair service, the output part after time 6548 can be analyzed as well.

5.2.9 The Repair Service Protocol

The repair service part is the main part of the AER/NCA protocol. The protocol specifies a system which receives variable-sized data blocks from a sender application, places the data in a number of data packets, and is responsible for transmitting all data packets in order to a group of receiver applications so that the original data blocks can easily be recovered.

Sender Protocol

The sender object peforms the following tasks:

- initialize the protocol at the other nodes by multicasting `SPMPacket(0)` packets;
- receive data blocks from the application layer of the protocol;
- turn each data block into fixed-sized data packets and multicast the data packets;
- multicast in regular intervals source path messages with the sequence number of the last (not retransmitted) data packet sent;
- handle repair requests by retransmitting the requested data packets.

The sender receives data blocks of the form `dataBlock`$(content, sender, size, \Delta)$ from the application layer, where *content* is an identifier of the content of the data, *sender* is the sender object (useful in case there are more than one multicast groups/trees), *size* is the number of data packets needed to represent the data block, and Δ is the desired interval between two consecutive sendings of (original) data packets.

The data packets are of the form

$$\texttt{dataPacket}(content, seqNo, timeStamp, firstSeqFlag, segFlag, retransFlag),$$

where *content* is the identifier of the data block being transmitted, *seqNo* is the sequence number of the data packet, and equals one plus the number of original data packets already sent in the current execution of the protocol, *timeStamp* is the time the packet is sent, *firstSeqFlag* is true iff the packet is the first data packet of a data block, *segFlag* is true iff the data packet is neither the first nor the last packet of a data block, and *retransFlag* is true iff the packet is a repair packet.

These messages are declared in the following module `AER-RS`:

```
(tomod AER-RS is
including LINK .
including RANDOM .
including COMMON-CLASSES .
including DETERMINISTIC-TICK-RULE .
including SHARED-MSGS .
including ENVIRONMENT .

msg dataBlock : Qid Oid NzNat Time -> Msg .
msg dataPacket : Qid NzNat Time Bool Bool Bool -> LargePacket .
```

The sender should store the parts of the data blocks it has yet to send. This is done in an attribute `unsentDataPackets` by a list of elements of the form `data`$(content, m, size, \Delta)$, where $m-1$ is the number of data packets already multicast from the data block, and *size* and Δ correspond to the same parameters of the data block.

```
sort DataBuffer .
op data : Qid NzNat NzNat Time -> DataBuffer .
op nil : -> DataBuffer .
op __ : DataBuffer DataBuffer -> DataBuffer [assoc id: (nil).DataBuffer] .
```

The sender class is declared as follows, where the meaning of some attributes will be explained below:

```
class RSsender |
      nextSeq : NzNat,                *** seqNo of next data packet to be sent
      SPMTimer : DefTime,
      reTransBuf : MsgConf,           *** Retransmission buffer
      unsentDataPackets : DataBuffer, *** Data yet to be multicast
      sendDataTimer : DefTime .       *** Try to send data when expires
subclass RSsender < Sender .
subclass RSsender < Clock .

class RSsenderAlone .
subclass RSsenderAlone < RSsender .
```

When a data block is received from the application layer, it is stored in the sender, and if it is the first block, the sending should wait five seconds to ensure that a nominee is set:

```
var DB : DataBuffer .
var RTB : MsgConf .

rl [A3getDataBlock] :
   { dataBlock(Q, Q', NZN, R)
   < Q' : RSsender | unsentDataPackets : DB, sendDataTimer : DT,
                     nextSeq : NZN' >  CF }
   =>
   { < Q' : RSsender | unsentDataPackets : (DB data(Q, 1, NZN, R)),
                       sendDataTimer : (if DT == noTimeValue then
                                          (if NZN' == 1 then 5000 else R fi)
                                        else DT fi) > CF } .
```

Every 4000 ms., triggered by the expiration of the `SPMTimer`, the sender multicasts a source path message with the sequence number of the last original data packet sent (which is given by `nextSeq`−1). Furthermore, the sender starts its operation by multicasting ten source path messages to initiate the repair servers and the receivers, and to allow them to set their `repairserver` attributes:

```
msg startRS : Oid -> Msg .
*** Initializes the sender and starts the protocol.

rl [A1] :
   { startRS(Q)
   < Q : RSsender | children : OS, nextSeq : NZN, SPMTimer : DT > CF  }
   =>
   { < Q : RSsender | nextSeq : 1, SPMTimer : 4000 >
   multiSend(SPMPacket(0), Q, OS)
   multiSend(SPMPacket(0), Q, OS)
   multiSend(SPMPacket(0), Q, OS)
   multiSend(SPMPacket(0), Q, OS)
   multiSend(SPMPacket(0), Q, OS)
   multiSend(SPMPacket(0), Q, OS)
```

```
   multiSend(SPMPacket(0), Q, OS)
   multiSend(SPMPacket(0), Q, OS)
   multiSend(SPMPacket(0), Q, OS)
   multiSend(SPMPacket(0), Q, OS) CF } .

rl [A2] :
   { < Q : RSsender | nextSeq : NZN, children : OS, SPMTimer : 0 >  CF }
   =>
   { < Q : RSsender | SPMTimer : 4000 >
   multiSend(SPMPacket(NZN minus 1), Q, OS) CF } .
```

To be able to serve retransmission requests for lost data packets, the sender must store each transmitted data packet in the attribute `retransBuf`, together with the *NAK count* value of each packet, which denotes (approximately) how many times a repair for the packet has been requested. The retransmission buffer consists of messages of the form `MsgAndNak`(*data packet*, *NAKcount*), where the wrapper `MsgAndNak` is declared

```
msg MsgAndNAK : Msg Nat -> Msg .
```

The data packet with sequence number `nextSeq` is multicast when `sendDataTimer` expires:

```
rl [A3send] :
    { < Q : RSsenderAlone | children : OS, clock : R, nextSeq : NZN'',
                            unsentDataPackets : data(Q', NZN, NZN', R') DB,
                            reTransBuf : RTB, sendDataTimer : 0 > CF }
    =>
    { < Q : RSsenderAlone | nextSeq : NZN'' + 1,
                       unsentDataPackets :
                         ((if NZN == NZN' then nil
                          else data(Q', NZN + 1, NZN', R') fi) DB),
                       reTransBuf :
                         (RTB MsgAndNAK(dataPacket(Q', NZN'', R, NZN == 1,
                                         NZN =/= 1 and NZN =/= NZN', false), 0)),
                       sendDataTimer : (if NZN =/= NZN' then
                                        R' else sendRate(DB) fi) >
    multiSend(dataPacket(Q', NZN'', R, NZN == 1, NZN =/= 1 and NZN =/= NZN',
                         false), Q, OS) CF } .

op sendRate : DataBuffer -> DefTime .
eq sendRate(nil) = noTimeValue .
eq sendRate(data(Q, NZN, NZN', R) DB) = R .
```

When a child has not received a data packet it should have received, or needs to see the data packet again, it requests a retransmission of the data packet by sending a `NAKPacket` message containing the sequence number of the missing packet as well as its NAK count. If the NAK count is less than or equal to the sender's NAK count for the packet, then this repair is already underway and is not performed:

```
msg NAKPacket : NzNat Nat Bool -> Packet .
```

```
*** Usage: NAKPacket(seqNo, NAKcount, fastRepairFlag)

rl [A4] :
   { (NAKPacket(NZN, N, X') from Q'' to Q)
   < Q : RSsender |  children : OS, reTransBuf :
                    (MC MsgAndNAK(dataPacket(Q', NZN, R, X, Y, Z), N')) > CF }
   =>
   { < Q : RSsender | reTransBuf :
                        (MC MsgAndNAK(dataPacket(Q', NZN, R, X, Y,
                                              if N' < N then true else Z fi),
                                       max(N, N'))) >
   (if N' < N then multiSend(dataPacket(Q', NZN, R, X, Y, true), Q, OS)
    else none fi) CF } .
```

Receiver Protocol

The receiver receives data packets and forwards them to the receiver application in increasing order of their sequence numbers, so that all data packets with sequence numbers up to the first data packet not received, are forwarded. Data packets which are received but cannot be forwarded are stored in the `dataBuffer` attribute, and the smallest sequence number among the non-received data packets is stored in the `readNextSeq` attribute.

The receiver detects missing data packets when data packets or source path messages with unexpectedly high sequence numbers are received. The receiver waits a small amount of time (in case some sibling or its repair server also detects the loss) before sending a NAK-request for each newly detected packet to its repair server, which, in turn, either sends the data packet from its cache or forwards the request upstream. The receiver requests another repair of the data packet if it does not receive a response to the repair request within a reasonable amount of time.

The time between consecutive repair requests for the same data packet should depend on the round trip times to the source, since the repair request may be propagated all the way up to the source in the worst case. The suppression time before sending the NAK packet depends on its siblings and its repair server, and therefore on their round trip times to their repair server. The NAK packet is sent without delay if the receiver's `fastRepairFlag` is set, which is the case for the nominee receiver.

The sequence number, the values of the suppression and retransmission timers, and the NAK count denoting the number of times the repair has been requested or suppressed, must be stored for each missing data packet:

```
sort DataInfo .
op info : NzNat DefTime DefTime Nat -> DataInfo .
*** info(seqNo, suppressionTimer, retransmissionTimer, NAKcount)
op none : -> DataInfo .
op __ : DataInfo DataInfo -> DataInfo [assoc comm id: (none).DataInfo] .
```

The receiver class is declared as follows:

```
class RSreceiver |
      isNominee : Bool,
```

```
      fastRepairFlag : Bool,
      readNextSeq : NzNat,         *** the NEXT expected seqNo to application.
      smoothedRTT : MachineInt,    *** smoothed RTT estimate to source
      rttVariance : MachineInt,
      suppressTO : Time,           *** time before sending NAK packet
      retransTO : Time,            *** time before resending NAK packet
      dataBuffer : MsgConf,        *** buffers dataPackets
      dataInfo : DataInfo,         *** store info about repairs
      SPMread : Bool .             *** SPM message read?
subclass RSreceiver < Receiver .

class RSreceiverAlone .
subclass RSreceiverAlone < RSreceiver .
```

SPMPackets are used to set the repairserver attribute and find lost packets. When a SPM value n is received, it means that the source has transmitted all data packet up to sequence number n, and all data packets with sequence numbers less than or equal to n which are not received before, must be considered missing. Upon detection of missing packets for which error recovery has not yet started, the receiver initiates error recovery by creating a new repair state for the sequence number n with NAK count 1, and, as already mentioned, suppresses its repair request by starting the suppression timer for the packet, and requests the repair only when this timer expires. All this is done by adding an element $\texttt{info}(n, r, \texttt{noTimeValue}, 1)$, for r the suppression time (which is 0 when the receiver is in fast repair mode), to the dataInfo attribute, which is done by the function updateDI:

```
vars DI DI' : DataInfo .

rl [firstSPM] :
   { (SPMPacket(N) from Q to Q')
     < Q' : RSreceiverAlone | SPMread : false > CF }
   =>
   { (SPMPacket(N) from Q to Q')
     < Q' : RSreceiverAlone | SPMread : true, repairserver : Q > CF } .

rl [B2a] :
   { (SPMPacket(0) from Q to Q')
     < Q' : RSreceiver | SPMread : true > CF }
   =>
   { < Q' : RSreceiver | repairserver : Q > CF } .

rl [B2b] :
   { (SPMPacket(NZN) from Q to Q')  < Q'' : RandomNGen | seed : N >
   < Q' : RSreceiver | SPMread : true, fastRepairFlag : X, readNextSeq : NZN',
                       suppressTO : R, dataBuffer : MC, dataInfo : DI > CF }
   =>
   { < Q'' : RandomNGen | seed : if X then N else
                                 repeatRandom(N, NZN minus
                                 max(NZN' minus 1, highestSeqNo(MC),
                                 highestNzNAKSeqNo(DI))) fi >
   < Q' : RSreceiver |  repairserver : Q,
                        dataInfo : updateDI(DI, X,
```

```
                                        max(NZN', highestSeqNo(MC) + 1,
                                            highestNzNAKSeqNo(DI) + 1),
                                        NZN, R, N) > CF } .

op updateDI : DataInfo Bool Nat Nat Time Nat -> DataInfo .
*** Usage: updateDI(dataInfo, fastRepairFlag, lowSeqNo, highSeqNo,
***                 suppressTO, randomSeed)

eq updateDI(DI, X, N, N', R, N'') =
     if N' < N then DI
     else (info(N, suppr(X, R, N'')), noTimeValue, 1)
           updateDI(removeSeqNo(DI, N), X, N + 1, N', R,
                    if X then N'' else random(N'') fi))
     fi .

*** Finds new suppression timer value:
op suppr : Bool Time Nat -> Time .
*** Usage: suppr(fastRepairFlag, suppressTO, randomSeed)
eq suppr(true, R, N) = 0 .
eq suppr(false, R, N) = random(N) % (((R * 3) / 2) plus 1) .
```

The function **removeSeqNo** removes repair information about the packet with the given sequence number. The function **highestSeqNo** finds the highest sequence in the data buffer. The function **highestNzNAKSeqNo** finds the highest sequence number with NAK count greater than 0. The function **_in_** decides whether there is repair information about the packet with the given sequence number. The definition of these functions is straightforward and is omitted in this exposition.

When a received data packet has been seen before—that is, it is either already delivered to the receiver application, in which case the sequence number is smaller that the **readNextSeq** value, or it is in the **dataBuffer** of the receiver—the timers triggering repairs are canceled:

```
crl [B3a] :
    { (dataPacket(Q, NZN, R, X', Y, Z) from Q' to Q'')
    < Q'' : RSreceiver | readNextSeq : NZN', dataBuffer : MC,
                         dataInfo : DI >  CF }
    =>
    { < Q'' : RSreceiver | dataInfo : cancelTimers(DI, NZN) > CF }
    if NZN < NZN' or NZN seqNoIn MC .

op cancelTimers : DataInfo NzNat -> DataInfo .
eq cancelTimers(none, NZN) = none .
eq cancelTimers((info(NZN, DT, DT', N) DI), NZN) =
      (info(NZN, noTimeValue, noTimeValue, N) DI) .
ceq cancelTimers(DI, NZN) = DI if not (NZN in DI) .

op _seqNoIn_ : NzNat MsgConf -> Bool .
eq NZN seqNoIn none = false .
eq NZN seqNoIn (NEMC NEMC') = (NZN seqNoIn NEMC) or (NZN seqNoIn NEMC') .
eq NZN seqNoIn dataPacket(Q, NZN', R, X, Y, Z) = (NZN == NZN') .
```

The error recovery for packets found missing when a new data packet is received is similar to the case when SPM packets are received. In addition, if the sequence number of the received packet equals **readNextSeq**, then this packet should be sent to the receiver application together with all subsequent packets until the next sequence number gap in received packets occurs.

To avoid passing on data packets to the receiver applications without ordering them, all the packets are sent in a message of the form

toApplication(*recvr*, dataPacket(*id*, n, ...) cat dataPacket(*id*$'$, $n+1$, ...)
cat ... cat dataPacket(*id*$''$, $n+m$, ...)).

The packets sent to the receiver application are out of the system, and should be removed from the sender's buffer and error recovery information store. The function toApp below sends the packet to the application, and rmDelivered and remove flush out the appropriate packets and error recovery information:

```
crl [B3b] :
    { (dataPacket(Q''', NZN, R, X, Y, Z) from Q to Q')
    < Q'' : RandomNGen | seed : N >
    < Q' : RSreceiverAlone | fastRepairFlag : X',
                             readNextSeq : NZN', suppressTO : R',
                             dataBuffer : MC, dataInfo : DI > CF }
    =>
    { < Q'' : RandomNGen | seed : (if X' then N else
                                   repeatRandom(N, (NZN minus 1) monus
                                                   max(NZN' minus 1,
                                                      highestNzNAKSeqNo(DI),
                                                      highestSeqNo(MC))) fi) >
    < Q' : RSreceiverAlone | readNextSeq : max(NZN', lastSeqNo(toApp(
                                             (dataPacket(Q''', NZN, R, X, Y, Z)
                                              MC), NZN')) + 1),
                             dataBuffer : rmDelivered(MC
                                             dataPacket(Q''', NZN, R, X, Y, Z), NZN'),
                             dataInfo : remove(updateDI(cancelTimers(DI, NZN),
                                           X', max(NZN', highestSeqNo(MC) + 1,
                                                   highestNzNAKSeqNo(DI) + 1),
                                           NZN minus 1, R', N),
                                        lastSeqNo(toApp(MC
                                             dataPacket(Q''', NZN, R, X, Y, Z),
                                                        NZN'))) >
    toApp((dataPacket(Q''', NZN, R, X, Y, Z) MC), NZN', Q') CF }
    if NZN' <= NZN and not (NZN seqNoIn MC) .

msg toApplication : Oid Msg -> Msg .
op _cat_ : MsgConf MsgConf -> MsgConf [assoc] .
eq MC cat none = MC .
eq none cat MC = MC .
op _cat_ : Msg Msg -> Msg [assoc] .
*** The following function defines which packets to send the application:
op toApp : MsgConf NzNat Oid -> MsgConf .
*** which calls the following workhorse:
op toApp : MsgConf NzNat -> MsgConf .
*** toApp(dataPackets, readNextSeq)
eq toApp(MC, NZN, Q) =  if toApp(MC, NZN) =/= none
```

```
                        then toApplication(Q, toApp(MC, NZN)) else none fi .
ceq toApp(MC, NZN) = (none).EmptyConf if not (NZN seqNoIn MC) .
eq toApp((dataPacket(Q, NZN, R, X, Y, Z) MC), NZN) =
     (dataPacket(Q, NZN, R, X, Y, Z) cat toApp(MC, NZN + 1)) .

*** The following function removes the dataPackets sent to the application.
op rmDelivered : MsgConf Nat -> MsgConf .
op lastSeqNo : MsgConf -> Nat .
eq lastSeqNo(none) = 0 .
eq lastSeqNo(dataPacket(Q, NZN, R, X, Y, Z)) = NZN .
eq lastSeqNo(MC cat dataPacket(Q, NZN, R, X, Y, Z)) = NZN .
eq rmDelivered(MC, N) = rmUpTo(MC, lastSeqNo(toApp(MC, N))) .
op rmUpTo : MsgConf Nat -> MsgConf .
*** Removes all dataPackets up to the one with seqNo = 2nd argument
eq rmUpTo(none, N) = none .
eq rmUpTo(MC, 0) = MC .
eq rmUpTo(NEMC NEMC', NZN) = rmUpTo(NEMC, NZN) rmUpTo(NEMC', NZN) .
eq rmUpTo(dataPacket(Q, NZN, R, X, Y, Z), NZN') =
      if NZN <= NZN' then none else dataPacket(Q, NZN, R, X, Y, Z) fi .

*** Removes from dataInfo all the elements with seqNo <= 2nd argument:
op remove : DataInfo Nat -> DataInfo .
eq remove(none, N) = none .
ceq remove(DI DI', N) = remove(DI, N) remove(DI', N)
    if DI =/= none and DI' =/= none .
eq remove(info(NZN, DT, DT', N), N') =
      if NZN <= N' then none else info(NZN, DT, DT', N) fi .
```

A repair request for a missing data packet is sent upstream when the suppression timer for the packet expires, and the retransmission timer is started. The receiver gives up if it has unsuccessfully requested a repair 48 times:

```
rl [B5] :
   { < Q : RSreceiver | readNextSeq : NZN, fastRepairFlag : X,
                        dataBuffer : MC, repairserver : Q',
                        dataInfo : (info(NZN', 0, DT, N) DI), retransTO : R > CF }
   =>
   { if (NZN' seqNoIn MC) or (NZN' < NZN)
     then (< Q : RSreceiver | dataInfo : (info(NZN', noTimeValue, DT, N) DI) > CF)
     else (if 48 < N then ERROR
           else (< Q : RSreceiver | dataInfo :
                                  (info(NZN', noTimeValue, R, N) DI) >
                 send(NAKPacket(NZN', N, X), Q, Q') CF) fi) fi } .

op ERROR : -> Configuration .
```

A new repair session is started when the retransmission timer for a (lost) data packet expires:

```
rl [B6] :
   { < Q : RandomNGen | seed : N >
   < Q' : RSreceiver | readNextSeq : NZN, suppressTO : R,
```

```
                      fastRepairFlag : X, dataBuffer : MC,
                      dataInfo : (info(NZN', DT, 0, N') DI) > CF }
   =>
   { if (NZN' < NZN) or (NZN' seqNoIn MC) then
      (< Q : RandomNGen | >
       < Q' : RSreceiver | dataInfo : (info(NZN', DT, noTimeValue, N) DI) > CF)
      else (< Q : RandomNGen | seed : (if X then N else random(N) fi) >
            < Q' : RSreceiver | dataInfo :
                      (info(NZN', suppr(X, R, N), noTimeValue, N' + 1) DI) >
            CF) fi } .
```

The receiver suppresses the current repair request by turning off its suppression timer and turning on its retransmission timer when it receives a relevant `NAKPacket` from its repair server:

```
rl [B7a] :
   { (NAKPacket(NZN, N, X) from Q to Q')
   < Q' : RSreceiver | readNextSeq : NZN', retransTO : R, dataBuffer : MC,
                       dataInfo : (info(NZN, DT, DT', N') DI) > CF }
   =>
   { CF (if (NZN seqNoIn MC) or (N < N') or (NZN < NZN')
         then < Q' : RSreceiver | >
         else < Q' : RSreceiver | dataInfo : (info(NZN, DT, R, N) DI) >
         fi) } .

crl [B7b] :
    { (NAKPacket(NZN, N, X) from Q to Q')
      < Q' : RSreceiver | readNextSeq : NZN', retransTO : R,
                          dataInfo : DI > CF }
    =>
    { < Q' : RSreceiver | dataInfo : (if NZN' <= NZN then
                                        (info(NZN, noTimeValue, R, N) DI)
                                      else DI fi) > CF }
    if not (NZN in DI) .
```

(Only) the nominee receiver should be in fast repair mode:

```
rl [B10B11] :
   { (NAMPacket(X) from Q' to Q) < Q : RSreceiver |  >  CF }
   =>
   { < Q : RSreceiver | isNominee : X, fastRepairFlag : X > CF } .
```

Finally, when the receiver in the RTT part of the protocol finds a new RTT value, it sends a method call **newRTT**(*newSourceRTT*, *newMaxUpRTT*, *objectId*) to itself, to be treated by the RS part of the protocol:

```
rl [B13a] :
   { newRTT(R, DT, Q)
   < Q : RSreceiver | smoothedRTT : I, rttVariance : I', suppressTO : R' > CF }
   =>
   { if I == 0 then
```

```
    (< Q : RSreceiver | smoothedRTT : R, rttVariance : R / 4,
                        retransTO : R plus R,
                        suppressTO : (if DT =/= noTimeValue then DT
                                      else R' fi) > CF)
  else
    (< Q : RSreceiver | smoothedRTT : ((R minus I) / 8) + I,
                        rttVariance : I' + ((abs(R minus I) minus I') / 4),
                        retransTO : (((R minus I) / 8) + I) +
                              (4 * (abs(R minus I) minus I') / 4),
                        suppressTO : (if DT =/= noTimeValue then DT
                                      else R' fi) > CF) fi } .

rl [B13b] :
   { newRTT(noTimeValue, DT, Q)
     < Q : RSreceiver | suppressTO : R >  CF }
   =>
   { < Q : RSreceiver | suppressTO :
                        (if DT =/= noTimeValue then DT else R fi) >  CF } .
```

Repair Server Protocol

A repair server has a cache in which it stores some of the data packets it has seen, as well as information about the repairs it is involved in. It tries to repair lost data packets in the same way that a receiver does. In addition, it sends NAKPackets to its children notifying them that a repair is requested.

Original data packets are stored in the cache and are subcast downstream. Repair data packets are stored and are subcast downstream only if a NAK is pending for the packet. A repair request from a child is treated by subcasting the requested data packet if it is in the cache, and by starting its own repair procedure for the packet otherwise.

The following data type is used to store a limited number of data packets in a message list. Our specification is based on the (unstated in the informal specification, but reasonable) assumption that data packets are stored in the buffer in reception order, not in sequence number order:

```
sort BoundedBuffer .
op bBuf : MsgList Nat NzNat -> BoundedBuffer .
*** Usage: bBuf(Buffer, Size, bound)
op add : BoundedBuffer Msg -> BoundedBuffer .
op _seqNoIn_ : NzNat BoundedBuffer -> Bool .
op _seqNoIn_ : NzNat MsgList -> Bool .

eq add(bBuf(M + ML, NZN, NZN), M') = bBuf(ML + M', NZN, NZN) .
ceq add(bBuf(ML, N, NZN), M) = bBuf(ML + M, N + 1, NZN) if N < NZN .
eq NZN seqNoIn bBuf(ML, N, NZN') = NZN seqNoIn ML .
eq NZN seqNoIn (nil).MsgList = false .
eq NZN seqNoIn dataPacket(Q, NZN', R, X, Y, Z) + ML =
     NZN == NZN' or NZN seqNoIn ML .
```

The repair server stores information about repairs in terms of the form

$$\texttt{nakState}(seqNo, NAKcount, downNAKcount, NAKpending, supprTimer, retransTimer),$$

where *seqNo* is the sequence number of the data packet, *NAKcount* is the upstream NAK count (which roughly corresponds to the number of upstream repair requests), *downNAKcount* is the downstream NAK count used in communication with the children, *NAKpending* is true when a child is waiting for the repair of the packet, and *supprTimer* and *retransTimer* are timer values which are used for upstream repair requests in the same way they were used in the receiver protocol, namely by suppressing the repair request in case the parent or a sibling also makes the same request, and by giving the time before another repair is attempted. The union of such NAK states is given by juxtaposition:

```
sort NAKstates .
op none : -> NAKstates .
op __ : NAKstates NAKstates -> NAKstates [assoc comm id: (none).NAKstates] .
op nakState : NzNat Nat Nat Bool DefTime DefTime -> NAKstates .
```

The repair server classes for the repair service part of the protocol are declared as follows:

```
class RSrepairserver |
      SPMread : Bool,             *** SPM packet read?
      maxSeqRcvd : Nat,
      smoothedRTT : MachineInt,
      rttVariance : MachineInt,
      retransTO : Time,
      suppressTO : Time,
      SPMWaitTimer : DefTime,
      NAKStates : NAKstates,      *** info about repairs
      dataBuffer : BoundedBuffer . *** data packet cache
subclass RSrepairserver < Repairserver .

class RSrepairserverAlone .
subclass RSrepairserverAlone < RSrepairserver .
```

Some operations in the combined protocol can only take place after an `SPMPacket` is read. The following rule sets the `SPMread` attribute value to `true` when an `SPMPacket` is seen, but without consuming the `SPMPacket`, which will be further treated by rule `C3`:

```
vars NS NS' : NAKstates .
var BB : BoundedBuffer .

rl [C1init] :
   { (SPMPacket(N) from Q to Q')
   < Q' : RSrepairserverAlone | SPMread : false > CF }
   =>
   { (SPMPacket(N) from Q to Q')
   < Q' : RSrepairserverAlone | SPMread : true > CF } .
```

The expiration of the `SPMTimer` indicates that no packets are sent by the sender or that this repair server is bypassed, in which case the "protocol state should be destroyed:"

```
rl [C2] :
   { < Q : RSrepairserver | SPMWaitTimer : 0 > CF } => { ERROR } .
```

The righthand side of the rule `C2` could instead be `{ CF }` if only the given repair server should be erased.

As before, `SPMPacket`s are used to detect the loss of data packets. No repair is undertaken if the repair server has yet to receive a data packet (that is, when `maxSeqRcvd` has the value 0). Otherwise, for all sequence numbers from `maxSeqRcvd` + 1 to the sequence number in the currently treated `SPMPacket`, the error recovery is performed for lost packets which do not yet have a NAK state in the object's `NAKStates` buffer. For each of these, a new NAK state is added by the function `updateNAKstates`, and a `NAKPacket` is sent to all its children by the function `newNAKPackets`. Furthermore, the received `SPMPacket` itself is forwarded to the children:

```
rl [C3] :
   { (SPMPacket(N) from Q to Q')
   < Q'' : RandomNGen | seed : N' >
   < Q' : RSrepairserver | SPMread : true, children : OS,  maxSeqRcvd : N'',
                           suppressTO : R, NAKStates : NS >  CF }
   =>
   { < Q'' : RandomNGen | seed : repeatRandom(N', noOfNewNAKs(NS, N'', N)) >
   < Q' : RSreceiver | repairserver : Q, SPMWaitTimer : 60000,
                       NAKStates : updateNAKstates(NS, N'', N, N', R) >
   multiSend(SPMPacket(N), Q', OS)
   newNAKPackets(NS, N'', N, Q', OS) CF } .
```

`noOfNewNAKs`(ns, n, n') returns 0 if $n = 0$ and otherwise gives the number of sequence numbers in the interval $n + 1$ to n' for which there is no NAK state in ns:

```
op noOfNewNAKs : NAKstates Nat Nat -> Nat .
eq noOfNewNAKs(NS, 0, N) = 0 .
eq noOfNewNAKs(NS, NZN, N) =
     if N <= NZN then 0
     else (if (NZN in NS) then 0 else 1 fi) + noOfNewNAKs(NS, NZN + 1, N) fi .

op _in_ : NzNat NAKstates -> Bool .
eq NZN in (none).NAKstates = false .
ceq NZN in (NS NS') = (NZN in NS) or (NZN in NS')
                       if NS =/= none and NS' =/= none .
eq NZN in nakState(NZN', N, N', X, DT, DT') = (NZN == NZN') .
```

`updateNAKstates`(ns, n, n', n'', r) updates the NAK state ns with new NAK states for sequence numbers between $n + 1$ and n' which are not already in ns. Again, $n = 0$ leaves ns unchanged. n'' is the seed to be used by the random number generator, and r is the value of `suppressTO`:

```
op updateNAKstates : NAKstates Nat Nat Nat Time -> NAKstates .
eq updateNAKstates(NS, 0, N, N', R) = NS .
eq updateNAKstates(NS, NZN, N, N', R) =
     if N <= NZN then NS else
       updateNAKstates(if (NZN in NS) then NS else
               (nakState(NZN, 1, 1, true, random(N') % (((R * 3) / 2) plus 1),
                         noTimeValue) NS) fi,
                       NZN + 1, N, random(N'), R)
     fi .
```

newNAKPackets(ns, n, n', q, os) sends NAKPackets to the children os from q for the sequence numbers in the interval $n+1$ to n' for which there is no NAK state in ns:

```
op newNAKPackets : NAKstates Nat Nat Oid OidSet -> Configuration .
eq newNAKPackets(NS, 0, N, Q, OS) = none .
eq newNAKPackets(NS, NZN, N, Q, OS) =
      if N <= NZN then (none).EmptyConf else
        (newNAKPackets(NS, NZN + 1, N, Q, OS)
         (if (NZN in NS) then none
          else multiSend(NAKPacket(NZN, 1, false), Q, OS) fi))
      fi .
```

A received data packet which is not stored in the cache (the condition `not (NZN seqNoIn BB)` below) is added to the `dataBuffer` cache of the repair server, and is subcast to its children if it is an original packet (the condition `not Z` below means that the retransmission flag of the packet is not set), or if a NAK is pending for the sequence number. In addition, repairs of the lost data packets are initiated as in the previous case and are handled by the functions `newNAKPackets` and `updateNAKstates`:

```
crl [C4a] :
    { (dataPacket(Q, NZN, R, X, Y, Z) from Q' to Q'')
    < Q'' : RSrepairserver | dataBuffer : BB > CF }
    =>
    { < Q'' : RSrepairserver | > CF }
    if NZN seqNoIn BB .

crl [C4b] :
    { (dataPacket(Q, NZN, R, X, Y, Z) from Q' to Q'')
    < Q''' : RandomNGen | seed : N >
    < Q'' : RSrepairserver | children : OS, maxSeqRcvd : N', dataBuffer : BB,
                             NAKStates : NS, suppressTO : R' > CF }
    =>
    { < Q'' : RSrepairserver | maxSeqRcvd : max(NZN, N'),
                               dataBuffer :
                                  add(BB, dataPacket(Q, NZN, R, X, Y, Z)),
                               NAKStates :
                                 (if (N' == 0) then
                                         cancelTimersAndSetFalse(NZN, NS)
                                  else updateNAKstates(cancelTimersAndSetFalse(NZN, NS),
                                                       N', NZN minus 1, N, R)
                                  fi) >
    < Q''' : RandomNGen | seed :
                          repeatRandom(N, noOfNewNAKs(NS, N', NZN minus 1)) >
    (if ((not Z) or NAKpending(NS, NZN))
     then multiSend(dataPacket(Q, NZN, R, X, Y, Z), Q'', OS)
     else none fi)
    newNAKPackets(NS, N', NZN minus 1, Q'', OS)  CF }
    if not (NZN seqNoIn BB) .

op NAKpending : NAKstates NzNat -> Bool .
*** Checks both if NAK state exists and then whether the NAK is pending:
```

```
ceq NAKpending(NS, NZN) = false if not(NZN in NS) .
eq NAKpending(nakState(NZN, N, N', X, DT, DT') NS, NZN) = X .

op cancelTimersAndSetFalse : NzNat NAKstates -> NAKstates .
*** Cancels the timers and sets NAK pending to false.
eq cancelTimersAndSetFalse(NZN, nakState(NZN, N, N', X, DT, DT') NS) =
      nakState(NZN, N, N', false, noTimeValue, noTimeValue) NS .
ceq cancelTimersAndSetFalse(NZN, NS) = NS if not NZN in NS .
```

A `NAKPacket` with count n received from the *upstream* repair server indicates that repair number n is attempted upstream. This means that the nth repair attempt is undertaken, and should be forwarded to the children if their NAK count is smaller than n or the NAK state does not exist. Furthermore, the retransmission timer should be reset for round $n + 1$. The `NAKPacket` comes from the upstream repair server of this object if the sender of the packet has the same value as the object's `repairserver` attribute:

```
*** Packet in cache; don't worry about repairs:
crl [C7a] :
    { (NAKPacket(NZN, N, X) from Q to Q')
    < Q' : RSrepairserver | repairserver : Q, dataBuffer : BB > CF }
    =>
    { < Q' : RSrepairserver | > CF }
    if NZN seqNoIn BB .

*** Packet not in cache; update and forward if higher NAK count received:
crl [C7b] :
    { (NAKPacket(NZN, N, X) from Q to Q')
    < Q' : RSrepairserver | repairserver : Q, dataBuffer : BB, retransTO : R,
                            NAKStates : (nakState(NZN, N', N'', Y, DT, DT') NS),
                            children : OS > CF }
    =>
    { < Q' : RSrepairserver | NAKStates : (nakState(NZN, max(N, N'),
                                                 if N' < N then N else N'' fi,
                                                    Y, noTimeValue, R) NS) >
    (if N' < N then multiSend(NAKPacket(NZN, N, false), Q', OS)
     else none fi) CF }
    if not (NZN seqNoIn BB) .

*** Packet not in cache, and no NAK state:
crl [C7c] :
    { (NAKPacket(NZN, N, X) from Q to Q')
    < Q' : RSrepairserver | repairserver : Q, dataBuffer : BB, retransTO : R,
                            children : OS, NAKStates : NS > CF }
    =>
    { < Q' : RSrepairserver | NAKStates : (nakState(NZN, N, N, true,
                                                     noTimeValue, R) NS) >
    multiSend(NAKPacket(NZN, N, false), Q', OS) CF }
    if not (NZN seqNoIn BB or NZN in NS) .
```

The following rules `C8a` to `C8e` treat unicast `NAKPacket`s from a child of the repair server object, as seen by the sender `Q` being also an element of the current object's `children` attribute. Essentially, if the data packet exists in the cache, it is subcast to the children. Otherwise, if the received NAK count is greater than the object's NAK count, it must start a new repair, and if the received NAK count is greater than the downstream NAK count, the other children should be aware of this current repair request. Note that if the `NAKPacket` has the fast repair flag set, the possible upstream repair should be undertaken with no delay (like the sender's repair when its `fastRepairFlag` is set). The following rules treat the different cases, depending on whether the requested data packet is in the object's cache, whether a NAK state exists for the requested packet, whether the fast repair flag of the packet is set, and depending on the NAK count values:

```
*** NAK does not exists, fastRepFlag is true, data not buffered:
crl [C8a] :
    { (NAKPacket(NZN, N, true) from Q to Q')
    < Q' : RSrepairserver | repairserver : Q'', children : Q OS, retransTO : R,
                            dataBuffer : BB, NAKStates : NS >  CF }
    =>
    { < Q' : RSrepairserver | NAKStates :
                              (nakState(NZN, N, N, true, noTimeValue, R) NS) >
    multiSend(NAKPacket(NZN, N, false), Q', Q OS)
    send(NAKPacket(NZN, N, true), Q', Q'')  CF }
    if not (NZN seqNoIn BB or NZN in NS) .

*** No NAK exists, data not buffered, fastRepFlag false:
crl [C8b] :
    { (NAKPacket(NZN, N, false) from Q to Q')  < Q'' : RandomNGen | seed : N' >
    < Q' : RSrepairserver | children : Q OS, suppressTO : R, dataBuffer : BB,
                            NAKStates : NS >  CF }
    =>
    { < Q'' : RandomNGen | seed : random(N') >
    < Q' : RSrepairserver | NAKStates :
                             (nakState(NZN, N, N, true,
                                    random(N') % (((3 * R) / 2) plus 1),
                                    noTimeValue) NS) >
    multiSend(NAKPacket(NZN, N, false), Q', Q OS) CF }
    if not (NZN seqNoIn BB or NZN in NS) .

*** Data not in buffer, NAK state exists:
crl [C8c] :
    { (NAKPacket(NZN, N, X) from Q to Q')
    < Q' : RSrepairserver | repairserver : Q'', children : Q OS,
                          NAKStates : (nakState(NZN, N', N'', Y, DT, DT') NS),
                          retransTO : R, dataBuffer : BB >  CF }
    =>
    { if (N' < N) then
        (< Q' : RSrepairserver | NAKStates :
                                 (nakState(NZN, N, N, true,
                                           if X then noTimeValue else DT fi,
                                           if X then R else DT' fi) NS) >
      multiSend(NAKPacket(NZN, N, false), Q', Q OS)
      (if X then send(NAKPacket(NZN, N, true), Q', Q'') else none fi) CF)
      else (if (N'' < N) then
              (< Q' : RSrepairserver | NAKStates :
                                       (nakState(NZN, N', N, Y, DT, DT') NS) >
```

```
              multiSend(NAKPacket(NZN, N, false), Q', Q OS) CF)
            else (< Q' : RSrepairserver | > CF)
            fi) fi}
    if not (NZN seqNoIn BB) .

*** The message is in the buffer, but the NAK state does not exist.
crl [C8d] :
    { (NAKPacket(NZN, N, X) from Q to Q')
    < Q' : RSrepairserver | children : Q OS, dataBuffer :
                             bBuf(ML + dataPacket(Q'', NZN, R, Y, Z, XX) + ML',
                                  N', NZN'),
                            NAKStates : NS >  CF }
    =>
    { < Q' : RSrepairserver | NAKStates :
                                (nakState(NZN, N, N, false, noTimeValue,
                                          noTimeValue) NS) >
    multiSend(dataPacket(Q'', NZN, R, Y, Z, true), Q', Q OS) CF }
    if not (NZN in NS) .

*** The sequence number exists in both data buffer and NAK state:
rl [C8e] :
   { (NAKPacket(NZN, N, X) from Q to Q')
   < Q' : RSrepairserver | children : Q OS,
                           dataBuffer :
                            bBuf(ML + dataPacket(Q'', NZN, R, Y, Z, XX) + ML', N''', NZN'),
                           NAKStates :
                             (nakState(NZN, N', N'', X', DT, DT') NS) >  CF }
   =>
   { if  N' < N then
        (< Q' : RSrepairserver | NAKStates :
                                  (nakState(NZN, N, N, false, DT, DT') NS) >
         multiSend(dataPacket(Q'', NZN, R, Y, Z, true), Q', Q OS) CF)
     else (< Q' : RSrepairserver | > CF) fi } .
```

A new repair is undertaken when the suppression timer expires for a packet with the NAK pending, unless the packet is cached:

```
rl [C11] :
   { < Q : RSrepairserver | dataBuffer : BB, NAKStates :
                              (nakState(NZN, N, N', X, 0, DT) NS),
                            retransTO : R, repairserver : Q' >  CF }
   =>
   { if (X and not (NZN seqNoIn BB)) then
       (< Q : RSrepairserver | NAKStates :
                                (nakState(NZN, N, N', X, noTimeValue, R) NS) >
       send(NAKPacket(NZN, N, false), Q, Q') CF)
     else
      (< Q : RSrepairserver | NAKStates :
                                (nakState(NZN, N, N', X, noTimeValue,DT) NS) >  CF)
     fi } .
```

Similarly, the NAK count is increased, the suppression timer is set to initiate a new repair in a short time, and a `NAKPacket` is sent to each child, when the retransmission timer expires for a packet needing repair:

```
rl [C12] :
   { < Q : RandomNGen | seed : N >
   < Q' : RSrepairserver | suppressTO : R, dataBuffer : BB, children : OS,
                           NAKStates : (nakState(NZN, N', N'', X, DT, 0) NS) >
   CF }
   =>
   { if (X and not (NZN seqNoIn BB)) then
       (< Q : RandomNGen | seed : random(N) >
        < Q' : RSrepairserver | NAKStates : (nakState(NZN, N' + 1, N'', X,
                                              random(N) % (((R * 3) / 2)
                                                    plus 1), noTimeValue) NS) >
        multiSend(NAKPacket(NZN, N' + 1, false), Q', OS) CF)
     else
      (< Q : RandomNGen | >
       < Q' : RSrepairserver | NAKStates :
                             (nakState(NZN, N', N'', X, DT, noTimeValue) NS) >
       CF) fi } .
```

Finally, new RTT values are treated as in the case of receivers:

```
rl [C13a] :
   { newRTT(R, DT, Q)
   < Q : RSrepairserver | smoothedRTT : I, rttVariance : I',
                          suppressTO : R' >  CF }
   =>
   { if I == 0 then
       (< Q : RSrepairserver | smoothedRTT : R, rttVariance : R / 4,
                               retransTO : R plus R,
                               suppressTO : (if DT =/= noTimeValue then DT
                                             else R' fi) > CF)
     else
       (< Q : RSrepairserver | smoothedRTT : ((R minus I) / 8) + I,
                               rttVariance : I' +
                                           ((abs(R minus I) minus I') / 4),
                               retransTO : ((R minus I / 8) + I) +
                                 (4 * (abs(R minus I) minus I') /  4),
                               suppressTO : (if DT =/= noTimeValue then DT
                                             else R' fi) > CF) fi } .

rl [C13b] :
   { newRTT(noTimeValue, DT, Q)
     < Q : RSrepairserver | suppressTO : R >  CF }
   =>
   { < Q : RSrepairserver | suppressTO :
                            (if DT =/= noTimeValue then DT else R fi) >  CF } .
```

Behavior in Time

Time acts on RS objects as before. That is, the function `delta` increases the clocks and decreases the timers with the elapse of time, and the function `mte` makes sure that time progress stops whenever a timer expires. The functions `delta` and `mte` must be extended to accommodate multiple timers in the `dataInfo` attribute of the receivers and the `NAKStates` attribute of the repair servers:

```
eq mte(< Q : RSsenderAlone | SPMTimer : DT', sendDataTimer : DT'' >) =
      default-min(DT', DT'') .
eq delta(< Q : RSsenderAlone | clock : R', SPMTimer : DT',
                               sendDataTimer : DT'' >, R) =
      < Q : RSsenderAlone | clock : R' plus R, SPMTimer : DT' default-monus R,
                            sendDataTimer : DT'' default-monus R > .

eq mte(< Q : RSreceiverAlone | dataInfo : DI >) = mte(DI) .
op mte : DataInfo -> TimeInf .
eq mte((none).DataInfo) = INF .
ceq mte(DI DI') = min(mte(DI), mte(DI')) if DI =/= none and DI' =/= none .
eq mte(info(NZN, DT, DT', N)) = default-min(DT, DT') .

eq delta(< Q : RSreceiverAlone | dataInfo : DI >, R) =
      < Q : RSreceiverAlone | dataInfo : delta(DI, R) > .
op delta : DataInfo Time -> DataInfo .
eq delta((none).DataInfo, R) = (none).DataInfo .
ceq delta(DI DI', R) = delta(DI, R) delta(DI', R)
    if DI =/= none and DI' =/= none .
eq delta(info(NZN, DT, DT', N), R) =
      info(NZN, DT default-monus R, DT' default-monus R, N) .

eq mte(< Q : RSrepairserverAlone | SPMWaitTimer : DT, NAKStates : NS >) =
      default-min(DT, mte(NS)) .
op mte : NAKstates -> TimeInf .
eq mte((none).NAKstates) = INF .
ceq mte(NS NS') = min(mte(NS), mte(NS')) if NS =/= none and NS' =/= none .
eq mte(nakState(NZN, N, N', X, DT, DT')) = default-min(DT, DT') .

eq delta(< Q : RSrepairserverAlone | SPMWaitTimer : DT, NAKStates : NS >, R) =
      < Q : RSrepairserverAlone | SPMWaitTimer : DT default-monus R,
                                  NAKStates : delta(NS, R) > .
op delta : NAKstates Time -> NAKstates .
eq delta((none).NAKstates, R) = (none).NAKstates .
ceq delta(NS NS', R) = delta(NS, R) delta(NS', R)
                       if NS =/= none and NS' =/= none .
eq delta(nakState(NZN, N, N', X, DT, DT'), R) =
      nakState(NZN, N, N', X, DT default-monus R, DT' default-monus R) .
endtom)
```

5.2.10 The Application Layer

The stand-alone RS protocol and the combined protocol receive data blocks from the application layer of the protocol, and should relay data packets (in order) to the receiver applications. We define here a simplistic model of the sender and receiver at the application level.

The sender application object stores data blocks with delays as a message list in its attribute `dataBlocksToBeSent`, and sends data blocks ready for delivery to the sender object in the protocol. The application-level receiver object stores in its attribute `dataRead` the concatenation of the data packets it has received from its associated receiver in the protocol, given by its `receiver` attribute:

```
(tomod AER-APPLICATIONS is
including AER-RS .

class SenderApplication | dataBlocksToBeSent : MsgList .

rl [appSend] :
   { < Q : SenderApplication | dataBlocksToBeSent : dly(M, 0) + ML > CF }
   =>
   { < Q : SenderApplication | dataBlocksToBeSent : ML > M CF } .

eq mte(< Q : SenderApplication | dataBlocksToBeSent : ML >) =
      leastDly(ML) .
eq delta(< Q : SenderApplication | dataBlocksToBeSent : ML >, R) =
     < Q : SenderApplication | dataBlocksToBeSent : delta(ML, R) > .

class RecvrApplication | receiver : Oid, dataRead : MsgConf .

rl [appRecv] :
   { toApplication(Q, M)
   < Q' : RecvrApplication | receiver : Q, dataRead : MC >  CF }
   =>
   { < Q' : RecvrApplication | receiver : Q, dataRead : MC cat M > CF } .

eq mte(< Q : RecvrApplication | >) = INF .
eq delta(< Q : RecvrApplication | >, R) = < Q : RecvrApplication | > .
endtom)
```

In the following initial state, with the same topology as `RTTstate2` and `NOMstate2` (and where the links are substituted by '`[ ... ]`' for brevity of exposition), the sender application wants to transmit two data blocks, labeled `'first` and `'second`, of sizes 20 and 1, respectively:

```
(tomod RS-APPLICATIONS is
including AER-APPLICATIONS .
[ ... ]
op RSstate2 : Nat -> System .
eq RSstate2(N) =
({ startRS('a)
 < 'SenderApp : SenderApplication | dataBlocksToBeSent :
                                    dly(dataBlock('first, 'a, 20, 5), 100) +
                                    dly(dataBlock('second, 'a, 1, 100), 800) >
 < 'RecvrAppB : RecvrApplication | receiver : 'b, dataRead : none >
 < 'RecvrAppE : RecvrApplication | receiver : 'e, dataRead : none >
 < 'RecvrAppF : RecvrApplication | receiver : 'f, dataRead : none >
 < 'RecvrAppG : RecvrApplication | receiver : 'g, dataRead : none >
 < 'a : RSsenderAlone | clock : 0, children : 'b 'c,
```

```
                      nextSeq : 1, SPMTimer : noTimeValue,
                      reTransBuf : none, unsentDataPackets : nil,
                      sendDataTimer : noTimeValue >
< 'b : RSreceiverAlone | repairserver : noOid, isNominee : false,
                        fastRepairFlag : false, readNextSeq : 1,
                        smoothedRTT : 0, rttVariance : 3000,
                        suppressTO : 100, retransTO : 6000,
                        dataBuffer : none, dataInfo : none, SPMread : false >
< 'c : RSrepairserverAlone | children : 'd 'e,  repairserver : noOid,
                            SPMread : false,
                            maxSeqRcvd : 0, SPMWaitTimer : noTimeValue,
                            smoothedRTT : 0, rttVariance : 3000,
                            suppressTO : 100, retransTO : 6000,
                            dataBuffer : bBuf(nil, 0, 10), NAKStates : none >
< 'd : RSrepairserverAlone | children : 'f 'g, repairserver : noOid,
                            SPMread : false,
                            maxSeqRcvd : 0, SPMWaitTimer : noTimeValue,
                            smoothedRTT : 0, rttVariance : 3000,
                            suppressTO : 100, retransTO : 6000,
                            dataBuffer : bBuf(nil, 0, 10), NAKStates : none >
< 'e : RSreceiverAlone | repairserver : noOid, isNominee : false,
                        fastRepairFlag : false, readNextSeq : 1,
                        smoothedRTT : 0, rttVariance : 3000,
                        suppressTO : 100, retransTO : 6000,
                        dataBuffer : none, dataInfo : none, SPMread : false >
< 'f : RSreceiverAlone | repairserver : noOid, isNominee : false,
                        fastRepairFlag : false, readNextSeq : 1,
                        smoothedRTT : 0, rttVariance : 3000,
                        suppressTO : 100, retransTO : 6000,
                        dataBuffer : none, dataInfo : none, SPMread : false >
< 'g : RSreceiverAlone | repairserver : noOid,  isNominee : false,
                        fastRepairFlag : false, readNextSeq : 1,
                        smoothedRTT : 0, rttVariance : 3000,
                        suppressTO : 100, retransTO : 6000,
                        dataBuffer : none, dataInfo : none, SPMread : false >
< 'random : RandomNGen | seed : N >
< 'env : Env | msgsFromEnv :   dly(NAMPacket(true) from 'd to 'f, 1444)
                              dly(newRTT(44, 56, 'b), 75)
                              dly(newRTT(56, 56, 'c), 89)
                              dly(newRTT(92, 48, 'e), 180)
                              dly(newRTT(114, 10, 'f), 362)
                              dly(newRTT(114, 10, 'g), 355)
                              dly(newRTT(104, 48, 'd), 200) >
[ ... ] }) .
```

Executing this example using Maude's default execution strategy should lead to a state where all the receiver applications, that is, the objects `'RecvrAppB`, `'RecvrAppE`, `'RecvrAppF`, and `'RecvrAppG`, have received all 21 data packets. Instead, we get the following result:

```
Maude> (rew- [3000] RSstate2(113) .)
rewrites: 867328 in 13020ms cpu (13030ms real) (66615 rewrites/second)
rewrite in RS-APPLICATIONS : RSstate2 .
result Error(ClockedSystem) : { ERROR } in time 17841
```

This error state must have been reached because of the inability to repair a data packet in less than 48 attempts (use case B5), since the elapsed time is less than 60000, which is the value the `SPMTimer` is set to. The connections should not break in our setting, so the execution seems to indicate that the protocol may be deficient. Indeed, a snapshot of the system after 2000 rewrites gives the following state[7]:

```
Maude> (rew- [2000] RSstate2(113) .)
rewrites: 810869 in 11950ms cpu (11960ms real) (67855 rewrites/second)
rewrite in RS-APPLICATIONS : RSstate2 (113) .
result Error(ClockedSystem) : { < 'e : RSreceiverAlone | repairserver : 'c,
    isNominee : false, dataInfo : (info(18, 61, noTimeValue, 41)
    info(19, noTimeValue, noTimeValue, 2) info(20,
    noTimeValue, noTimeValue, 1)), dataBuffer : (dataPacket('first, 19,
    5190, false, true, false) dataPacket('first, 20, 5195, false,
    false, true) dataPacket('second, 21, 5295, true, false, false)),
    [ ... ] >
  < 'a : RSsenderAlone | clock : 15792, children : 'b 'c,
    sendDataTimer : noTimeValue, unsentDataPackets : nil, reTransBuf :
    (MsgAndNAK(dataPacket('first, 1, 5100, true, false, false), 0)
    MsgAndNAK(dataPacket('first, 2, 5105, false, true, false), 0)
    [ ... ]
    MsgAndNAK(dataPacket('first, 20, 5195, false, false, true), 1)
    MsgAndNAK(dataPacket('second, 21, 5295, true, false, false), 0)),
    SPMTimer : 208, nextSeq : 22 >
  < 'SenderApp : SenderApplication | dataBlocksToBeSent : nil >
  < 'b : RSreceiverAlone | repairserver : 'a, isNominee : false,
    SPMread : true, dataInfo : none, dataBuffer : (none).Configuration,
    retransTO : 88, suppressTO : 56, rttVariance : 11, smoothedRTT : 44,
    readNextSeq : 22, fastRepairFlag : false >
  < 'c : RSrepairserverAlone | children : 'e 'd, repairserver : 'a,
    dataBuffer : bBuf(dataPacket('first, 5, 5120, false, true, true) +
    dataPacket('first, 11, 5150, false, true, true) +
    dataPacket('first, 17, 5180, false, true, true) + dataPacket(
    'first, 14, 5165, false, true, true) + dataPacket('first, 16,
    5175, false, true, true) + dataPacket('first, 8, 5135, false,
    true, true) + dataPacket('first, 13, 5160, false, true, true) +
    dataPacket('first, 15, 5170, false, true, true) + dataPacket(
    'first, 20, 5195, false, false, true) + dataPacket('first, 19,
    5190, false, true, true), 10, 10), NAKStates : [ ... ]  nakState(17,
    2, 2, false, noTimeValue, noTimeValue) nakState(18, 40, 40, true,
    noTimeValue, noTimeValue) nakState(19, 2, 2, false, noTimeValue,
    noTimeValue) nakState(20, 1, 1, false, noTimeValue, noTimeValue)),
    maxSeqRcvd : 21, [ ... ] >
  < 'env : Env | msgsFromEnv : none >
  < 'RecvrAppB : RecvrApplication | dataRead :
    (dataPacket('first, 1, 5100, true, false, false) cat
    dataPacket('first, 2, 5105, false, true, false) cat [ ... ]
    cat dataPacket('first, 20, 5195, false, false, false) cat
    dataPacket('second, 21, 5295, true, false, false)), receiver : 'b >
```

[7]Note that the choice of rules to apply may not be the same using Real-Time Maude timed rewrite and ordinary rewrite command. Therefore, we cannot just rewrite to, say, time 15000 using the timed rewrite command to execute the system.

```
  < 'RecvrAppE : RecvrApplication | dataRead :
    (dataPacket('first, 1, 5100, true, false, false) cat
    dataPacket('first, 2, 5105, false, true, false) cat [ ... ]
    cat dataPacket('first, 17, 5180, false, true, false)), receiver : 'e >
  [ ... ] } in time 15792
```

The receiver application `'RecvrAppB` has received all 21 data packets, and so have all the other receivers which have been removed from the output in this exposition. The receiver application `'RecvrAppE` has only received 17 packets, and should wait for the 18th packet. Its associated protocol node `'e` also has packet 18 missing, and has a dangerously high NAK count 41 for this packet! We continue to analyze how the next repair for this packet happens. Note first that the suppression timer of packet 18 in `'e` is 61, so that if the above resulting term is rewritten until time 15853, it should start its repair:

```
Maude> (trew [0] ({ [ ... ] } in time 15792) mode gm dti 1000 in time
        le 15853 .)
[ ... ]
Result: { < 'e : RSreceiverAlone | repairserver : 'c, dataInfo :
    (info(18, 0, noTimeValue, 41) [ ... ]),
    dataBuffer : (dataPacket('first, 19, 5190, false, true, false)
    dataPacket('first, 20, 5195, false, false, true) dataPacket(
    'second, 21, 5295, true, false, false)), readNextSeq : 18, [ ... ] >
  < 'c : RSrepairserverAlone | children : 'e 'd, repairserver : 'a,
    dataBuffer : bBuf(dataPacket('first, 5, 5120, false, true, true) +
    dataPacket('first, 11, 5150, false, true, true) + dataPacket(
    'first, 17, 5180, false, true, true) + dataPacket('first, 14,
    5165, false, true, true) + dataPacket('first, 16, 5175, false,
    true, true) + dataPacket('first, 8, 5135, false, true, true) +
    dataPacket('first, 13, 5160, false, true, true) + dataPacket(
    'first, 15, 5170, false, true, true) + dataPacket('first, 20,
    5195, false, false, true) + dataPacket('first, 19, 5190, false,
    true, true), 10, 10), retransTO : 112, suppressTO : 56, rttVariance
    : 14, smoothedRTT : 56, NAKStates :
    ( [ ... ] nakState(18, 40, 40, true, noTimeValue, noTimeValue)),
    maxSeqRcvd : 21, [ ... ] > [ ... ] }  in time 15853 .
```

We observe that the repair server `'c` does not have data packet 18 in its cache, probably overwritten by later repair requests. The repair server's NAK status for packet 18 is that both its suppression and retransmission timers are turned off, essentially implying that it does not try to repair packet 18. (This is probably because, as seen in use cases C8/C9, it had packet 18 when a request for it came, subcasted packet 18, and created a NAK state with the timers turned off.) We analyze how the NAK request will be handled, and it is enough to focus of on the objects `'e` and `'c` being in the above resulting state, and on the link between them:

```
Maude> (rew- [0] { < 'e : RSreceiverAlone | [ ... ] >
                   < 'c : RSrepairserverAlone | [ ... ] >
                   < 'ce : Link | [ ... ] >  } in time 15853 .)
rewrites: 4362 in 100ms cpu (110ms real) (43620 rewrites/second)
[ ... ]
result Error(ClockedSystem) : { < 'e : RSreceiverAlone |
```

```
    dataInfo : (info(18, noTimeValue, 166, 41) [ ... ]),
    readNextSeq : 18, [ ... ] >
  < 'c : RSrepairserverAlone | NAKStates :
    nakState(18, 41, 41, true, noTimeValue, noTimeValue) [ ... ]), [ ...] >
  < 'ce : Link | downMsgs : dly(NAKPacket(18, 41, false), 18),
    up : 'c, down : 'e, [ ... ] >
  send(NAKPacket(18, 41, false), 'c, 'd) } in time 15871
```

Object `'c` has received the NAK request 41 from `'e`, as acknowledged by the packets sent to `'e` and `'d`. However, as seen, `'c` still has not sent any repair request for this packet to its repair server, nor has it set its suppression or retransmission timer for such a repair, so it will not undertake this repair. Meanwhile, `'e` has set its retransmission timer in case the repair has not arrived, and will start another error recovery attempt in time 166 from now, eventually leading to its 48th repair attempt.

The anomalous behavior above can be traced back to the informally given original protocol as follows: When a NAK is received by a repair server from its child, use case C8 is entered, from which use case C9 is entered though the test `if (nakCount > NAK count for data packet seq)`, and from use case C9, control goes to use case C10 with `fastRepairFlag` and `newNakStateFlag` both false, and indeed no further action is taken. We will not analyze the informally given use cases further, but maybe the `newNakStateFlag` should have been set to `true` when the received NAK count is greater than the stored NAK count? Maybe this flag is superfluous?

Another Execution

Another test configuration, with three protocol nodes, was suggested by Mark Keaton to test the repair service protocol:

```
op RSstate3 : Nat -> System .
eq RSstate3(N) =
({ startRS('a)
 < 'SenderApp : SenderApplication | dataBlocksToBeSent :
                                    dly(dataBlock('first, 'a, 35, 5), 100) >
 < 'RecvrAppC : RecvrApplication | receiver : 'c, dataRead : none >
 < 'a : RSsenderAlone | clock : 0, children : 'b,
                        nextSeq : 1, SPMTimer : noTimeValue,
                        reTransBuf : none, unsentDataPackets : nil,
                        sendDataTimer : noTimeValue >
 < 'b : RSrepairserverAlone | children : 'c,  repairserver : noOid,
                              SPMread : false,
                              maxSeqRcvd : 0, SPMWaitTimer : noTimeValue,
                              smoothedRTT : 0, rttVariance : 3000,
                              suppressTO : 100, retransTO : 6000,
                              dataBuffer : bBuf(nil, 0, 10), NAKStates : none >
 < 'c : RSreceiverAlone | repairserver : noOid, isNominee : false,
                          fastRepairFlag : false, readNextSeq : 1,
                          smoothedRTT : 0, rttVariance : 3000,
                          suppressTO : 100, retransTO : 6000,
                          dataBuffer : none, dataInfo : none, SPMread : false >
 < 'random : RandomNGen | seed : N >
```

```
 < 'env : Env | msgsFromEnv :   dly(NAMPacket(true) from 'b to 'c, 2)
                                dly(newRTT(220, 220, 'b), 3)
                                dly(newRTT(210, 4, 'c), 4) >
 < 'ab : Link | up : 'a, down : 'b, bound : 10, propDelay : 100,
                linkSpeed : 1, downMsgs : nil, downSize : 0,
                upMsgs : nil, upSize : 0 >
 < 'bc : Link | up : 'b, down : 'c, bound : 10, propDelay : 1,
                linkSpeed : 10, downMsgs : nil, downSize : 0,
                upMsgs : nil, upSize : 0 >}) .
```

35 data packets should be sent from `'a` to `'c` via `'b`. The sending rate is one new packet every 5 ms, and the propagation delay of 100 ms in the link from `'a` to `'b` ensures that many packets will be lost along this link with bound 10. The other link is much faster and should not lose many packets. An execution gives the following result:

```
Maude> (rew- [2000] RSstate3(113) .)
rewrites: 748291 in 7910ms cpu (7930ms real) (94600 rewrites/second)
rewrite in RS-APPLICATIONS : RSstate3 (113) .
result Error(ClockedSystem) : { < 'a : RSsenderAlone | children : 'b,
    reTransBuf : (MsgAndNAK(dataPacket('first, 1, 5100, true, false, false), 0)
    [ ... ]
    MsgAndNAK(dataPacket('first, 35, 5270, false, false, true), 1)), [ ... ] >
  < 'b : RSrepairserverAlone | children : 'c, repairserver : 'a,
    dataBuffer : bBuf(dataPacket('first, 25, 5220, false, true, true) +
      dataPacket('first, 27, 5230, false, true, true) +
      dataPacket('first, 28, 5235, false, true, true) +
      dataPacket('first, 30, 5245, false, true, true) +
      dataPacket('first, 32, 5255, false, true, true) +
      dataPacket('first, 23, 5210, false, true, true) +
      dataPacket('first, 21, 5200, false, true, true) +
      dataPacket('first, 34, 5265, false, true, true) +
      dataPacket('first, 35, 5270, false, false, true) +
      dataPacket('first, 22, 5205, false, true, true), 10, 10),
    NAKStates : (nakState(11, 2, 2, false, noTimeValue, noTimeValue) [ ... ]
      nakState(22, 1, 1, false, noTimeValue, noTimeValue)
      nakState(23, 1, 1, false, noTimeValue, noTimeValue)
      nakState(25, 3, 3, false, noTimeValue, noTimeValue)
      nakState(27, 3, 3, false, noTimeValue, noTimeValue)
      nakState(28, 3, 3, false, noTimeValue, noTimeValue)
      nakState(30, 3, 3, false, noTimeValue, noTimeValue)
      nakState(32, 3, 3, false, noTimeValue, noTimeValue)
      nakState(34, 1, 1, false, noTimeValue, noTimeValue)
      nakState(35, 1, 1, false, noTimeValue, noTimeValue)),
    maxSeqRcvd : 35, [ ... ] >
  < 'c : RSreceiverAlone | repairserver : 'b, isNominee : true,
    dataInfo : (info(25, noTimeValue, noTimeValue, 3)
      info(27, noTimeValue, noTimeValue, 3) [ ... ]
      info(35, noTimeValue, noTimeValue, 1)),
    dataBuffer : (dataPacket('first, 25, 5220, false, true, true)
      dataPacket('first, 26, 5225, false, true, false) [ ... ]
      dataPacket('first, 35, 5270, false, false, true)),
    readNextSeq : 24, fastRepairFlag : true, [ ... ] >
```

```
< 'RecvrAppC : RecvrApplication | dataRead : (dataPacket
  ('first, 1, 5100, true, false, false) cat [ ... ] cat
  dataPacket('first, 22, 5205, false, true, true) cat
  dataPacket('first, 23, 5210, false, true, true)), receiver : 'c >
[ ... ] } in time 460103
```

This execution reveals another serious problem: Packet 24 has not been delivered to the receiver application, and is missing at both the receiver and the repair server. Furthermore, neither of them has any repair state for this packet, which is alarming since all "holes" should be repaired when a received SPM packet or data packet indicates that some packets are lost. A stepwise analysis similar to the previous case can be undertaken to find the problem. We will not give the details here, but only describe the result of the analysis leading to this undoubtedly strange behavior:

`'a` sends data packets to `'b`. The first 10 data packets enters the link, which then becomes full. However, `'a` continues to send packets, so the packets 11–23 are lost, and 24 is the next packet in the link which is not lost. When `'b` reads packet 24, it discovers the holes 11–23, and sends NAK packets for these to `'c` with NAK count 1. At the same time, it also subcasts packet 24 to `'c`. This makes 14 messages sent from `'b` to `'c` at the same time. Since the link can only take 10 packets, four packets are lost, among them data packet 24. Back in the link from `'a` to `'b` packet 25 is lost, and 26 arrives safely at `'b`, which then discovers the hole for 25 and sends a NAK for 25 to `'c`, as well as the data packet 26. `'c` then reads and stores the NAK packet for 25 (with NAK count 1), and then reads the packet 26. This was the golden opportunity to discover the hole for packet 24, since it will never be discovered by the repair server, which had the packet. Instead of discovering the hole at 24, `'c` goes backwards in use case B4 and finds that a NAK state with NAK count 1 exists for packet 25, and stops its look for holes. Therefore, no repair will be attempted for packet 24!

5.2.11 The Combined Protocol

In the full protocol, which combines the four parts of the protocol, the transitions modeling the treatment of a received packet and the sending of the data packets, must be given by transitions which combine the treatment of these cases in the subprotocols. For example, when a data packet is received by a receiver object, the receiver must update its window used to calculate the loss probability estimate (in the NOM protocol part), the rate control part must respond with a `ccmPacket` if it is the nominee, and the repair service part must check for lost data packets and initiate the repair of these. Similarly, the sending of an original data packet, which in our division of the tasks is performed by the repair service protocol, must be combined with the rate control algorithm to certify that a sending is allowed.

As mentioned earlier, the class `SenderCombined` is a subclass of `RTTsender`, `NOMsender`, `RCsender`, and `RSsender`, and therefore inherits (the union of) all the attributes of these classes, as well as the rewrite rules involving objects of these classes. Transitions which are shared between different parts of the protocol are put together "manually" into single transitions. The exception is that upon the discovery of a new RTT value, the object calls its own method `newRTT` (by sending a message to itself), to be treated by another part of the protocol. The reason is that it seems that combining these rules together would give some unnecessarily complicated rules. This approach of calling methods in the same object to be treated by other parts of the protocol, could also have been applied to the other cases, so that, say, a `dataPacket` is treated by, say, the repair service part of the protocol, which, upon reading this packet, calls its own methods nomDataPacket and rcDataPacket. In this way, the protocol would be defined entirely by its subparts without the need for "combined" rules. (The possible exception is combining the sending of a data packet with testing if the rate control allows for the sending of a packet, but this can be solved by an exchange of messages.)

The given solution with explicit combined rules is chosen because we think it is clearer and closer to the original specification. In addition, the events should take place simultaneously. If the method call approach is used, some actions may be performed between the treatment of a dataPacket and its associated nomDataPacket. This may or may not affect the behavior of the protocol, and can anyway be avoided in various ways. We think that it is not crucial for the updates to take place simultaneously with the associated discovery of new RTT values.

When to send data packets? The rate control protocol does not specify when data packets are to be sent, only whether a new data packet *can* be sent at the "current" time. There are various choices of when packets should be sent. They could be sent urgently in the sense that new packets are sent as early as possible. Or sending could be attempted at certain times? If so, should we test for sending after each the application of each instantaneous rule application, since the application of an instantaneous rule could change the sending condition? How long should the protocol wait if sending is not possible at the desired time? Another interval? The next time sending is possible?

Allowing the sending to happen at any time when the sending condition is true avoids overspecification of this part, and requires a nondeterministic tick rule. It could lead to two packets being sent at the same time, with the possibility of the packet with the higher sequence number to arrive just before the one with the lower sequence number, possibly leading to unnecessary repair requests. The other options may overspecify the protocol. Urgent sending could be modeled by not letting time advance when a sending could be performed, either by adding the test to the tick rule, or by making the tick rule lazy. (Note that since the test condition can only change as result of instantaneous actions, the tick rule may remain deterministic in this case.) Modifications of the above approaches, like requiring that two consecutive sendings take place at least time Δ apart, can also be modeled in a straightforward way. In our current version, a multicast of a new data packet is attempted at intervals given by the data block. If a multicast cannot be performed, sending is attempted one ms later.

The classes used in the combined protocol are given as follows:

```
(tomod COMBINED-PROTOCOL is

including AER-RTT .
including NCA-NOM .
including NCA-RC .
including AER-RS .

subclass SenderCombined < RTTsender .
subclass SenderCombined < NOMsender .
subclass SenderCombined < RCsender .
subclass SenderCombined < RSsender .
subclass RepairserverCombined < RTTrepairserver .
subclass RepairserverCombined < NOMrepairserver .
subclass RepairserverCombined < RCrepairserver .
```

```
subclass RepairserverCombined < RSrepairserver .
subclass ReceiverCombined < RTTreceiver .
subclass ReceiverCombined < NOMreceiver .
subclass ReceiverCombined < RCreceiver .
subclass ReceiverCombined < RSreceiver .

vars MSS WLS MBC BOF : Nat .
var CCMTO : Time .
var DB : DataBuffer .
var RTB : MsgConf .
vars WS : PosFixedPointNumber .
var FRF : Bool .
var DI : DataInfo .
var W : Window .
var NS : NAKstates .
vars DN DN' : DefNat .
```

An original data packet can only be sent when the rate control algorithm allows it. That is, when the `sendAllowed` holds. Sending is attempted once when the `sendDataTimer` expires, even though the sending condition may change at this time due to instantaneous actions. If sending is not possible when attempted, another attempt will be made in 1 ms. Otherwise, the `sendDataTimer` is reset to the sending interval of the data block:

```
rl [A3sendG11G12] :
   { < Q : SenderCombined | children : OS, clock : R,
                            unsentDataPackets : data(Q', NZN, NZN', R') DB,
                            reTransBuf : RTB, nextSeq : NZN'',
                            sendDataTimer : 0, maxSeqSent : MSS,
                            winLowerSeq : WLS,  csmNominee : DQ,
                            winSize : WS, fastRecovFlag : FRF,
                            maxBurstCount : MBC, ccmTimer : DT,
                            timerNominee : DQ', backOffFactor : BOF,
                            ccmTimeout : CCMTO >  CF }
    =>
    { if (NZN'' == MSS plus 1) and sendAllowed(MSS, WLS, WS, FRF, MBC) then
        (< Q : SenderCombined | unsentDataPackets :
                       ((if NZN == NZN' then nil
                        else data(Q', NZN + 1, NZN', R') fi) DB),
                     reTransBuf :
                       (RTB MsgAndNAK(dataPacket(Q', NZN'', R, NZN == 1,
                                     NZN =/= 1 and NZN =/= NZN', false), 0)),
                     nextSeq : NZN'' + 1,
                     sendDataTimer : (if NZN =/= NZN' then
                                      R' else sendRate(DB) fi),
                     maxSeqSent : MSS + 1,
                     ccmTimer :
                        (if WLS <= MSS then (BOF * CCMTO)
                         else noTimeValue fi),
                     maxBurstCount : MBC + 1,
                     timerNominee : (if WLS <= MSS then DQ else DQ' fi) >
    multiSend(dataPacket(Q', NZN'', R, NZN == 1, NZN =/= 1 and NZN =/= NZN',
                       false), Q, OS) CF)
```

```
    else (< Q : SenderCombined | sendDataTimer : 1 > CF) fi } .
```

The sender in the RC part of the protocol may need to reset its `ccmTimeout` value when the nominee is set to `noOid` as the result of reading a `csmPacket` in the NOM protocol:

```
rl [D2D3G14] :
   { (csmPacket(DN, DT, DQ') from Q to Q')
   < Q' : SenderCombined | clock : R, csmNominee : DQ, csmLPE : DN',
                           csmRTT : DT', csmSetTime : R', NAMTimer : DT'',
                           ccmTimeout : CCMTO > CF }
   =>
   { if newNomValues(DN, DT, DQ', DQ, DN', DT', R, R') then
       (< Q' : SenderCombined | csmNominee : DQ', csmLPE : DN,
                                csmRTT : DT, csmSetTime : R,
                                NAMTimer : (if DQ =/= DQ' then 7000 else DT'' fi),
                                ccmTimeout : (if DQ' == noOid and DQ' =/= DQ
                                              then 1000 else CCMTO fi) >
      (if DQ =/= DQ' and DQ' =/= noOid
       then (NAMPacket(true) from Q' to DQ') else none fi)
      (if DQ =/= DQ' and DQ =/= noOid then (NAMPacket(false) from Q' to DQ)
       else none fi)
      CF)
   else  (< Q' : SenderCombined | > CF) fi } .
```

The repair server and receiver objects are initialized when they receive their first `SPMPacket`s:

```
rl [C1initR2F1] :
   { (SPMPacket(N) from Q to Q')  < Q'' : RandomNGen | seed : N' >
   < Q' : RepairserverCombined | SPMread : false >  CF }
   =>
   { (SPMPacket(N) from Q to Q')   < Q'' : RandomNGen | seed : random(N') >
   < Q' : RepairserverCombined | SPMread : true, csmTimer : 7000,
                                 getRTTResendTimer : random(N') % 31 >  CF } .

rl [init] :
   { (SPMPacket(N) from Q to Q')
     < Q'' : RandomNGen | seed : N' >
     < Q' : ReceiverCombined | SPMread : false >   CF }
   =>
   { (SPMPacket(N) from Q to Q')
     < Q'' : RandomNGen | seed : repeatRandom(N', 2) >
     < Q' : ReceiverCombined | SPMread : true,
                               getRTTResendTimer : random(N') % 31,
                               csmTimer : repeatRandom(N', 2) % 5001 >  CF } .
```

The reception of a new data packet by a receiver combines the reception and error recovery treatment in the use case B3 of the RS protocol with the use case E2 in the NOM protocol, which updates the window used to determine the loss probability estimate, and the use case H1 in the RC protocol, which sends a `ccmPacket` if the receiver is the nominee receiver. The lowest outstanding packet number is now given by the attribute `readNextSeq`:

```
crl [B3bE2H1] :
    { (dataPacket(Q''', NZN, R, X, Y, Z) from Q to Q')
    < Q'' : RandomNGen | seed : N >
    < Q' : ReceiverCombined | isNominee : X', fastRepairFlag : FRF,
                              repairserver : Q'''', readNextSeq : NZN',
                              suppressTO : R', dataBuffer : MC,
                              dataInfo : DI, msgWindow : W > CF }
    =>
    { < Q'' : RandomNGen | seed : (if FRF then N else
                                   repeatRandom(N, (NZN minus 1) monus
                                                   max(NZN' minus 1,
                                                      highestNzNAKSeqNo(DI),
                                                      highestSeqNo(MC))) fi) >
    < Q' : ReceiverCombined | readNextSeq : max(NZN', lastSeqNo(toApp(
                                             (dataPacket(Q''', NZN, R, X, Y, Z)
                                              MC), NZN')) + 1),
                             dataBuffer : rmDelivered(MC
                                      dataPacket(Q''', NZN, R, X, Y, Z), NZN'),
                             dataInfo : remove(updateDI(cancelTimers(DI, NZN),
                                             FRF, max(NZN', highestSeqNo(MC) + 1,
                                                     highestNzNAKSeqNo(DI) + 1),
                                             NZN minus 1, R', N),
                                           lastSeqNo(toApp(MC
                                              dataPacket(Q''', NZN, R, X, Y, Z),
                                                           NZN'))),
                             msgWindow : add(NZN, W) >
    toApp((dataPacket(Q''', NZN, R, X, Y, Z) MC), NZN', Q')
    (if X' then
       send(ccmPacket(Q',
              max(NZN', lastSeqNo(toApp(dataPacket(Q''', NZN, R, X, Y, Z) MC,
                                        NZN')) + 1),
                      R, Z), Q', Q'''')
     else none fi)  CF }
    if NZN' <= NZN and not (NZN seqNoIn MC) .
```

The above rule only applies if the data packet has not been seen before. The receiver objects in the NOM and RC protocols are not affected by the reception of data packets with sequence numbers they have already seen, so that case is handled by the rule `B3a` in the RS protocol.

Finally, the behavior in time of the combined objects is straightforward:

```
eq mte(< Q : SenderCombined | NAMTimer : DT, ccmTimer : DT', SPMTimer : DT'',
                              sendDataTimer : DT''' >) =
     default-min(DT, DT', DT'', DT''') .
eq delta(< Q : SenderCombined | clock : R, NAMTimer : DT,
                                ccmTimer : DT', sendDataTimer : DT'',
                                SPMTimer : DT''' >, R') =
     < Q : SenderCombined | clock : R plus R' , NAMTimer : DT default-monus R',
                            ccmTimer : DT' default-monus R',
                            sendDataTimer : DT'' default-monus R',
                            SPMTimer : DT''' default-monus R' > .

eq mte(< Q : RepairserverCombined | getRTTResendTimer : DT, csmTimer : DT',
```

```
                                    SPMWaitTimer : DT'', NAKStates : NS >) =
     default-min(DT, DT', DT'', mte(NS)) .
eq delta(< Q : RepairserverCombined | clock : R, getRTTResendTimer : DT,
                                      csmTimer : DT', SPMWaitTimer : DT'',
                                      NAKStates : NS >, R') =
     < Q : RepairserverCombined | clock : R plus R',
                                  getRTTResendTimer : DT default-monus R',
                                  csmTimer : DT' default-monus R',
                                  SPMWaitTimer : DT'' default-monus R',
                                  NAKStates : delta(NS, R') > .

eq mte(< Q : ReceiverCombined | getRTTResendTimer : DT, csmTimer : DT',
                                dataInfo : DI >) =
     default-min(DT, DT', mte(DI)) .
eq delta(< Q : ReceiverCombined | clock : R, getRTTResendTimer : DT,
                                  csmTimer : DT', dataInfo : DI >, R') =
     < Q : ReceiverCombined | clock : R plus R',
                              getRTTResendTimer : DT default-monus R',
                              csmTimer : DT' default-monus R',
                              dataInfo : delta(DI, R') > .
endtom)
```

We cannot expect the combined protocol to work as desired in light of the problems found already during the execution of the repair service part of the protocol, although it *could* be possible that the introduction of the rate control part excludes the scenarios which lead to the undesired behaviors there, in particular the second problem. The following module gives a set of initial states with the same topology as the state `RSstate3`(n) in the repair service protocol:

```
(tomod COMBINED-PROTOCOL1 is
including COMBINED-PROTOCOL .
including AER-APPLICATIONS .
[ ... ]
op State3 : Nat -> System .
eq State3(N) =
 ({ startRS('a)
  < 'SenderApp : SenderApplication | dataBlocksToBeSent :
                                    dly(dataBlock('first, 'a, 35, 5), 100) >
  < 'RecvrAppC : RecvrApplication | receiver : 'c, dataRead : none >
  < 'a : SenderCombined | clock : 0, sourceRTT : 0, children : 'b,
                      maxDownRTT : noTimeValue, maxDownRTTSetTime : 0,
                      maxRecentDownRTT : 0,  NAMTimer : noTimeValue,
                      csmLPE : noNat, csmRTT : 0, csmSetTime : 0,
                      csmNominee : noOid, timerNominee : noOid,
                      winSize : (fixed) 1, winLowerSeq : 1, dupLowerSeq : 0,
                      maxSeqSent : 0, smoothedRTT : 0, rttVariance : 0,
                      ccmTimeout : 1000, ssThreshold : (fixed) 64,
                      backOffFactor : 1, fastRecovFlag : false,
                      fastRecovSeq : 0, maxBurstCount : 0,
                      ccmTimer : noTimeValue, nextSeq : 1,
                      SPMTimer : noTimeValue, reTransBuf : none,
                      unsentDataPackets : nil, sendDataTimer : noTimeValue >
  < 'b : RepairserverCombined | clock : 0, sourceRTT : noTimeValue,
```

```
                            children : 'c, maxDownRTT : noTimeValue,
                            maxDownRTTSetTime : 0, maxRecentDownRTT : 0,
                            repairserver : noOid, resendInterval : 200,
                            maxUpRTT : noTimeValue, myUpRTT : noTimeValue,
                            getRTTResendTimer : noTimeValue, csmLPE : noNat,
                            csmRTT : 0, csmAddress : noOid, csmSetTime : 0,
                            csmTimer : noTimeValue, SPMread : false,
                            maxSeqRcvd : 0, SPMWaitTimer : noTimeValue,
                            smoothedRTT : 0, rttVariance : 3000,
                            suppressTO : 100, retransTO : 6000,
                            dataBuffer : bBuf(nil, 0, 10), NAKStates : none >
  < 'c : ReceiverCombined | clock : 0, sourceRTT : noTimeValue,
                        repairserver : noOid, resendInterval : 200,
                        maxUpRTT : noTimeValue, myUpRTT : noTimeValue,
                        getRTTResendTimer : noTimeValue, isNominee : false,
                        msgWindow : initWindow(4), csmTimer : noTimeValue,
                        fastRepairFlag : false, readNextSeq : 1,
                        smoothedRTT : 0, rttVariance : 3000,
                        suppressTO : 100, retransTO : 6000,
                        dataBuffer : none, dataInfo : none, SPMread : false >
  < 'random : RandomNGen | seed : N >
  < 'ab : Link | up : 'a, down : 'b, bound : 10, propDelay : 100,
                  linkSpeed : 1, downMsgs : nil, downSize : 0,
                  upMsgs : nil, upSize : 0 >
  < 'bc : Link | up : 'b, down : 'c, bound : 10, propDelay : 1,
                  linkSpeed : 10, downMsgs : nil, downSize : 0,
                  upMsgs : nil, upSize : 0 >}) .
endtom)
```

A quick execution of this state gives the following result:

```
Maude> (rew- [5000] State3(1) .)
rewrites: 1139977 in 14970ms cpu (29500ms real) (76150 rewrites/second)
rewrite in COMBINED-PROTOCOL1 : State3(1) .
result Error(ClockedSystem) : {
  < 'a : SenderCombined | children : 'b, csmNominee : 'c,
    sourceRTT : 0, maxDownRTT : 202, csmRTT : 206, csmLPE : 0,
    ccmTimer : noTimeValue, maxBurstCount : 0, fastRecovSeq
    : 36, fastRecovFlag : true, backOffFactor : 1, ssThreshold : [2000000],
    ccmTimeout : 301, maxSeqSent : 35, dupLowerSeq : 4, winLowerSeq : 36,
    winSize : [8000000], timerNominee : 'c, unsentDataPackets : nil,
    reTransBuf :  (MsgAndNAK(dataPacket('first, 1, 5100, true, false, false), 0)
      MsgAndNAK(dataPacket('first, 2, 5318, false, true, false), 0)
      MsgAndNAK(dataPacket('first, 3, 5323, false, true, false), 0)
      MsgAndNAK(dataPacket('first, 4, 5536, false, true, false), 0)
      MsgAndNAK(dataPacket('first, 5, 5541, false, true, false), 0)
      MsgAndNAK(dataPacket('first, 6, 5548, false, true, false), 0)
      MsgAndNAK(dataPacket('first, 7, 5553, false, true, false), 0)
      [ ... ]
      MsgAndNAK(dataPacket('first, 32, 6190, false, true, false), 0)
      MsgAndNAK(dataPacket('first, 33, 6195, false, true, false), 0)
      MsgAndNAK(dataPacket('first, 34, 6202, false, true, false), 0)
      MsgAndNAK(dataPacket('first, 35, 6207, false, false, false), 0)),
```

```
    nextSeq : 36, [ ... ] >
  < 'SenderApp : SenderApplication | dataBlocksToBeSent : nil >
  < 'b : RepairserverCombined | children : 'c, repairserver : 'a,
    sourceRTT : 202, maxRecentDownRTT : 0, maxUpRTT : 202, csmRTT : 206,
    csmLPE : 0, csmAddress : 'c, dataBuffer : bBuf(
    dataPacket('first, 32, 6190, false, true, false) + dataPacket(
    'first, 33, 6195, false, true, false) + dataPacket('first, 34,
    6202, false, true, false) + dataPacket('first, 35, 6207, false,
    false, false) + dataPacket('first, 26, 6032, false, true, true) +
    dataPacket('first, 27, 6037, false, true, true) + dataPacket(
    'first, 28, 6044, false, true, true) + dataPacket('first, 29,
    6049, false, true, true) + dataPacket('first, 30, 6056, false,
    true, true) + dataPacket('first, 31, 6061, false, true, true),
    10, 10), NAKStates : (nakState( 26, 3, 3, false, noTimeValue, noTimeValue)
      nakState(27, 3, 3, false, noTimeValue, noTimeValue)
      nakState(28, 3, 3, false, noTimeValue, noTimeValue)
      nakState(29, 3, 3, false, noTimeValue, noTimeValue)
      nakState(30, 3, 3, false, noTimeValue, noTimeValue)
      nakState(31, 3, 3, false, noTimeValue, noTimeValue)),
    maxSeqRcvd : 35, [ ... ] >
  < 'c : ReceiverCombined | repairserver : 'b, isNominee : true,
    sourceRTT : 206, maxUpRTT : 4, resendInterval : 3000, csmTimer : 2834,
    msgWindow : window(32 33 34 35, 4, 4), dataInfo : none,
    dataBuffer : (none).Configuration, readNextSeq : 36,
    fastRepairFlag : true, [ ... ] >
  < 'RecvrAppC : RecvrApplication | dataRead : (dataPacket(
    'first, 1, 5100, true, false, false) cat dataPacket('first, 2,
    5318, false, true, false) cat [ ... ] cat
    dataPacket('first, 34, 6202, false, true, false) cat
    dataPacket('first, 35, 6207, false, false, false)), receiver : 'c >
  [ ... ] } in time 172641
```

Everything seems to have gone well in this execution, since all data packets have been delivered in order to the receiver application. Executing the combined protocol is also a good way of analyzing the rate control protocol, since there is now an attempt to repair lost packets.

Although the current combined protocol executes as expected, we found a significant flaw/omission in an earlier version of the protocol during execution. Only one data packet was sent because the data packet was sent before a nominee was found. No receiver would then acknowledge the first data packet by sending a `ccmPacket`, and the analysis of the rate control protocol given above showed that the second packet cannot be sent before the first one is acknowledged. We have addressed this problem by not sending the first data packet before a nominee is found. There are still some troubling scenarios in the combined protocol related to the fact that only the nominee receiver sends `ccmPacket`s. We showed one such scenario in our analysis of the nominee selection protocol, where we demonstrated that there are moments when no receiver knows that it is the nominee. A data packet received at those times may not be acknowledged.

Another related scenario is the following: The first data packet is received by a receiver rcv_1 before rcv_1 becomes the nominee in place of a receiver rcv_2 due to new RTT values, and is received by rcv_2 after this change of nominees. Then, neither will acknowledge the packet, and no more packets can be transmitted. Similar unpleasant behaviors can be given for various attempts at addressing this problem, suggesting that acknowledgment by only the knowing nominee is somewhat problematic.

5.3 Methodology and Experience

We discuss in this section the methodology used and the experience gained in using Real-Time Maude to specify and analyze the AER/NCA real-time network protocol suite. We outline the formal methodology supported by Real-Time Maude to specify and analyze the protocol, and discuss the specification formalism used. We then discuss the new version of the informal specification which was developed after—and was influenced by—our specification and analysis effort.

5.3.1 Formal Methodology

As mentioned in [Mes00a], the fact that (Real-Time) Maude specifications are executable allows us to have a range of increasingly stronger formal methods, to which a system specification can be subjected. Only after less costly methods have been used, leading to a better understanding and to important improvements and corrections of the original specification(s), is it meaningful and worthwhile to invest effort in costlier validation methods. We indicate how Real-Time Maude was, and can be, used to support the following increasingly stronger methods:

- Formal specification,
- execution,
- model checking analysis,
- narrowing analysis, and
- formal proof of correctness of parts of the specification.

Formal Specification

The formalization process results in a formal model which makes essential implicit knowledge explicit and spells out fuzzy requirements in a precise way. For example, no assumptions were stated in the informally described protocol about the communication model, even though the protocol depends crucially on aspects such as the fact that packets sent along the same link arrive in order. Furthermore, formalizing communication aspects, such as taking packet sizes into account, led to new questions that needed to be clarified: Should the RTT protocol compute the round trip times of small or large packets? Does the bound of a link depend on the size of the packets traveling along the link? Should packets ready for delivery be released from a link before a new message is entered into the link or is it lost due to a full link? Other examples of hidden assumptions involved the treatment of "default values" in some cases (such as when a new RTT value is `noTimeValue`), the still unresolved question about *when* the sender should attempt to send a new data packet, how data packets are delivered to the receiver application (in order? when a packet is received?), and whether the repair server can store repair information about an unlimited number of data packets.

Execution of the Specification

Immediate execution using the Real-Time Maude default interpreter gives a feel for the specification, can find errors in the specification, and can give a first validation of the protocol, at no additional cost.

Real-Time Maude's (timed or untimed) default rewrite strategy can be used to immediately execute the specification. Prototyping can give a feel for the system prior to deeper analysis. For example, in the rate control protocol we were able to get a helpful indication of the appropriate sending rates prior to further analyzing the protocol. Furthermore, the result of executing the repair service protocol immediately drew our attention to the large set of NAK states in the repair server objects, which may go against the protocol's intention of requiring as little memory and processing time of the repair servers as possible.

We validated the RTT and NOM protocols using the default timed command by just executing a state and checking that the resulting state satisfied our expectations.

Default executions of the repair service protocol returned undesired states which seemed to invalidate the protocol. Repeated use of Real-Time Maude's timed and untimed commands made it easy to find the undesired behaviors of the protocol, and we were able to easily point out the corresponding scenarios in the original informal specification.

We also quickly found some errors, including the following, in earlier versions of the protocol by executing the specifications:

- As mentioned above, a nominee receiver must be found before data packets are sent. Otherwise, only one data packet can be sent, since no receiver will acknowledge the first data packet by sending a `ccmPacket`. In this connection, we also mentioned some potential problems related to the acknowledgment of data packets only by the nominee which must know it is the nominee.
- An execution led to a state where a new RTT value was `noTimeValue`, and the execution of the protocol could not proceed, because these cases were not handled by the original use cases B13 and C13.
- Execution stopped when the `ccmTimer` was given a negative value. The problem was traced to the erroneous statement `newRTT = (CCM packet timestamp - currentTime in ms)` in the original use case G3, which was immediately corrected.

Model Checking Analysis

A distributed and nondeterministic system may exhibit many behaviors which are not captured by the default execution of a system. To find errors not revealed by default executions, or to increase our confidence in the correctness of the specification before a costly verification effort is undertaken, the specification can be subjected to model checking analysis that considers all behaviors of a system from an initial state, up to some level or condition.

Real-Time Maude's search strategies and user-defined strategies built on top of Real-Time Maude's strategy library can be used to model check the specification. We defined two search functions for exploratory analysis of the RTT and NOM protocols (see Sections 5.2.6 and 5.2.7). These strategies extended the strategy library in the module `TIMED-META-LEVEL` and could therefore be defined very easily. We applied these search strategies to some initial states of the RTT and NOM protocols, and got the expected results, increasing our confidence in the correctness of the specification of these (sub)protocols. A model checking strategy also found a (potential) error—although we were aware of the possibility of the problem before the analysis—which could be difficult to find by just executing the specification using the default interpreter strategies. We are currently analyzing the output from default executions of the rate control protocol, and if they seem reasonable, we have to state some desired properties for further analysis of this protocol. Until a new specification of the repair service is arrived at, there is for the moment no need to further analyze the repair service (and therefore the combined) protocol, since default execution could already point out undesired behaviors of this protocol.

Narrowing Analysis

By using symbolic expressions with variables, one can analyze the system's behavior not just from a single initial state, but from a (possibly infinite) set of states described by a symbolic expression. Our specification has not yet been subjected to this form of symbolic simulation.

Formal Proof

For highly critical properties it is also possible to carry out a formal proof of correctness. The *initial model* of the specification provides a precise model of the protocol which can be subjected to formal proof. For example, the initial model can be regarded as the Kripke structure about which timed temporal logics can state abstract properties. We did not yet try to formally prove parts of the specifications correct with respect to some properties, but this should be done in the future.

5.3.2 The Specification Formalism

The rewriting logic formalism is fairly simple and intuitive, and gives an operational specification of the protocol at an appropriate level of abstraction. The formalism involved, namely real-time rewriting logic, is very well suited for specifying distributed systems, in which local concurrent transitions can be specified as timed rewrite rules. The formalism allows the specification of the static and the dynamic aspects to be given in essentially the same formalism, by equations and rewrite rules, and does not have to introduce language constructions for expressing concurrency and/or nondeterminism. This is in contrast to many other protocol specification languages such as LOTOS [ISO87, BB87] and SDL (see e.g. [BH89, FO94]), where the static, the reactive, and the real-time aspects of a system are described in very different formalisms. Furthermore, the intuitive object-oriented style of specification supported by Real-Time Maude allowed us to specify both the separate subprotocols and the combined protocol with maximal reuse of common attributes and common transitions. The simplicity of the formalism was emphasized by the very fruitful discussions about the Real-Time Maude specification we had with our colleagues at TASC, who had no previous experience with term rewriting.

The rewriting logic specification style allows us to abstract away from aspects which do not concern the essential features of the protocol. We can, for example, abstract away from all irrelevant communication aspects such as how the routers direct packets, how the repair servers "intercept" packets sent through their co-located routers, and so on.

The specification is operational in nature, it describes how the protocol works, while it abstracts away from unnecessary details and avoids overspecifying the protocol. Such an operational specification seems necessary to describe protocols where the emphasis is on *how* it operates rather than on what properties it is supposed to satisfy (such as "all packets should eventually be received by all receivers"). However, the operational specification should be complemented by a more abstract, property-oriented, specification defining the desired properties that the protocol should satisfy. There is, for example, no criteria stating the assumptions about the network under which the protocol is required to deliver all packets. Should delivery be guaranteed under all circumstances or only when the links have a certain bandwidth and the repair servers' caches are not too small?

The operational specification can then be verified against the properties stated in the requirement specification. The initial model of the rewrite specification, together with the durations associated with the arrows, can, for example, be regarded as the Kripke structure about which real-time temporal logic requirements are stated. We are investigating versions of explicit-clock temporal logics suitable to specify properties about real-time rewrite theories. Details will appear elsewhere.

5.3.3 Experience and Benefits Gained using Real-Time Maude for Real-Time Network Protocol Specification and Analysis

We think that this case study has shown that Real-Time Maude is a suitable formalism with adequate tool support in which to specify in a natural way the operational aspects of sophisticated real-time communication protocols.

The experience gained about the Real-Time Maude tool includes:

- We found the timed rewrite command quite useful, even in the absence of nondeterministic tick rules, since it allowed us to relate the execution to the simulated elapsed time, and to include a lazy tick rule in case the sending of data packets should be performed urgently. However, the command may in some cases be a little slow, and for efficiency purposes we may want to have an optimized timed rewrite command to apply when we know that time increase is deterministic.
- The search commands of Real-Time Maude seem sufficient to express most properties we wanted to analyze—but the commands are executed too slowly for larger states, which prompted the need for writing optimized application-specific strategies. Although this was easy enough to do, the most satisfying solution lies in developing an optimized version of the strategy library that will often substantially decrease the need for application-specific strategies.
- Useful search strategies could be defined very easily as extensions of the `TIMED-META-LEVEL` module.
- It could be useful to have some built-in random function to be used by the default interpreter when a time variable is introduced in the righthand side of a rule. This could be used on both tick rules and in other cases such as when a timer should be given an arbitrary start value.
- Earlier specifications showed that executions of object-oriented specifications using tokens '`*`', and tick rules of the form described in Section 2.2.5 could not be efficiently executed, even though the special strategy which instantiates the variable of sort `Tokens` provided some increase in performance. As mentioned before, one should instead regard a theory with global instantaneous rules as an implementation of the corresponding system with tokens '`*`'.
- The current, straightforward specification of the positive rational numbers presented in Appendix A is too inefficient, requiring sometimes up to 300 000 rewrites for some simple calcu-

lations. This should be addressed in the future by developing a high-performance implementation of the rational numbers in Maude.

The informal specification of the AER/NCA protocol suite has changed substantially in the 1.1 version from the 1.0 version since the analysis effort described in this chapter was undertaken. Among the problems we have pointed out that have been addresses are:

- Data packets cannot "miss" the nominee receiver, since each data packet takes as additional parameter the nominee receiver responsible for acknowledging the data packet.
- The repair service part has changed significantly to avoid the flaws in the error recovery process reported above.
- The repair servers can no longer longer store an unlimited amount of NAK information.
- More information about state variables available seems to make essential knowledge about the state explicit.

Chapter 6

Conclusions and Future Work

In this thesis we have investigated, at both the theoretical and the methodological levels, the application of rewriting logic to model and analyze real-time and hybrid systems.

At the theoretical level, we have shown that many well-known models of real-time and hybrid systems can naturally be expressed as real-time rewrite theories, and that such theories can be mapped into ordinary rewriting logic theories, by adding a clock, in a way which preserves all their interesting properties.

Based on this result, we have designed a language and tool, called Real-Time Maude, for the formal specification and analysis of real-time rewrite theories. The functor

$$(\_)^{C} : RTRWTh \to RWTh$$

mapping real-time rewrite theories to ordinary rewrite theories is represented in the specification of Real-Time Maude by the function `timedPreModuleToPreModule`, that transforms a timed (pre)module to an untimed (pre)module. We have identified certain commonly occurring characteristics of typical real-time systems—such as the need for eager and lazy rules and that tick rules have a nondeterministic time increase—and have taken them into account in the design of the specification language and the tool's timed rewrite interpreter and model checking strategies, as well as in the extensible strategy library to suit the specification, execution, and formal analysis of such typical real-time systems.

The tool has been built as an extension of Full Maude, to allow Real-Time Maude specifications to use Full Maude's support for structured and object-oriented specification. To reuse Full Maude's parsing and module handling facilities as much as possible, the tool performs the transformation from a timed module into an untimed module already at the "premodule" stage. Maude's reflective capabilities and support for defining other (specification and execution) tools in Maude, such as its sophisticated parsing facilities, were heavily used in the design of the tool, and in the specification of the model checking strategies.

We have tested the Real-Time Maude language and tool in practice using the formal methodology outlined in [Mes00a] to specify and analyze a sophisticated new suite of real-time active network communication protocols. The protocol components were specified by object-oriented timed modules, and subclass inheritance allowed us to specify and execute the different protocol components both as stand-alone protocols and together in the combined protocol, with maximal reuse of common rules and attributes.

The specification process for this protocol suite required us to make explicit essential implicit assumptions and to clarify fuzzy arguments in the informal specification. Executions of the protocol components using Real-Time Maude's default interpreter revealed some errors which had not been found before, and led to new versions of the informal protocol where some of the errors have been corrected.

The protocol components were then subjected to formal analysis by model checking involving the exploration of all rewrite paths from a given initial state. Although it turned out that the built-in model checking strategies were too general and, therefore, too inefficient, we defined more specialized strategies to model check the protocols. The ease with which these new model checking strategies were defined by extending the tool's strategy library, was encouraging. This model checking technique did not find any errors in the RTT protocol, but found an error in a property we were concerned about in the NOM protocol.

Although our experience in using Real-Time Maude for the specification and analysis of real-time and hybrid systems is quite encouraging, this thesis is just a first step in the application of these methods, and opens up some interesting research directions we would like to investigate.

First of all, the tool should be further polished and improved by adding more features, such as interactive simulation, and more flexible "default" timed rewrite strategies. The search functions should be more efficient, perhaps specialized for some frequently occurring subset of timed modules. The AER/NCA case study showed that some model checking can be performed in a reasonable amount of time when they are slightly specialized. We also expect to use the built-in search facilities of the upcoming 2.0 version of Maude to make model checking more efficient.

We have only investigated some of the ways mentioned in the introduction in which real-time rewrite theories could help bridge the gap between property-oriented (requirement) specification and actual implementation.

At one end of the spectrum, we should investigate what sublanguages of real-time temporal logic are most suitable to describe abstract properties about real-time rewrite theories, and should develop and implement proof techniques for proving a specification correct, using Maude's reflection and interoperability features to integrate existing verification tools, and tools under development, such as inductive theorem provers.

At the other end of the spectrum, we should develop methods to pass from correct Real-Time Maude specifications to correct actual implementations by generating code from the specifications.

More case studies are also needed. We have till now specified real-time and hybrid systems such as the railroad crossing [HL94], a scheduling problem, and the network protocol presented in this thesis. Other application domains where Real-Time Maude could be applied include embedded systems, control systems, and robotics—which combined with code generation techniques could lead to Maude-programmed robots.

Bibliography

[Aal93] W. M. P. van der Aalst. Interval timed coloured Petri nets and their analysis. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 453–472. Springer, 1993.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AER00] Active error recovery (AER): AER/NCA software release version 1.1. `http://www.tascnets.com/panama/AER/`, May 2000.

[AH92] R. Alur and T.A. Henzinger. Logics and models of real time: a survey. In J.W. de Bakker, K. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 74–106. Springer, 1992.

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14, 1987.

[Bee85] M. J. Beeson. *Foundations of Constructive Mathematics.* Springer, 1985.

[BH89] F. Belina and D. Hogrefe. The CCITT specification and description language SDL. *Computer Networks and ISDN Systems*, 16:311–341, 1989.

[BKK+96] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science.* Elsevier, 1996. `http://www.elsevier.nl/locate/entcs/volume4.html`.

[BMSU97] N. Bjørner, Z. Manna, H. B. Sipma, and T. E. Uribe. Deductive verification of real-time systems using STeP. In M. Bertran and T. Rus, editors, *Proc. of ARTS'97*, volume 1231 of *Lecture Notes in Computer Science*, pages 22–43. Springer, 1997.

[BST98] S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In W.-P. de Roever, H. Langmaack, and A. Pnueli, editors, *Proc. Compositionality'97*, volume 1536 of *Lecture Notes in Computer Science*, pages 103–129. Springer, 1998.

[BT87] J. A. Bergstra and J. V. Tucker. Algebraic specification of computable and semicomputable data types. *Theoretical Computer Science*, 50:137–181, 1987.

[BW99] M. Barr and C. Wells. *Category Theory for Computing Science*. Centre de Recherches Mathématiques, third edition, 1999.

[CDE+98] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Martí-Oliet, and J. Meseguer. Metalevel computation in Maude. In C. Kirchner and H. Kirchner, editors, *Proc. 2nd Intl. Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[CDE+99a] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic*. Computer Science Laboratory, SRI International, Menlo Park, 1999. `http://maude.csl.sri.com`.

[CDE+99b] M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a formal metatool. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Proc. FM'99*, volume 1709 of *Lecture Notes in Computer Science*, pages 1684–1703. Springer, 1999.

[CDE+00] M. Clavel, F. Durn, S. Eker, Patrick Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. A maude tutorial. Tutorial at ETAPS 2000, 2000.

[Cla98] M. Clavel. *Reflection in general logics and in rewriting logic, with applications to the Maude language*. PhD thesis, University of Navarre, 1998.

[CM96a] M. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In G. Kiczales, editor, *Proc. Reflection'96*, pages 263–288, 1996. `http://jerry.cs.uiuc.edu/reflection/`.

[CM96b] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Proc. 1st Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www.elsevier.nl/locate/entcs/volume4.html`.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6. Elsevier, Amsterdam, 1990.

[DM98] F. Durán and J. Meseguer. An extensible module algebra for Maude. In C. Kirchner and H. Kirchner, editors, *Proc. 2nd Intl. Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[DM99] F. Durán and J. Meseguer. The Maude specification of Full Maude. Manuscript, May 1999. `http://maude.csl.sri.com/papers/full-maude/`.

[Dur99] F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, University of Málaga, 1999.

[FD98] K. Futatsugi and R. Diaconescu. CafeOBJ report. AMAST Series, World Scientific, 1998.

[FO94] O. Færgemand and A. Olsen. Introduction to SDL–92. *Computer Networks and ISDN Systems*, 26:1143–1167, 1994.

[GM92] J. A. Goguen and J. Meseguer. Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[Han93] H. M. Hanisch. Analysis of place/transition nets with timed arcs and its application to batch process control. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 282–299. Springer, 1993.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HHWT97] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997. See also HyTech home-page at `http://www-cad.eecs.berkeley.edu/~tah/HyTech/`.

[HL94] C. Heitmeyer and N. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *Proc., IEEE Real-Time System Symposium, San Juan*, pages 120–131. IEEE Computer Society Press, 1994.

[ISO87] Information systems processing – open systems interconnection – LOTOS. Tech. rep. International Standards Organization DIS 8807, 1987.

[KBK+99] S. Kasera, S. Bhattacharyya, M. Keaton, D. Kiwior, J. Kurose, D. Towsley, and S. Zabele. Scalable fair reliable multicast using active services. Technical Report TR 99-44, University of Massachusetts, Amherst, CMPSCI, 1999.

[Knu81] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, second edition edition, 1981.

[KW97] P. Kosiuczenko and M. Wirsing. Timed rewriting logic with an application to object-based specification. *Science of Computer Programming*, 28(2-3):225–246, 1997.

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997. See also UPPAAL home-page at `http://www.uppaal.com/`.

[Lyn96] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[Mes89] J. Meseguer. General logics. In H.-D. Ebbinghaus et al., editor, *Logic Colloquium'87*, pages 275–329. North-Holland, 1989.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[Mes93] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[Mes96] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In U. Montanari and V. Sassone, editors, *Proc. Concur'96*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372. Springer, 1996.

[Mes98a] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Proc. WADT'97*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.

[Mes98b] J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany, July 29 – August 6, 1997*, NATO ASI Series F: Computer and Systems Sciences 165, pages 347–398. Springer, 1998.

[Mes00a] J. Meseguer. Rewriting logic and Maude: a wide-spectrum semantic framework for object-based distributed systems. In S. Smith and C.L. Talcott, editors, *Formal Methods for Open Object-based Distributed Systems, FMOODS 2000*, pages 89–117. Kluwer, 2000.

[Mes00b] J. Meseguer. Rewriting logic and Maude: Concepts and applications. In L. Bachmair, editor, *Rewriting Techniques and Applications*, volume 1833 of *Lecture Notes in Computer Science*, 2000.

[MM90] J. Meseguer and U. Montanari. Petri nets are monoids. *Information and Computation*, 88:105–155, 1990.

[MÖS00] J. Meseguer, P. C. Ölveczky, and M.-O. Stehr. Rewriting logic as a unifying framework for Petri nets. Submitted for publication, 2000.

[MP93] Z. Manna and A. Pnueli. Models for reactivity. *Acta Informatica*, 30:609–678, 1993.

[MP96] Z. Manna and A. Pnueli. Clocked transition systems. *Logic and Software Engineering*, pages 3–42, 1996. Also available as Stanford University CSD technical report STAN-CS-TR-96-1566.

[MPT91] S. Morasca, M. Pezzè, and M. Trubian. Timed high-level nets. *The Journal of Real-Time Systems*, 3:165–189, 1991.

[MS98] Z. Manna and H. Sipma. Deductive verification of hybrid systems using STeP. In T. A. Henzinger and S. Sastry, editors, *Hybrid Systems: Computation and Control*, volume 1386 of *Lecture Notes in Computer Science*, pages 305–318. Springer, 1998.

[ÖKW96] P. C. Ölveczky, P. Kosiuczenko, and M. Wirsing. An object-oriented algebraic steam-boiler control specification. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Application: Specifying and Programming the Steam-Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*, pages 379–402. Springer, 1996.

[Ölv00] P. C. Ölveczky. Specifying and analyzing the AER/NCA active networks protocols in real-time maude. `http://www.csl.sri.com/~peter/AER/AER.html`, 2000.

[ÖM96] P. C. Ölveczky and J. Meseguer. Specifying real-time systems in rewriting logic. In J. Meseguer, editor, *Proc. First Intl. Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996. `http://www.elsevier.nl/locate/entcs/volume4.html`.

[ÖM00a] P. C. Ölveczky and J. Meseguer. Real-Time Maude: A tool for simulating and analyzing real-time and hybrid systems. In *Third International Workshop on Rewriting Logic and its Applications*, 2000. To appear.

[ÖM00b] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. To appear in *Theoretical Computer Science*. Available at `http://maude.csl.sri.com/papers`, September 2000.

[Rab60] M. Rabin. Computable algebra: General theory and theory of computable fields. *Transactions of the American Mathematical Society*, 95:341–360, 1960.

[Rei85] W. Reisig. *Petri Nets*, volume 4 of *EATCS monographs on Theoretical Computer Science*. Springer, 1985.

[SK98] L. J. Steggles and P. Kosiuczenko. A timed rewriting logic semantics for SDL: A case study of the alternating bit protocol. In C. Kirchner and H. Kirchner, editors, *Proc. 2nd International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[Sta] The Stanford Temporal Prover. `http://www-step.stanford.edu/`.

[Vir94] P. Viry. Rewriting: An effective model of concurrency. In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *Proc. PARLE'94*, volume 817 of *Lecture Notes in Computer Science*, pages 648–660. Springer, 1994.

[Wir90] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 13. Elsevier, Amsterdam, 1990.

[Yov97] S. Yovine. Kronos: A verification tool for real-time systems. *Software Tools for Technology Transfer*, 1(1/2), 1997. See also Kronos home-page at `http://www-verimag.imag.fr/TEMPORISE/kronos/`.

Appendix A

Time Domains

This appendix gives the specifications of three of the time domains we have used in real-time specifications. For efficiency purposes they are all built on top of the machine integers. Furthermore, they all specify useful functions not required by the theory *TIME*. Finally, the time domains are extended by adding the infinity value `INF` and appropriate supersorts containing `INF`.

A.1 The Machine Natural Numbers

The following specification specifies the machine natural numbers with a sort for the nonzero naturals, and is used in the specification of the AER/NCA protocol.

```
(fmod NATURALS is
including MACHINE-INT .

sorts Nat NzNat .
subsort Nat < MachineInt .
subsort NzNat < Nat .
subsort NzNat < NzMachineInt .

op abs : MachineInt -> Nat .
op _minus_ : MachineInt MachineInt -> MachineInt [prec 33 gather(E e)] .
op _monus_ : Nat Nat -> Nat  [prec 33 gather(E e)] .
vars I I' : MachineInt .
var NZI : NzMachineInt .
vars N N' : Nat .

mb 0 : Nat .
cmb NZI  : NzNat if NZI > 0 .

eq I minus I' = _-_(I, I') .
eq abs(I) = if I < 0 then - I else I fi .
eq N monus N' = if N < N' then 0 else N minus N' fi .
endfm)
```

A.2 The Positive Machine Rationals

This appendix gives the specification of the positive rationals used in the specification of scheduling in Section 3.3. For efficiency purposes they are all built on top of the machine integers.

```
(fmod POSITIVE-RATIONALS is
including NATURALS .

*** PosRat denotes the nonnegative rationals.
sorts PosRat NzPosRat .
subsort NzPosRat < PosRat .
subsort NzNat < NzPosRat .
subsort Nat < PosRat .

op _plus_ : PosRat PosRat -> PosRat  [prec 33 gather (E e) assoc comm] .
op _monus_ : PosRat PosRat -> PosRat [prec 33 gather (E e)] .
op _times_ : PosRat PosRat -> PosRat [prec 31 gather (E e) assoc comm] .

*** div is the constructor for rationals
op div : PosRat NzPosRat -> PosRat .
op div : NzPosRat NzPosRat -> NzPosRat .

op gcd : NzMachineInt NzMachineInt -> NzNat .
op _lt_ : PosRat PosRat -> Bool [prec 37] .

vars I I' : MachineInt .
vars NZP NZP' : NzNat .
vars PR PR' : PosRat .
vars NZPR NZPR' : NzPosRat .

*** Use the basic definitions:
eq I lt I' = I < I' .
eq I plus I' = I + I' .
eq I times I' = I * I' .

*** Reduce all "nested" div's:
eq div(PR, div(NZPR, NZPR')) = div(PR times NZPR', NZPR) .
eq div(div(PR, NZPR), NZPR') = div(PR, NZPR times NZPR') .
*** simplify result:
ceq div(I, NZP) = div(I / gcd(I, NZP), NZP / gcd(I, NZP))
                       if gcd(I, NZP) =/= 1 .
*** The condition is (only) necessary for termination
*** remove the div if possible:
eq div(PR, 1) = PR .
eq div(0, NZPR) = 0 .

eq div(PR', NZPR) plus PR = div(PR' plus (PR times NZPR), NZPR) .
eq PR times div(PR', NZPR) = div(PR times PR', NZPR) .
eq PR monus div(PR', NZPR) = div((PR times NZPR) monus PR', NZPR) .
eq div(PR, NZPR) monus PR' = div(PR monus (PR' times NZPR), NZPR) .

*** Comparators:
eq I lt div(I', NZP) = (I times NZP) lt I' .
eq div(I', NZP) lt I = I' lt (I times NZP) .
eq div(I, NZP) lt div(I', NZP') = (I times NZP') lt (I' times NZP) .

eq gcd(NZP, NZP') = if NZP == NZP' then NZP
```

```
                    else (if NZP > NZP' then gcd(NZP monus NZP', NZP')
                          else gcd(NZP, NZP' monus NZP) fi) fi .
endfm)
```

A.3 The Fixed Point Numbers

The fixed point number `[n]` represents the real number $\frac{n}{BASE}$, where the choice of the base can be done by the user according to the need of the application. We have chosen to keep the sorts `MachineInt` and `FixedPointNumber` in different connected components of the poset of sorts. Therefore, all conversions between fixed points numbers and integers must be explicit, using conversion functions declared to have C-like syntax, as in e.g., `(int) [50000000]` and `(fixed) 50`.

The base should be carefully chosen to avoid overflow, in particular when using the multiplication operation.

```
(fmod FIXED-POINT-REALS is

 including MACHINE-INT .

 sorts FixedPointNumber NzFixedPointNumber .
 subsort NzFixedPointNumber < FixedPointNumber .

 sorts PosFixedPointNumber NzPosFixedPointNumber .
 subsort NzPosFixedPointNumber < PosFixedPointNumber .
 subsort NzPosFixedPointNumber < NzFixedPointNumber .
 subsort PosFixedPointNumber < FixedPointNumber .

 vars N M : MachineInt .
 var NZ : NzMachineInt .
 vars F G : FixedPointNumber .

 op BASE : -> NzMachineInt .
 eq BASE = 1000000 .

 *** Constructor:
 op `[_`] : MachineInt -> FixedPointNumber .
 op `[_`] : NzMachineInt -> NzFixedPointNumber .

 *** The following takes a number into its fixed-point "BASE"-ial form:
 op `(fixed`)_ : MachineInt -> FixedPointNumber [prec 1] .
 eq (fixed) N = [N * BASE] .

 *** The following gives back the integer part of a fixed point number.
 op `(int`)_ : FixedPointNumber -> MachineInt [prec 1] .
 eq (int) [N] = N / BASE .

 *** The following gives a more accurate "rounded value":
 op integerRounded : FixedPointNumber -> MachineInt .
 eq integerRounded([N]) = if N >= 0 then ((N + (BASE / 2)) / BASE)
                          else (_-_(N, (BASE / 2)) / BASE) fi .

 *** The usual operations:
 op _plus_ : FixedPointNumber FixedPointNumber ->
            FixedPointNumber [prec 33 gather (E e) assoc comm] .
```

```
 op _minus_ : FixedPointNumber FixedPointNumber ->
             FixedPointNumber [prec 33 gather (E e) ] .
 op _monus_ : PosFixedPointNumber PosFixedPointNumber ->
             PosFixedPointNumber [prec 33 gather (E e) ] .
 op _times_ : FixedPointNumber FixedPointNumber ->
             FixedPointNumber [prec 31 gather (E e) assoc comm] .
 op _div_ : FixedPointNumber NzFixedPointNumber ->
           FixedPointNumber [prec 31 gather (E e)] .
 op _lt_ : FixedPointNumber FixedPointNumber -> Bool [prec 37] .
 op _le_ : FixedPointNumber FixedPointNumber -> Bool [prec 37] .
 op _gt_ : FixedPointNumber FixedPointNumber -> Bool [prec 37] .
 op _ge_ : FixedPointNumber FixedPointNumber -> Bool [prec 37] .
 op min : FixedPointNumber FixedPointNumber -> FixedPointNumber [assoc comm] .
 op max : FixedPointNumber FixedPointNumber -> FixedPointNumber [assoc comm] .

 *** Other useful operators:
 op floor : FixedPointNumber -> FixedPointNumber .
 op abs : FixedPointNumber -> FixedPointNumber .
 op isInteger : FixedPointNumber -> Bool .
 op closestInteger : FixedPointNumber -> FixedPointNumber .

 mb [0] : PosFixedPointNumber .
 cmb [N] : NzPosFixedPointNumber if N > 0 .

 eq [N] plus [M] = [N + M] .
 eq [N] minus [M] = [_-_(N, M)] .
 eq [N] monus [M] = if N >= M then [_-_(N, M)] else [0] fi .
 eq [N] times [M] = [(N * M) / BASE] .
 eq [N] div [NZ] = [(N * BASE) / NZ] .
 eq [N] lt [M] = N < M .
 eq [N] le [M] = N <= M .
 eq [N] gt [M] = N > M .
 eq [N] ge [M] = N >= M .
 ceq min(F, G) = F if F le G .
 ceq max(F, G) = F if F ge G .
 eq floor([N]) = if N >= 0 then [(N / BASE) * BASE]
                 else [(_-_(N, BASE) / BASE) * BASE] fi .
 eq abs(F) = if F le [0] then ([0] minus F) else F fi .
 eq isInteger([N]) = ((N * BASE) / BASE) == ((N / BASE) * BASE) .
 eq closestInteger([N]) = if N >= 0 then [((N + (BASE / 2)) / BASE) * BASE]
                          else [(_-_(N, (BASE / 2)) / BASE) * BASE] fi .

endfm)

(fmod FIXED-POINT-REALS-AND-INF is

 including FIXED-POINT-REALS .

 sort FixedPointNumberInf .
 subsort FixedPointNumber < FixedPointNumberInf .
 op INF : -> FixedPointNumberInf .

 *** The usual operators are extended to INF:
 op _plus_ : FixedPointNumberInf FixedPointNumberInf ->
            FixedPointNumberInf [prec 33 gather (E e) assoc comm] .
 op _minus_ : FixedPointNumberInf FixedPointNumber ->
             FixedPointNumberInf [prec 33 gather (E e) ] .
 op _monus_ : FixedPointNumberInf FixedPointNumber ->
```

```
             FixedPointNumberInf [prec 33 gather (E e) ] .
 op _lt_ : FixedPointNumberInf FixedPointNumberInf -> Bool [prec 37] .
 op _le_ : FixedPointNumberInf FixedPointNumberInf -> Bool [prec 37] .
 op _gt_ : FixedPointNumberInf FixedPointNumberInf -> Bool [prec 37] .
 op _ge_ : FixedPointNumberInf FixedPointNumberInf -> Bool [prec 37] .
 op min : FixedPointNumberInf FixedPointNumberInf
         -> FixedPointNumberInf [assoc comm] .
 op max : FixedPointNumberInf FixedPointNumberInf
         -> FixedPointNumberInf [assoc comm] .

 vars FI FI' : FixedPointNumberInf .
 var F : FixedPointNumber .

 eq INF plus FI = INF .
 eq INF minus F = INF .
 eq INF monus F = INF .
 eq F lt INF = true .
 eq INF lt FI = false .
 eq FI le INF = true .
 eq INF le F = false .
 eq FI gt FI' = FI' lt FI .
 eq FI ge FI' = FI' le FI' .
 eq max(INF, FI) = FI .
 eq min(INF, FI) = FI .

endfm)
```

Appendix B

The Original Specification of the AER/NCA Protocol

This appendix presents the specification of the AER/NCA protocol which was the basis for our Real-Time Maude specification of the protocol. In addition to using the following specification, we communicated with Mark Keaton on various issues of the protocol to arrive at the Real-Time Maude specification given earlier.

The specification is written entirely by Dr. G. Stephen Zabele, Mark Keaton, and Diane Kiwior at Litton-TASC, Reading, Mass., and Dr. Don Towsley, Dr. Jim Kurose, Supratik Bhattacharyya, and Sneha Kasera at The University of Massachusetts, Amherst, Mass., and is a test version which probably was not intended to be a finished product. It is given here to document the kind of specifications that we used as the basis for developing our formal specification. The specification can also be useful to allow a comparison between the different specification notations and methods for protocol specifications.

B.1 The AER-RS Protocol

```
         AER Repair Service Use Cases
         ----------------------------
                  1999-12-08  TEST
                 Version: 1.0


Section A: Sender Protocol

  State Information:
    nextSeq     - The sequence number to use for the next original data packet
                  sent.

  A.1 Instantiation of the Protocol

  This use case begins when the sender protocol is instantiated (by upper
layers). The protocol data structures are initialized and endpoints (e.g.,
sockets) are opened for the transmission and reception of packets. The state
```

information is initialized as follows:

```
nextSeq     = 1
```

A number of source path message (SPM) packets (e.g, 10) are formed and sent to the multicast group address (obtained from the entity that instantiates the protocol). The SPM packet format is described in Appendix A. Each SPM packet contains the IP address of the sender, a multicast group address and a port number, and the current highest data packet sequence number of the transmitted data packets, which is equal to (nextSeq - 1). A highest data packet sequence number of 0 indicates that no data packets have been transmitted by the sender. The SPM timer is started after sending out the SPM packets. The duration of the SPM timer is set to some constant value (e.g., 4 seconds). This use case ends when the protocol is taken to a wait state (waiting for the events described in use cases A.2 - A.5.

A.2 SPM Timer Service Routine

This use case begins when the SPM timer expires. An SPM packet is transmitted to the destination multicast address. The SPM packet contains the current highest data packet sequence number sent, whic is equal to nextSeq minus 1. The SPM timer is then reset. This use case ends when the SPM timer is reset.

A.3 Sending Data

This use case begins when data (a variable length data block) is received from the upper layer. This data is placed in one or more data packets. The first data packet always has the first segment flag set, and all data packets except the last have the segmentation flag set. Each data packet has the retransmission flag cleared and is assigned a sequence number of nextSeq, then nextSeq is incremented for the next data packet. All data packets are then buffered in a retransmission buffer. The current sender NAK count (maximum value of NAK count seen in NAK packets so far) for each data packet is initialized to 0. These data packets are then sent out to the multicast group as the Testing if an Original Data Packet Can Be Sent Use Case in the rate control algorithm allows. The timestamp of each data packet is filled in with the current time in milliseconds just before the data packet is sent. The data packet format is described in Appendix A. This use case ends when the data packet transmission(s) are completed.

A.4 Processing a Received NAK Packet

This use case begins when a NAK packet is received from the lower layer. The NAK packet contains the sequence number of the data packet whose retransmission has been requested. It also contains a NAK count for that data packet. The NAK packet format is shown in Appendix A. If the NAK packet NAK count is greater than the current sender NAK count of that data packet in the retransmission buffer, then the current sender NAK count for the data packet in retransBuf is set to the NAK packet NAK count, the requested data packet is fetched from the retransmission buffer retransBuf, and the data packet is sent out to the multicast group address. Otherwise, the NAK packet is ignored and no retransmission is sent out. This use case ends when the repair data packet is sent or the NAK packet is ignored.

A.5 Processing a Stop

This use case begins when a stop protocol request is received from the upper layer. All of the resources allocated to the protocol layer are freed. This use case ends when the resources are freed.

Section B: Receiver Protocol

```
  State Information:
    fastRepairFlag  - If true, the repair procedure should not be delayed.
                      If false, NAK suppression is used.
    repairServer    - The address of the repair server for the receiver.
    readNextSeq     - The sequence number of the last data packet delivered
                      to the application.
    smoothedRTT     - The smoothed receiver to sender RTT estimate in
                      milliseconds.
    rttVariance     - The mean deviation of the receiver to sender RTT
                      estimate in milliseconds.
    retransTO       - The current NAK retransmission timer duration in
                      milliseconds.
    suppressTO      - The current NAK suppression timer duration in
                      milliseconds before being adjusted by a uniform random
                      variate.
```

B.1 Instantiation of the Protocol

This use case begins when the receiver protocol is instantiated (by upper layers). The protocol data structures are initialized and endpoints (e.g., sockets) are opened for the transmission and reception of packets. This is followed by a multicast join operation for joining the multicast group on which data and NAK packets are expected. The multicast address and port numbers are received from the entity that instantiates the protocol.The state information is initialized as follows:

```
    fastRepairFlag  = false
    repairServer    = 0
    readNextSeq     = initial data packet sequence number
    smoothedRTT     = 0
    rttVariance     = 3000
    retransTO       = (smoothedRTT + (2 * rttVariance))
    suppressTO      = 100
```

This use case ends when the protocol is taken to a wait state where it waits to receive data, NAK or SPM packets and also waits for expiration of timers as described below.

B.2 Processing a Received SPM Packet

This use case begins when an SPM packet is received. The following processing is performed (rs and maxSeq are the SPM packet repair server address and highest data packet sequence number):

```
    repairServer = rs
    if (maxSeq != 0)
    {
      Call Use Case B.4 using (maxSeq + 1)
    }
```

This use case ends when either repairServer has been updated or Use Case B.4

returns.

B.3 Processing a Received Data Packet

This use case begins when a data packet is received. The following processing is performed (seq is the received data packet sequence number):

```
Cancel any NAK retransmission timer for data packet seq
Cancel any NAK suppression timer for data packet seq
if ((data packet seq is not currently buffered in dataBuf) AND
    (seq >= readNextSeq))
{
  Buffer received data packet
  Call Use Case B.8
}
else
{
  Discard received data packet
}
```

This use case ends when either Use Case B.8 returns or the received data packet is discarded.

B.4 Detecting Gaps in Data Packet Sequence Numbers

This use case begins when gaps in received data packet sequence numbers must be detected and handled. The following processing is performed (seq is the sequence number passed to this use case):

```
nakSeq = (seq - 1)
while (nakSeq >= readNextSeq)
{
  if ((data packet nakSeq is currently buffered) OR
      (NAK count for data packet nakSeq in dataBuf > 0))
  {
    break while loop
  }
  NAK count for data packet nakSeq = 1
  Call Use Case B.9
  nakSeq = (nakSeq - 1)
}
```

This use case ends when the above processing is complete.

B.5 NAK Suppression Timer Service Routine

This use case begins when the NAK suppression timer for a missing data packet expires. The following processing is performed (seq is the sequence number of the missing data packet):

```
if ((data packet seq is currently buffered) OR (seq < readNextSeq))
{
  End Use Case
}
if (NAK count for data packet seq > 48)
{
  Error, connection is broken, cannot continue
}
```

```
    Unicast a NAK packet for data packet seq with the receiver's NAK count and
      fastRepairFlag to repairServer
    Start a NAK retransmission timer for data packet seq with a duration of
      retransTO
```

This use case ends when either the timer is ignored, an unrecoverable error occurs or a NAK retransmission timer for the data packet is started.

B.6 NAK Retransmission Timer Service Routine

This use case begins when the NAK retransmission timer for a missing data packet expires. The following processing is performed (seq is the sequence number of the missing data packet):

```
    if ((data packet seq is currently buffered) OR (seq < readNextSeq))
    {
      End Use Case
    }
    NAK count for data packet seq is incremented by 1
    Call Use Case B.9
```

This use case ends when either the timer is ignored or Use Case B.9 returns.

B.7 Processing a Received NAK Packet

This use case begins when a NAK packet is received. The following processing is performed (seq and nakCount are the NAK packet sequence number and NAK count):

```
    if ((data packet seq is currently buffered) OR (seq < readNextSeq))
    {
      End Use Case
    }
    if (nakCount >= NAK count for data packet seq)
    {
      NAK count for data packet seq in dataBuf = nakCount
      Cancel any NAK retransmission timer for data packet seq
      Cancel any NAK suppression timer for data packet seq
      Start a NAK retransmission timer for data packet seq with a duration
        of retransTO
    }
```

This use case ends when either the NAK packet is ignored or the NAK retransmission timer is started.

B.8 Processing a Buffered Data Packet

This use case begins when a received data packet is buffered. The following processing is performed (seq is the data packet sequence number):

```
    Call Use Case B.4 using seq
    if (application is using an asynchronous receive mode)
    {
        Check for data packets to deliver to the application
        Update readNextSeq
    }
```

Note that in delivering any data packets to application, a "Reassembly" process is performed using readNextSeq along with the "First Segment" and "Segmentation" flags in the data packet headers. This use case ends when either Use Case B.4 returns or when readNextSeq is updated.

B.9 Start Suppression Interval

This use case begins when a NAK suppression interval is to be started for the missing data packet sequence number seq. The following processing is performed:

```
if (fastRepairFlag == false)
{
  Start a NAK suppression timer for data packet seq with a duration of
    a random variate, uniformly distributed between 0 and an upper limit
    (e.g., 1.5), times suppressTO
}
else
{
  Call Use Case B.5
}
```

This use case ends when the either the NAK suppression timer is started or Use Case B.5 returns.

B.10 Enable Fast Repair

This use case begins when the fast repair option is to be enabled. fastRepairFlag is set to true. This use case ends when fastRepairFlag is set to true.

B.11 Disable Fast Repair

This use case begins when the fast repair option is to be disabled. fastRepairFlag is set to false. This use case ends when fastRepairFlag is set to false.

B.12 Processing a Stop

This use case begins when a stop protocol request is received from the upper layer. All of the resources allocated to the protocol layer are freed. This use case ends when the resources are freed.

B.13 Updating Round Trip Time Estimates

This use case begins when a new RTT estimate is available. The following processing is performed (senderRTT is a new sender RTT estimate in milliseconds and maxUpRTT is a new maximum peer group RTT estimate in milliseconds. The maximum peer group RTT is the maximum RTT between this receiver's repair server and all receivers and repair servers serviced by it.):

```
if (smoothedRTT == 0)
{
  smoothedRTT = senderRTT
  rttVariance = (senderRTT / 4)
}
else
```

```
      {
        error       = (senderRTT - smoothedRTT)
        smoothedRTT = (smoothedRTT + (error / 8))
        rttVariance = (rttVariance + ((ABS(error) - rttVariance) / 4))
      }
      retransTO  = (smoothedRTT + (4 * rttVariance))
      suppressTO = maxUpRTT
```

This use case ends when suppressTO has been updated.

B.13 Processing a Stop

This use case begins when a stop protocol request is received from the upper layer. All of the resources allocated to the protocol layer are freed. This use case ends when the resources are freed.

Section C: Repair Server Protocol

State Information:

```
    repairServer    - The address of the repair server for this repair server.
    maxSeqRcvd      - The maximum data packet sequence number received.
    smoothedRTT     - The smoothed repair server to sender RTT estimate in
                      milliseconds.
    rttVariance     - The mean deviation of the repair server to sender RTT
                      estimate in milliseconds.
    retransTO       - The current NAK retransmission timer duration in
                      milliseconds.
    suppressTO      - The current NAK suppression timer duration in
                      milliseconds before being adjusted by a uniform random
                      variate.
```

C.1 Instantiation of the Protocol

This use case begins when the active node receives the first SPM packet. The active node searches for the repair server object for the source address and multicast group and, not finding any, creates a new repair server object for the stream. The source address and multicast group are extracted from the SPM packet and stored by the repair server. The repair server object is added to the active node cache using the source address and multicast group. The state information is initialized as follows:

```
    repairServer = 0
    maxSeqRcvd   = 0
    smoothedRTT  = 0
    rttVariance  = 3000
    retransTO    = (smoothedRTT + (2 * rttVariance))
    suppressTO   = 100
```

Use Case C.3 is then called for further SPM packet processing. This use case ends when Use Case C.3 returns.

C.2 SPM Wait Timer Service Routine

This use case begins when the SPM wait timer expires. This event indicates that either no packets are being sent by the sender anymore on this multicast group or the routes have changed. All timers for the repair server are canceled, all buffer resources for the repair server are freed and the repair server object is removed from the active node cache. This use case ends when the protocol state is destroyed.

C.3 Processing a Received SPM Packet in the Wait State

This use case begins when an SPM packet is received in the wait state. The SPM packet is processed in this use case and is not forwarded down the multicast tree until this use case subcasts it. The following processing is performed (rs and maxSeq are the SPM packet repair server address and highest data packet sequence number):

```
Cancel any current SPM wait timer
Start an SPM wait timer with a fixed duration (e.g., 60 seconds)
Call Use Case C.5 using (maxSeq + 1)
repairServer = rs
Subcast SPM packet downstream with the SPM packet repair server
  address field set to the local outgoing interface address
```

Note that when subcasting the SPM packet downstream, it is sent once out each outgoing interface for the multicast tree, and each of these SPM packets has the outgoing interface IP address in the repair server field. This use case ends when the the SPM packet is subcast downstream.

C.4 Processing a Received Data Packet

This use case begins when a data packet with sequence number seq is received by the repair server. Note that original data packets are first forwarded out all outgoing interfaces for the multicast tree before this processing takes place. Repair data packets are processed first and forwarded out all outgoing interfaces for the multicast tree only when these use cases call for subcasting the repair data packet downstream. The following processing is performed:

```
if (data packet seq is not currently buffered)
{
  if (NAK state exists for data packet seq)
  {
    if ((data packet seq is a repair data packet) AND
        (NAK pending flag for data packet seq == true))
    {
      Subcast data packet seq downstream
    }
    Cancel any current NAK suppression timer for data packet seq
    Cancel any current NAK retransmission timer for data packet seq
    NAK pending flag for data packet seq = false
  }
  Buffer data packet seq
  Call Use Case C.5 using seq
  if (seq > maxSeqRcvd)
  {
    maxSeqRcvd = seq
  }
}
else
{
  Discard received data packet
}
```

Note that the repair server data packet buffer uses a certain buffer management policy (e.g., limited size FIFO). Also, the NAK state consists of a NAK pending flag, a NAK count and a downstream NAK count. This use case ends when either Use Case C.4 returns, maxSeqRcvd is updated or the data packet is discarded.

C.5 Detecting Gaps in Data Packet Sequence Numbers

This use case begins when lost data packets must be detected and handled. The following processing is performed (seq is the sequence number passed to this use case):

```
if (maxSeqRcvd == 0)
{
  End Use Case
}
if (seq > (maxSeqRcvd + 1))
{
  for (i = (seq - 1); i > maxSeqRcvd; i--)
  {
    if (no NAK state exists for data packet seq)
    {
      Call Use Case C.6 using i
    }
  }
}
```

This use case ends when either Use Case C.6 returns for the last missing data packet or no action is taken.

C.6 Starting Data Packet Recovery

This use case begins when data packet seq is detected as missing and the data packet recovery process must be started. The following processing is performed:

```
Create nakState for data packet seq
NAK count for data packet seq                    = 1
Downstream NAK count for data packet seq         = 1
NAK pending flag for data packet seq             = true
Subcast a NAK packet for data packet seq with the repair server's NAK
  count and the fast repair flag set to false to the multicast group
Start a NAK suppression timer for data packet seq with a duration of
  a random variate, uniformly distributed between 0 and an upper limit
  (e.g., 1.5), times suppressTO
```

This use case ends when either a NAK suppression timer for data packet seq is started.

C.7 Processing a Received Multicast NAK Packet from Upstream

This use case begins when a multicast NAK packet is received from upstream. The following processing is performed (seq and nakCount are the NAK packet sequence number and NAK count):

```
if (data packet seq is not currently buffered)
{
  if (NAK state exists for data packet seq)
```

```
      {
        if (nakCount > NAK count for data packet seq)
        {
          NAK count in nakState for data packet seq = nakCount
          Downstream NAK count for data packet seq  = nakCount
          Subcast a NAK packet for data packet seq with nakCount and the
            fast repair flag set to false to the multicast group
        }
        Cancel any current NAK suppression timer for data packet seq
        Cancel any current NAK retransmission timer for data packet seq
      }
      else
      {
        Create nakState for data packet seq
        NAK count in nakState for data packet seq        = nakCount
        Downstream NAK count for data packet seq         = nakCount
        NAK pending flag in nakState for data packet seq = true
        Subcast a NAK packet for data packet seq with nakCount and the fast
          repair flag set to false to the multicast group
      }
      Start a NAK retransmission timer for data packet seq with a duration
        of retransTO
    }
```

This use case ends when either the NAK retransmission timer is started.

C.8 Processing a Received Unicast NAK Packet from Downstream

This use case begins when a unicast NAK packet is received from downstream. The following processing is performed (seq, nakCount and fastRepairFlag are the NAK packet sequence number, NAK count and fast repair flag):

```
  if (no NAK state exists for data packet seq)
  {
    Create NAK state for data packet seq
    Call Use Case C.9 using seq, nakCount, fastRepairFlag and true
  }
  else
  {
    if (nakCount > NAK count for data packet seq)
    {
      Call Use Case C.9 using seq, nakCount, fastRepairFlag and false
    }
    else
    {
      if (data packet seq is not currently buffered)
      {
        if (nakCount > downstream NAK count for data packet seq)
        {
          Downstream NAK count for data packet seq = nakCount
          Subcast NAK packet for data packet seq with nakCount and the fast
            repair flag set to false to the multicast group
        }
      }
    }
  }
```

Note that the repair server downstream NAK count is the highest NAK count seen be the repair server from unicast NAK packets for data packet seq sent from downstream. It is used only for the purpose of preventing unnecessary downstream multicasts of NAK packets for NAK suppression. This use case ends when either C.9 returns, a NAK packet is subcast downstream or no action is taken.

C.9 Processing a New NAK Packet with NAK State

This use case begins when a received unicast NAK packet is a new NAK packet. The following processing is performed (seq, nakCount, fastRepairFlag and newNakStateFlag are passed into this use case.):

```
NAK count for data packet seq            = nakCount
Downstream NAK count for data packet seq = nakCount
if (data packet seq is currently buffered)
{
  NAK pending flag for data packet seq = false
  Get data packet seq from buffer
  Set the retransmission flag in the data packet seq
  Subcast data packet seq downstream
}
else
{
  NAK pending flag for data packet seq = true
  Subcast NAK packet for data packet seq with nakCount and the fast
    repair flag set to false to the multicast group
  Call Use Case C.10 using seq, nakCount, fastRepairFlag and
    newNakStateFlag
}
```

This use case ends when either data packet seq is subcast downstream or Use Case C.10 returns.

C.10 Start Suppression Interval

This use case begins when a received unicast NAK packet is a new NAK packet. The following processing is performed (seq, nakCount, fastRepairFlag and newNakStateFlag are passed into this use case):

```
if (fastRepairFlag == false)
{
  if (newNakStateFlag == true)
  {
    Start a NAK suppression timer for data packet seq with a duration of
      a random variate, uniformly distributed between 0 and an upper limit
      (e.g., 1.5), times suppressTO
  }
}
else
{
  if (newNakStateFlag == false)
  {
    Cancel any current NAK suppression timer for data packet seq
    Cancel any current NAK retransmission timer for data packet seq
  }
  Unicast a NAK packet for data packet seq with nakCount and
    fastRepairFlag to repairServer
  Start a NAK retransmission timer for data packet seq with a duration of
```

```
        retransTO
    }
```

This use case ends when either a NAK suppression timer is started, a NAK retransmission timer is started, or no action is taken.

C.11 NAK Suppression Timer Service Routine

This use case begins when the NAK suppression timer for data packet seq expires. The following processing is performed:

```
    if ((data packet seq is currently buffered) OR
        (No NAK state exists for data packet seq) OR
        (NAK pending flag for data packet seq == false))
    {
      End Use Case
    }
    Unicast a NAK packet for data packet seq with the NAK count for data packet
      packet  and fastRepairFlag set to false to repairServer
    Start a NAK retransmission timer for data packet seq with a duration of
      retransTO
```

This use case ends when the NAK retransmission timer is started.

C.12 NAK Retransmission Timer Service Routine

This use case begins when the NAK retransmission timer for data packet seq expires. The following processing is performed:

```
    if ((data packet seq is currently buffered) OR
        (No NAK state exists for data packet seq) OR
        (NAK pending flag for data packet seq == false))
    {
      End Use Case
    }
    NAK count for data packet seq is incremented by 1
    Subcast a NAK packet for data packet seq including repair server NAK count
      and fastRepairFlag set to false downstream
    Start a NAK suppression timer for data packet seq with a duration of
      a random variate, uniformly distributed between 0 and and upper limit
      (e.g., 1.5), times suppressTO
```

This use case ends when the NAK suppression timer is started.

C.13 Updating Round Trip Time Estimates

This use case begins when a new RTT estimate is available. The following processing is performed (senderRTT is a new sender RTT estimate in milliseconds and maxUpRTT is a new maximum peer group RTT estimate in milliseconds. The maximum peer group RTT is the maximum RTT between this receiver's repair server and all receivers and repair servers serviced by the it.):

```
      if (smoothedRTT == 0)
      {
        smoothedRTT = senderRTT
        rttVariance = (senderRTT / 4)
      }
```

```
      else
      {
        error       = (senderRTT - smoothedRTT)
        smoothedRTT = (smoothedRTT + (error / 8))
        rttVariance = (rttVariance + ((ABS(error) - rttVariance) / 4))
      }
      retransTO  = (smoothedRTT + (4 * rttVariance))
      suppressTO = maxUpRTT

This use case ends when suppressTO has been updated.


Appendix A: Packet Formats

SPM: Version Number, Packet Type, Multicast Group Address, Port Number,
     Repair Server Address, Maximum Transmitted Data Packet Sequence Number.
     The IP header of an SPM contains the SPM option.

Data Packet: Version Number, Packet Type, Sequence Number, Timestamp, First
             Segment Flag, Segmentation Flag, Retransmission Flag, Payload

Unicast NAK: Version Number, Packet Type, Data Packet Sequence Number,
             NAK Count, Fast Repair Flag, Source Address

Repair Packet: Same as Data Packet. In addition, the IP header contains
               the repair/NAK option.

Multicast NAK: Same as Unicast NAK Packet. In addition, the IP header contains
               the repair/NAK option.
```

B.2 The AER-RTT Protocol

```
          NCA Use Cases : Estimation of Suppression Timer and
                      Retransmission Timer Values
          ---------------------------------------------------
                              1999-12-08
                             Version: 1.0


In order to facilitate operation of the AER protocol, network-adaptive values
for the NAK suppression timer and the NAK retransmission timer are needed.
Upon assessment of the behavioral characteristics of AER, the following
heuristics seem reasonable:

  * The suppression timer should be set to some scalar value times the maximum
    round-trip time between the nearest upstream repair server and members of
    the NAK originator's peer group. Members of the peer group are identically
    clients and repair servers that are directly supplied by the originator's
    upstream repair server (i.e., there are no intervening repair servers
    between the originator's repair server and any of the members of the
    originator's peer group)

  * The retransmission timer should be set to some scalar value times the
    round trip time from the originator of the NAK to the sender.
```

The above settings directly reflect AER's NAK suppression strategy. Whenever an entity capable of generating a NAK detects a packet loss, the entity first waits for some period of time before transmitting a NAK. During this interval, the entity listens on the multicast channel for repairs or NAKs of the missing packet. If neither a repair nor a NAK for the missing packet arrives during the suppression interval, the NAK processing thread then sends a NAK upstream and subsequently sets and awaits the expiration of a retransmission timer. Alternately, if a repair arrives during the suppression interval, the NAK processing thread simply terminates. Finally, if a NAK arrives during the suppression interval, then the NAK processing thread terminates the suppression timer and subsequently sets and awaits the expiration of a retransmission timer.

Repair operations following the initiation of a retransmission timer are straightforward. If no repair arrives during the retransmission interval, then the NAK processing thread assumes that the repair was dropped and begins the NAK processing thread from the top: i.e., a NAK suppression timer is set and the thread begins listening for NAKs or repairs for the missing packet.

Nominally, each entity (receiver or repair server) picks a random number uniformly distributed between 0 and 1 as a scalar multiplier on a suppression timer maximum value. The hope is that some of the entities will wait very little time to issue NAKs, while others wait longer periods to see if others have issued NAKs. Assuming that one of the entities issues a NAK immediately, then the minimum time required before the NAK can be observed by a peer is the sum of: the transit time between the NAK originator and the repair server; the queueing delays within the repair server; and the transit time from the repair server to the peer. In the worst case, the entity choosing the smallest uniform random variable value has the longest transit time to the repair server, and its peer has a nearly identical transit time; for this case the minimum observation time is asymptotically the round trip time from the furthest entity to the repair server. Hence we set the suppression timer maximum value to some constant (e.g., 1.5) times the estimated maximum round trip time for the peer group.

For establishing the retransmission timer, we assume that in the worst case the repair request must propagate all the way back to the sender in order to be serviced. Hence, we set the retransmission timer for each entity to some constant (e.g., 1.75) times the estimated round trip time between it and the sender.

The following discussion describes the mechanism used for estimating the maximum round trip time for the peer group and the round trip time between an entity (repair server or receiver) and the sender. The mechanism is embodied in the Get-RTT algorithm described below.

Description of the Get-RTT Algorithm

1. High-level description

Both the maximum round trip time for a peer group and the round trip time to the sender are determined via the Get-RTT algorithm. The algorithm functions via request-response messages exchanged between subscribers (receivers or repair servers) and their nearest upstream providers (repair servers or the sender). The Get-RTT request message has the format:

```
--------------------
```

```
  | xmitTime | upRTT |
  --------------------
```

and the Get-RTT response message has the format:

```
  --------------------------------------
  | xmitTime | peerGroupRTT | globalRTT |
  --------------------------------------
```

```
2. State variable definitions and initial settings

  Algorithm Parameters:

    implosionSuppressionInterval
      The maximum amount of time that a receiver or repair server should wait
      before starting the first Get-RTT sequence.
      Nominal value: 30 milliseconds

    initialGetRTTCycleTime
      The initial amount of time that a receiver or repair server should wait
      before starting the next Get-RTT sequence.
      Nominal value: 200 milliseconds

    maxGetRTTCycleTime
      The maximum amount of time that a receiver or repair server should wait
      before starting the next Get-RTT sequence.
      Nominal value: 3 seconds

    updateWindowLength
      The maximum amount of time that the sender or a repair server should
      wait before updating its estimate of the maximum round trip time for its
      downstream peer group.
      Nominal value: maxGetRTTCycleTime + 0.5 seconds


  Sender:

    Peer Group RTT Variables:

      maxDownRTTSetTime
        Time that maxDownRTT was last updated.
        Initialized to the currentTime in milliseconds.

      maxDownRTT
        Value currently being used as the largest RTT received from a directly
        supplied downstream receiver or repair server.
        Initialized to -1.

      maxRecentDownRTT
        Largest RTT received from a directly supplied downstream receiver or
        repair server since maxDownRTT was last set.
        Initialized to 0.

    Sender RTT Variables:

      sourceRTT:
        RTT to the sender.
```

```
      Always 0.


Repair servers:

  Common Variables:

    resendInterval
      Time between successive Get-RTT requests.
      Initialized to the value of initialGetRTTCycleTime.

  Peer Group RTT Variables:

    maxDownRTTSetTime
      Time that maxDownRTT was last updated.
      Initialized to the currentTime in milliseconds.

    maxDownRTT
      Value currently being used as the largest RTT received from a directly
      supplied downstream receiver or repair server.
      Initialized to -1.

    maxRecentDownRTT
      Largest RTT received from a directly supplied downstream receiver or
      repair server since maxDownRTT was last set.
      Initialized to 0.

    maxUpRTT
      Largest RTT observed by the nearest upstream repair server (or
      sender). Value is used to derive the suppression timer value.
      Initialized to -1.

    myUpRTT
      RTT observed to the nearest upstream repair server (or sender).
      Initialized to -1.

  Sender RTT Variables:

    sourceRTT
      RTT to the sender. Value is used to derive the retransmit timer
      value.
      Initialized to -1.


Receivers:

  Common Variables:

    resendInterval
      Time between successive Get-RTT requests.
      Initialized to the value of initialGetRTTCycleTime.

  Peer Group RTT Variables:

    maxUpRTT
      Largest RTT observed by the nearest upstream repair server (or
      sender). Value is used to derive the suppression timer value.
      Initialized to -1.
```

```
  myUpRTT
    RTT observed to the nearest upstream repair server (or sender).
    Initialized to -1.

Sender RTT Variables:

  sourceRTT
    RTT to the sender. Value is used to derive the retransmit timer
    value.
    Initialized to -1.
```

3. Estimation Processing Sequence

The maximum peer group RTT value is the maximum RTT between a repair server (or the sender) and the next downstream level of repair servers or receivers. The source RTT is the round trip time between the sender and an repair server or receiver.

1) (Initialization: this step is not repeated) Upon receipt of the first SPM packet, each receiver or repair server sets a Get-RTT resend timer to a value that is the product of implosionSuppressionInterval and a random number that is uniformly distributed between 0 and 1.

2) Upon expiration of the Get-RTT resend timer, the Get-RTT resend timer is reset using the current value of resendInterval and the process continues at step 3).

3) The receiver or repair server sends a Get-RTT request to the nearest upstream repair server (or sender if it is the first repair server). The host's current time in milliseconds is inserted into the message as the value for xmitTime, and the hosts's current value for myUpRTT is inserted into the message as the value for UpRTT.

4) Upon receipt of a Get-RTT request the upstream repair server (or sender) compares the upRTT in the message with the local value of maxDownRTT. If the value of upRTT is greater than the current value of maxDownRTT, then it sets maxDownRTT to the value of upRTT, the value of maxDownRTTSetTime is set to the current time in milliseconds, and the value of maxRecentDownRTT is set to 0. If the value of upRTT is not greater than maxDownRTT, then a check is made to see if upRTT is greater than maxRecentDownRTT; if it is, the value of maxRecentDownRTT is set to the value of upRTT; i.e.,

```
if (upRTT > maxDownRTT)
{
  maxDownRTT = upRTT
  maxDownRTTSetTime = currentTime in milliseconds
  maxRecentDownRTT = 0
}
else
{
  if (upRTT > maxRecentDownRTT)
  {
    maxRecentDownRTT = upRTT
  }
}
```

A check is then made to determine if maxDownRTT should be updated to the value of maxRecentRTT by comparing the update time against the current time in milliseconds.

```
if (currentTime > (maxDownRTTSetTime + updateWindowLength))
{
  maxDownRTT = maxRecentDownRTT
  maxDownRTTSetTime = currentTime in milliseconds
  maxRecentDownRTT = 0
}
```

The repair server (or sender) then sends a Get-RTT response message. The value of xmitTime in the response is set to the value of xmitTime contained in the request. The value of peerGroupRTT in the response is set to the local value of maxDownRTT. The value of globalRTT in the response is set to the local value of sourceRTT.

5) Upon receipt of Get-RTT response message, the repair server or receiver computes the local round trip time as

```
myUpRTT = (currentTime - xmitTime)
```

If the value of maxUpRTT is equal to -1 OR the value of peerGroupRTT is equal to -1, it then: cancels the current Get-RTT resend timer; sets the local value of resendInterval equal to myUpRTT; rearms the Get-RTT timer with the value of resendInterval; and immediately sends another Get-RTT request.

The value of resendInterval is then doubled and limited, if necessary, to a maximum value given by maxGetRTTCycleTime.

The value for maxUpRTT is either set to the value of peerGroupRTT from the message or the new value of myUpRTT, whichever is greater: i.e.,

```
if (myUpRTT > peerGroupRTT)
{
  maxUpRTT = myUpRTT
}
else
{
  maxUpRTT = peerGroupRTT
}
```

If the value of globalRTT in the response message is nonnegative, the receiver or repair server then computes and sets the value of sourceRTT as the sum of the value of globalRTT and the value of myUpRTT; i.e.,

```
if (globalRTT >= 0)
{
  sourceRTT = (globalRTT + myUpRTT)
}
```

4. Use Cases

R.1 Initialization

This use case begins when Get-RTT is initialized. The variables are initialized as follows:

```
Sender:
  maxDownRTTSetTime = currentTime in milliseconds
  maxDownRTT        = -1
  maxRecentDownRTT  = 0
  sourceRTT         = 0

Repair Server:
  maxDownRTTSetTime = currentTime in milliseconds
  maxDownRTT        = -1
  maxRecentDownRTT  = 0
  maxUpRTT          = -1
  myUpRTT           = -1
  sourceRTT         = -1
  resendInterval    = initialGetRTTCycleTime

Receiver:
  maxUpRTT       = -1
  myUpRTT        = -1
  sourceRTT      = -1
  resendInterval = initialGetRTTCycleTime
```

This use case ends when the variables have been initialized.

R.2 Processing the First Received SPM Packet

This use case begins when the first SPM packet is received at a receiver or repair server. Each receiver or repair server starts a Get-RTT resend timer with a duration of a random variate, uniformly distributed between 0 and 1.0, times implosionSuppressionInterval

This use case ends when the Get-RTT resend timer has been set.

R.3 Get-RTT Resend Timer Service Routine

This use case begins when the Get-RTT resend timer expires at a receiver or repair server. Each receiver or repair server resets the Det-RTT resend timer using the current value of resendInterval, and subsequently sends a Get-RTT request packet to the nearest upstream repair server (or sender). The Get-RTT request packet fields are set as follows:

```
xmitTime = currentTime
upRTT    = myUpRTT
```

This use case ends when the Get-RTT request packet has been sent.

R.4 Processing a Received Get-RTT Request Packet

This use case begins when a Get-RTT request packet is received at a repair server or sender. The following processing is performed (xmitTime and upRTT are Get-RTT request packet fields):

```
if (upRTT > maxDownRTT)
{
  maxDownRTT        = upRTT
  maxDownRTTSetTime = currentTime in milliseconds
```

```
      maxRecentDownRTT  = 0
    }
    else
    {
      if (upRTT > maxRecentDownRTT)
      {
        maxRecentDownRTT = upRTT
      }
    }
```

A check is then made to determine if maxDownRTT should be updated to the value of maxRecentRTT by comparing the update time against the current time in milliseconds:

```
    if (currentTime > (maxDownRTTSetTime + updateWindowLength))
    {
      maxDownRTT        = maxRecentDownRTT
      maxDownRTTSetTime = currentTime in milliseconds
      maxRecentDownRTT  = 0
    }
```

The repair server or sender then sends a Get-RTT response packet. The Get-RTT response packet fields are set as follows:

```
       xmitTime     = xmitTime,
       peerGroupRTT = maxDownRTT
       globalRTT    = sourceRTT
```

This use case ends when the repair server or sender sends a Get-RTT response packet.

R.5 Processing a Received Get-RTT Response Packet

This use case begins when a Get-RTT response packet is received at a receiver or repair server. The receiver or repair server computes the local round trip time as:

```
    myUpRTT = (currentTime - xmitTime)
```

Next, if either the value of maxUpRTT is equal to -1 or the value of peerGroupRTT is equal to -1, it immediately issues another Get-RTT request as follows:

```
    if ((maxUpRTT == -1) || (peerGroupRTT == -1))
    {
      resendInterval = myUpRTT
      Cancel the current Get-RTT resend timer
      Start the Get-RTT timer with a duration of resendInterval
      Send a Get-RTT request packet (xmitTime = currentTime, upRTT = myUpRTT)
    }
```

The resend interval is doubled and limited to the value of maxGetRTTCycleTime:

```
    resendInterval = (resendInterval * 2)
    if (resendInterval > maxGetRTTCycleTime)
    {
      resendInterval = maxGetRTTCycleTime
    }
```

```
maxUpRTT is set to either the value of peerGroupRTT from the packet or the new
value of myUpRTT, whichever is greater:

    if (myUpRTT > peerGroupRTT)
    {
      maxUpRTT = myUpRTT
    }
    else
    {
      maxUpRTT = peerGroupRTT
    }

Finally, if the value of globalRTT in the response packet is nonnegative, the
receiver or repair server then computes and sets the value of sourceRTT as the
sum of the value of globalRTT and the value of myUpRTT:

    if (globalRTT >= 0)
    {
      sourceRTT = (globalRTT + myUpRTT)
    }
```

B.3 The NCA-NOM Protocol

```
                NCA Use Cases : Nomination Algorithm
                -------------------------------------
                             1999-12-08
                            Version: 1.0


Section D: Sender Protocol

  State Information:
    csmNominee     - The address of the current nominee receiver.
    csmLPE         - The current loss probability estimate reported by the
                     nominee receiver.
    csmRTT         - The current round trip time to the sender reported by the
                     nominee receiver.
    csmSetTime     - The time, in milliseconds, when csmLPE, csmRTT and
                     csmNominee were set.

  D.1 Initialization

  This use case begins when the sender is instantiated. The state information
is initialized as follows:

    csmNominee     = 0
    csmLPE         = -1.0
    csmRTT         = 0
    csmSetTime     = currentTime in milliseconds

This use case ends when the state information is initialized.

  D.2 Processing a Received CSM Packet

  This use case begins when a CSM packet arrives at the sender. The following
```

test is performed (addr, lpe and rtt are the CSM packet receiver address, loss probability estimate and source to receiver round trip time in milliseconds. currentTime is the current time in milliseconds. The timeout parameter is set to some constant (e.g., 17000 milliseconds).):

```
    if ((csmNominee == 0) OR
        (addr == csmNominee) OR
        ((0.0 <= lpe <= 1.0) AND (csmLPE < 0.0)) OR
        ((0.0 <= lpe <= 1.0) AND (0.0 <= csmLPE <= 1.0) AND
         (g(lpe, rtt) > g(csmLPE, csmRTT))) OR
        ((lpe < 0.0) AND (csmLPE < 0.0) AND (rtt > csmRTT)) OR
        (currentTime > (csmSetTime + 17000)))
    {
      Call Use Case D.3
    }
```

The function g(lpe,rtt) is as follows:

```
    g(lpe,rtt) = (rtt * squareroot(lpe)), 0 <= lpe <= 1, rtt >= 0
               = 0,                       otherwise
```

This use case ends when either Use Case D.3 returns or the CSM packet is ignored.

D.3 Update Nominee Receiver Parameters

This use case begins when a CSM is receiver from the nominee receiver. The CSM packet is processed as follows:

```
      oldNominee = csmNominee
      csmNominee = CSM packet receiver address
      csmLPE     = CSM packet loss probability estimate
      csmRTT     = CSM packet receiver to source round trip time
      csmSetTime = currentTime in milliseconds
      if (oldNominee != csmNominee)
      {
        if (csmNominee != 0)
        {
          Send a NAM packet to csmNominee (nominee address = csmNominee)
        }
        if (oldNominee != 0)
        {
          Send a NAM packet to oldNominee (nominee address = csmNominee)
        }
        Cancel NAM Timer
        Start NAM Timer with a constant duration (e.g., 7000 milliseconds)
      }
```

This use case ends when either NAM Timer is restarted or csmSetTime is set.

D.4 NAM Timer Service Routine

This use case begins when the NAM Timer expires. The following processing is performed:

```
    if (csmNominee != 0)
    {
      Send a NAM packet to csmNominee (nominee address = csmNominee)
```

```
      Restart NAM Timer with a constant duration (e.g., 7000 milliseconds)
    }
```

This use case ends when either the NAM Timer is restarted or no action is taken.

Section E: Receiver Protocol

```
  State Information :
    isNomineeFlag - If true, indicates that this receiver has been nominated
                    to send congestion control signals to the sender,
                    otherwise false.
```

E.1 Initialization

This use case begins when the receiver protocol is initialized by the upper layers. The state information is initialized as follows:

```
    isNomineeFlag = false
```

A sliding loss probability estimate (LPE) window with a fixed maximum size (e.g., 200 packets) is created for recording the arrival of original data packets and computing the LPE. The CSM Timer is started with a duration of a random variate, uniformly distributed between 0 and 1.0, times some fixed time value (e.g., 5000 milliseconds). This use case ends when the CSM Timer has been started.

E.2 Processing a Received Original Data Packet

This use case begins when an original data packet (not a retransmitted data packet) is received. The sliding LPE window is updated to record the reception of the data packet using the following rules:

- the window may expand from being empty to the maximum size (e.g., 200)
- the window never shrinks in size
- once at the maximum window size, the window may be slid forward while the window size remains constant
- the window may only be slid forward (never backward)

This use case ends when the sliding LPE window has been updated.

E.3 CSM Timer Service Routine

This use case begins when the CSM Timer expires. A CSM packet is created with the LPE and sourceRTT fields set as follows:

```
    if (current sliding LPE window size < 250)
    {
      CSM packet LPE field = -1
    }
    else
    {
      CSM packet LPE field = (1000 * (windowSize - capsulesReceived) /
                              windowSize)
    }
    if (receiver to source round trip time (RTT) is unavailable)
    {
```

```
      CSM packet source RTT field = -1
    }
    else
    {
      CSM packet source RTT field = current receiver to source RTT estimate
        in milliseconds
    }
```

The completed CSM packet is unicast to the receiver's upstream repair server.
The CSM Timer is restarted using a constant duration (e.g., 5000 milliseconds).
This use casse ends when the CSM Timer is restarted.

E.4 Processing a Received NAM Packet

This use case begins when a NAM packet is received. The following processing is performed:

```
  if (NAM packet nominee address == receiver's local address)
  {
    isNomineeFlag = true
    Call Enable Fast Repair Use Case in the repair service algorithm
  }
  else
  {
    isNomineeFlag = false
    Call Disable Fast Repair Use Case in the repair service algorithm
  }
```

This use case ends when isNomineeFlag has been updated.

Section F: Repair Server Protocol

State Information:

```
    csmAddress      - The address of the worst receiver downstream.
    csmLPE          - The current loss probability estimate reported by the
                      worst receiver downstream.
    csmRTT          - The current round trip time to the sender reported by
                      the worst receiver downstream.
    csmSetTime      - The time, in milliseconds, when csmLPE, csmRTT and
                      csmNominee were set.
```

F.1 Initialization

This use case begins when the repair server receives the first SPM packet.
The state information is initialized as follows:

```
    csmAddress     = 0
    csmLPE         = -1.0
    csmRTT         = 0
    csmSetTime     = currentTime in milliseconds
```

The CSM Timer is started using a constant duration (e.g., 7000 milliseconds).
This use case ends when the CSM Timer has been started.

F.2 Processing a Received CSM Packet

This use case begins when a CSM packet arrives at the repair server. The following test is performed (addr, lpe and rtt are the CSM packet receiver address, loss probability estimate and sender to receiver round trip time in milliseconds. currentTime is the current time in milliseconds. The timeout parameter is set to some constant (e.g., 17000 milliseconds)):

```
if ((csmAddress == 0) OR
    (addr == csmAddress) OR
    ((0.0 <= lpe <= 1.0) AND (0.0 <= csmLPE <= 1.0) AND
     (g(lpe, rtt) > g(csmLPE, csmRTT))) OR
    ((lpe < 0.0) AND (csmLPE < 0.0) AND (rtt > csmRTT)) OR
    (currentTime > (csmSetTime + 17000)))
{
  Call Use Case F.3
}
```

The function g(lpe,rtt) is as follows:

```
g(lpe,rtt) = (rtt * squareroot(lpe)), 0 <= lpe <= 1, rtt >= 0
           = 0,                       otherwise
```

This use case ends when either Use Case F.3 returns or the CSM packet is ignored.

F.3 Update Worst Receiver Parameters

This use case begins when a CSM packet is received from the worst receiver downstream. The CSM packet is processed as follows:

```
csmAddress = CSM packet receiver address
csmLPE     = CSM packet loss probability estimate
csmRTT     = CSM packet receiver to source round trip time
csmSetTime = currentTime
if (csmAddress != 0)
{
  Send a CSM packet upstream to this repair server's repair server
    (lpe = csmLPE, sourceRTT = csmRTT, receiver's address = csmAddress)
}
Cancel the CSM Timer
Start the CSM Timer with a constant duration (e.g., 7000 milliseconds)
```

This use case ends when the CSM Timer is restarted.

F.4 CSM Timer Service Routine

This use case begins when the CSM Timer expires. The following processing is performed:

```
if (currentTime in milliseconds > (csmSetTime + 17000))
{
  csmAddress = 0
  csmLPE     = -1.0
  csmRTT     = 0
  csmSetTime = currentTime in milliseconds
}
if (csmAddress != 0)
{
  Send a CSM packet with the receiver address set to csmAddress, the
    LPE set to csmLPE and the sender RTT set to csmRTT to this repair
```

```
        server's repair server
    }
    Restart the CSM Timer with a constant duration (e.g., 7000 milliseconds)

This use case ends when the CSM Timer has been started.


Appendix A: Packet Formats

 Congestion Status Message (CSM) Packet:
   CSM packets are unicast from a receiver or repair server to its upstream
   repair server, which may be the sender.

   Fields: Version Number, Packet Type, Multicast Group Address, Port Number,
           Loss Probability Estimate, Source to Receiver RTT.

 Nominee Activation Message (NAM) Packet:
   Unicast from the sender to a receiver with no intervention by any repair
servers.

   Fields: Version Number, Packet Type, Multicast Group Address, Port Number,
           Nominee Address
```

B.4 The NCA-RC Protocol

```
                NCA Use Cases : Rate Control Algorithm
                --------------------------------------
                              1999-12-08
                             Version: 1.0


Section G: Sender Protocol

  State Information:
    nominee          - The current nominee receiver address.  There is no
                       nominee if equal to 0.
    timerNominee     - The address of the nominee receiver when the CCM Timer
                       was set.
    winSize          - The current congestion window sizer. Must be greater
                       than or equal to 1.0 at all times.
    ssThreshold      - The slow start threshold window size.
    winLowerSeq      - The sequence number of the oldest original data packet
                       that has not received a CCM packet response. Always the
                       lower end of the congestion window. Must be greater
                       than or equal to 0.
    dupLowerSeq      - The number of duplicate CCM packets received for the
                       winLowerSeq data packet.
    maxSeqSent       - The sequence number of the most recent original data
                       packet sent. This may be either in or outside of the
                       congestion window. Must be greater than or equal to 0.
    smoothedRTT      - The smoothed RTT estimate in milliseconds.
    rttVariance      - The mean deviation of the RTT estimate in milliseconds
                       (a la TCP).
    ccmTimeout       - The maximum amount of time in milliseconds to wait for
                       a CCM packet. If this time expires and no CCM packet is
```

```
                     received, then the congestion window size is dropped to
                     1.0.
   backoffFactor    - The exponential backoff factor used in case of
                     successive timeouts.
   fastRecovFlag    - A flag for recording if fast recovery is active or not.
   fastRecovSeq     - The sequence number of the last data packet in the
                     congestion window when fast recovery is activated.
                     Set to 0 when fast recovery is not active.
   maxBurstCount    - The number of original data packets sent since the last
                     CCM packet was received.
```

G.1 Initialization

This use case begins when the sender is instantiated. The constants used by NCA are as follows:

```
    MAX_WINDOW      = 256
    MAX_BURST       = 4
    FR_MAX_BURST    = 2
    MIN_SS_THRESH   = 2
    MAX_BACKOFF     = 64
```

The state information is initialized as follows:

```
    nominee       = 0
    timerNominee  = 0
    winSize       = 1.0
    ssThreshold   = 64
    winLowerSeq   = initial data packet sequence number
    dupLowerSeq   = 0
    maxSeqSent    = (winLowerSeq - 1)
    smoothedRTT   = 0
    rttVariance   = 0
    ccmTimeout    = 1000
    backoffFactor = 1
    fastRecovFlag = false
    fastRecovSeq  = 0
    maxBurstCount = 0
```

This use case ends when the state information has been initialized.

G.2 Processing a Received CCM Packet

This use case begins when a CCM packet arrives at the sender. The CCM packet is processed as follows (ccmAddr, ccmRetransFlag, seq and timestamp are the CCM packet IP source address, retransmission flag, sequence number and time stamp):

```
    if (ccmAddr == nominee)
    {
      backoffFactor = 1
      maxBurstCount = 0
      if (ccmRetransFlag == false)
      {
        Call Use Case G.3 passing timestamp
      }
      Call Use Case G.5 passing seq
      if (winLowerSeq > maxSeqSent)
```

```
      {
        Cancel CCM Timer
      }
    }
```

This use case ends when either Use Case G.4 returns or the CCM Timer is canceled.

G.3 Updating the RTT Estimate

This use case begins when the sender receives a CCM and has to update its RTT estimate. The CCM packet is processed as follows (timestamp is the time stamp value of the CCM packet passed to this use case):

```
    newRTT = (timestamp - currentTime in milliseconds)
    if (smoothedRTT == 0)
    {
      smoothedRTT = newRTT
      ccmTimeout  = smoothedRTT
    }
    else
    {
      Call Use Case c.4 passing the value of newRTT
    }
```

This use case ends when either smoothedRTT and ccmTimeout are updated or Use Case G.4 returns.

G.4 Updating the RTT Estimate and Variance

This use case begins when both the RTT estimate and variance must be updated given a new value of newRTT. The following processing is performed:

```
    error = (newRTT - smoothedRTT)
    smoothedRTT = (smoothedRTT + (error / 8))
    if (rttVariance == 0)
    {
      rttVariance = ABS(error)
    }
    else
    {
      rttVariance = (rttVariance + ((ABS(error) - rttVariance) / 4))
    }
    ccmTimeout = (smoothedRTT + (4 * rttVariance))
```

Note that ABS() is the absolute value function. This use case ends when ccmTimeout has been updated.

G.5 Updating the Congestion Window

This use case begins when the congestion window needs to be updated due to the arrival of a CCM packet. The congestion window is updated as follows (seq is the sequence number of the CCM packet passed to this use case):

```
    if (seq == winLowerSeq)
    {
      dupLowerSeq = (dupLowerSeq + 1)
    }
```

```
    else
    {
      if (seq > winLowerSeq)
      {
        Call Use Case C.6
      }
      if (fastRecovFlag == true)
      {
        if (winLowerSeq > fastRecovSeq)
        {
          Call Use Case G.9
        }
        else
        {
          Call Use Case G.8
        }
      }
      else
      {
        if (dupLowerSeq >= 3)
        {
          Call Use Case G.10
        }
      }
```

This use case ends when the congestion window has been updated.

G.6 Sliding the Congestion Window Forward

This use case begins when the congestion window is ready to be slid forward. The congestion window is updated as follows (seq is the sequence number of the CCM packet passed to this use case):

```
    if (fastRecovFlag == false)
    {
      Call Use Case G.7
    }
    winLowerSeq = seq
    dupLowerSeq = 0
```

This use case ends when dupLowerSeq is set to 0.

G.7 Increasing the Congestion Window Size

This use case begins when a CCM packet is received and the congestion window size must be increased while not in fast recovery. The congestion window is updated as follows:

```
    if (winSize <= ssThreshold)
    {
      winSize = (winSize + 1.0)
    }
    else
    {
      winSize = (winSize + (1.0 / FLOOR(winSize)))
    }
    winSize = MIN(winSize, MAX_WINDOW)
    winSize = MAX(winSize, 1.0)
```

Note that the FLOOR() function returns the largest value that is not greater than the argument and is equal to a mathematical integer. This use case ends when winSize has been updated.

G.8 Increasing the Congestion Window Size in Fast Recovery

This use case begins when a CCM packet is received and the congestion window size must be increased while in fast recovery. The congestion window is updated as follows:

```
winSize = (winSize + 1.0)
winSize = MIN(winSize, MAX_WINDOW)
winSize = MAX(winSize, 1.0)
```

This use case ends when winSize has been updated.

G.9 Leaving Fast Recovery

This use case begins when the sender must exit the fast recovery state. The following processing is performed:

```
fastRecovFlag = false
fastRecovSeq  = 0
winSize       = ssThreshold
winSize       = MIN(winSize, MAX_WINDOW)
winSize       = MAX(winSize, 1.0)
```

This use case ends when winSize has been updated.

G.10 Entering Fast Recovery

This use case begins when the sender must enter the fast recovery state. The following processing is performed:

```
fastRecovFlag = true
fastRecovSeq  = (winLowerSeq + FLOOR(winSize) - 1)
ssThreshold   = MAX((winSize / 2.0), MIN_SS_THRESH)
winSize       = (ssThreshold + 3.0)
winSize       = MIN(winSize, MAX_WINDOW)
winSize       = MAX(winSize, 1.0)
```

This use case ends when winSize has been updated.

G.11 Testing if an Original Data Packet Can Be Sent

This use case begins when an original data packet with a sequence number of (maxSeqSent + 1) is to be sent. The following processing is performed:

```
if (maxSeqSent >= (winLowerSeq + FLOOR(winSize) - 1))
{
  The original data packet cannot be sent at the current time
  End Use Case
}
if (fastRecovFlag == true)
{
  if (maxBurstCount >= FR_MAX_BURST)
  {
```

```
      The original data packet cannot be sent at the current time
      End Use Case
    }
  }
  else
  {
    if (maxBurstCount >= MAX_BURST)
    {
      The original data packet cannot be sent at the current time
      End Use Case
    }
  }
  The original data packet can be sent at the current time
```

This use case ends when the determination is made to either allow or disallow the original data packet to be sent.

G.12 Sending an Original Data Packet

This use case begins when an original data packet with a sequence number of (maxSeqSent + 1) is ready to be sent and Use Case G.11 allows the packet to be sent. The following processing is performed:

```
  maxSeqSent    = (maxSeqSent + 1)
  maxBurstCount = (maxBurstCount + 1)
  Cancel CCM Timer
  if (maxSeqSent >= winLowerSeq)
  {
    timerNominee = nominee
    Start the CCM Timer with a duration of (backoffFactor * ccmTimeout)
  }
```

This use case ends when the CCM Timer is either canceled or canceled and restarted.

G.13 CCM Timer Service Routine

This use case begins when the CCM Timer expires. The following processing is performed:

```
  if (timerNominee == 0)
  {
    ssThreshold   = 64
    backoffFactor = 1
  }
  else
  {
    ssThreshold   = MAX((winSize / 2.0), MIN_SS_THRESH)
    backoffFactor = MIN((backoffFactor * 2), MAX_BACKOFF)
  }
  winSize       = 1.0
  winLowerSeq   = (maxSeqSent + 1)
  fastRecovFlag = false
  fastRecovSeq  = 0
  maxBurstCount = 0
```

This use case ends when winLowerSeq has been updated.

G.14 Updating the Nonimee Receiver

This use case begins when the nominee receiver address has changed and must be updated. The following processing is performed (addr is the new nominee address):

```
nominee = addr
if (nominee == 0)
{
  ccmTimeout = the initial value (e.g., 1000 milliseconds)
}
```

This use case ends when nominee has been updated.

Section H: Receiver Protocol

State Information:

```
isNomineeFlag - If true, indicates that this receiver has been nominated
                to send congestion control signals to the sender,
                otherwise false (from NCA Nomination Algorithm).
```

H.1 Processing a Received Data Packet

This use case begins when a data packet is received. All receivers must maintain the lowest sequence number of all outstanding data packets at all times. The following processing is performed:

```
if ((isNomineeFlag == true) AND
    (the data packet has not been received before))
{
  Send a CCM packet to the sender
    (seq = lowest outstanding data packet sequence number,
     timestamp = data packet timestamp,
     retrans = data packet retransmission flag)
}
```

This use case ends when either a CCM packet is sent to the sender or no processing occurs.

Appendix A: Packet Formats

Congestion Control Message (CCM) Packet:
Unicast from the nominated receiver to the sender.

Fields: Version Number, Packet Type, Multicast Group Address, Port Number, Sequence Number, Timestamp, Retransmission Flag