MODELING AND VERIFICATION OF REAL-TIME AND CYBER-PHYSICAL SYSTEMS

by

Neda Saeedloei

To my mother.

MODELING AND VERIFICATION OF REAL-TIME AND CYBER-PHYSICAL SYSTEMS

by

NEDA SAEEDLOEI, B.S., M.S.

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

December 2011

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor Dr. Gopal Gupta, whose continuous support, guidance, and patience made this research work possible. He will be always my role model not just as a great professor but as a wonderful person who made my entire PhD work a fruitful and enjoyable experience. I will always be grateful to him for giving me wise and valuable advice in academic and non-academic matters. He sets a great example for all the current and future professors.

I would like to thank our committee members: Dr. Farokh Bastani, Dr. Haim Schweitzer and Dr. Kevin Hamlen, for providing valuable feedback. Special thanks to Dr. Bastani for his thoughtful suggestions and wise advice that helped me throughout job search.

I had the privilege of having Dr. Feliks Kluźniak work in our research group for three out of the four years of my PhD work. I would like to express my gratitude for Dr. Kluźniak whose valuable advice and critiques have helped me in developing theoretical foundations of this dissertation. I am grateful to him for long and fruitful discussions, proof reading of my papers and this dissertation, and maintaining a friendly relationship. I could always go to him when I needed to discuss something, academic or personal, and he would always give me sound and thougtful advice.

I thank my colleagues Brian Devries and Kyle Marple for all discussions and suggestions. I always enjoyed our weekly group meetings and benefited from our constructive discussions. I thank my friend Julie Rauer for her support and help.

Finally, I would like to thank my family, especially my mother, for their persistent love, support and encouragement. I thank my sister Mona, whose voice always gives me a positive energy.

September 2011

MODELING AND VERIFICATION OF REAL-TIME AND CYBER-PHYSICAL SYSTEMS

Publication No. ___________________

Neda Saeedloei, Ph.D.
The University of Texas at Dallas, 2011

Supervising Professor: Dr. Gopal Gupta

There has been a tremendous amount of work on incorporation of time in computation for modeling and implementing real-time systems. Most efforts incorporate time by discretizing it. Discretizing means that time is represented through finite time intervals. As a result, *infinitesimally small time intervals cannot be represented or reasoned about in these approaches.* In practical real-time systems, e.g., a nuclear reactor, two or more events *can* occur within an infinitesimally small interval. In this dissertation, we develop techniques for including continuous time in computations. These techniques result in frameworks for modeling and verification of real-time systems. Our proposed framework is based on *logic programming* (*LP*) extended with *constraints* and *coinduction.* To build the theoretical foundations necessary for this work, we present a new programming paradigm, *co-constraint logic programming* (*co-CLP* for brevity), that merges two programming paradigms: *constraint logic programming* and *coinductive logic programming.* Constraint logic programming (CLP) has been proposed as a declarative paradigm for merging constraint solving and logic programming. Coinductive logic programming, on the other hand, has been proposed as a powerful extension of logic programming for handling infinite (rational) objects and reasoning about their properties. Since CLP's declarative semantics is given in terms of a least fixed-point, i.e., it is inductive, it cannot be directly used for reasoning about infinite and cyclical objects and their properties. Coinductive logic programming does not include constraints. *Co-CLP* combines both constraint logic programming and coinduction and enables us to reason about infinite (rational) structures in the presence of constraints.

In this dissertation, we investigate incorporation of continuous time in various domains of computer science. In particular we study the extension of ω-grammars with continuous time (timed ω-grammars) and also continuous time extension of π-calculus (timed π-calculus). We present the implementation of these formalisms in co-CLP and show how these co-CLP-based realization can be used for modeling and verifying real-time systems. We also present a co-CLP framework for modeling hybrid automata, of which pushdown timed automata and timed automata are instances; we use this logic programming realization of hybrid automata to develop a framework for modeling and verification of cyber-physical systems, including real-time systems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

In a real-time system the correctness of the system's behavior depends not only on the tasks that the system is designed to perform, but also on the physical instants at which these tasks are performed. Real-time systems can be described as the class of systems that have to respond to externally generated signals or inputs within specified time limits. Also, the future behavior of such systems depends on the times at which the external signals are received. This type of systems includes many safety critical systems such as in robotics and flight control. Errors in design and modeling of real-time systems can be extremely dangerous, due to the nature of such systems. Proving the correctness of real-time systems and verifying their various properties such as safety, utility and liveness are non trivial tasks.

There has been tremendous amount of work on design, specification, and verification of real-time systems, which has resulted in various formalisms and frameworks for modeling time and incorporation of time in computations. These approaches can be divided in to three categories, depending on how they model time.

The first approach is the *fictitious clock* model, in which a special transition called *tick* is introduced in the model. Time is viewed as a global state variable that ranges over the domain of natural numbers and is incremented by one with every *tick* transition (Aggarwal and Kurshan, 1983, Alur and Henzinger, 1990). In this model, the behavior of a system is specified as a timed trace of events, in which events occur in the specified order at real-valued times; however, only the integer readings of the actual times with respect to a digital clock are recorded in the trace. This model allows arbitrary many transitions between two successive *tick* transitions. The timing delay between two events is measured by counting the number of *ticks* between them. When it is required to have k ticks between two transitions, we can only infer that the delay between them is larger than $K - 1$ time units and smaller than $K + 1$ time units. As a result, certain simple requirements on delays such as "the delay between two transitions equals 5 units of time" is impossible to express in this model.

The second approach is based on *discrete-time models*, in which the domain of integers is used to model time. When time is modeled as a discrete quantity, all the events in a real-time system occur at nonnegative integer time values. In order to model real-time systems using discrete time, some fixed time quantum is chosen *a priori*, and the delay between any two events in the system is represented as a multiple of this time quantum. Therefore, the continuous time has to be approximated. While modeling real-time systems by discretizing time might affect the accuracy of the system being modeled, this approach has a lot of applications in certain kinds of real-time systems in which inaccuracy up to some level is acceptable.

The last approach is based on *dense-time* model, in which the domain of reals is used to model time, and it is more realistic physically. We next present a very brief overview of discrete and continuous real-time systems.

1.1 Discrete Real-Time Systems

Symbolic Model Checking techniques have been used to express and verify some discrete real-time systems; however, complex timing properties are very difficult or impossible to be expressed in this approach. Temporal logic is a formalism for describing sequences of transitions between states in a reactive/real-time system. CTL^* is a powerful example of temporal logic which can be used for verifying some discrete real-time systems; however, time is not mentioned explicitly in this formalism. For instance, it is possible to express, using special temporal operators, the property that "event `e` will happen in the future", or "event `e` will never happen"; however, it is not simple to express the property that "event `e` will happen in at most `t` time units". Moreover, obtaining quantitative information such as response time or the number of occurrences of events are not straightforward. In summary, many types of real-time systems that occur frequently in practice, cannot be modeled and verified in a natural and efficient way by using these techniques.

Symbolic Model Checking techniques have also been *extended* for handling real-time systems (Cleaveland et al., 1996, Cleaveland and Sims, 1996, Yang et al., 1993); However, these methods only determine if the system satisfies a given property, and do not provide detailed information on its behavior. Some restricted quantitative analysis, e.g., computing minimum/maximum delays, can be performed on discrete real-time systems. RTCTL is an extension of CTL, obtained by introducing bounds in the CTL temporal operators (Emerson et al., 1991). RTCTL has been proposed as an effective way to allow the verification of time-bounded properties. For instance, it is possible to define the time interval in which some property p must be true using the basic RTCTL temporal operator, such as the *bounded until* operator.

1.2 Continuous Real-Time

While some real-time systems can be modeled using discrete time, discretizing time in general has some drawbacks. Discretizing means that time is represented through finite time intervals. As a result, *infinitesimally small time intervals cannot be represented or reasoned about in these approaches.* In practical real-time systems, e.g., a nuclear reactor, two or more events *can* occur within an infinitesimally small interval. As another example, it has been shown that the reachability problem for asynchronous circuits with bounded delays cannot be solved correctly when time is assumed to be discrete (Brzozowski and Seger, 1991).

Different models of continuous time have been proposed (Alur et al., 1990, Dill, 1990, Henzinger et al., 1992, Yoneda et al., 1993, Merritt et al., 1991). Automata based real-time formalisms such as timed graphs (Alur et al., 1990), timed automata (Alur and Dill, 1994, Heitmeyer and Lynch, 1994) and timed transition systems have been proposed to model and analyze a wide range of real-time systems. The *timed automata* model of Alur et. al (Alur and Dill, 1994, Heitmeyer and Lynch, 1994) has been extensively studied and has become the standard. A timed automaton is a finite state automaton equipped with a set of clocks which are used to measure the delays between transitions. Timed automata are used to model finite-state real-time systems by allowing us to express constant bounds on the delays between the system transitions. Most of the research on modeling and verification of real-time systems is based on timed automata. Algorithms for CTL Model Checking (Alur et al., 1990) and LTL Model Checking (Alur, 1991) based on timed automata have been proposed. The inclusion problem for timed automata, closure properties of timed automata (Alur and Dill, 1994), and the emptiness of the language of a given timed automaton have been also studied.

Typically, verification of real-time, concurrent systems using timed automata reduces to solving the language inclusion problem. This consists of the following three steps. First, an implementation of a concurrent real-time system is modeled as a parallel composition of components of the system. In other words, the first step is to construct a single timed automaton A that describes the entire system. The parallel composition of a set of timed automata is the product of the automata. While building the product automaton is an entirely syntactical operation, it is computationally expensive. The second step includes building another timed automaton S, which describes the specification of the system. The last step is to check that the language of A is a subset of the language of S. This is equivalent to checking the emptiness of the intersection of A with the complement of S. However, the complement of the language of a timed automaton cannot always be described as a timed automaton; even if it can, complementing an arbitrary timed automaton might involve an exponential blow-up in the number of states. The problem of checking the language inclusion is undecidable, in general. However, for two timed automata A and B, checking $L(A) \subseteq L(B)$ is decidable, if B is a deterministic timed automaton. The undecidability of the language inclusion problem for timed automata is introduced by the arbitrary clock resets.

The most interesting feature of a timed automaton is the *reachability* of a given final state or set of final states, which is used for verifying *safety* properties. A safety property is usually characterized by the fact that some states cannot be reached. The infinite state-space of a timed automaton has to be finitely partitioned into symbolic states using clock constraints known as *clock regions*, for algorithms related to timed automata, e.g., reachability, to be decidable. Region reachability is useful but has intrinsic limitations. In many real-world applications, one might want to check whether a timed automaton satisfies a non-region property. For instance, one might want to check whether the difference between clocks x_1 and x_2 is always greater than the difference between clocks x_3 and x_4 i.e., $x_1 - x_2 > x_3 - x_4$.

There has been some interesting work on implementing timed automata that is based on logic programming (Gupta and Pontelli, 1997). Timed automata are implemented as set of logic programming rules, extended with constraints, in this work. However, this work does not handle the infinite behavior of timed automata. Jaffar (Jaffar et al., 2004) also builds on the work of Gupta and Pontelli (1997) and translates timed automata to CLP programs and uses them for proving assertions. This work also does not model timed automata faithfully, as it cannot handle infinite timed words. UPPAAL has been proposed as a toolbox for verification of real-time systems (Behrmann et al., 2004). The tool is designed to verify real-time systems modeled as network of timed automata. The network of timed automata is the parallel composition of a set of timed automata. Since building the composition of a set of timed automata is computationally expensive, UPPAAL builds the product automaton on-the-fly during verification. The verification engine of UPPAAL is written in C++, and the query language, which is used to specify the properties to be checked, is a simplified version of CTL (UPPAAL does not allow nesting of path formulae).

Extension of timed automata to pushdown timed automata (PTA) has also been proposed (Dang, 2001). A pushdown timed automaton is a timed automaton with dense clocks, augmented with a pushdown stack. PTA were considered by Dang (Dang, 2001) and used to give a decidable characterization of the binary reachability of PTA. The results can be used to verify a class of safety properties containing linear relations over dense variables and unbounded discrete variables. However, the treatment in that work is largely theoretical, there is not much focus on how to efficiently realize a pushdown timed automaton.

Hybrid automata were introduced as a model and specification language for hybrid systems: dynamical systems whose behavior exhibits both discrete and continuous change (Alur et al., 1992). Hybrid automata are a generalization of timed automata, in which the behavior of variables is governed in each state by a set of differential equations. Hybrid automata combine automata transitions for capturing discrete change with differential equations for capturing continuous change.

Model checking of hybrid systems requires exploration of the entire state space of the system. However, the state space for hybrid systems is infinite; so, an enumerative approach that considers each state individually is impossible. However, a symbolic approach for a class of linear hybrid automata has been proposed (Henzinger and hsin Ho, 1995, Alur et al., 1997). The implementation of this approach is a verifier for hybrid systems, called HYTECH (Henzinger and hsin Ho, 1995, Alur et al., 1997). HYTECH is a symbolic model checker that can be used for verifying a limited class of hybrid systems: those that can be specified by linear hybrid automata. The termination of the model checking procedure is not guaranteed in HYTECH —the model checking problem for linear hybrid automata is undecidable— but the method is still of practical interest. Termination can be enforced in models in which the termination does not happen naturally by considering the behavior of a system over a bounded interval of time. Many physical quantities involved in a hybrid automaton modeling a hybrid system, such as temperature, exhibit nonconstant derivatives. These types of hybrid automata are called non-linear hybrid automata. When using linear hybrid automata for modeling, as in HYTECH, physical quantities whose rate of change are not constants must be either translated into piecewise-linear quantities or approximated by rate intervals. These two approaches for converting non-linear hybrid automata to linear hybrid automata are called *clock translation* and *rate translation*, respectively.

1.3 Dissertation Objective

The goal of this research is to develop techniques for incorporation of continuous time in computations and to generalize them to techniques for handling various continuous physical quantities. This research results in formalisms and frameworks for modeling and verification of real-time systems, hybrid systems and cyber-physical systems: formalisms and frameworks based on logic programming extended with coinduction and constraints. It also includes developing theoretical foundations that will support our proposed frameworks.

Combining coinductive logic programming and constraint logic programming into a single programming paradigm, coinductive constraint logic programming (co-CLP), constitutes the first main contribution of this dissertation. Each of these programming paradigms have been realized individually; however, their integration into a single programming paradigm has never been attempted. The principles of coinductive constraint logic programming provides the fundamental foundations for the rest of our work in this dissertation. This formalism will be used when we develop logic programming-based techniques for realizing timed automata and pushdown timed automata. We employ these realizations to propose a framework for modeling and verifying real-time systems.

The second main contribution of this dissertation is to propose timed ω-grammars. In particular the time extension of context-free grammars are proposed. Timed languages, realized by timed automata, have been used for describing the behavior of real-time systems; however, the concept of timed grammars and timed ω-grammars is introduced for the first time (Saeedloei and Gupta, 2010). Timed grammars provide a simple and natural way for describing the behavior of real-time systems at a higher level of abstraction. We will use the principles of co-CLP to develop effective and practical parsers for timed context-free ω-languages, recognized by timed context-free ω-grammars.

The third main contribution of this dissertation is to propose a time extension of π-calculus. We extend π-calculus with real-time by adding clocks and assigning time to actions. The proposed timed π-calculus provides a simple and novel way to annotate transition rules of π-calculus with timing constraints. Timed π-calculus is an expressive way of describing mobile, concurrent, real-time systems in which the behaviors of systems are modeled by finite or infinite sequences of timed events. We provide an operational semantics and present the outline of an implementation of this operational semantics, an implementation based on co-CLP and coroutining. We show how timed π-calculus can be used for modeling and verification of complex real-time systems.

The last contribution of this dissertation is to investigate foundations of cyber-physical systems (CPS), and propose a framework for modeling and verification of such systems. CPS consist of perpetually and concurrently executing physical and computational components. The presence of physical components requires the computational components to deal with continuous quantities. A formalism that can model discrete and continuous quantities together with concurrent, perpetual execution is lacking. We report on the development of a formalism based on co-CLP and coroutining that allows CPS to be elegantly modeled. This logic programming realization can be used for verifying interesting properties of CPS.

Thus the main contributions of this work are as follows:

(i) Proposing a new programming paradigm, co-CLP, as a merger of two programming paradigms: constraint logic programming and coinductive logic programming, along with a proof of correctness (soundness and completeness),

(ii) Modeling pushdown timed automata and timed automata using principles of co-CLP,

(iii) Employing the logic programming realization of pushdown timed automata/timed automata to propose a general framework for modeling and verification of real-time systems (including CPS),

(iv) Proposing timed ω-grammars and timed context-free ω-grammars and developing practical parsers for timed context-free ω-languages in coinductive constraint logic programming,

(v) Extending the π-calculus with continuous time, developing an operational semantics for the resulting timed π-calculus and outlining its implementation using co-CLP extended with coroutining,

(vi) Proposing a framework for modeling and verification of cyber-physical systems using co-CLP extended with coroutining.

1.4 Dissertation Outline

The remainder of this dissertation is organized as follows.

Chapter 2 presents the background concepts that are used throughout the dissertation. It contains a brief overview of *logic programming*, *coinductive logic programming* and *constraint logic programming*. It also presents the formalisms of timed automata and pushdown timed automata.

We present a new programming paradigm, co-CLP, which merges two powerful paradigms: constraint logic programming and coinductive logic programming in Chapter 3. This chapter is organized in two parts. In the first part we consider coinductive logic programming with constraints. The resulting programming paradigm is called *coinductive constraint logic programming* (*coinductive CLP*). We present a new declarative semantics and also an operational semantics based on greatest fixed-point semantics. In the second part we extend co-LP with constraints; we call this new paradigm *co-constraint logic programming* (*co-CLP*). Declarative and operational semantics of co-CLP are also presented in this chapter. This chapter also presents the correctness as well as the completeness results for coinductive CLP and co-CLP.

We develop a framework for modeling timed automata and pushdown timed automata using the principles of co-CLP in Chapter 4. Next, we propose a logic programming framework for verifying complex continuous real-time systems, based on pushdown timed automata and timed automata. The application of this framework to the well known generalized railroad crossing (GRC) problem is also presented in this chapter.

Chapter 5 introduces timed ω-grammars and their application to modeling real-time systems in a natural fashion. It also presents a practical and effective technique for parsing timed languages recognized by timed ω-grammars. The equivalence of pushdown timed automata and timed context-free grammars is also discussed in this chapter.

An extension of π-calculus with continuous time is presented next in Chapter 6. This chapter also includes the theoretical results for proposed timed π-calculus, in particular timed bisimilarity and expansion theorem for concurrent, mobile, real-time processes. The outline of an implementation of the proposed timed π-calculus, based on logic programming, is also presented in this chapter.

Chapter 7 reports on the development of a formalism based on *logic programming* extended with *coinduction*, *constraints over reals*, and *coroutining* for modeling cyber-physical systems (CPS). This chapter also includes the application of our formalism to the reactor temperature control system, a traditional example of CPS.

Chapter 8 presents conclusions and some possible future research work to extend the current research presented in this dissertation.

CHAPTER 2
BACKGROUND

In this chapter we present the requisite concepts and results which are used throughout this dissertation. First, we overview the requisite basic mathematical concepts in Section 2.1. An overview of logic programming (LP) is presented in Section 2.2, coinductive logic programming in Section 2.3, and constraint logic programming in Section 2.4. All of these concepts are well-known and their detailed presentation can be found elsewhere (Lloyd, 1987, Simon, 2006, Jaffar and Maher, 1994). These basic concepts will be used explicitly in developing coinductive constraint logic programming: a formalism for merging coinductive logic programming and constraint logic programming. We also present a brief overview of timed automata and pushdown timed automata in Section 2.5 and Section 2.6, respectively. The reader is referred to (Alur and Dill, 1990, 1994) and (Dang et al., 2000) for a full account of these formalisms.

2.1 Mathematical Preliminaries

In this section, we present the requisite concepts and results concerning monotonic mappings and their fixpoints, taken directly from Lloyd (1987).

2.1.1 Fixed points

Let S be a set. A *relation* R on S is a subset of $S \times S$. For $(x, y) \in R$, we write xRy. A relation R on S is a *partial order* if (i) xRx, for all $x \in S$; (ii) xRy and yRx imply $x = y$, for all $x, y \in S$; and (iii) xRy and yRz imply xRz, for all $x, y, z \in S$. Adopting the standard notation, we use $\leq$ to denote a partial order. Therefore we have (i) $x \leq x$; (ii) $x \leq y$ and $y \leq x$ imply $x = y$; and (iii) $x \leq y$ and $y \leq z$ imply $x \leq z$, for all $x, y, z \in S$.

Let S be a set with a partial order $\leq$. Then $u \in S$ is the *least upper bound* of a subset X of S if u is an upper bound of X and, for all upper bounds u' of X, we have $u \leq u'$. Similarly, $l \in S$ is the *greatest lower bound* of a subset X of S if l is a lower bound of X and, for all lower bounds l' of X, we have $l' \leq l$. The least upper bound of X, denoted by $lub(X)$, is unique, if it exists. Similarly, the greatest lower bound of X, denoted by $glb(X)$, is unique, if it exists. A partially ordered set L is a *complete lattice* if $lub(X)$ and $glb(X)$ exist for every subset X of L.

Let L be a complete lattice and $T : L \to L$ be a mapping. Then T is *monotonic* if $T(x) \leq T(y)$, whenever $x \leq y$. Further for $X \subseteq L$, X is *directed* if every finite subset of X has an upper bound in X. For a complete lattice L, and a mapping $T : L \to L$, T is *continuous* if $T(lub(X)) = lub(T(X))$, for every directed subset X of L.

Let L be a complete lattice and $T : L \to L$ be a mapping. $x \in L$ is a *fixed-point* of T if $T(x) = x$; $l \in L$ is the *least fixed-point* of T if l is a fixed-point and for all fixed-points l' of T, we have $l \leq l'$. The *greatest fixed-point* is defined similarly.

The following proposition, which is a weak form of a theorem due to Tarski (), is stated without a proof; interested readers are referred to (Lloyd, 1987) for the proof.

Proposition 2.1.1. *Let L be a complete lattice and $T : L \to L$ be monotonic. Then T has a least fixed-point and a greatest fixed-point. Furthermore, $lfp(T) = glb\{x \mid T(x) = x\} = glb\{x \mid T(x) \leq x\}$ and $gfp(T) = lub\{x \mid T(x) = x\} = lub\{x \mid x \leq T(x)\}$.*

Next we present the concept of *ordinal numbers* (*ordinals* for brevity), *ordinal powers* and some of their properties. The first ordinal 0 is defined to be $\emptyset$. Then $1 = \{\emptyset\} = \{0\}$, $2 = \{\emptyset, \{\emptyset\}\} = \{0, 1\}$, $3 = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$, and so on. These are the finite ordinals, the non-negative integers. The first infinite ordinal is $\omega = \{0, 1, 2, \dots\}$ which is the set of all non-negative integers. After ω comes $\omega + 1 = \omega \cup \{\omega\}$, $\omega + 2 = (\omega + 1) + 1$, and so on. Next we specify an ordering $<$ on the collection of all ordinals (finite and infinite) by defining $\alpha < \beta$ if $\alpha \in \beta$. If α is an ordinal, the *successor* of α is the ordinal $\alpha + 1 = \alpha \cup \{\alpha\}$, which is the least ordinal greater than α. An ordinal α is said to be a *limit ordinal* if it is not the successor of any ordinal. The smallest limit ordinal is ω. The next limit ordinal is $\omega 2$, which is the set consisting of all n, where $n \in \omega$, and all $\omega + n$, where $n \in \omega$. Then come $\omega 2 + 1, \omega 2 + 2, \ldots, \omega 3, \omega 3 + 1, \ldots, \omega 4, \ldots, \omega n, \ldots$.

Next, we present the *principle of transfinite induction* as follows. Let $P(\alpha)$ be a property of ordinals. Assume that for all ordinals β, if $P(\gamma)$ holds for all $\gamma < \beta$, then $P(\beta)$ holds. Then $P(\alpha)$ holds for all ordinals α. Next, the *ordinal powers* are defined as follows. Let L be a complete lattice and $T : L \to L$ be monotonic. Let $\top$ denote the *top element* $lub(L)$ and $\bot$ denote the *bottom element* $glb(L)$. Then we define

$T \uparrow 0 = \bot$

$T \uparrow \alpha = T(T \uparrow (\alpha - 1))$, if α is a successor ordinal

$T \uparrow \alpha = lub\{T \uparrow \beta : \beta < \alpha\}$, if α is a limit ordinal

$T \downarrow 0 = \top$

$T \downarrow \alpha = T(T \downarrow (\alpha - 1))$, if α is a successor ordinal

$T \downarrow \alpha = glb\{T \downarrow \beta : \beta < \alpha\}$, if α is a limit ordinal

A well-known characterization of $lfp(T)$ and $gfp(T)$ in terms of ordinal powers, taken directly from Lloyd (1987), is presented next.

Proposition 2.1.2. *Let L be a complete lattice and $T : L \to L$ be monotonic. Then, for any ordinal α, $T \uparrow \alpha \leq lfp(T)$ and $T \downarrow \alpha \geq gfp(T)$. Furthermore, there exist ordinals β_1 and β_2 such that $\gamma_1 \geq \beta_1$ implies $T \uparrow \gamma_1 = lfp(T)$ and $\gamma_2 \geq \beta_2$ implies $T \downarrow \gamma_2 = gfp(T)$.*

The least α such that $T \uparrow = lfp(T)$ is called the *closure ordinal* of T. The next result, which is attributed to Kleene, shows that if T is continuous (stronger assumption), the closure ordinal of T is $\leq \omega$.

Proposition 2.1.3. *Let L be a complete lattice and $T : L \to L$ be continuous. Then $lfp(T) = T \uparrow \omega$.*

Note that proposition 2.1.3 does not hold for $gfp(T)$, that is $gfp(T)$ may not be equal to $T \downarrow \omega$.

2.2 Logic Programming

Logic programming (LP) languages belong to the class of languages called *declarative languages*. Declarative programming is a programming paradigm in which the logic of computation is expressed rather than the control flow. Declarative languages are high-level languages, in the sense that they allow the programmer to specify *what* the program should accomplish rather than describing *how* to accomplish it. Thus, declarative programming languages reduce the burden and side-effects introduced by the traditional imperative programming paradigm.

In logic programming, a subset of First Order Logic, Horn Clause Logic, is used to express the program (or a theory) in logical form. Given a program specified as a set of axioms (theory) and a goal (query), an automated theorem prover is used to prove the truth of the query.

An *alphabet* of logic programming consists of six classes of symbols: (a) variables, (b) constants, (c) function symbols, (d) predicate symbols, (e) connectives, and (f) punctuation symbols. A *term* is defined inductively as follows: (i) A variable is a term, (ii) A constant is a term, (iii) If f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term. A *ground term* is a term that does not contain variables. If p is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is an *atom* (or a *predicate*).

A *definite program clause* (called Horn's Clause) is of the form $A \leftarrow B_1, \ldots, B_n$. where $A, B_1, \ldots, B_n$ are *atoms* (or *predicates*), A is called the *head* of the clause, and $B_1, \ldots, B_n$ is called the *body* of the clause. A *definite program* is a finite set of definite program clauses. A clause without a body is called a *fact* which is written as A. A clause without a head is called a *query* (goal) which is written as $\leftarrow B_1, \ldots, B_n$.. Each B_i is called a *subgoal*.

We next present the declarative semantics of logic programming, based on the *minimal Herbrand model* semantics, following the account of Lloyd (Lloyd, 1987).

2.2.1 Declarative Semantics of Logic Programming

Let P be a logic program. The *Herbrand universe* U_P for P is the set of all ground terms, which can be formed out of the constants and function symbols appearing in P. The *Herbrand base* B_P for P is the set of all ground atoms which can be formed by using predicate symbols from P with ground terms from the Herbrand universe as arguments. The *Herbrand pre-interpretation* for P consists of the following: (i) the *domain* of pre-interpretation U_P; (ii) for each constant in P, the assignment of an element in U_P; (iii) for each n-ary function symbol in P, the assignment of a mapping from $(U_P)^n$ into U_P. An *Herbrand interpretation* I for P, consists of a pre-interpretation with domain U_P together with the following: for each n-ary predicate symbol in P, the assignment of a mapping from $(U_P)^n$ into $\{true, false\}$ (or, equivalently, a relation on $(U_P)^n$). The mapping $T_P : 2^{B_P} \rightarrow 2^{B_P}$ is defined as follows. Let I be an Herbrand interpretation. Then $T_P(I) = \{A \in B_P \mid A \leftarrow A_1, \ldots, A_n$ is a ground instance of a clause in P and $\{A_1, \ldots, A_n\} \subseteq I\}$. The *least Herbrand model* of P, denoted by M_P, is the least fixed-point of T_P. In other words, $M_P = lfp(T_P) = T_P \uparrow \omega$.

An atom A of program P is true if and only if the set of all groundings of A of B_P, with all the substitutions ranging over the U_P, is a subset of M_P. Next we present the operational semantics of logic programming.

2.2.2 Operational Semantics of Logic Programming

The declarative semantics of logic programming based on the least Herbrand model, provides an effective way to assign meaning (model) to programs; however, it does not provide a practical way to compute such meanings (models). Therefore, the language must have an operational semantics. The following account of logic programming's operational semantics, called SLD-resolution, is taken directly from Lloyd (1987).

A *substitution* θ is a finite set of the form $\{v_1/t_1, \ldots, v_n/t_n\}$, where each v_i is a variable, each t_i is a term distinct from v_i and the variables $v_1, \ldots, v_n$ are distinct. Each element v_i/t_i is called a *binding* for v_i. θ is called a *ground substitution* if the t_i are all ground terms, and *variable-pure substitution* if the t_i are all variables. The substitution given by the empty set is called the *identity substitution.* If E is an expression, $E\theta$ is the expression obtained from E by simultaneously replacing each occurrence of the variable v_i in E by the term t_i, where $i = 1, \ldots, n$. $E\theta$ is called a *ground instance* of E, if $E\theta$ is ground.

If $\theta_1 = \{v_1/t_1, \ldots, v_i/t_i\}$ and $\theta_2 = \{v_{i+1}/t_{i+1}, \ldots, v_n/t_n\}$ are substitutions, then the *composition* $\theta_1\theta_2$ of θ_1 and θ_2 is the substitution obtained from the set $\{v_1/t_1\theta_2, \ldots, v_i/t_i\theta_2, v_{i+1}/t_{i+1}, \ldots, v_n/t_n\}$ by deleting any binding $v_j/t_j\theta_2$ for which $v_j = t_j\theta_2$ for $1 \leq j \leq i$, and deleting any binding v_k/t_k for which $v_k \in \{v_1, \ldots, v_i\}$.

An *expression* is either a term, a literal or a conjunction or disjunction of literals. A *simple expression* is either a term or an atom. Two expressions E and F are *variants*, if there exist substitutions θ_1 and θ_2 such that $E = F\theta_1$ and $F = E\theta_2$. In this case, E is a variant of F or F is a variant of E. Let S be a finite set of simple expressions, a substitution θ is called a *unifier* for S if $S\theta$ is a singleton. A unifier θ for S is a *most general unifier* (*mgu*) for S if, for each unifier θ' of S, there exists a substitution γ such that $\theta' = \theta\gamma$. The *disagreement* set of a set of simple expressions S, is defined as follows. Locate the leftmost symbol position p, at which not all expressions in S have the same symbol and extract from each expression in S the subexpression beginning at p. The set of all such subexpressions is the disagreement set.

Next, we present the unification algorithm and unification theorem following the account of Lloyd (Lloyd, 1987). Let S be a finite set of simple expressions, The *unification algorithm* is defined as follows:

1. Initial k to 0 and θ_0 to ϵ.

2. If $S\theta_k$ is a singleton, then stop, where θ_k is an mgu of S. Otherwise, find the disagreement set D_k of $S\theta_k$.

3. If there exist v and t in D_k such that v is a variable that does not occur in t, then put $\theta_{k+1} = \theta_k\{v/t\}$, increment k and go to 2. Otherwise, stop; S is not unifiable.

Theorem 2.2.1. *Let S be a finite set of simple expressions. If S is unifiable, then the unification algorithm terminates and gives an mgu for S. If S is not unifiable, then the unification algorithm terminates and reports this fact.*

Let G be $\leftarrow A_1, \ldots, A_i, \ldots, A_k$ and C be $A \leftarrow B_1, \ldots, B_n$. Then G' is *derived* from G and C using mgu θ if the following conditions hold: (i) A_i is an atom, called the *selected atom*, in G; (ii) θ is an mgu of A_i and A; (iii) G' is the goal $\leftarrow (A_1, \ldots, B_1, \ldots, B_n, A_{i+1}, \ldots, A_k)\theta$. G' is called a *resolvent* of G and C. For a definite program P, and a definite goal G, an *SLD-derivation* of $P \cup \{G\}$ consists of a (finite or infinite) sequence $G_0 = G, G_1, \ldots$ of goals, a sequence $C_1, C_2, \ldots$ of variants of program clauses of P and a sequence $\theta_1, \theta_2, \ldots$ of mgu's such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}. An *SLD-refutation* of $P \cup \{G\}$ is a finite SLD-derivation of $P \cup \{G\}$ which has the empty clause $\square$ as the last goal in the derivation. For a program P, and a definite goal G, let $\theta_1, \ldots, \theta_n$ be the sequence of mgu's used in an SLD-refutation of $P \cup \{G\}$, then a *computed answer* θ for $P \cup \{G\}$ is the composition $\theta_1 \ldots \theta_n$ restricted to the variables of G.

Theorem 2.2.2 (Soundness of SLD-Resolution)**.** *(Lloyd, 1987) Let P be a definite program and G a definite goal. Then every computed answer for $P \cup \{G\}$ is a correct answer for $P \cup \{G\}$.*

2.3 Coinductive Logic Programming

Standard logic programming (which computes least fixed-points) cannot be used to reason about infinite objects (which belong to the greatest fixed-points). Recently *coinduction* (Barwise and Moss, 1996) has been introduced into logic programming by Simon et al (Gupta et al., 2007, Simon et al., 2007) to overcome this problem. Coinductive logic programming can be used for reasoning about unfounded sets, behavioral properties of (interactive) programs, etc., as well as elegantly proving liveness properties in model checking, type inference in functional programming, etc. (Gupta et al., 2007).

Coinductive logic programming is a dual of logic programming: it supports reasoning directly about infinite objects and infinite properties. The declarative semantics of coinductive logic programming allows the universe of terms to contain infinite terms, in addition to the traditional finite terms. It also allows for the model to contain ground goals that have either finite or infinite idealized proofs. Coinductive logic programming provides an operational semantics—similar to SLD resolution (Sterling and Shapiro, 1994)—for computing the greatest fixed-point of a logic program. This operational semantics (called co-SLD resolution) relies on the *coinductive hypothesis rule* and systematically computes elements of the *gfp* via backtracking. It is briefly described below through a simple example. Consider a normal logic programming definition of a stream (list) of bits given as program P1 below:

```
stream([]).
stream([H | T]) :- bit(H), stream(T).
bit(0).
bit(1).
```

Under SLD resolution, the query `?- stream(X).` will systematically produce all finite streams of bits one by one, starting from the empty stream `[]`. Suppose we remove the base case and obtain the program P2:

```
stream([H | T]) :- bit(H), stream(T).
bit(0).
bit(1).
```

In standard logic programming the query `?- stream(X).` fails, since the lfp model of P2 does not contain any instances of `stream/1`. The problems are two-fold: (i) the Herbrand universe does not contain infinite terms, (ii) the least Herbrand model does not allow for infinite proofs, such as the proof of `stream(X)`; yet these concepts are commonplace in computer science, and a sound mathematical foundation exists for them in the field of hyperset theory

(Barwise and Moss, 1996). Coinductive logic programming extends the traditional declarative and operational semantics of LP to allow reasoning over infinite and cyclic structures and properties. In the coinductive logic programming paradigm the declarative semantics of the predicate `stream/1` above is given in terms of *infinitary Herbrand* (or *co-Herbrand*) *universe, infinitary Herbrand* (or *co-Herbrand*) *base* (Lloyd, 1987), and *maximal models* (computed using greatest fixed-points) (Simon et al., 2007).

The query `?- stream(X).` under the coinductive interpretation of P2 should produce all infinite sized streams as answers, e.g.,

```
X = [1, 1, 1, ...  ],
X = [1, 0, 1, 0, ...  ],
```

etc. The model of P2 does contain instances of `stream/1` (but proofs may be of infinite length).

If we take a coinductive interpretation of program P1, then we get all finite and infinite streams as answers to the query `?- stream(X).`. Coinductive logic programming allows programmers to manipulate infinite structures. As a result, unification must be extended and the "occurs check" removed: unification equations such as `X = [1 | X]` are allowed in coinductive logic programming; in fact, such equations will be used to represent infinite (rational) structures in a finite manner.

The operational semantics under co-induction is given in terms of the *coinductive hypothesis rule*: during execution, if the current resolvent R contains a call C' that unifies with an ancestor call C, then the call C' succeeds; the new resolvent is $R'\theta$ where $\theta = mgu(C, C')$ and R' is obtained by deleting C' from R. With this extension, a clause such as

```
p([1 | T]) :- p(T).
```

and the query `?- p(Y).` will produce an infinite answer `Y = [1 | Y]`.

In coinductive logic programming the alternative computations started by a call are not only those that begin with unifying the call and the head of a clause, but also those that begin with unifying the call and one of its ancestors in a proof tree.

Regular logic programming execution extended with the coinductive hypothesis rule is termed *co-logic programming* (Simon, 2006). The coinductive hypothesis rule works for only those infinite proofs that are *regular* (or *rational*), i.e., infinite behavior is obtained by a finite number of finite behaviors interleaved an infinite number of times. More general implementations of coinduction are possible (Simon, 2006). More complex examples of coinductive LP can be found elsewhere (Gupta et al., 2007). Even with the restriction to rational proofs, there are many applications of coinductive logic programming. These include model checking, modeling ω-automata, non-monotonic reasoning, etc.

2.4 Constraint Logic Programming

Constraint Logic Programming (CLP) is a many-sorted version of logic programming in which different sorts are associated with different interpretation domains, and corresponding formulae are manipulated using predefined constraint solvers. The intuitive idea is to introduce special classes of formulae (constraints) which are not handled using traditional resolution, but are interpreted under a predefined specific interpretation and handled by external constraint solvers. For a more precise and complete presentation of CLP the reader is referred to the literature (Jaffar and Lassez, 1987, Jaffar and Maher, 1994).

Constraint logic languages are built on two collections of symbols, Σ, containing all the function symbols, and Π, containing the predicate symbols. Furthermore, predicate symbols are divided into two separate classes, $\Pi = \Pi_c \cup \Pi_p$; Π_p contains the user-defined predicates, while Π_c contains the constraint predicates. Constraint predicates will be interpreted with respect to a predefined interpretation structure, while user-defined predicates will be subject to the user definitions. Π_c is assumed to contain the equality symbol $=$.

Terms are objects created using the symbols from Σ and $\mathcal{V}$, where $\mathcal{V}$ is a collection of variables. A term is either a simple variable or the application $f(t_1, \ldots, t_n)$ of a n-ary symbol $f \in \Sigma$ to n terms $t_1, \ldots, t_n$ $(n \geq 0)$.

An atom is the application $p(t_1, \ldots, t_n)$ of a predicate symbol p to n terms $t_1, \ldots, t_n$. If $p \in \Pi_C$, then the atom is said to be a *constraint*. A program consists of a collection of clauses, where each clause has the form:

$$\mathit{head} \text{ :- } c \mid b_1, \ldots, b_k$$

head, b_i are user defined atoms while c is an arbitrary conjunction of constraints.

From the semantic point of view, constraints are interpreted using a predefined interpretation (i.e., a domain $\mathcal{D}$ together with an interpretation function I_D). In particular, a constraint c is solvable if $\mathcal{D} \models \exists(c)$. A solution θ for c is a mapping from the variables in c to $\mathcal{D}$, such that $\mathcal{D} \models c\theta$.

From the procedural point of view, execution of a constraint program requires the use of constraint solvers capable of deciding the solvability of each possible constraint formula[1]. Resolution is extended in order to embed calls to the constraint solvers. If ?- $c_1 \mid g_1, \ldots, g_n$ is a goal, and p :- $c_2 \mid b_1, \ldots, b_k$ is a clause in the program, then the resolvent of the goal w.r.t. the given clause is

$$\text{?- } (c_1, c_2, g_1 = p) \mid b_1, \ldots, b_k, g_2, \ldots, g_n$$

as long as $\mathcal{D} \models (c_1 \wedge c_2 \wedge (g_1 = p))$. The constraint solver is used to test the validity of the condition on the constraints.

Frequently, constraint solvers are capable not only of checking solvability, but also of simplifying the constraints (eventually computing explicit solutions whenever possible); in this case, in the resolvent the constraint $c_1, c_2, (g_1 = p)$ is replaced with its simplified form.

An example of a CLP system is *CLP($\mathcal{R}$)*, where the constraint domain is the domain of real numbers, Σ contains real numbers and arithmetic operations ($+$, $*$, etc.), and Π_C contains the equality $=$ and the disequation/relational predicates ($\leq$, $\geq$, etc.).

[1]It is common for the programmer to identify only special types of constraint formulae, the *admissible* constraints; these are the only constraints which are admitted during the execution of a program. This is because it is not possible to devise a constraint solver that will solve any arbitrary set of constraints.

2.5 Timed Automata

A *timed automaton* (Alur and Dill, 1994) is a tuple $M = \langle \Sigma, Q, Q_0, C, E, F \rangle$, where

- Σ is a finite alphabet;
- Q is the (*finite*) set of states;
- $Q_0 \subseteq Q$ is the set of initial states;
- C is a finite set of *clocks*;
- $E \subseteq Q \times Q_0 \times \Sigma \times 2^C \times \phi(C)$ gives the set of transitions. An edge $\langle q, q', a, \lambda, \delta \rangle$ represents a transition from state q to state q' on input symbol a. The set $\lambda \subseteq C$ gives the clocks to be reset with this transition, and δ is a clock constraint over C;
- F is a subset of 2^Q.

A *clock interpretation* ν for a set X of clocks assigns a real value to each clock; that is, it is a mapping from X to $\mathbf{R}$. We say that a clock interpretation ν for X satisfies a clock constraint δ over X iff δ evaluates to true using the values given by ν.

For $t \in \mathbf{R}$, $\nu + t$ denotes the clock interpretation which maps every clock x to the value $\nu(x) + t$, and the clock interpretation $t.\nu$ assigns to each clock x the value $t.\nu(x)$. For $Y \subseteq X$, $[Y \mapsto t]\nu$ denotes the clock interpretation for X which assigns t to each $x \in Y$, and agrees with ν over the rest of the clocks.

A run r, denoted by $(\bar{q}, \bar{\nu})$, of a timed automaton over a timed word $(\sigma_1, t_1), (\sigma_2, t_2), \ldots$ is an infinite sequence of the form

$$r : \langle q_0, \nu_0 \rangle \xrightarrow[t_1]{\sigma_1} \langle q_1, \nu_1 \rangle \xrightarrow[t_2]{\sigma_2} \langle q_2, \nu_2 \rangle \xrightarrow[t_3]{\sigma_3} \ldots$$

with $q_i \in Q$ and $\nu_i \in [C \mapsto \mathbf{R}]$, for all $i \geq 0$, satisfying two requirements:

- $q_0 \in Q_0$, and $\nu_0(x) = 0$ for all clocks $x \in C$.

- for all $i \geq 1$, there is an edge in E of the form $\langle q_{i-1}, q_i, \sigma_i, \lambda_i, \delta_i \rangle$ such that $(\nu_{i-1}+t_i-t_{i-1})$ satisfies δ_i and ν_i equals $[\lambda_i \mapsto 0](\nu_{i-1} + t_i - t_{i-1})$.

The set *inf(r)* consists of $q \in Q$ such that $q = q_i$ for infinitely many $i \geq 0$.

Different notions of acceptance have been proposed. A run $r = (\bar{q}, \bar{\nu})$ of a timed *Büchi* automaton over a timed word $(\sigma_1, t_1), (\sigma_2, t_2), \ldots$ is called an *accepting run* iff F is a singleton, and $inf(r) \cap F \neq \emptyset$. A run $r = (\bar{q}, \bar{\nu})$ of a timed *Müller* automaton over a timed word $(\sigma_1, t_1), (\sigma_2, t_2), \ldots$ is called an *accepting run* iff $inf(r) \in F$.

Figure 2.1 shows a simple Büchi timed automaton, with clock x and final state $F = \{q_0\}$. It describes a system in which signals are recognized. Each signal a can (but need not) be followed by a signal b, with the constraint that the signal b must arrive at least one time unit and at most two time units after a.

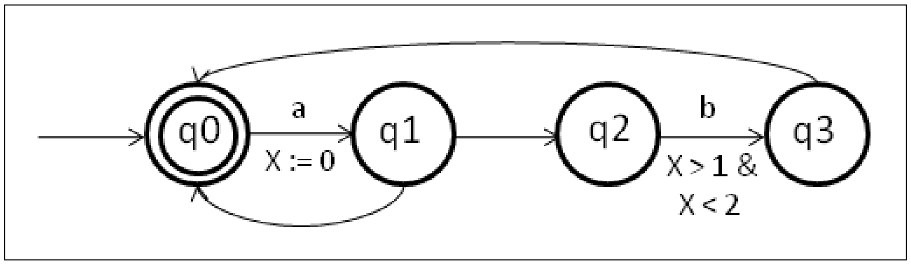

Figure 2.1. A Timed Automaton

2.6 Pushdown Timed Automata (PTA)

Pushdown timed automata extend timed automata with stack in exactly the same manner that pushdown automata extend finite automata. Thus, a pushdown timed automaton is obtained from a timed automaton by adding:

- ϵ (empty string) to the input alphabet Σ;
- a stack alphabet $\Gamma_\epsilon = \Gamma \cup \epsilon$, where Γ is a set of symbols disjoint from Σ;
- a stack represented by Γ_ϵ^*;

- $PD : E \mapsto \Gamma \times \Gamma_\epsilon$ assigns a pair (a, γ) with $a \in \Gamma$ and $\gamma \in \Gamma_\epsilon$, called a *stack operation*, to each transition in E. A stack operation (a, γ) replaces the top symbol a of the stack with a string (possibly empty) in Γ_ϵ.

Acceptance conditions for an infinite string for PTA are similar to those for timed automata but, additionally, the stack must be empty in every final state.

As an example of a pushdown timed automaton, consider a language in which sequences of a's are followed by sequences of an equal number of b's (each such string has at least two a's and at least two b's). For each pair of equinumerous sequences of a's and b's, the first b must appear within 5 units of time from the first symbol and the final b must appear within 20 units of time from the first a.

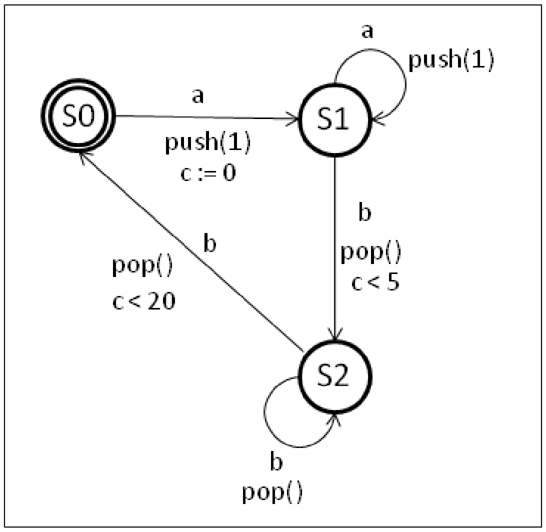

Figure 2.2. A Pushdown Timed Automaton

The pushdown timed automaton recognizing this timed language is shown in Figure 2.2. In this figure, s_0 is the final state, c is the clock, and s is the stack. Actions $s.push(1)$ and $s.pop()$, respectively push 1 onto the stack and pop the stack (this automaton accepts also the empty string; we allow this for simplicity of presentation). The requirement that the stack be empty ensures that only strings with equal numbers of a's and b's are accepted.

Note that the wall clock time keeps advancing at the normal uniform rate, as the automaton makes transitions.

Note that in many cases real-time systems that are naturally modeled as PTA can be modeled as timed automata by imposing restrictions (such as limiting the size of the string, i.e., limiting the number of allowable events), but our experience indicates that such a timed automaton will have an enormous number of states, and thus would be unwieldy and time consuming to specify. Proving its safety and liveness properties will also be quite cumbersome simply due to the large size of the automaton.

CHAPTER 3
COINDUCTIVE CONSTRAINT LOGIC PROGRAMMING

3.1 Introduction

Constraint logic programming (Jaffar and Maher, 1994) is a natural and expressive paradigm which combines two declarative paradigms: logic programming (Lloyd, 1987) and constraint solving. A program written in CLP defines a set of inference rules extended with constraints which is used to effectively construct the automatic proofs for various logical statements in the presence of constraints. On the one hand CLP's declarative semantics is defined in terms of a least fixed-point, i.e., it is inductive. Therefore, it cannot be directly used for reasoning about infinite and cyclical objects and their properties. Coinductive logic programming, on the other hand, has been proposed as a powerful technique for finitely reasoning about infinite (rational) structures and their properties (Gupta et al., 2007, Simon et al., 2007, Simon, 2006). Coinductive logic programming does not handle constraints, while CLP's declarative and operational semantics is inadequate for various programming techniques which involve infinite computations besides constraint solving. Such techniques have interesting applications for example in Model Checking and in verifying real-time systems and cyber-physical systems (Lee, 2008, Gupta, 2006a). A formalism that can support both infinite (rational) behavior and constraints is missing. This chapter is focused on developing the needed formalism.

Consider the following program, which describes an infinite list with a constraint on its elements. This program is not semantically meaningful in constraint logic programming, and likewise in coinductive logic programming.

```
stream([X, Y | T]) :- {Y - X >= 3}, stream(T).
```

However, we would like a query such as `?- stream(L).` to return a positive answer in a finite amount of time. The operational semantics of CLP does not allow for infinite proofs such as a proof for the query above and the operational semantics of coinductive logic programming does not allow constraints. While this is only an artificial, simple example to illustrate the point, there are lots of real world applications that exhibit (i) infinite computation, (ii) interaction with the environment, and that operate under constraints imposed by physical entities such as time, temperature, etc.

For example a reactor temperature control system is a traditional example of a cyber-physical system which exhibits both aspects mentioned above: one of the components of this system, the core controller, runs forever in order to keep the temperature of the system between two thresholds. Two physical quantities involved in this system are time and temperature, and the behavior of the system is specified by placing constraints on them. Another example is the generalized railroad crossing problem. This system is controlled by several components: controller, gate and tracks that work in parallel and run forever. The behavior of this system is also specified by constraints between the times at which different events in the system take place. These systems can be naturally modeled as coinductive constraint logic programs.

In this chapter we explore the theoretical foundations of coinductive logic programming (co-LP) in the presence of constraints. We show how the operational semantics of co-LP is extended in order to obtain soundness results for coinductive constraint logic programming. First, we extend the constraint logic programming scheme to coinductive constraint logic programming in Section 3.3. We describe a new paradigm for constraint logic programming which allows for a natural approach to direct reasoning about infinite (rational) objects and their properties when constraints are used to express the relation between the objects. We redefine the declarative semantics of constraint logic programming (Jaffar and Maher, 1994) so that infinite and cyclical objects can be reasoned about. The declarative semantics of coinductive constraint logic programming (coinductive CLP) is defined in terms of a greatest fixed-point. We also extend the operational semantics of CLP with a *coinductive* *hypothesis rule* in order to obtain finite derivations for infinite proofs. In this work, we limit our attention to coinductive constraint logic programs without negation.

A constraint logic program is restricted only to finite objects and proofs, while a coinductive constraint logic program can involve infinite objects and infinite proofs. However, there are interesting cases in which both finite and infinite objects and proofs are involved. To be able to handle these cases we combine constraint logic programming with coinductive constraint logic programming in Section 3.4. The resulting *co-constraint logic programming (co-CLP)* allows predicates to be annotated as either inductive or coinductive. Inductive predicates can invoke coinductive predicates and vice versa; however, no cycles through alternating induction and coinduction are allowed. The declarative semantics of co-CLP is defined in terms of greatest fixed-point semantics. The operational semantics is defined as an extension of the top-down execution model of CLP with a *coinductive hypothesis rule*, with the restriction that inductive and coinductive predicates cannot be mutually recursive. The operational semantics of co-CLP allow us to obtain finite derivations for infinite (rational) proofs.

The contributions of this work include the extension of constraint logic programming with coinduction (coinductive CLP), which includes new declarative semantics as well as operational semantics. To prove that a logical statement is true (belongs to the greatest fixed-point) in a given coinductive constraint logic program, we use the coinductive hypothesis rule. Our contributions also include proposing a new paradigm, called co-CLP, which combines traditional constraint logic programming and coinductive constraint logic programming, as well as its declarative and operational semantics. Co-CLP allows for logic programming with finite and infinite and cyclical data structures in the presence of constraints; that is, co-CLP can be used directly to reason about logical statements regarding both finite and infinite objects and their properties with constraints imposed on them.

3.2 Notation and Terminology

In the rest of the chapter we follow the notations and terminologies that are used in Jaffar and Maher (1994) and Simon (2006). However, in traditional constraint logic programming (Jaffar and Maher, 1994), the only domain considered is the domain of constraints; therefore, defining new function symbols and terms is not allowed. As a result the terms and atoms that are used in a constraint logic program are limited to what is defined in Σ (defined below), and terms and atoms are interpreted in the domain of constraints. In this dissertation, we consider not only the domain of constraints, but also the Herbrand universe. Therefore, user-defined terms will be allowed. User-defined functions and predicates will be given the standard interpretation in the Herbrand universe and the Herbrand base. However, constraints will be interpreted using a predefined interpretation (provided by the domain of computation).

A *signature* defines a set of function and predicate symbols with their corresponding arities. If Σ is a signature, a Σ-*structure* $\mathcal{D}$ consists of a set D and an assignment of functions and relations on D to the symbols of Σ which respects the arities of the symbols (a Σ-structure $\mathcal{D}$ is an interpretation of Σ in domain D). Note that D may contain finite as well as infinite objects.

We assume an infinite number of variables; x, y, z, w will denote variables, h, t will denote terms, p, q will denote predicate symbols, f, g will denote function symbols, a will denote an atom, c will denote a constraint, r will denote a rule, P will denote a program. $\tilde{x}$ and $\tilde{h}$ denote sequences of terms $x_1, x_2, \ldots, x_n$ and $h_1, h_2, \ldots, h_n$, respectively. $\mathcal{D}$ will denote a structure, D will denote its set of elements. $\exists_{-\tilde{x}}\ \phi$ denotes the full existential closure of the formula ϕ except for the variables $\tilde{x}$, which remain unquantified. $\tilde{\exists}\ \phi$ denotes the full existential closure of the formula ϕ. $\mathcal{D} \models c$ denotes that c is true in $\mathcal{D}$. $\mathcal{D} \models \exists c$ means that the constraint c is satisfiable in $\mathcal{D}$.

A first-order Σ-*formula* is built upon variables, function and predicate symbols of Σ, the logical connectives $\wedge, \vee, \neg, \leftarrow, \rightarrow, \leftrightarrow$ and quantifiers over variables $\exists, \forall$. A closed formula is a formula in which all variable occurrences are bound by quantifiers. A Σ-*theory* is a collection of closed Σ-formulas.

Analogously to the class of CLP languages, co-CLP($\mathcal{X}$) can be obtained by instantiating the parameter $\mathcal{X}$, where $\mathcal{X}$ is an abbreviation for a 4-tuple $(\Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T})$. Σ is a signature containing the binary predicate symbol $=$ interpreted as identity in D, and a special binary relation $=_u$ which will be used for unification. The expression $x =_u a$ will be interpreted as substituting the term a for variable x. $\mathcal{D}$ is a Σ-structure, $\mathcal{L}$ is a class of Σ-formulas that contains a constraint that is identically true and one that is identically false, and $\mathcal{T}$ is a first-order Σ-theory. $\mathcal{L}$ is assumed to be closed under variable renaming, conjunction and existential quantification. For any signature Σ, a Σ-structure $\mathcal{D}$ (the domain of computation) and a class of Σ-formulas $\mathcal{L}$ (the constraints), the pair $(\mathcal{D}, \mathcal{L})$ is called a *constraint domain*. Sometimes the constraint domain is denoted by $\mathcal{D}$. A Σ-structure $\mathcal{D}$ is a model of a Σ-theory $\mathcal{T}$ if all formulas of $\mathcal{T}$ evaluate to *true* under the interpretation provided by $\mathcal{D}$. We limit our attention to such $\mathcal{T}$'s whose satisfiability of constraints is decidable in $\mathcal{D}$, in other words, for a constraint domain $(\mathcal{D}, \mathcal{L})$ with signature Σ, $\mathcal{T}$ is a decidable theory, i.e., (i) $\mathcal{D}$ is a model of $\mathcal{T}$, (ii) for every constraint $c \in \mathcal{L}$, either $\mathcal{T} \models \exists c$ or $\mathcal{T} \models \neg\exists c$ (i.e., each model of $\mathcal{T}$ is a model of $\exists c$ or each model of $\mathcal{T}$ is a model of $\neg\exists c$). This second property is called *satisfaction completeness.*

A *term* is a tree where every leaf node is labeled with a variable or with some $f/0$ (constant) and every inner node with n children, $n > 0$, is labeled with some f/n. We write $f(t_1, \ldots, t_n)$ for the term with root f and direct subtrees $t_1, \ldots, t_n$. A term t is called *finite* if all paths in the tree t are finite, otherwise it is *infinite.* A term (tree) is *rational* if it only contains finitely many different subterms (subtrees).

We consider an extension of Σ to Σ^D in which there is a constant for every element of D. Terms that are built upon variables and function symbols in Σ^D are denoted by T^D. Let $\mathcal{L}^D$ be a class of Σ^D-formulas [1]. Moreover, let Σ^D_P be the extension of Σ^D with the set of all user defined function and predicate symbols in program P. The set of predicate symbols defined

[1]In the literature Σ^D and $\mathcal{L}^D$ are often denoted as Σ^* and $\mathcal{L}^*$.

by Σ is denoted by Π_c and the set of predicate symbols defined by a program is denoted by Π_P. Note that Π_c and Π_P are disjoint.

An *atom* is a tree $p(t_1, \ldots, t_n)$, where $p \in \Pi_p$ and $t_1, \ldots, t_n$ are rational terms. A *primitive constraint* is an expression of the form $p(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms from T^D and p is a predicate symbol from Π_c. In other words, primitive constraints only contain functors from Σ. This will ensure that the constraint solver will only deal with primitive constraints which it can solve. A *constraint* is a first-order formula built from primitive constraints.

Example 3.2.1. *Let Σ contain a collection of constants, the binary predicate $=$, a binary function symbol '.' and a special constant* nil. *Let D be the set of finite as well as infinite (rational) lists. Let $\mathcal{D}$ interpret the function symbol '.' of Σ as the list constructor, where '.' maps a flat list L to a binary tree whose root and inner nodes are labeled with '.' and all elements of L appear as leaves of the binary tree. The empty list is represented by* nil. *The primitive constraints are equations between terms. Let $\mathcal{L}$ be the class of constraints generated by these primitive constraints. Then $(\mathcal{D}, \mathcal{L})$ is the constraint domain of finite and (rational) infinite lists.*

Example 3.2.2. *Let Σ contain two binary predicate symbols $=$ and $<$, and a binary function symbol $+$. Let D be the set of natural numbers and $\mathcal{D}$ interpret the symbols of Σ as usual (i.e., $+$ is interpreted as addition, etc). Let f be a unary function symbol defined in the program. Then, for example, $7 + 2$ is a term in T^D, but not a formula; however, $7 + 2 < 8$ is a formula in $\mathcal{L}^D$, which is a legal constraint; however, $f(2) < 7$ is not allowed.*

3.3 Coinductive Constraint Logic Programming

3.3.1 Syntax

A coinductive constraint logic program P is syntactically identical to a constraint logic program : it is a collection of rules of the form $a \leftarrow c, b_1, \ldots, b_n$. a is an atom and called the head of the rule. $c, b_1, \ldots, b_n$ is called the body of the rule; c is the conjunction of constraints in the body and b_i's are atoms.

Example 3.3.1. *In the following program the predicate p is a coinductive predicate (note that there is no base case). This predicate generates/accepts infinite (rational) lists of numbers in which, the difference between elements of lists with some constant is less than 5.*

```
p([X | T], Y) :- {X - Y =< 5}, p(T, Y).
p([X | T], Y) :- {Y - X =< 5}, p(T, Y).
```

3.3.2 Declarative Semantics

In this section, we present the declarative semantics of coinductive constraint logic programming in terms of a greatest fixed-point. Since we are dealing with two types of predicate and function symbols, namely those that are defined by Σ and those that are defined by the user in program P, we must define a new interpretation that handles both. First, we extend the Herbrand universe of a program P to contain the domain D ($D \subset U_P$). We assume that the Herbrand Universe contains both finite and (rational) infinite terms (Colmerauer, 1982). User-defined terms and atoms will be given the standard interpretation in the Herbrand universe and the Herbrand base of P (U_P and B_P) (Lloyd, 1987); while terms and formulas built from elements of Σ are given the interpretation provided by $\mathcal{D}$.

We denote the interpretation of a predicate symbol p of program P by $\mathbf{p}$. Similarly, the interpretation of a user-defined function symbol f will be denoted by $\mathbf{f}$. We take $v : V \rightarrow U_P$ to be a valuation of variables to ground terms in the Herbrand universe (Lloyd, 1987), such that if variable x occurs in a constraint, then $v(x) \in D$. We consider the natural extension of v that maps user-defined terms and atoms over the Herbrand universe and Herbrand base, terms in T^D to ground terms, and formulas in $\mathcal{L}^D$ to ground $\mathcal{L}^D$-formulas. We call this valuation a *constraint consistent valuation.* For an atom a of the form $p(t_1, \ldots, t_n)$ in P, $v(a)$ is $\mathbf{p}(v(t_1), \ldots, v(t_n))$. For a user defined function of the form $f(t_1, \ldots, t_n)$, $v(f)$ is $\mathbf{f}(v(t_1), \ldots, v(t_n))$.

Definition 3.3.1. *A $\mathcal{D}$-base of a coinductive CLP program P over the domain D, denoted by $B_P^{\mathcal{D}}$, is the set*

$$\{\boldsymbol{p}(x_1, \ldots, x_n) \mid p \textit{ is an n-ary user-defined predicate in } P \textit{and each } x_i \textit{ is an element in } U_P\}$$

Definition 3.3.2. *A $\mathcal{D}$-interpretation of a formula in $\mathcal{L}^D$ is an interpretation that agrees with $\mathcal{D}$ on the interpretation of the symbols in Σ.*

Definition 3.3.3. *A $\mathcal{D}$-interpretation of a fact of the form $p(\tilde{x}) \leftarrow c$, is the set*

$$\{\boldsymbol{p}(\tilde{h}) \mid \textit{there exists a constraint consistent valuation } v, \textit{ such that } \mathcal{D} \models v(c), v(\tilde{x}) = \tilde{h}\}$$

Definition 3.3.4. *Let P be a coinductive constraint logic program over the domain of computation $\mathcal{D}$. Let I be a $\mathcal{D}$-interpretation. The $\mathcal{D}$-model of P is the fixed-point of $T_P^{\mathcal{D}} : 2^{B_P^{\mathcal{D}}} \rightarrow 2^{B_P^{\mathcal{D}}}$*

$T_P^{\mathcal{D}}(I) = \{\boldsymbol{p}(\tilde{h}) \mid p(\tilde{x}) \leftarrow c, b_1, \ldots, b_n$ is a rule of P, and there exists a constraint consistent valuation v such that $\mathcal{D} \models v(c), v(\tilde{x}) = \tilde{h}, v(b_i) \in I, i = 1, \ldots, n\}$

The greatest $\mathcal{D}$-model of a program P, denoted by $gm(P, \mathcal{D})$, is the greatest fixed-point of $T_P^{\mathcal{D}}$. $gm(P, \mathcal{D})$ is defined to be the declarative semantics of a coinductive constraint logic program P.

Note that the function $T_P^{\mathcal{D}}$ is defined on $\mathcal{D}$-interpretations. The set of $\mathcal{D}$-interpretations forms a complete lattice under the subset ordering. $T_P^{\mathcal{D}}$ is a monotonic function on this lattice. Therefore the least and greatest fixed-points exist for $T_P^{\mathcal{D}}$.

Example 3.3.2. *Recall the stream example from the introduction. Let the domain of computation be the set of real numbers with the binary predicate symbol $\geq$ and let the binary function symbol '.' represent the list constructor. The greatest $\mathcal{D}$-model of this program contains* `stream(t)` *where* `t` *is an infinite list of real numbers in which, if the ith and $i+1$th elements are t_i and t_{i+1} respectively, $\mathtt{t_{i+1}} - \mathtt{t_i} \geq 3$. Coinductive constraint logic programming is required for this kind of program since* `stream/1` *has no meaning in constraint logic programming.*

The truth of a coinductive constraint logic program P over a constraint domain $\mathcal{D}$ is defined in terms of inclusion in the greatest $\mathcal{D}$-model of program P. Note that the $T_P^{\mathcal{D}}$ operator will be applied a finite number of times for any given atom in the least $\mathcal{D}$-model of a program, while it may have to be applied an infinite number of times for an atom to be in the greatest $\mathcal{D}$-model of a program. The model of a coinductive constraint logic program is a subset of the greatest fixed point: it consists of those elements for which there exist *idealized rational proofs*. Intuitively, idealized proofs are trees of ground atoms, such that a parent is deduced from the idealized proofs of its children. An idealized proof is *rational* if it is represented as a rational tree.

Definition 3.3.5. *Let node(a,T) be a constructor of a tree with root a and subtrees T, where a is an atom and T is a list of trees. The set of idealized proofs for a coinductive constraint logic program P is the greatest fixed-point of $\Sigma_P^{\mathcal{D}}$, denoted by $\nu\Sigma_P^{\mathcal{D}}$,where*

$$\Sigma_P^{\mathcal{D}}(S) = \{node(\boldsymbol{p}(\tilde{h}), [T_1, \ldots, T_n]) \mid p(\tilde{x}) \leftarrow c, b_1, \ldots, b_n \textit{ is a rule of } P, v \textit{ is a constraint consistent valuation such that } \mathcal{D} \models v(c), v(\tilde{x}) = \tilde{h},\ T_i \in S,\ i = 1, \ldots, n, \textit{the root of } T_i \textit{ is } v(b_i)\}$$

We next define the declarative semantics of coinductive constraint logic programs in terms of idealized proofs. If $S = \{a \mid \exists T \in \nu\Sigma_P^{\mathcal{D}}, a \text{ is the root of } T\}$, then $S = gm(P, \mathcal{D})$.

Any element in the gm$(P, \mathcal{D})$ has an idealized proof and anything that has an idealized proof is in gm$(P, \mathcal{D})$. The greatest model consists of all ground atoms that have a finite as well as an infinite (rational) idealized proof. This equivalence will be used in the completeness proof of operational semantics of coinductive CLP which will be discussed in Appendix A.

3.3.3 Operational Semantics

The operational semantics given for coinductive constraint logic programming is similar to that of constraint logic programming in the sense that it is presented in terms of state transition systems. However, there are two major differences between these two semantics: (i) While the operational semantics of CLP uses multisets of atoms and constraints along with multisets of constraints as states, coinductive CLP uses finite trees of atoms along with a multiset of constraints as states. (ii) Operational semantics of CLP allows program rules (clauses) along with transition rules that manipulate constraints (checking consistency, adding constraints and inferring new constraints) as transition rules. The operational semantics of coinductive CLP, in addition to these transitions, allows the coinductive hypothesis rule. The coinductive hypothesis rule is used to obtain finite derivations for goals with (rational) infinite idealized proofs. Intuitively, it states that if, during execution, a call A is encountered and unified with an ancestor call A', and moreover *the set of accumulated constrains is consistent*, then A is deleted from the resolvent and the substitution resulting from unification of arguments of A and A' is applied to the resolvent. Note that each ancestor of a goal constitutes a coinductive hypothesis, which can be accessed in the recursion stack. Note also that in coinductive CLP, unification must be extended and the "occurs check" removed: unification equations such as $X =_u [1|X]$ are allowed; in fact, such equations will be used to represent infinite (rational) structures in a finite manner.

Definition 3.3.6. *(Simon, 2006) Let N^* denote the set of all finite sequences of positive integers, including the empty sequence ϵ. We call these sequences paths. A singleton path is denoted by i for some positive integer i. The concatenation of two paths π and π' is denoted by $\pi.\pi'$. $D \subseteq N^*$ is prefix-closed, if $\pi_1.\pi_2 \in D$ implies $\pi_1 \in D$. A tree of elements of a set S, called an S-tree, is defined as a partial function from paths to elements of S, such that the domain is non-empty and prefix-closed. Each node in the tree is unambiguously denoted by a path. The root of tree t is denoted by $t(\epsilon)$; the node defined by a path π is denoted by $t(\pi)$. A tree t is described by the paths π from the root $t(\epsilon)$ to the nodes $t(\pi)$ of the tree, and the nodes are labeled with elements of S. A child of node π in tree t is any path $\pi.i$ that is in the domain of t, where i is some positive integer. The subtree of t rooted at π, denoted by $t' = t\backslash\pi$, is the partial function $t'(\pi') = t(\pi.\pi')$. $node(L, T_1, \ldots, T_n)$ is a constructor of an S-tree with root labeled L and subtrees T_i, where $L \in S$ and each T_i is an S-tree, such that $1 \leq i \leq n$, $node(L, T_1, \ldots, T_n)(\epsilon) = L$, and $node(L, T_1, \ldots, T_n)(i.\pi) = T_i(\pi)$.*

Definition 3.3.7. *A state is a triple (T, E, C), where*

- *T is a finite tree with nodes labeled with atoms,*
- *E is the set of equalities of the form $x =_u t$, in which x is a variable and t is a term and each variable x appears at most once in the left hand side of equations in E,*

- *C is the set of constraints from $\mathcal{L}^D$, it is called the constraint store.*

There is one other state, called the failure state.

Intuitively, E is the system of equations that captures the accumulated results of unification. It can be seen as a representation of a substitution. A substitution is *rational* if it only substitutes rational terms for variables. The collection of as-yet-unseen atoms is represented by tree T. When an atom labeling a leaf in tree T is proved to be true, T is modified (using function δ, defined below) by removing the leaf and the maximum number of its ancestors, such that the result is still a tree. If all nodes in a tree are removed, the tree itself is removed.

Definition 3.3.8. *(Simon, 2006) Given a tree of atoms T and a path π in the domain of T, let $\rho(T, \pi.i)$ be the partial function equal to T, except that it is undefined at $\pi.i$, Then*

$$\begin{array}{rcl} \delta(T,\pi) = & T & \text{, if } \pi \text{ has children in } T \\ \delta(T,\epsilon) = & \emptyset & \text{, if } T \text{is empty} \\ \delta(T,\pi) = & \delta(\rho(T,\pi),\pi') & \text{, if } \pi = \pi'.i \text{ is a leaf in } T \end{array}$$

Example 3.3.3. *Let predicate* `p/2` *of program P be defined as follows:*

```
p([X | T], Y) :- {X - Y = 2}, p(T, Y).
```

The result of asking the query: `p(Z, 3).` *is a substitution $E = \{Z =_u [X \mid T], Y =_u 3\}$, and a set of constraints $C = \{X - Y = 2\}$. Substituting 3 for Y in C will result in $X = 5$. Since there is no inconsistency, the execution can proceed with X bounded to 5.*

The operational semantics of coinductive CLP relies on three types of transitions:

- Transitions that arise from resolution.
- Transitions that arise from coinductive hypothesis rules.
- Transitions that manipulate the collection of constraints in constraint store. These transitions include adding constraints to constraint store, checking the consistency of current constraints in constraint store and inferring a new collection of constraints from the current collection.

Definition 3.3.9. *A transition rule r of a co-constraint logic program P is one of the following four forms:*

- *an instance of a clause in P, with variables consistently renamed for freshness,*
- *a coinductive hypothesis of the form $\eta(\pi, \pi')$: π and π' are paths such that π is a proper prefix of π', π is labeled by $p(t_1, \ldots, t_n)$, π' is labeled by $p(t'_1, \ldots, t'_n)$, $t_1, \ldots, t_n$ and $t'_1, \ldots, t'_n$ are unifiable,*
- *an inference rule for simplifying the current collection of constraints; the function* $\mathrm{infer}(C, E)$ *takes the current set of constraints and current substitution and computes a new set of constraints C', provided that $\mathcal{D} \models C \leftrightarrow C'$,*
- *a consistency check rule; the predicate* $\mathrm{consistent}(C)$ *expresses a test for consistency of C. It can be defined by: if $\mathcal{D} \models \tilde{\exists} C$ then* $\mathrm{consistent}(C)$ *holds (Jaffar and Maher, 1994).*

Definition 3.3.10. *A state (T, E, C) transitions to another state (T', E', C') by transition rule r when*

- *r is a definite clause of the form $p(t'_1, \ldots, t'_n) \leftarrow c, b_1, \ldots, b_m$, renamed to new variables, π is a leaf in T, $T(\pi) = p(t_1, \ldots, t_n)$. Let E'' be the substitution obtained from unification of $t_1, \ldots, t_n$ and $t'_1, \ldots, t'_n$, then $E' = E \cup E''$, $C' = C \cup c$ and T' is obtained as follows:*
  - *$m = 0$: $T' = \delta(T, \pi)$.*
  - *$m > 0$: T' is equal to T except at $\pi.i$ and $T'(\pi.i) = b_i$, for $1 \leq i \leq m$.*
- *r is of the form $\eta(\pi, \pi')$, a leaf π in T is labeled with $p(t_1, \ldots, t_n)$, π' the proper ancestor of π, is labeled with $p(t'_1, \ldots, t'_n)$. Let E'' be the substitution obtained from unification of $t_1, \ldots, t_n$ and $t'_1, \ldots, t'_n$, then $E' = E \cup E''$, $C' = C \cup c$ and $T' = \delta(T, \pi')$.*
- *r is an inference rule, $T' = T$, $E' = E$ and $C' = infer(C, E)$.*

- *r is a consistency check rule,*
  - *if consistent(C) holds then $(T', E', C') = (T, E, C)$,*
  - *if $\neg$consistent(C) holds then (T', E', C') = failure.*

Note that in the state transition system of a coinductive constraint logic program, the set of accumulated constraints in C must be satisfied for a coinductive hypothesis rule to be applied. We limit our attention to those constraint solvers in which the inference of new set of constraints and test for consistency is performed each time the collection of constraints in the constraint solver is changed.

Definition 3.3.11. *A transition sequence in program P consists of a sequence of states $(T_1, E_1, C_1), (T_2, E_2, C_2), \ldots$, and a sequence of transition rules $r_1, r_2, \ldots$, such that (T_i, E_i, C_i) transitions to $(T_{i+1}, E_{i+1}, C_{i+1})$ by rule r_i of definition 3.3.9.*

A transition sequence defines the trace of an execution. Execution terminates when it reaches a final state: either all atoms have been proved or the execution path reaches a state where no further transitions are possible.

Definition 3.3.12. *An accepting state is of the form $(\emptyset, E, C)$, where $\emptyset$ denotes an empty tree. A failure state is a non-accepting state which does not have any outgoing edges. A final state is either an accepting state or a failure state.*

Given a substitution specified as a system of equations E, an atom a, and a constraint c, $E(a)$ and $E(c)$ denote the result of applying the substitution E to a and c, respectively.

Definition 3.3.13. *A derivation of a state (T, E, C) in program P is a state transition sequence where the first state is (T, E, C). A derivation is successful if it ends in an accepting state, and a derivation is failed if it ends in an failure state. For a goal G with free variables $\tilde{x}$, if $(G, \emptyset, \emptyset)$ has a successful derivation in P ending at $(\emptyset, E, C)$, then $\exists_{-\tilde{x}} E(C)$ is called the answer constraint of the derivation.*

Example 3.3.4. *Let* `stream/2` *and* `number/1` *be coinductive predicates. The execution of the following program can be used to generate/recognize infinite streams of natural numbers and ω, that is infinity.*

```
stream([H | T], X) :- number(H), stream(T, Y), {Y - X >= 3}.
number(0).
number(s(N)) :- number(N).
```

The following is an execution trace for the query `?- stream([0, s(0), s(s(0)) | R], W).`*, which shows the recursion stack and also the set of constraints after each recursive call (substitution is not shown).*

```
1.  stream([0, s(0), s(s(0)) | R], W), C = ∅
2.  number(0), C = ∅
3.  stream([s(0), s(s(0)) | R], U), C = {U - W >= 3}
4.  number(s(0)), C = {U - W >= 3}
5.  number(0), C = {U - W >= 3}
6.  stream([s(s(0)) | R], V), C = {U - W >= 3, V - U >= 3}
7.  number(s(s(0))), C = {U - W >= 3, V - U >= 3}
8.  number(s(0)), C = {U - W >= 3, V - U >= 3}
9.  number(0), C = {U - W >= 3, V - U >= 3}
```

The next goal call is `stream(R, Z)`*, which unifies with ancestor* `(1)` *and immediately succeeds, since the set of constraints C is consistent. Hence the original query succeeds with* `R = [0, s(0), s(s(0)) | R]` *with the answer constraint* $\{$`U -W` $\geq$ `3, V - U` $\geq$ `3`$\}$.

Theorem 3.3.1. *(soundness) If atom a has a successful derivation in a coinductive constraint logic program P over the constraint domain $\mathcal{D}$, which ends at state $(\emptyset, E, C)$, then $E(a)$ is true in program P if $\mathcal{D} \models E(C)$.*

Example 3.3.5. *Let atom a be $p(x, y)$. Assume a has a successful derivation in a coinductive constraint logic program P which ends at $(\emptyset, E_1, C_1)$, where $E_1 = \{x =_u f(w), y =_u g(z)\}$,* *and $C_1 = \{z > 1\}$. Then $p(f(h(w)), g(z))$ is true in P for those values of z that are greater than 1. This can be represented by the fact:*

$$p(f(w), g(z)) \leftarrow \{z > 1\}.$$

Theorem 3.3.2. *(completeness) If atom $a \in gm(P, \mathcal{D})$ has a rational idealized proof, then a has a successful derivation in coinductive constraint logic program P over the domain of constraint $\mathcal{D}$.*

The proofs for soundness and completeness of the operational semantics of coinductive CLP are straightforward extension of the proofs for coinductive logic programming (Simon, 2006), which are not presented here but shown in Appendix A.

3.4 co-Constraint Logic Programming (co-CLP)

There are predicates that can be described in traditional constraint logic programming, but not in coinductive constraint logic programming. There are also predicates that are naturally coded in coinductive constraint logic programming but not in traditional constraint logic programming. It would be useful to have a constraint logic programming language that combines these two paradigms. The combination is called co-constraint logic programming (co-CLP), which is presented next.

3.4.1 Syntax

co-CLP's syntax extends the syntax of CLP by allowing predicate symbols to be declared as being either inductive or coinductive. In analogy with co-LP (Simon, 2006), the stratification restriction in co-CLP would not allow the inductive and coinductive predicates to be mutually recursive; that is, execution cannot loop between inductive and coinductive calls. This restriction is for preventing ambiguity in the semantics of co-CLP programs. Without this restriction, it would not be possible to assign meaning to the programs that simultaneously exhibit the semantics of both inductive and coinductive constraint logic programming.

A *co-constraint logic program* is defined in the exact same manner as CLP and coinductive CLP programs. An *annotated program* is a co-constraint logic program extended with a set of declarations. The set of declarations specify for every predicate p in program P, whether p is inductive or coinductive. In the annotated program corresponding to program P, for every inductive predicate q, there is a declaration of the form `:-inductive(q)`, and for every coinductive predicate p, there is a declaration of the form: `:-coinductive(p)`.

Definition 3.4.1. *(Simon, 2006) Let P be an annotated program. We say that a predicate p directly depends on a predicate q if and only if $p = q$ or P contains a clause $p \leftarrow c, b_1, \ldots, b_n$. such that some b_i contains q. The dependency graph of program P has the set of its predicates as vertices, and the graph has an edge from p to q if and only if p directly depends on q.*

Definition 3.4.2. *(Simon, 2006) The reduced graph for a co-constraint logic program has vertices consisting of the strongly connected components of the dependency graph of P. There is an edge from vertex u_1 to vertex u_2 in the reduced graph if and only if some predicate in u_1 depends directly on some predicate in u_2. A vertex in a reduced graph is said to be coinductive (resp. inductive) if it contains only coinductive (resp. inductive) predicates.*

3.4.2 Declarative Semantics

The semantics of co-CLP follows that of coinductive constraint logic programming. However, both the declarative and operational semantics of coinductive CLP must be changed to allow both inductive and coinductive predicates.

Since the set of predicates in program P is stratified into a collection of mutually disjoint strata of predicates, a vertex in a reduced graph of P is called a stratum. All the vertices in a stratum are of the same kind, either coinductive or inductive. A stratum u depends on stratum u', if there is an edge from u to u' in the reduced graph. A stratum u is said to be lower than stratum u' (u' is higher than u), if there is a path from u to u' in the reduced graph. A stratum u is said to be strictly lower than stratum u' (u' is strictly higher than u), if u is lower than u' and $u \neq u'$.

The model of each stratum, i.e., the model of a vertex in the reduced graph of a co-CLP is defined using the $T^{\mathcal{D}}_{u,P}$ operator below. Since a co-constraint logic program is stratified, its reduced graph is always a DAG. Moreover, all predicates in the same stratum are the same kind, coinductive or inductive. Therefore, when $T^{\mathcal{D}}_{u,P}$ is used, the atoms defined as true in lower strata are treated as facts when proving atoms containing predicates in the current stratum.

Definition 3.4.3. *If R is the union of the models of strata that are strictly lower than stratum u, then:*

$T^{\mathcal{D}}_{u,P}(I) = \ R \ \cup \ \{\boldsymbol{p}(\tilde{h}) \mid p(\tilde{x}) \leftarrow c, b_1, \ldots, b_n$ *is a rule of P, $p \in u$, and there exists a constraint consistent valuation v such that* $\mathcal{D} \models v(c), v(\tilde{x}) = \tilde{h}, v(b_i) \in I, i = 1, \ldots, n\}$

The model of a stratum u of P is the least fixed-pint of $T^{\mathcal{D}}_{u,P}$ if u is inductive, and the greatest fixed-point of $T^{\mathcal{D}}_{u,P}$ if u is coinductive.

Definition 3.4.4. *The model of a co-CLP program P over a constraint domain $\mathcal{D}$, written $M^{\mathcal{D}}(P)$, is the union of the models of every stratum of P.*

The model of a co-constraint logic program is built by creating the inductive model for inductive strata, starting the lower strata, and the coinductive model for coinductive strata. This process is repeated from the lowest strata up to the topmost strata. Note that lowest and topmost strata exist since the program is stratified. Note also that the set of $\mathcal{D}$-interpretations forms a complete lattice under the subset ordering. $T^{\mathcal{D}}_{u,P}$ is a monotonic function on a complete lattice. Therefore the least and greatest fixed-points exist for $T^{\mathcal{D}}_{u,P}$.

Definition 3.4.5. *The set of idealized proofs of a stratum u of P over domain $\mathcal{D}$ equals $\mu\Gamma^{\mathcal{D}}_{u,P}$ if u is inductive and $\nu\Gamma^{\mathcal{D}}_{u,P}$ if u is coinductive, such that if R is the union of the sets of idealized proofs of the strata strictly lower than u then*

$\Gamma^{\mathcal{D}}_{u,P}(S) = R \cup \{node(\boldsymbol{p}(\tilde{h}), T_1, \ldots, T_n) \mid p(\tilde{x}) \leftarrow c, b_1, \ldots, b_n$ *is a rule of P, $p \in u$, and there exists a constraint consistent valuation v such that* $\mathcal{D} \models v(c), v(\tilde{x}) = \tilde{h},\ T_i \in S, T_i(\epsilon) = v(b_i),\ i = 1, \ldots, n\}$

Definition 3.4.6. *The set of idealized proofs generated by a co-constraint logic program P over a constraint domain $\mathcal{D}$, denoted by $\Gamma_P^{\mathcal{D}}$, is the union of the sets of idealized proofs of every stratum of P.*

We next define the declarative semantics of co-constraint logic programs in terms of idealized proofs. If $S = \{a \mid \exists T \in \Gamma_P^{\mathcal{D}}, a \text{ is the root of } T\}$, then $S = M^{\mathcal{D}}(P)$. Any element in the model has an idealized proof and anything that has an idealized proof is in the model. This formulation will be used in soundness and completeness proofs of operational semantics of co-CLP in order to distinguish between the cases when a query has a finite derivation, and when there are only infinite derivations of the query.

3.4.3 Operational Semantics

The operational semantics for co-CLP is defined using state transition systems. Part of the state is the set of unproved goals, modeled by a forest of finite trees of atoms, the other part of the state includes the multiset of constraints and substitutions. The operational semantics of co-CLP is similar to that of coinductive constraint logic programming in the sense that it allows program rules (clauses) along with transition rules that manipulate constraints (checking consistency, adding constraints and inferring new constraints) and also the coinductive hypothesis rule. However, the set of unproved goals is modeled as a forest rather than a tree. This will become clear when we present the operational semantics of co-CLP.

Definition 3.4.7. *A state is a triple (F, E, C), where*

- *F is a finite multiset of finite trees with nodes labeled with atoms,*
- *E is the set of equalities of the form $x =_u t$, in which x is a variable and t is a term and each variable x appears at most once in the left hand side of equations in E,*
- *C is the set of constraints from $\mathcal{L}^D$, it is called the constraint store.*

There is one other state, called the failure state.

Intuitively, the collection of as-yet-unseen atoms is represented by forest F. When an atom labeling a leaf in tree T of forest F is proved to be true, T is modified (using function δ, defined earlier in this chapter) by removing the leaf and the maximum number of its ancestors, such that the result is still a tree. If all nodes in a tree are removed, the tree itself is removed. C is the collection of accumulated constraints built from the function and relation symbols of Σ, and E is the system of equations that captures the accumulated results of unification.

A transition rule r of a co-constraint logic program P is defined in exactly the same manner as in coinductive constraint logic programming; however, the operational semantics of co-CLP, presented next, is different from that of ciunductive CLP.

Definition 3.4.8. *Let T be a tree in forest F. A state (F, E, C) transitions to another state $((F \backslash \{T\}) \cup F', E', C')$ by transition rule r of program P when*

- *r is a definite clause of the form $p(t'_1, \ldots, t'_n) \leftarrow c, b_1, \ldots, b_m$, renamed to new variables, π is a leaf in T, $T(\pi) = p(t_1, \ldots, t_n)$. Let E'' be the substitution obtained from unification of $t_1, \ldots, t_n$ and $t'_1, \ldots, t'_n$, then $E' = E \cup E''$, $C' = C \cup c$ and F' is obtained as follows:*
  - *$m = 0$: $F' = \{\delta(T, \pi)\}$.*
  - *$m > 0$ and p is coinductive: $F' = \{T'\}$ where T' is equal to T except at $\pi.i$ and $T'(\pi.i) = b_i$, for $1 \leq i \leq m$.*
  - *$m > 0$ and p is inductive:*
    * *$\pi = \epsilon$: $F' = \{node(b_i) \mid 1 \leq i \leq m\}$.*
    * *$\pi = \pi'.j$ for some positive integer j : Let T'' be equal to T except at $\pi'.k$ for all $k \geq j$, where T'' is undefined. $F' = \{T'\}$ where T' is equal to T'' except at $\pi'.i$, where $T'(\pi'.i) = b_i$, for $1 \leq i \leq m$, and $T'(\pi'.(j-1+m+k)) = T(\pi'.k)$, for $k > j$.*

- *r is of the form $\eta(\pi, \pi')$, p is a coinductive predicate, π is a proper prefix of π', which is a leaf in T, $T(\pi) = p(t'_1, \ldots, t'_n)$, $T(\pi') = p(t_1, \ldots, t_n)$. Let E'' be the substitution obtained from unification of $t_1, \ldots, t_n$ and $t'_1, \ldots, t'_n$, then $E' = E \cup E''$, $C' = C \cup c$ and $F' = \{\delta(T, \pi')\}$.*

- *r is an inference rule, $F' = \{T\}$, $E' = E$ and $C' = infer(C, E)$.*

- *r is a consistency check rule, if consistent(C) holds then $F' = \{T\}$, $E' = E$ and $C' = C$, if $\neg$consistent(C) holds then $((F \backslash \{T\}) \cup F', E', C') =$ failure.*

Note that when a state is transformed by using a coinductive hypothesis, the set of accumulated constraints should be consistent for the coinductive hypothesis to be applied. The state in a transition system can also change by applying an instance of a clause from the program. When a subgoal is proved true, the unneeded nodes will be pruned from the forest. Note that the body of an inductive clause is overwritten on top of the leaf of a tree. When the tree contains only the root, the old tree will be replaced with one or more singleton trees. Coinductive subgoals will be remembered, in the form of a forest, so that infinite proofs can be recognized[2]. The state can also change by applying the inference rule and also by checking the consistency of constraints.

Definition 3.4.9. *A transition sequence in a co-CLP program P consists of a sequence of states $(F_1, E_1, C_1), (F_2, E_2, C_2), \ldots$, and a sequence of transition rules $r_1, r_2, \ldots$, such that (F_i, E_i, C_i) transitions to $(F_{i+1}, E_{i+1}, C_{i+1})$ by rule r_i of Definition 3.4.8.*

A transition sequence defines the trace of an execution. Execution terminates when it reaches a final state: either all atoms have been proved or the execution path reaches a state where no further transitions are possible.

Definition 3.4.10. *An accepting state is of the form $(\emptyset, E, C)$, where $\emptyset$ denotes the empty set. A failure state is a non-accepting state which does not have any outgoing transitions. A final state is an accepting state or a failure state.*

[2]Note that the ancestors of a subgoal can be accessed in the recursion stack.

Definition 3.4.11. *A derivation of a state (F, E, C) in a co-constraint logic program P is a state transition sequence where the first state is (F, E, C). A derivation is successful if it ends in an accepting state, and a derivation is failed if it ends in a failure state. A goal G of the form $a_1, \ldots, a_n$, has a derivation in program P, if $(\{node(a_i)|1 \leq i \leq n\}, \emptyset, \emptyset)$ has a derivation in P. For the goal G with free variables $\tilde{x}$, if $(G, \emptyset, \emptyset)$ has a successful derivation in P ending at $(\emptyset, E, C)$, then $\exists_{-\tilde{x}}E(C)$ is called the answer constraint of the derivation.*

Theorem 3.4.1. *(soundness) If the query $a_1, \ldots, a_n$ has a successful derivation in co-CLP program P over the constraint domain $\mathcal{D}$, which ends at state $(\emptyset, E, C)$, then $E(a_1, \ldots, a_n)$ is true in program P if $\mathcal{D} \models E(C)$.*

Theorem 3.4.2. *(completeness) Let $a_1, \ldots, a_n \in M^{\mathcal{D}}(P)$. If each $a_1, \ldots, a_n$ has a rational idealized proof, then the query $a_1, \ldots, a_n$ has a successful derivation in co-constraint logic program P over the domain of constraint $\mathcal{D}$.*

The proof of correctness for co-CLP is the straightforward extension of the proofs for correctness of co-LP (Simon, 2006), which is not presented here, but shown in Appendix A.

3.5 Conclusions

In the first part of this chapter, we proposed a new programming paradigm, coinductive constraint logic programming, which extends coinductive logic programming with constraints. Coinductive CLP can be used to reason about infinite (rational) and cyclical objects and structures in the presence of constraints. We have proposed a new declarative and also operational semantics for coinductive CLP. The operational semantics of coinductive CLP relies on the coinductive hypothesis rules to obtain finite proofs for logical statements that would otherwise have infinite proofs.

In the second part of this chapter, we proposed a new programming paradigm, co-CLP, which merges constraint logic programming and coinductive logic programming. The paradigm allows us to define inductive and coinductive predicates with constraints imposed on their arguments. It is the programmer's responsibility to specify for each predicate whether it is coinductive or inductive. The inductive and coinductive predicates cannot be mutually recursive in co-CLP programs. The operational semantics of co-CLP uses the coinductive hypothesis rule for proving logical statements which would otherwise have infinite proofs. The set of accumulated constraints should be consistent when using the coinductive hypothesis rule.

Co-CLP has interesting real world applications in which infinite (rational) computations are combined with constraints. An example application is a reactor temperature control system in which the controllers run forever and the system requirements are specified as a set of constraints on physical quantities: temperature and time. Many cyber-physical systems also exhibit these two features. In summary, co-CLP is a new paradigm that can be used for modeling real-time, hybrid and cyber-physical systems, which exhibit infinite behavior and also run under some constraints imposed by the environment or by other systems.

CHAPTER 4
VERIFYING COMPLEX REAL-TIME SYSTEMS

4.1 Introduction

Using timed automata is a popular approach to designing, specifying, and verifying real-time systems (Alur and Dill, 1990, 1994). Timed automata can also provide foundational basis for cyber-physical systems (CPS) (Lee, 2008, Gupta, 2006b) that are currently receiving a lot of attention. Timed automata are ω-automata (Thomas, 1990) extended with clocks (or stop watches). Transitions from one state to another are not only made on the alphabet symbols of the language, but also on constraints imposed on clocks (e.g., at least 2 units of time must have elapsed). Timed automata are suitable for specifying a large class of real-time systems; however, they suffer from the same limitations that any automaton suffers, in that they can recognize only *timed regular languages*. This restriction to regular languages renders them unsuitable for many complex, useful applications where the language involved may not be regular. To overcome this problem, timed automata have been extended to pushdown timed automata (PTA) (Dang, 2001). A pushdown timed automaton recognizes sequences of timed words, where a timed word is a symbol from the alphabet of the language the automaton accepts, paired with the time-stamp indicating the time that symbol was seen. The sequence of timed words in a string accepted by a pushdown timed automaton must obey the rules of syntax laid down by the underlying untimed pushdown automaton, while the time-stamps must obey the timing constraints imposed on the times at which the symbols appear.

The work in Gupta and Pontelli (1997) showed how timed automata can be modeled via constraint logic programming over reals (or CLP(R)) and their properties verified. However, this work does not model the ω-automata naturally, as it cannot model infinite timed words.

In this chapter we extend the efforts of Gupta and Pontelli (1997), and employ the principles of coinductive constraint logic programming to elegantly model timed automata and PTA. We propose a general framework based on coinductive constraint logic programming over reals (coinductive CLP(R)) for modeling/verifying real-time systems (including CPS).

The formalism that are used in this framework are timed automata and pushdown timed automata which can be computationally modeled by coinductive CLP(R). Note that the formulation of PTA is our own, though they were first introduced in Dang (2001).

We show how a coinductive CLP(R) rendering of a timed automaton and a pushdown timed automaton can be used to verify safety and liveness properties of a system. We illustrate the effectiveness of our approach by showing how the well-known *generalized railroad crossing (GRC) problem* of Lynch and Heitmeyer (Heitmeyer and Lynch, 1994) can be naturally modeled, and how its various safety and utility properties can be easily verified. Our approach based on coinductive CLP(R) for handling the GRC is considerably more elegant and simpler than other proposed approaches, e.g., Puchol (1995).

4.2 Modeling Timed Automata and Pushdown Timed Automata

Timed automata/PTA have been used as a powerful formalism for specifying, designing and analyzing real-time systems. However, modeling timed automata/PTA is not an easy task. This difficulty is introduced by two fundamental features of these formalisms: (i) they are involved with time, a special type of continuous quantity which always advances, and clock constraints which are posed over continuously flowing time, (ii) they accept infinite timed words.

In the previous chapter, we proposed co-CLP as a new paradigm for merging constraints and coinduction. While co-CLP can be used for modeling infinite structures in presence of constraints, it cannot be directly used for modeling timed automata and PTA. The reason why is because the operational semantics of co-CLP (and also coinductive CLP) relies on the coinductive hypothesis rule: if during execution, a call A is encountered that *unifies* with an ancestor call A', then a coinductive success can be obtained if the set of accumulated constraints are satisfied. When we are dealing with time, some of the arguments in A might be time-related arguments. Since time is always advancing, it would be impossible to unify a time-related argument in A with that of A'. In other words, it would be impossible to obtain a coinductive success, if coinductive predicates are involved with time-realted arguments. Therefore, for modeling timed automata/PTA or in general infinite behavior of real-time systems, we found it convenient to use a slight extension of the co-CLP. This extension includes a slight change in the operational semantics of co-CLP. This extension is not reflected in the declarative semantics of co-CLP described in Chapter 3.

First, we describe the extension of the operational semantics of co-CLP, which we call it extended co-CLP. Next, we present a framework based on this extended co-CLP for modeling timed automata/PTA in which both aspects, mentioned above, can be handled naturally and faithfully. Next, we show how the LP-programming based realization of timed automata can be extended in order to obtain a LP-based realization of pushdown timed automata.

4.2.1 Extension of co-CLP for Modeling Time-Related Arguments

The syntax of the extended co-CLP is similar to that of co-CLP; however, since we cannot obtain unification on time-related arguments, we define two types of arguments: coinductive arguments and non-coinductive arguments. For each coinductive predicate p, the programmer must specify on which arguments it will be coinductive. Therefore, an extended coinductive constraint logic program is a normal co-constraint logic program extended with a set of declarations which are used to specify coinductive arguments of predicates. For every n-ary coinductive predicate p, there is a declaration of the form: `p(+,...,+,-,...,-)`. The predicate p will be coinductive only on those arguments that are mapped to $+$; these arguments are called *coinductive arguments*. Arguments that are mapped to $-$ are called *non-coinductive arguments* and must satisfy the following requirements. Each coinductive argument of a coinductive predicate p,

must appear as a fresh variable in the heads of clauses for p, must occur in some constraint in the body of the clause,

should not appear as a coinductive argument for some coinductive predicate in the bodies of clauses for p.

Example 4.2.1. *In the following program the predicate p is declared as coinductive on its first argument and inductive on its second argument. Note that Y appears in constraints* `{Y2 > Y}`, `{X - Y =< 3}` *in the body of p.*

```
p(+, -).
p([X | T], Y) :- {Y2 > Y}, {X - Y =< 3}, p(T, Y2).
```

The operational semantics of the extended co-CLP is similar to that of co-CLP, except for the coinductive hypothesis rule which is as follows: if during execution, a call A is encountered whose coinductive arguments unify with those of an ancestor call A', and moreover the set of accumulated constrains is consistent, then A is deleted from the resolvent and the substitution resulting from unification of coinductive arguments of A and A' is applied to the resolvent.

The state of program can be described as pairs of forests of unproved atoms along with the set of accumulated constraints and a substitution, as in co-CLP. The transition rules of the extended co-CLP are similar to those of co-CLP; however, a coinductive hypothesis rule is of the form $\eta(\pi, \pi')$ in which, π and π' are paths such that π is a proper prefix of π', π is labeled by $p(t_{c_1}, \ldots, t_{c_n}, t_1, \ldots, t_k)$, π' is labeled by $p(t'_{c_1}, \ldots, t'_{c_n}, t'_1, \ldots, t'_k)$, $t_{c_1}, \ldots, t_{c_n}$ and $t'_{c_1}, \ldots, t'_{c_n}$ are coinductive arguments and they are unifiable.

Definition 4.2.1. *Let T be a tree in forest F. A state (F, E, C) transitions to another state $((F \backslash \{T\}) \cup F', e', C')$ by transition rule r of program P when*

- *r is a definite clause of the form $p(t'_1, \ldots, t'_n) \leftarrow c, b_1, \ldots, b_m$, renamed to new variables, π is a leaf in T, $T(\pi) = p(t_1, \ldots, t_n)$. Let E'' be the substitution obtained from unification of $t_1, \ldots, t_n$ and $t'_1, \ldots, t'_n$, then $E' = E \cup E''$, $C' = C \cup c$ and F' is obtained as follows:*

- *$m = 0$: $F' = \{\delta(T, \pi)\}$.*
- *$m > 0$ and p is coinductive: $F' = \{T'\}$ where T' is equal to T except at $\pi.i$ and $T'(\pi.i) = b_i$, for $1 \leq i \leq m$.*
- *$m > 0$ and p is inductive:*
  - *$\pi = \epsilon$: $F' = \{node(b_i) \mid 1 \leq i \leq m\}$.*
  - *$\pi = \pi'.j$ for some positive integer j : Let T'' be equal to T except at $\pi'.k$ for all $k \geq j$, where T'' is undefined. $F' = \{T'\}$, where T' is equal to T'' except at $\pi'.i$, where $T'(\pi'.i) = b_i$, for $1 \leq i \leq m$, and $T'(\pi'.(j-1+m+k)) = T(\pi'.k)$, for $k > j$.*

- *r is of the form $\eta(\pi, \pi')$, p is a coinductive predicate, π is a proper prefix of π', which is a leaf in T, $T(\pi) = p(t'_{c_1}, \ldots, t'_{c_n}, t'_1, \ldots, t'_k)$, $T(\pi') = p(t_{c_1}, \ldots, t_{c_n}, t_1, \ldots, t_k)$. Let E'' be the substitution obtained from unification of $t_{c_1}, \ldots, t_{c_n}$ and $t'_{c_1}, \ldots, t'_{c_n}$, then $E' = E \cup E''$, $C' = C \cup c$ and $F' = \{\delta(T, \pi')\}$.*
- *r is an inference rule, $F' = \{T\}$, $E' = E$ and $C' = infer(C, E)$.*
- *r is a consistency check rule, if $consistent(C)$ holds then $F' = \{T\}$, $E' = E$ and $C' = C$, if $\neg consistent(C)$ holds then $((F \backslash \{T\}) \cup F', E', C') = failure$.*

Note that when a state is transformed by using a coinductive hypothesis, only coinductive arguments are taken into account and the non-coinductive arguments will be ignored; however, the set of accumulated constraints should be consistent for the coinductive hypothesis to be applied.

A transition sequence, an accepting state and a derivation of a state, can be defined for the extension of co-CLP in the exact same manner as in co-CLP. Note that the extension of co-CLP is not reflected in the declarative semantics of co-CLP, described in Section 3.4.2.

Next, we propose a framework for modeling timed automata, using the extended co-CLP.

4.2.2 Modeling Timed Automata via Co-CLP(R)

We propose co-CLP(R) as the underlying formalism for modeling timed automata, and show how both fundamental aspects of them can be elegantly handled within this formalism. We show how coinduction can enable us to develop language processors that recognize infinite strings. Further, incorporation of constraints into coinduction allows modeling of time aspects of timed automata. The general method of converting transitions of timed automata (with one clock) to set of CLP(R) rules, presented in Table 4.1, is outlined next. Note that this method can be also used for converting PTA (with one clock) to CLP(R) programs, which is explained in Section 4.2.4. The general method that handles any number of clocks is presented in Appendix B.

The method takes the description of a timed automaton and generates a coinductive constraint logic program over reals. The generated logic program models the timed automaton as a collection of transition rules (one rule per transition of the timed automaton), where each rule is extended with clock constraints. For simplicity of presentation the method is explained for a timed automaton with one clock, but it can handle any number of clocks in a similar manner.

The description of a timed automaton contains a list of states of the automaton, a list of accepting states, a list of clocks, and a list of tuples of the form `(Si, Input, So, ResetClocks, Constraints)`, representing the transitions of the automaton. The first three arguments of this tuple are self-explanatory. `ResetClocks` is a list of clocks which get reset in this transition. `Constraints` is a formula that should be satisfied by the current values of clocks for transition to take place. `Constraints` is implemented as a possibly empty list of propositions of the form $c \sim x$; where c is a clock, x is a constant, and $\sim \in \{=, <, \leq, >, \geq\}$. The method takes each and every transition of the automaton and inserts its corresponding logic rule in which the head is of the form `trans(Si, Input, So, W, Ti, To)`, and the body is composed of CLP(R) constraints, where the CLP(R) constraints are enclosed within the curly braces. To illustrate, we describe the logic programming rendering of the timed automaton shown in Figure 4.1, automatically generated by our system. The transitions of

Table 4.1. The General Method of Converting Transitions of Timed Automata (with one clock) to CLP(R) Rules

```
insert-ta-one-clock :-
 transitions(Transitions),
 open('file', append, OpenId),
 assert_transitions(Transitions, OpenId),
 create_driver(OpenId), close(OpenId).

assert_transitions([H| T], OpenId) :-
 assert_transition(H, OpenId), nl(OpenId),
 assert_transitions(T, OpenId).

assert_transitions([], OpenId) :- close(OpenId).

assert_transition((S1, Input, S2, Resets, Consts), OpenId) :-
 assert_constraints(Consts, Resets, Body),
 W = 'W', Ti = 'Ti', To = 'To',
 Rule = ':-'(trans(S1, Input, S2, W, Ti, To), Body),
 write(OpenId, Rule).

assert_transition((S1, Input ,S2, Action, Resets, Consts), OpenId) :-
 assert_constraints(Consts, Resets, Body),
 W = 'W', Ti = 'Ti', To = 'To', C = 'C',
 (Action = 'push(1)' ->
     Rule = ':-'(trans(S1, Input, S2, W, Ti, To, C, [ 1 | C ]), Body)
    ;Rule = ':-'(trans(S1, Input, S2, W, Ti, To, [ 1 | C ], C), Body)),
 write(OpenId, Rule).

assert_constraints([Const | Consts], Clocks, ','(A, As) ) :-
 atom_chars(Const, [Clock | Rest]),
 append(['{', ' ', 'W', ' ', -, ' ', 'T', 'i'], Rest, R1),
 append(R1, ['}'], R2),
 atom_chars(A, R2),
 assert_constraints(Consts, Clocks, As).

assert_constraints([], Clocks, A ) :-
 (not_empty(Clocks) ->
   atom_chars(A, ['{', 'T', 'o', =, 'W', '}', '.'])
  ;atom_chars(A, ['{', 'T', 'o', =, 'T', 'i', '}', '.'])).

empty([]).
not_empty([_|X]).
```

Table 4.2. The Transitions of Timed Automaton of Figure 4.1

```
transitions( [(s0, a, s1, [c], []       ),
              (s1, a, s1, [c], []       ),
              (s1, b, s2, [],  ['c < 5']),
              (s2, b, s2, [],  []       ),
              (s2, b, s0, [],  ['c < 20']) ]).
```

this timed automaton can be specified through a simple fact, as follows. After specifying the transitions, calling the predicate `insert-ta-one-clock.` will generate the set of CLP(R) clauses whose head is `trans`. This logic programming is presented in Table 4.2.

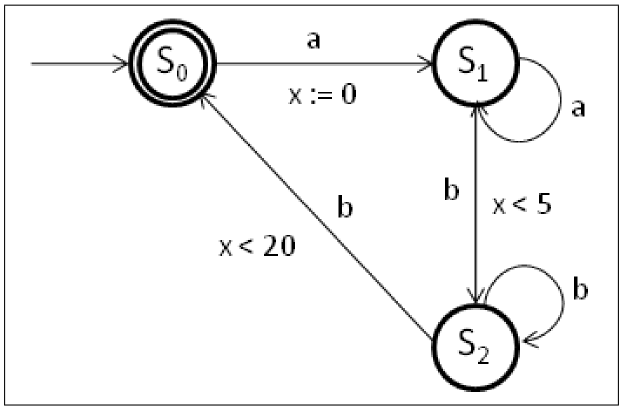

Figure 4.1. A Timed Automaton

In `trans/6`, `W` represents the wall clock time; the pair of arguments, `Ti` and `To`, represent the clock c of the timed automaton. In fact, a pair of arguments have to be added for each clock that is used in the automaton. The first argument of this pair is used to remember the last (wall clock) time this clock was reset, while the second one is used to pass on this clock's value to the next transition. Note that the CLP(R) constraints are enclosed within curly braces, as is the convention in most Prolog systems. Resetting a clock is modeled as a constraint, e.g., the constraint {`To = W`} in the first clause of `trans/6` expresses the clock being reset with this transition. Assigning the wall clock W to To can be understood as remembering the wall clock time at which the clock was reset.

Table 4.3. The Logic Programming Realization of Timed Automaton in Figure 4.1

```
trans(s0, a, s1, W, Ti, To) :- {To = W}.
trans(s1, b, s2, W, Ti, To) :- {W - Ti < 5,  To = Ti}.
trans(s1, a, s1, W, Ti, To) :- {To = Ti}.
trans(s2, b, s2, W, Ti, To) :- {To = Ti}.
trans(s2, b, s0, W, Ti, To) :- {W - Ti < 20, To = Ti}.

final(s0).

:- coinductive(auto(+, +, +, -, -, -)).
auto([X | T], Si, [Si | S], W, Ti, [(X, W) | R]) :-
     trans(Si, X, So, W, Ti, To),
     {W2 > W},
     auto(T, So, S, W2, To, R).
```

Once the set of transition rules of the timed automaton are defined in constraint logic programming, through `trans/6`, the coinductive predicate `auto/6` realizes the automaton, calling `trans/6` rule repeatedly. The constraint `W2 > W` advances the time on the wall clock after every transition. `auto/6` generates the timed trace of events as output. Note that `trans/6` is declared as inductive; while `auto/6` is declared as coinductive on its first three arguments and inductive on the last three arguments. The third argument of `auto/6` is the list of visited states. The final state `s0` must appear infinitely often in the set of visited states, for a timed word to be accepted by the automaton. This can be checked using the coinductive predicate `comember/2` defined below.

```
comember(X, L1) :- drop(X, L1, L2), comember(X, L2).

drop(X, [X | T], T).
drop(X, [_ | T], R) :- drop(X, T, R).
```

Note that the coinductive CLP(R) predicate `auto/6` is automatically generated by our system. The outline of this system is presented in Appendix B. Once the timed automaton is realized as a coinductive CLP(R), asking the query

```
auto(Y, s0, V, 0, 0, R), comember(s0, V),
```

generates all the cyclic solutions along with the constraint answers.

Next, we argue that the coinductive success obtained by the extended co-CLP, in modeling timed automata, is indeed sound. As we mentioned earlier, time is a special quantity that always advances; as a result, we cannot obtain coinductive success by unifying time-related arguments of coinductive predicates. We will declare the time-related arguments of coinductive predicates as non-coinductive arguments, while modeling timed automata and pushdown timed automata. We argue that if we obtain a coinductive success by unifying the coinductive arguments, and the set of clock constraints accumulated in the first traversal of a cycle p of a timed automaton are satisfiable, then p is an accepting cycle. In other words, the set of constraints accumulated in all the future traversals of p will remain satisfiable when time advances.

In the next section we study timed automata by a case analysis on how the constraints on clocks are related to the clock resets. We show that resetting the clocks, represented as constraints in our framework, does not affect the soundness of the coinductive CLP(R) programs realizing timed automata.

4.2.3 Justification of Coinductive CLP(R) Realization of Timed Automata

The intuitive idea behind using co-CLP for realizing timed automata is as follows. If the set of clock constraints accumulated in one traversal of a cycle of a timed automaton are satisfiable, they remain satisfiable in all the future cycles when time advances; in particular the set of clock constraints will be satisfiable in the nth cycle when n approaches to infinity (n approaching to infinity corresponds to time advancing and approaching to infinity). Therefore, a co-constraint logic program can be used to capture the infinite (rational) behavior of a timed automaton. We identify two classes of timed automata based on how clock constraints appear in their transitions: after some clock resets or before some clock resets. We argue that, in both classes clock resets, which are implemented as constraints, will not affect the soundness of the coinductive CLP(R) programs realizing timed automata. We investigate this problem by case analysis on clock resets of timed automata.

Definition 4.2.2. *A basic cycle of a timed automaton A is a traversal of states of A which starts at some state s and ends at s, without revisiting s in between.*

Definition 4.2.3. *A cycle is either a basic cycle or composition of more than one basic cycle.*

Definition 4.2.4. *An accepting cycle of a timed automaton A with starting state s_0 and accepting state s_a is a cycle which starts at s_0 and contains s_a.*

Definition 4.2.5. *For a timed automaton A with an accepting cycle p, p^ω is an accepting traversal of A.*

Proposition 4.2.1. *In a timed automaton A with an accepting basic cycle p, if the clock constraints accumulated in one traversal of p are satisfiable, they remain satisfiable in all the future traversals of p.*

Proof. Note that in timed automata we are dealing with a finite number of states and a finite number of edges. We consider two cases:

Case 1: p contains a constraint of one of the forms $x \leq a$, $x < a$ or $x = a$, and there is a clock reset of the form $x := 0$ which is also part of p. We identify two subcases:

case 1a: Every clock constraint in p is preceded by a clock reset in p. An example of this behavior is shown in the timed automaton of Figure 4.2(a), where s_0 is the starting state, $p = s_0s_1s_2s_3s_0$ is a basic cycle, x and y are clocks, and both clock constraints on x and y are preceded by clock resets on x and y in p.

case 1b: p contains a clock constraint on x which appears before resetting the clock x in p. An example of this behavior is shown in the timed automaton of Figure 4.2(b). Note that in this example checking the constraint $x < 2$ in the first traversal of $p = s_0s_1s_2s_3s_0$ is performed against the clock reset which was performed before the first traversal of p started.

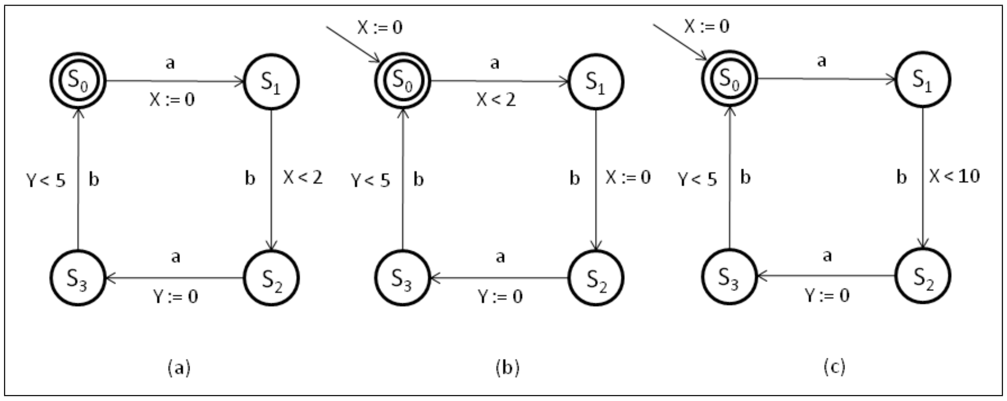

Figure 4.2. Timed Automata Examples

observation: The automaton of case 1b after the first traversal of p will be in a situation similar to case 1a, in the sense that for each clock x such that x appears in a constraint of one of the forms: $x < a, x \leq a, x = a$ in p, x is reset within p; moreover, all the constraint checks on x will be performed against a clock reset on x which is performed within p. To avoid the distinction between two cases 1a and 1b, we ignore the first traversal of p and we argue that if the set of constraints accumulated in the second traversal of p in A are satisfiable, they will remain satisfiable in all future traversals of p (it is obvious that if the set of constraints are satisfiable in the second traversal of p, they were satisfiable in the first traversal of p also).

We consider two cases:

(a) The set of accumulated constraints in the second traversal of p are satisfiable. Assume the set of accumulated constraints in the kth traversal of p are satisfiable, we prove they will be satisfiable in the $(k+1)$th traversal of p.

Assume that in the kth traversal of p, n constraints are accumulated. Moreover, let the time-stamps associated with the transitions in the kth traversal of p be $T_0, T_1, \ldots, T_n$; so, T_0 and T_n are the time-stamps associated with the first and last transitions of the kth traversal of p, respectively. In other words, the kth traversal of p takes $T_n - T_0$ units of time to complete. Assume the set of constraints accumulated in the kth traversal of p is as follows:

$$T_1 - T_0 \sim a_1$$

$$.$$

$$.$$

$$.$$

$$T_n - T_{n-1} \sim a_n$$

Note that in the set of constraints above a_i's are constants with real values and $\sim \in \{>, \geq, <, \leq, =\}$. Assume this set of constraints is satisfied by the values of clocks at the time of transitions. We define a mapping from $\mathbf{R}^+$ to $\mathbf{R}^+$, which takes a time-stamp T_i of the set of constraints above and maps it to $T_i + \Delta$, in which $\Delta = T_n - T_0$. We do this transformation uniformly and simultaneously on all the time-stamps appearing in the set of constraints above. By performing this mapping we obtain the set of constraints in the $(k+1)$th traversal of the cycle p. All the time-stamps appearing in the $(k+1)$th traversal of p are those of the kth traversal of p plus the time taken for the kth traversal to be completed (Δ). By performing this transformation the difference between any pair of time-stamps $T_{i+1} + \Delta$ and $T_i + \Delta$ appearing in the $(k+1)$th traversal of p are equal to those of T_{i+1} and T_i appearing in the kth traversal of p. Therefore, the resulting constraints by this transformation remain satisfiable. This justifies the coinductive success obtained by the extended co-CLP(R) system, when the cycle p is detected. Note that detecting the cycle p is characterized by unification of coinductive arguments. For instance, in the logic programming modeling of timed automaton of Figure 4.1, presented in Table 4.3, the cycle $p = s_0 s_1 s_2 s_0$ is detected by the fact that the first three arguments in the first call to `auto/6` are unifiable with those in some predecessor call.

(b) Assume that the set of accumulated constraints in the second traversal of p are not satisfiable. In this case, the underlying constraint solver will detect this inconsistency. Therefore, the co-CLP system will report the failure; this failure corresponds to the emptiness of the language of A.

Note that in case 1 we only considered clock constraints of the form $x \leq a$, $x < a$, and $x = a$. The reason is that for clock constraints of the form $x \geq a$ or $x > a$ in p, the clock x is not required to get reset within p for the automaton to accept infinite timed words. Intuitively, as time advances, it is always possible to find a value on clock x that satisfies $x \geq a$ or $x > a$ in the nth traversal of p when n approaches to infinity, even if clock x does not get reset.

Case 2: The timed automaton has a basic cycle p with at least one clock constraint of the form $x \leq a$, $x < a$, or $x = a$, but the clock x does not get reset within p.

In this case the timed automaton does not accept any infinite timed words, as there is no infinite timed word that satisfies the *progress* property. The progress property of timed automata states that only a finite number of events can happen in a bounded interval of time. The constraints of the form $x \leq a$, $x < a$, or $x = a$ in p, where clock x is not reset in p, put an upper bound on the time-stamps of the symbols corresponding to transitions of the automaton. Therefore, having a constraint of one of the forms mentioned above in p without x being reset in p, forces the time-stamps of all transitions of the timed automaton to be bounded by a, which violates the progress condition. Any timed word whose time-stamps are bounded by an upper bound a will not be considered as a valid infinite timed word and therefore will not be accepted by the timed automaton. An example of this situation is shown in Figure 4.2(c). Note that the language of this timed automaton is empty as there is no infinite timed word that satisfies the progress property for this timed automaton. Note also that in this case, the set of accumulated constraints in the first finite number of traversals of p might be satisfiable; however, the coinductive success should not be invoked. Since coinductive is used for proving infinite (rational) behavior of infinite structures, the behavior of such automata cannot be justified by coinduction. A dynamic analysis on the set of constraints and resets that appear in transitions of a timed automaton can be used to check this situation. $\square$

The previous proposition can be understood as follows: for a timed automaton A with a basic cycle p which contains a clock constraint of one of the forms: $x < a, x \leq a$, and $x = a$, p^ω is an accepting traversal of A if the clock x is reset an infinite number of times in p^ω.

In general, an accepting traversal of a timed automaton might be composed of one or more basic cycles. The idea is that for every constraint of the form $x \leq a$, $x < a$, or $x = a$ appearing in a cycle p, there must exist a basic cycle p' in which the clock x is reset and p' is traversed an infinite number of times. For instance, in the timed automaton of Figure 4.3(a) $p = s_0s_1s_2s_3s_0$ and $p' = s_0s_1s_0$ are two basic cycles. The constraint $x < 2$ is part of both p and p', clock x is reset in p but not in p'; therefore, p must appear an infinite number of times in any accepting traversal of this timed automaton. An example of accepting traversal of A is $((s_0s_1)^n(s_0s_1s_2s_3)^m)^\omega$, where $n \geq 0, m > 0$. Similarly, for the timed automaton of Figure 4.3(b), an accepting traversal is $((s_0s_1)^n(s_0s_1s_2s_3)^m)^\omega$, where $n > 0, m \geq 0$. The coinductive success can be justified by case analysis on basic cycles and clock resets, identical to the argument that we presented for basic cycles.

Note that if the cycle p contains a clock constraint of the form $y \geq a$ or $y > a$, then the clock y is not required to get reset for the cycle to be an accepting cycle. This is similar to the previous case where p was a basic cycle.

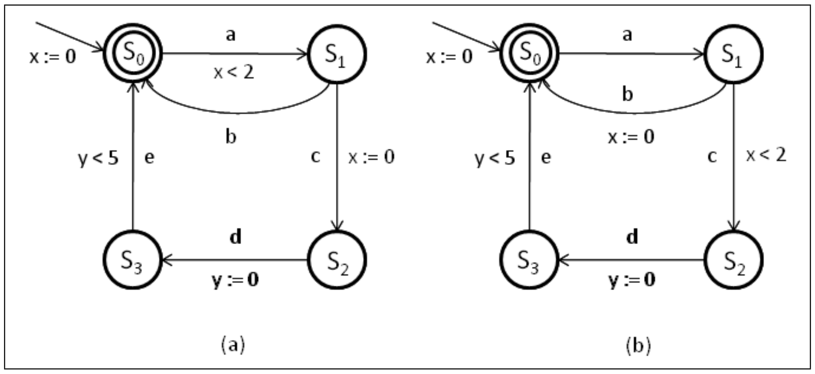

Figure 4.3. Timed Automata Examples

Next we show how the LP realization of timed automata can be extended in order to model pushdown timed automata.

Table 4.4. The Logic Programming Realization of The Pushdown Timed Automaton of Figure 2.6

```
trans(s0, a, s1, W, Ti, To, [],    [1]  ) :- {To = W}.
trans(s1, a, s1, W, Ti, To, C,     [1|C]) :- {To = Ti}.
trans(s1, b, s2, W, Ti, To, [1|C], C    ) :- {W - Ti < 5,  To = Ti}.
trans(s2, b, s2, W, Ti, To, [1|C], C    ) :- {To = Ti}.
trans(s2, b, s0, W, Ti, To, [1|C], C    ) :- {W - Ti < 20, To = Ti}.

final(s0).

:- coinductive(auto(+, +, +, +, -, -, -)).
auto(Si, [X | Inputs], [Si | S], Ci, W, Ti, [(X, W) | Outputs]) :-
       trans(Si, X, So, W, Ti, To, Ci, Co),
       {W2 > W},
       auto(So, Inputs, S, Co, W2, To, Outputs).
```

4.2.4 Modeling PTA with Coinductive CLP(R)

The general method of converting timed automata to coinductive CLP(R) can be extended with stack actions in order to obtain the LP realization of pushdown timed automata. In our framework, PTA are modeled as set of transition rules, as in timed automata; however, two extra arguments must be added to the transition rules of timed automata for modeling stacks. The stack can be realized in many ways, in this section we simply implement it as a list, while describing the LP-implementation of stacks. In the example described in Section 4.3.1 we implement it as a counter.

Next, we describe the logic programming rendering of the pushdown timed automaton shown in Figure 2.6. This logic programming is presented in Table 4.4. The first six arguments of `trans/8` are the exact same arguments for `trans/6`, described in Section 4.2.2; the last two arguments are lists which are used to implement the stack.

The coinductive `auto/7` rule takes the current state of the pushdown timed automaton, the list of input symbols, the wall clock time along with the clock and stack of the automaton as input; and generates the timed trace of events as output. It realizes the automaton, calling `trans/8` rule repeatedly, until the coinductive success is obtained. Note that the final state, in this case `s0`, must be traversed infinitely often for a timed word to be accepted by the pushdown timed automaton. The coinductive predicate `comember/2` can be used for this check, as it was explained in the previous section. The third argument of `trans/8` is used to accumulate the set of visited states. Note also that the stack must be empty while in state `s0`; this requirement is specified by the empty list [], in the first clause of `trans/8`. Note that `auto/7` is declared as coinductive only on the first four arguments (the current state, the input symbol seen, the set of visited states, and the stack, respectively), i.e., the wall-clock time and other arguments will be ignored to check if `auto/7` is cyclical. However, the set of accumulated constraints until the cycle is detected should be satisfied for a coinductive success.

Given this program, one can pose queries to it to check if a timed string satisfies the timing constraints. Alternatively, one can generate possible (cyclical) legal timed strings (note that a CLP(R) system will output timed strings, where time-stamps of terminal symbols in the output string will be represented as variables; constraints that these time-stamps must satisfy will be output as well). Finally, one can verify properties of this timed language (e.g., checking the simple property that all the `a`'s are generated within 5 units of time, in any timed string that is accepted).

4.3 Application: The GRC Problem

We next present the effectiveness of our approach by applying it to the well-known *generalized railroad crossing (GRC)* problem. The GRC problem has been proposed (Heitmeyer and Lynch, 1994) as a benchmark problem in order to compare the formal methods that have been invented for specifying, designing, and analyzing real-time systems. It also provides a better way to understand the use of these methods in developing practical real-time systems. The formal statement of the GRC problem, taken directly from Heitmeyer and Lynch (1994), is as follows.

The system to be developed operates a gate at a railroad crossing. The railroad crossing I lies in a region of interest R, i.e., $I \subseteq R$. A set of trains travel through R on multiple tracks in both directions. A sensor system determines when each train enters and exits region R. To describe the system formally, we define a gate function $g(t) \in [0, 90]$, where $g(t) = 0$ means the gate is down and $g(t) = \mathit{90}$ means the gate is up. We also define a set $\{\lambda_i\}$ of *occupancy intervals*, where each occupancy interval is a time interval during which one or more trains are in *I*. The *i*th occupancy interval is presented as $\lambda_i = [\tau_i, \nu_i]$, where τ_i is the time of the *i*th entry of a train into the crossing when no other train is in the crossing and ν_i is the first time since τ_i that no train is in the crossing (i.e., the train that entered at τ_i has exited as have any trains that entered the crossing after τ_i). Given two constants ξ_1 and ξ_2, $\xi_1 > 0$, $\xi_2 > 0$, the problem is to develop a system to operate the crossing gate that satisfies the following two properties:

Safety Property: $t \in \cup_i \lambda_i \Rightarrow g(t) = 0$

Utility Property: $t \notin \cup_i [\tau_i - \xi_1, \nu_i + \xi_2] \Rightarrow g(t) = 90$

Note that in the problem description above, I and R are used respectively to denote the railroad crossing, and the region from where a train passes a sensor until it exits the crossing. Some positive real-valued constants are defined by the GRC as follows:

- ϵ_1, a lower bound on the time from when a train enters R until it reaches I.
- ϵ_2, an upper bound on the time from when a train enters R until it reaches I.
- γ_{down}, an upper bound on the time to lower the gate completely.
- γ_{up}, an upper bound on the time to raise the gate completely.
- ξ_1, an upper bound on the time from the start of lowering the gate until some train is in I.

- ξ_2, an upper bound on the time since the last train leaves I until the gate is up (unless the raising is interrupted by another train getting close to I).
- β, an arbitrarily small constant used for some technical race conditions.
- δ, the minimum useful time for the gate to be up.

Some restrictions are placed on the values of the various constants as follows:

1. $\epsilon_1 \leq \epsilon_2$.
2. $\epsilon_1 > \gamma_{down}$. (The time from when a train arrives until it reaches the crossing is sufficiently large to allow the gate to be lowered.)
3. $\xi_1 \geq \gamma_{down} + \beta + \epsilon_2 - \epsilon_1$. (The time allowed between the start of lowering the gate and some train reaching I is sufficient to allow the gate to be lowered in time for the fastest train, and then to accommodate the slowest train.)
4. $\xi_2 \geq \gamma_{up}$. (The time allowed for raising the gate is sufficient.)

4.3.1 Modeling the GRC with Coinductive CLP(R)

To model the GRC, the number of tracks has to be given as an input to the system so that any number of tracks can be handled. The system is composed of: *gate* automaton that controls the gate, *controller* automaton that acts as an overall controller, and a *track* automaton per track that models the behavior of trains traveling through each track (Figure 4.4). In Figure 4.4, for ease of understanding, we assign the wall clock time (`W`) to clock variables when they are reset, since this is how clocks are realized in our implementation.

Each track is modeled as a timed automaton, called a *track* automaton. These track automata work in parallel, in the sense that when an event takes place in a specified track, only that track responds to this event and all automata for other tracks remain in their current states. A track automaton has five states (Figure 4.4, (i)) and takes actions based on three events: *(i) approach* indicates a train approaching the crossing; *(ii) in* indicates the

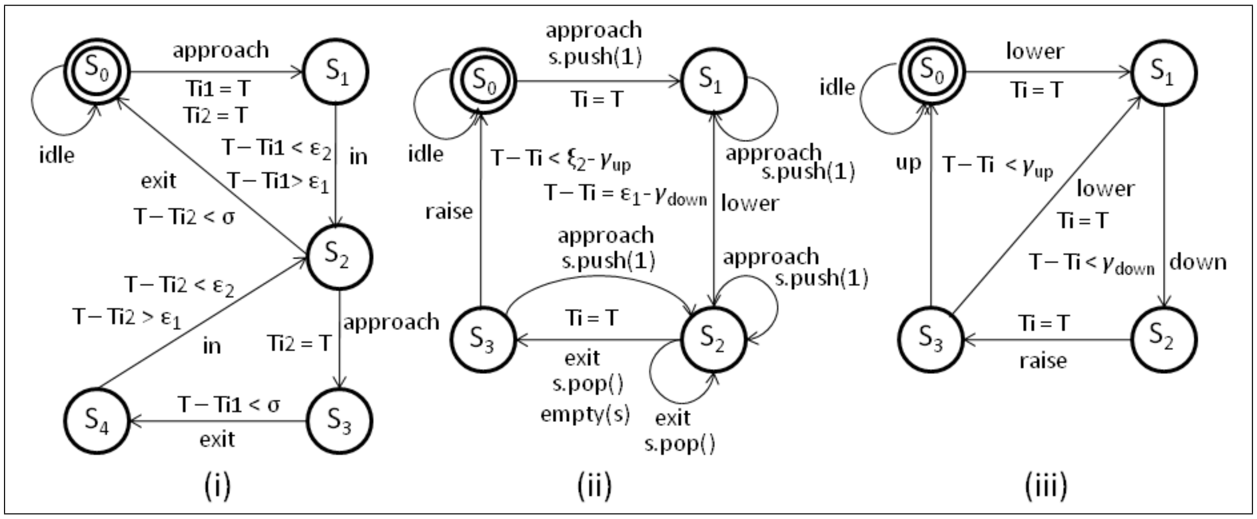

Figure 4.4. (i) Track Automaton, (ii) Controller Automaton, (iii) Gate Automaton

train being in the crossing; and, *(iii) exit* indicates that the train has left the crossing. The track automaton assumes that there cannot be two trains at the same time in the crossing area in each track. In other words trains travel at a safe distance from each other. The range of sensors is such that the *approach* signal of only at most one approaching train is registered for each track. Therefore, on receiving a new *approach* signal from a train on a given track, the system will respond to it assuming that there is no other train in that track or that any trains on that track will exit the crossing area before the new train would arrive there (while in state `s2` the track automaton can receive a new approach). There could be a situation in which multiple trains travel one after the other (at a safe distance from each other) in one track. In this situation, the system will work properly in the sense that the first train will enter the crossing area, the second train will enter the crossing area after the first train exits, the third train will take the place of the second train and so on. Therefore the gate remains down until the last train exits the crossing area. Note that the gate crossing system is not responsible for ensuring safe distance between trains, its task is to ensure the safety and utility of the crossing.

The *controller* automaton is modeled as a pushdown timed automaton with four states (Figure 4.4, (ii)). This automaton must keep track of trains currently in the system (i.e., those trains whose approach signal has been received): it has to ensure that the number of *approach* events is identical to those of *exit* events. Finite timed automata are not appropriate for specifying the controller, for two reasons: *(i)* we do not know the number of *approach* events in advance, so we cannot design one general timed automaton that works for an arbitrary number of tracks and trains. In other words, we would have to have different controller automata for different numbers of tracks. *(ii)* the automaton would become too complicated as the number of tracks increases. More tracks means more states and transitions, therefore a more complicated automaton. *Use of a stack in the pushdown timed automaton eliminates the need for new extra states and transitions as the number of approach signals and tracks increase.*

The *controller* must respond to four events: *approach*, *lower*, *exit*, and *raise. lower* and *raise* indicate starting of lowering and raising the gate respectively. On receiving an *approach* at state `s0` the controller's clock will be reset. This will ensure lowering of the gate before the train gets into the crossing area. The controller's clock will not get reset if an *approach* is received while in other states. The stack in pushdown timed automaton is used to keep track of the number of trains in the system. On receiving an *approach* the controller pushes the symbol "1" onto the stack and on receiving an *exit*, it pops the stack (implemented as a counter). When the stack is empty, the controller sends *raise* to the gate as the last train has left the system and it is safe to raise the gate. A transition is activated by the signal (event) received and also the state of the counter. Testable states of the counter are "$= 0$" and "$\neq 0$", and counter actions are *increment* and *decrement.* The automaton may ignore the state of the counter and act on the input signal. For example on receiving an *approach* at any state, the automaton increments the counter regardless of its current state. The automaton can act based on both the input signal, and the counter state. For instance, on receiving an *exit* in state `s2`, the automaton checks the state of the counter. If the counter is equal to zero, the controller will go to state `s3` and reset its clock; this will ensure that *raise* will be sent to the gate automaton within $\xi_2 - \gamma_{up}$ units of time after the controller clock is reset. If the counter is not equal to zero, the controller remains in state `s2`.

Finally, *gate* is modeled as a timed automaton with four states (Figure 4.4, (iii)) which takes actions based on four different events: *lower*, *down*, *raise*, and *up*. *down* and *up* indicate the gate being down and up respectively.

For modeling the GRC problem we set $\epsilon_1 = 2$, $\epsilon_2 = 3$, $\gamma_{down} = 1$, $\gamma_{up} = 2$, $\xi_1 = 2$, $\xi_2 = 3$. Note that these values are taken directly from Alur and Dill (1994). A real-time system designer can choose other values for these parameters. The GRC does not put any restrictions on how long a train can take to pass the gate crossing (theoretically speaking, a train can even stop at the gate and stay there indefinitely). To disallow such behaviors, we put an upper bound on the maximum time a train should take to exit the crossing. We introduce a constant σ, which is the maximum time in which an *exit* should appear since an *approach* was seen. For GRC, $\sigma = \infty$; following (Alur and Dill, 1994) we set $\sigma = 5$. The behavior of *track* is specified by the CLP(R) rules in Table 4.5.

The first argument of *track/9* is the track number. The second argument is current state of the track. The third argument is one of the events triggering an action explained above. The fourth argument is the new state that results. `W` represents the wall clock time. As we mentioned before there could be two trains at the same time in one track, i.e., one train in the crossing area and another one approaching the crossing. Therefore two clocks in the track automaton are needed to handle this situation. `Ti1` and `Ti2` are used to remember the last wall clock time these clocks were reset, while `To1` and `To2` are used to pass on these clock's values to the next transition.

The behavior of *controller* and *gate* can be specified similarly; the CLP(R) realization of *controller* and *gate* is represented in Table 4.6.

Once the components of GRC are realized in logic programming, the coinductive predicate `auto/8`, presented in Table 4.7, can be called for composing them. `auto/8` takes as input (i) the current states of controller, and gate; (ii) the list of events; (iii) the current wall clock time; (iv) the last reset time for controller's clock (CC), and gate's clock (CG); and, (v) the number of tracks in the system. The `initialize_tracks/2` predicate is used to set all the tracks in the system to the initial status in which all the tracks are in state `s0` and

Table 4.5. The Logic Programming Realization of *track*

```
track(Trk, s0, approach, s1, W, Ti1, Ti2, To1, To2) :-
      {To1 = W, To2 = W}.

track(Trk, s1, in,       s2, W, Ti1, Ti2, To1, To2) :-
      {W - Ti1 > 2, W - Ti1 < 3, To1 = Ti1, To2 = Ti2}.

track(Trk, s2, approach, s3, W, Ti1, Ti2, To1, To2) :-
      {To1 = Ti1, To2 = W}.

track(Trk, s3, exit,     s4, W, Ti1, Ti2, To1, To2) :-
      {W - Ti1 < 5, To1 = Ti1, To2 = Ti2}.

track(Trk, s4, in,       s2, W, Ti1, Ti2, To1, To2) :-
      {W - Ti2 > 2, W - Ti2 < 3, To1 = Ti1, To2 = Ti2}.

track(Trk, s2, exit,     s0, W, Ti1, Ti2, To1, To2) :-
                  {W - Ti2 < 5, To1 = Ti1, To2 = Ti2}.

track(_,    X, lower,     X, W, Ti1, Ti2, Ti1, Ti2).
track(_,    X, down,      X, W, Ti1, Ti2, Ti1, Ti2).
track(_,    X, raise,     X, W, Ti1, Ti2, Ti1, Ti2).
track(_,    X, up,        X, W, Ti1, Ti2, Ti1, Ti2).
```

their clocks are set to 0. The number of tracks in the system is given as an input to this predicate, and the output is the list of initial states of the form `s(0, 0, s0)`. The status of each track is specified by a structure of the form `s(time1, time2, state)`; `time1` and `time2` indicate the last time the track's clocks were reset, and `state` indicates the current state of the track automaton. The status of all tracks is specified as a list `TrkSts`. `Trains` is a list (initialized to empty list) containing all the trains currently in the system, in which each train is identified by the number of the track it is in. For example if `Trains` gets bound to [2, 1, 3, 2], it means that there are currently four trains in the system: the first train is in track 2; the second train is in track 1; the third train is in track 3; and, yet another train is in track 2. On receiving *approach* and *exit* signals on a track, the track number is added to

Table 4.6. The Logic Programming Realization of *Controller* and *Gate*

```
contr(s0, approach, s1, W, Ti, To, Ci, Co) :- Co is Ci + 1, {To = W}.
contr(s1, approach, s1, W, Ti, To, Ci, Co) :- Co is Ci + 1, {To = Ti}.
contr(s1, lower,    s2, W, Ti, To, Ci, Ci) :- {W - Ti = 1, To = Ti}.
contr(s2, approach, s2, W, Ti, To, Ci, Co) :- Co is Ci + 1, {To = Ti}.
contr(s3, approach, s2, W, Ti, To, Ci, Co) :- Co is Ci + 1, {To = Ti}.
contr(s3, raise,    s0, W, Ti, To, Ci, Ci) :- {W - Ti < 1, To = Ti}.
contr(s2, exit,     s3, W, Ti, To, Ci, Co) :-
                                   Ci = 1, Co is Ci - 1, {To = W}.
contr(s2, exit,     s2, W, Ti, To, Ci, Co) :-
                                   Ci > 1, Co is Ci - 1, {To = Ti}.
contr(X,  in,       X,  W, Ti, Ti, Ci, Ci).
contr(X,  up,       X,  W, Ti, Ti, Ci, Ci).
contr(X,  down,     X,  W, Ti, Ti, Ci, Ci).

gate(s0, lower,     s1, W, Ti, To) :- {To = W}.
gate(s1, down ,     s2, W, Ti, To) :- {To = Ti, W - Ti < 1}.
gate(s2, raise,     s3, W, Ti, To) :- {To = W}.
gate(s3, up,        s0, W, Ti, To) :- {To = Ti, W - Ti < 2}.
gate(s3, lower,     s1, W, Ti, To) :- {To = W}.
gate(X, approach,   X,  W, Ti, Ti).
gate(X, in,         X,  W, Ti, Ti).
gate(X, exit,       X,  W, Ti, Ti).
```

the end of the list `Trains`, and the first occurrence of that track number is removed from the list `Trains` respectively. Therefore, our framework can handle situations where *approaches* and *exits* of various trains on various tracks may be inter-mixed: for example, a long slow train in one track that approaches first, may be overtaken by a fast short train in another track that approaches later but exits first. Coinductive `auto/8` calls *controller*, *gate*, and *track* repeatedly, advances the clock after each call, and updates the status of the tracks through a call to `update/4` until the coinductive termination is reached. `auto/8` generates its output as a list of `(X, Track, W)` triples where `X` is an event, `Track` is the track number in which the event happened, and `W` is the time at which the event occurred.

Table 4.7. Logic Programming Realization of GRC

```
auto(SC, SG, Events, W, CC, CG, NoTrks, Timed_Events) :-
  initialize_tracks(NoTrks, TrkSts),
  create_list(NoTrks, Trks),
  auto(SC, SG, Events, W, CC, CG, Trks, TrkSts, [], 0, Timed_Events).

auto(SC, SG, [X|E], W, CC, CG, Trks, TrkSts, Trains, Ci, [(X,Trk,W)|TE]):-
  contr(SC, X, SCo, W, CC, CCo, Ci, Co),
  gate( SG, X, SGo, W, CG, CGo),
  {W2 > W},
  ((X = approach, select(Trk, Trks, _)
  ;(X = in; X = exit), member(Trk, Trains))->
     nthElement(Trk, TrkSts, K),
     arg(1, K, CT1),
     arg(2, K, CT2),
     arg(3, K, ST),
     track(Trk, ST, X, STo, W, CT1, CT2, CTo1, CTo2),
     update(Trk, TrkSts, s(CTo1, CTo2, STo), NewTrkSts),
     ( X = approach -> add_to_list(Trk, Trains, NewTrains)
     ;(X = exit ->    delete_first(Trk, Trains, NewTrains)
     ; NewTrains = Trains))
  ; NewTrains = Trains,
    NewTrkSts = TrkSts),
  auto(SCo, SGo, E, W2, CCo, CGo, Trks, NewTrkSts, NewTrains, Co, TE).

% create_list/2 is used to create the list of tracks in the system,
  where each track is identified by an integer between one and the
  number of tracks in the system.
```

Table 4.8. Verifying Safety and Utility

```
unsafe(N) :-
             auto(s0, s0, X, 0, 0, 0, N, R),
             append(C, [(in, _, _)| D], R),
             append(A, [(up, _, _)| B], C),
             not_member((down, _, _), B).

% not_member/2 takes an element X, and a list L, and succeeds if X is not
% a member of L and fails otherwise.

unutilized(N) :-
                 auto(s0, s0, X, 0, 0, 0, N, R),
                 append(A, [(down, _, _)| B], R),
                 find_first_up(B, C),
                 not_member((in, _, _), C).

% find_first_up/2 returns all the signals after down up to signal up.
```

Given the logic programming definitions of *controller*, *track*, and *gate* and the *auto* routine, one can check if a given sequence of timed events is legal or not, i.e., use the logic program as a simulator. One can also generate a sample sequence of timed events accepted by the system (note that in this case a CLP(R) system will output timed strings, where time-stamps of terminal symbols in the output string will be represented as variables; constraints that these time-stamps must satisfy will be output as well).

4.3.2 Verifying Safety and Utility Properties

Properties of interests are verified as follows: given a property `Q` to be verified, we specify its negation as a logic program, with top level predicate `notQ`. If the query `notQ` fails w.r.t. the logic program that models the system, the property `Q` holds. If the query `notQ` succeeds, the answer provides a counter example to why the property `Q` does not hold.

To prove the *safety* property, we define the `unsafe/1` predicate which looks for an accepting string that contains an `in` signal a little after an `up` signal and with no intervening `down` signal, i.e., we look for any possibility that a train is in the crossing area before the gate goes down, with the gate being up initially. `unsafe/1` is parameterized on the number of tracks in the system. The call to this predicate for any values for `N` fails which proves the safety of our system. The logic programming corresponding to verifying safety is presented in Table 4.8. Similarly we can check the utility property using the `unutilized/1` predicate, presented in Table 4.8. As we mentioned before, the *utility* property stipulates that the gate must be up (or going up) when there is no train in the crossing area. Equivalently, whenever the gate goes down a train must enter the crossing. `unutilized/1` looks for possibility of situations in which the gate is down without any train being in the crossing area. If a call to `unutilized/1` fails we know that the utility property is satisfied.

Table 4.9. safety and utility verification times

Number of tracks	safety	utility
1	0.006	0.006
2	0.065	0.072
3	0.6	0.587
4	5.666	5.634
5	60.013	60.430
6	426.300	453.544

Note that as the number of tracks in the system increases, the number of combinations in the system increases leading to an increase in the size of the state space. The execution time for verifying safety and utility properties therefore also increases. To keep this execution time down, and not explore irrelevant state space, we fold the negated properties into the `auto/8` predicate itself. Use of logic programming is again helpful here, where the negated property can be called before the call to `auto/8` in both `unsafe/1` and `unutilized/1`. Appropriate *delay declarations* must be included to ensure that the calls corresponding to the negated property are invoked only when appropriate bindings have been established by the driver. Table 4.9 shows the verification time in seconds for the safety and the utility properties for a system with up to 6 tracks (an average of five run time is taken). Our results show that our LP-based method is a practical method to verify complex real-time systems.

Table 4.10. distance/2

```
distance(D1, D2):-
  auto(s0, s0, X, 0, 0, 0, 1, R),
  append([(approach, 1, T1)| B],
  [(approach, 1, T2)| _], R),
  not_member((approach, 1, _), B),
  {T2 - T1 > D1, T2 -T1 < D2, D1 > 0, D2 > 0}.
```

Other interesting properties of the system can also be verified using appropriate queries. For example one can compute the minimum time distance between two consecutive trains (i.e., two consecutive *approach* signals) in one track through a call to a simple predicate `distance/2`, presented in Table 4.10. The solution produced by asking the query `?- distance(D1, D2)` for a system with one track is `D2 > 2`, and `D1 < 5`, which indicates the range of this minimum for the system to operate safely.

4.4 Conclusions and Related Work

In this chapter we showed how co-CLP(R) can be used for realizing implementations of pushdown timed automata and timed automata. We showed how this co-CLP(R)-based realization of PTA and timed automata can be used to elegantly and faithfully model the generalized railroad crossing problem as well as verify its safety and utility properties. Our modeling of the GRC based on PTA, and coinductive CLP(R) is simpler and more elegant than other approaches. In addition, unlike other realizations of timed-systems that discretize time, we treat time as a continuous quantity, and do not discretize it.

The GRC problem has been posed as a canonical problem so that various approaches to modeling and verifying real-time systems can be compared. Because of the difficulty of representing time faithfully (most approaches will discretize time), few solutions have been proposed for the GRC problem. The most notable is that of Puchol (Puchol, 1995), which is based on the ESTEREL programming language. In Puchol's work, time is discretized and thus is not faithful to the original problem. In contrast, our solution treats time as continuous. Verifying safety properties of the system in Puchol's approach is extremely complex: this complexity is such that one cannot be sure if the verification process itself is trustworthy. In our approach, in contrast, safety properties as well as other properties can be verified easily by posing simple queries.

UPPAAL has been also proposed as a toolbox for verification of real-time systems (Behrmann et al., 2004). In fact, a model for the simple 1-track Train-Gate example is distributed with UPPAAL. UPPAAL is based on a timed automata formalism and was not designed to handle PTA directly. However, UPPAAL allows arbitrary C-code to be executed during transitions. The inclusion of this facility can be used to model PTA; however, users have to be careful as one has to be wary of using low-level C-code in any verification process. In contrast, in our approach, the modeling of all operations of PTA is done directly at the higher level of logic programming.

To conclude, a combination of constraints over reals, coinduction, and logic programming provides an expressive, and easy-to-use formalism for modeling and analyzing complex real-time systems and CPS modeled as timed automata and PTA. In fact, our framework is a general framework that can be applied to complex systems to handle not only time but also other continuous quantities. The LP based approach is simpler and more elegant than other approaches that have been proposed for this purpose.

CHAPTER 5
TIMED GRAMMARS

Introduction

In the previous chapter we considered pushdown timed automata and presented a logic-based approach for modeling them. Pushdown timed automata are extension of timed automata with stacks. Since timed automata are equivalent to timed regular languages, it seems natural to think of PTA being equivalent to timed context-free languages. Regular expressions are unsuitable for many complex (and useful) applications; in many situations one needs context-free languages. For real-time systems this means that timed regular languages may not be powerful enough, and one has to resort to timed context-free languages.

In this chapter we propose timed grammars as a simple and natural method for describing timed languages. Timed grammars describe words that have real-time constraints placed on the times at which the words' symbols appear. We extend the concept of context-free grammars (CFGs) to timed context-free grammars and timed context-free ω-grammars. Informally, a timed context-free grammar is obtained by associating clock constraints and clock resets with terminal and non-terminal symbols of the productions of a CFG. The timed language accepted by a timed CFG contains those strings that are accepted by the underlying untimed grammar but which also satisfy the timing constraints imposed by the associated clock constraints. The language accepted by a timed ω-CFG (ω-TCFG) contains timed strings that are infinite in size. Such languages are useful for modeling complex CPS including real-time systems that run forever.

First, we introduce timed grammars and illustrate them through examples. We show how DCGs together with coinductive CLP(R) can be used to develop an effective and elegant method for parsing the languages generated by them. To illustrate this, we show how the *controller* component of the *generalized railroad crossing problem* (Heitmeyer and Lynch, 1994) can be specified naturally using timed grammars.

Developing the notion of timed grammars and using them for elegantly modeling complex real-time systems is the main subject of this chapter. Our novel contributions also include developing practical and faithful realization of timed grammars as coinductive constraint logic programs over reals. What is noteworthy in our realization of timed grammars is that, in contrast to other approaches, we do not discretize time. Rather, time is treated as a continuous quantity, with relationship between various time instances modeled as constraints over real numbers.

5.1 Timed Context-free ω-Grammars

A timed grammar is a grammar extended with real-valued variables.[1] These variables model the clocks in the grammar. All clocks advance at the same rate (identical to the rate at which a wall clock advances). Clock constraints are used to restrict the kind of strings generated by the underlying untimed grammar. Clocks may be reset to zero when a particular symbol of the language is seen. Informally, timed grammars are collections of production rules in which clock expressions (clock constraints and resets) can appear after both the terminal and non-terminal symbols on the right hand side. The timed language accepted by a timed grammar consists of strings that can be derived from rules of the grammar and that satisfy the clock constraints appearing in that grammar. Since CFGs are suitable for describing a large class of complex applications, we consider timed CFGs only (though, in general, one could have timed versions of context-sensitive grammars, as well as timed Turing machines). We use a timed CFG to describe a timed language by generating each string of that language in a manner similar to a CFG. Informally, during the derivation, the right hand side of a rule can be substituted for a non-terminal symbol only if the timing constraints accompanying that non-terminal symbol are satisfied. Similarly, a terminal symbol cannot be generated if

[1]Grammars extended with real-valued variables can be used to model other cyber-physical phenomenon, however, in this chapter our focus is on real-time systems.

its accompanying clock constraint is violated. Timed context-free grammars extend CFGs with:

- A set of *clocks*, which may be reset to zero. Clock names are global to all the production rules, i.e., all occurrences of a clock name `c` refer to the same clock.
- *Clock resets*, which are written within curly braces and can appear after a terminal or a non-terminal symbol on the right hand side of a production rule. Resetting clock `c` after a terminal symbol `a`, denoted by $\mathtt{a}\{\mathtt{c} := \mathtt{0}\}$, is used to remember the time at which `a` has been seen; while resetting the clock after a non-terminal symbol `B` is used to remember the time at which the *last* terminal symbol in the string that is reduced to `B` has been seen.
- *Clock constraints*, which are put in the timed grammar in exactly the same manner as *clock resets*, i.e., within curly braces; however, they are used to indicate the timing constraints between the time-stamps of the various symbols that appear in an accepted string. For example $\mathtt{a}\{\mathtt{c} < \mathtt{2}\}$ indicates that the symbol `a` must appear within two units of time since the clock `c` was reset.

Each *terminal* as well as *non-terminal* symbol appearing on the right hand side of a production rule maybe followed by curly braces that enclose a non-empty sequence of *clock resets* and *clock constraints*. Before we formally define timed grammars, let us consider some examples.

Example 5.1.1. *The following grammar describes a language in which sequences of* `a`*'s are followed by a final symbol* `b`*, which must appear within 5 units of time from the last symbol* `a`*.*

$S \rightarrow a \ \{c := 0\} \ S$

$S \rightarrow b \ \{c < 5\}$

Note that in this timed grammar `c` is a clock. Note also that, for example, in the first production of this grammar, the set of clock constraints associated with `S` is empty, and is therefore omitted.

Example 5.1.2. *With a slight change in the timed grammar in the previous example we can capture a language in which sequences of* `a`*'s are followed by a final symbol* `b` *that appears within 5 units of time from the* first *symbol* `a`*. The timed grammar for this timed language is as follows.*

$S \rightarrow a \ \{c := 0\} \ R$

$R \rightarrow a \ R$

$R \rightarrow b \ \{c < 5\}$

Note the difference in how the clocks are reset in these two examples. The clock `c` in Example 5.1.1 is reset on every occurrence of the symbol `a`; while in Example 5.1.2 the clock `c` is reset only on the first occurrence of symbol `a`.

Definition 5.1.1. *A timed context-free grammar is a 6-tuple $G = \langle V, T, C, E, R, S \rangle$, where*

- *V is a finite set of non-terminal symbols;*
- *T is a finite set of terminal symbols, disjoint from V, which is the alphabet of the language defined by the grammar;*
- *C is a finite set of clock identifiers;*
- *E is a set of clock expressions over C (clock constraints and clock resets);*
- *R is a finite set of productions of the form $A \rightarrow (a\delta)^*$, where $A \in V$, $a \in (V \cup T)$, and δ denotes a (possibly empty) collection of clock expressions from E (* denotes Kleene closure);*
- *$S \in V$ is a special symbol called the start symbol.*

The set E of clock expressions is limited to expressions of the form $\{c := 0\}$ and $\{c \sim x\}$, where c is a free variable, x is a constant, and $\sim \in \{=, <, \leq, >, \geq\}$.

Definition 5.1.2. *A timed language is a set of timed strings. A timed string is a sequence of pairs of the form $(\mathtt{a}, \mathtt{t_a})$ where $\mathtt{a}$ is a symbol from the alphabet, and $\mathtt{t_a}$ is the time at which the symbol $\mathtt{a}$ was seen (by time we mean wall clock or physical time). If $(\mathtt{a}, \mathtt{t_a})$ is immediately followed by $(\mathtt{b}, \mathtt{t_b})$ in a timed string, then $\mathtt{t_b} > \mathtt{t_a}$, i.e., two or more symbols cannot appear at the same instant.*

We now formally define the language generated by a timed context-free grammar $G = \langle V, T, C, E, R, S \rangle$. Note that because time flows linearly we only consider left to right *derivations*. Note also that because clocks may reset during a derivation, the reset times have to be recorded as part of the state. We could carry this state locally with each step of the derivation; however, to keep the exposition below simple we maintain this state as a global entity, which is accessed during each step of the derivation. For each clock `c`, `reset(c)` denotes the wall clock time at which clock `c` was last reset. The wall clock is treated as a global variable whose value can be read at any time. We consider two cases, one where we reduce using a production that has a terminal symbol occurring in the leftmost position on its RHS, and the other where this production has a non-terminal symbol as the leftmost symbol in the RHS.

Case I: If $A \rightarrow a\sigma B$ is a production of R, where $a \in T$, σ is set of clock expressions (possibly empty) from E, and B is in $(V \cup T \cup E)^*$, then $A\gamma F$ *yields* $(a, t_a)B\gamma F$, written $A\gamma F \underset{G}{\Rightarrow} (a, t_a)B\gamma F$, where F is in $(V \cup T \cup E)^*$, γ is set of clock expressions (possibly empty) from E associated with non-terminal symbol A, and for each clock expression $k \in \sigma$:

- if $k = \{c := 0\}$, then $reset(c) = t_a$, i.e., we record that the clock c was reset at wall clock time t_a.
- if $k = \{c \sim x\}$, then $t_a - reset(c) \sim x$ must hold. If this clock constraint does not hold, then the derivation fails.

Case II: If $A \rightarrow D\sigma B$ is a production of R, where $D \in V, \sigma$ is set of clock expressions (possibly empty) from E, and B is in $(V \cup T \cup E)^*$, then $A\gamma F \underset{G}{\Rightarrow} D\sigma B\gamma F$, where F is in $(V \cup T \cup E)^*$, and γ is set of clock expressions (possibly empty) from E associated with non-terminal symbol A.

If two sets of clock expressions, γ and σ, appear next to each other during a derivation, they are replaced by $\gamma \cup \sigma$.

Definition 5.1.3. *Suppose that $\alpha_1, \alpha_2, \ldots, \alpha_m$ are strings in $(V \cup T \cup E \cup (T \times \mathbf{R}^+))^*$, $m \geq 1$, and*

$$\alpha_1 \underset{G}{\Rightarrow} \alpha_2,\ \alpha_2 \underset{G}{\Rightarrow} \alpha_3,\ \ldots,\ \alpha_{m-1} \underset{G}{\Rightarrow} \alpha_m.$$

Then we say that $\alpha_1 \overset{*}{\underset{G}{\Rightarrow}} \alpha_m$, or α_1 derives α_m in grammar G. Alternatively, $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ if β follows from α by application of zero or more productions of R. Note that $\alpha \overset{*}{\underset{G}{\Rightarrow}} \alpha$ for each string α. If it is clear which grammar G is involved, we use $\Rightarrow$ for $\underset{G}{\Rightarrow}$ and $\overset{*}{\Rightarrow}$ for $\overset{*}{\underset{G}{\Rightarrow}}$.

Definition 5.1.4. *The language generated by grammar G [denoted by $L(G)$], with the starting symbol S is*

$$\{w \mid w = (w_1, t_1), (w_2, t_2), \ldots, (w_n, t_n) \text{ where } w_i \in T, t_i \in \mathbf{R}^+, t_1 < t_2 \cdots < t_n, \text{and } S \overset{*}{\underset{G}{\Rightarrow}} w\}.$$

That is, a timed string is in $L(G)$ if:

1. *The timed string (timed word) consists of a sequence of pairs; the first element of each pair is a terminal symbol and the second is a real number.*

2. *The timed string can be derived from S and satisfies the time constraints imposed by the grammar.*

We call L a timed context-free language (TCFL) for some timed CFG G, if $L = L(G)$.

Definition 5.1.5. *Timed context-free ω-grammars (ω-TCFG) are CFGs with co-recursive grammar rules (i.e., recursive rules with no base cases). ω-TCFGs generate TCFLs with infinite sized words.*

Example 5.1.3. *Consider a language in which each sequence of* `a`*'s is followed by a sequence of an equal number of* `b`*'s, with each accepted string having at least two* `a`*'s and two* `b`*'s. For each pair of equinumerous sequences of* `a`*'s and* `b`*'s, the* first *symbol* `b` *must appear within 5 units of time from the* first *symbol* `a` *and the* final *symbol* `b` *must appear within 20 units of time from the* first *symbol* `a`*. The grammar annotated with clock expressions is shown below:* `c` *is a clock which is reset when the first symbol* `a` *is seen.*

1. $S \rightarrow R\ \ S$
2. $R \rightarrow a\ \ \{c := 0\}\ \ T\ \ b\ \ \{c < 20\}$
3. $T \rightarrow a\ \ T\ \ b$
4. $T \rightarrow a\ \ b\ \ \{c < 5\}$

Note that in the grammar above, the first rule is coinductive (i.e., a recursive rule with no base case). Thus, this grammar is a timed ω-grammar. Next, we give an example of a derivation of a timed word accepted by a timed grammar.

Example 5.1.4. *Consider the timed ω-grammar of Example 5.1.3. Consider a timed word*

$$(a, 2), (a, 4), (b, 5), (b, 10), \ldots$$

Below we give the initial segment of the derivation of this timed word. The global state will record the time at which clock `c` *is reset;* `c` *will be set to the value 2 when the first symbol* `a` *is seen.*

$$\begin{aligned}
S &\Rightarrow R\ \ S \\
&\Rightarrow a\ \ \{c := 0\}\ \ T\ \ b\ \ \{c < 20\}\ \ S \\
&\Rightarrow (a, 2)\ \ T\ \ b\ \ \{c < 20\}\ \ S \\
&\Rightarrow (a, 2)\ \ a\ \ b\ \ \{c < 5\}\ \ b\ \ \{c < 20\}\ \ S \\
&\Rightarrow (a, 2)\ \ (a, 4)\ \ b\ \ \{c < 5\}\ \ b\ \ \{c < 20\}\ \ S \\
&\Rightarrow (a, 2)\ \ (a, 4)\ \ (b, 5)\ \ b\ \ \{c < 20\}\ \ S \\
&\Rightarrow (a, 2)\ \ (a, 4)\ \ (b, 5)\ \ (b, 10)\ \ S \ldots
\end{aligned}$$

As we mentioned before pushdown timed automata are equivalent to timed context-free ω-grammars. The proof is straightforward extension of the proof that shows the equivalence of pushdown automata and context-free grammars, found in any automata theory text book, e.g., (Sipser, 1996), and it is not included here; but shown in Appendix C.

5.2 Modeling ω-TCFGs with Coinductive CLP(R)

To model and reason about ω-TCFGs we should be able to handle the fact that: (i) the underlying language is context-free, (ii) accepted strings are infinite, and (iii) clock constraints are posed over continuously flowing time. All three aspects can be elegantly handled within LP. It is well known that the definite clause grammar (DCG) facility of Prolog allows one to obtain parsers for context-free grammars (and even for context-sensitive grammars) with minimal effort. By extending LP with coinduction, one can develop language processors that recognize infinite strings. DCGs extended with coinduction can act as recognizers for ω-grammars. Further, incorporation of coinduction and CLP(R) into the DCG allows modeling of time aspects of ω-TCFGs. Once an ω-TCFG is modeled as a coinductive CLP(R) program, it can be used to (i) check whether a particular timed string will be accepted or not; and, (ii) systematically generate all possible timed strings that can be accepted (note that a CLP(R) system will represent time-stamps of terminal symbols as variables in the output, with constraints imposed on them). The LP realization of the system based on coinduction and CLP(R) can also be used to verify system properties by posing appropriate queries.

We model ω-TCFGs as sets of DCG rules augmented with coinduction and CLP(R). This coinductive constraint logic program acts as a parser for the ω-TCFL recognized by the ω-TCFG. The general method of this system is outlined next.[2] The method takes an ω-TCFG as input and generates a parser as a collection of DCG rules (one rule per production in the ω-TCFG), where each rule is extended with clock expressions. For simplicity of presentation the method is explained for a timed grammar with one clock, but it can handle any number

[2]The details of the method can be found in `http://www.utdallas.edu/~nxs048000/tGrammar.yap`.

of clocks in a similar manner. We assume `c` is the clock; `Ti` and `To` are used to remember the last (wall clock) time this clock was reset and to pass on this clock's value to the next step in the derivation respectively. `W` and `Wn` are used to represent the current wall clock time and the new wall clock time after each step in the derivation respectively. The method replaces every component of the timed grammar with its corresponding timed component as follows.

- A *non-terminal* symbol `s` is replaced with predicate `s(W, Ti, Wn, To)`;
- A *terminal* symbol `a` is replaced with $[(\mathtt{a}, \mathtt{W_a})]$, where $\mathtt{W_a}$ is the wall clock time at which the symbol `a` appeared;
- A *clock constraint* of the form $\{\mathtt{c} \sim \mathtt{x}\}$, where $\sim \in \{=, <, \leq, >, \geq\}$ is replaced with $\{\{\mathtt{W} - \mathtt{Ti} \sim \mathtt{x}\}\}$;
- A *clock reset* of the form $\{\mathtt{c} := \mathtt{0}\}$ is replaced with $\{\{\mathtt{Ti} = \mathtt{W}\}\}$ or $\{\{\mathtt{To} = \mathtt{W}\}\}$ (depending on where it appears in the production).

Note that everything within curly braces in DCG's is treated as a standard Prolog code, i.e., curly braces would be simply dropped and the code within them would be executed. Constraint solving is done using the CLP(R) system (after dropping a pair of braces; since it is the convention in most CLP(R) systems to put constraints inside a pair of curly braces).

For each production in the timed grammar, the method replaces each component (starting with the left hand side) with its corresponding timed component as explained above, advances the clock if necessary, passes on the wall clock time and last reset time to the next component, and repeats this step until the end of production rule is reached. If `W` represents the wall clock time, then time is advanced by posting the constraint $\mathtt{W}' > \mathtt{W}$ where $\mathtt{W}'$ represents the current wall clock time. Advancing the clock is necessary since the underlying assumption is that multiple symbols of the timed string are not seen at the same instant. if $(\mathtt{a}, \mathtt{W_a})$ is immediately followed by $(\mathtt{b}, \mathtt{W_b})$ in a timed string, then $\mathtt{W_b} > \mathtt{W_a}$. Thus the wall clock should be advanced after each symbol in the timed string.

Table 5.1. The parser for the language of the pushdown timed automaton of Figure 2.2, implemented in coinductive CLP

```
s(W, Ti, Wn, To) -->
  r(W, Ti, W1, To1), {{W2 > W1}}, s(W2, To1, Wn, To).

r(W, Ti, Wn, To) -->
  [(a, W)], {{Ti = W, W1 > W}}, t(W1, Ti, W2, To),
  {{Wn > W2}}, [(b, Wn)], {{Wn - To < 20}}.

t(W, Ti, Wn, To) -->
  [(a, W)], {{W1 > W}}, t(W1, Ti, W2, To), {{Wn > W2}}, [(b, Wn)].

t(W, Ti, Wn, To) -->
  [(a, W)], {{Wn > W}}, [(b, Wn)], {{Wn - Ti < 5, To = Ti}}.
```

To illustrate the process, we next describe the logic programming rendering of the timed ω-grammar shown in Example 5.1.3 in Section 5.1, generated by our system.

Note that the predicates `s`, `r`, and `t` are defined as DCG rules; therefore, two arguments (for the difference lists) will be added to each of them by a Prolog compiler. The first explicit argument of these predicates is the wall clock time; `Wn` is the new wall clock time after each predicate call. The pair of arguments, `Ti` and `To`, represent the clock `c` of the timed grammar. Note also that the coinductive termination of `s/6` will depend only on the two arguments (for the difference lists) that are added by the Prolog compiler, i.e., the wall-clock time and other arguments will be ignored when checking if the `s/6` predicate is cyclical.

Given this program, one can pose queries to it to check if a timed string satisfies the timing constraints imposed in the timed grammar. Alternatively, one can generate possible legal timed strings (note that in this case a CLP(R) system will output timed strings, where time-stamps of terminal symbols in the output string will be represented as variables; constraints that these time-stamps must satisfy will be output as well). Finally, one can verify properties of this timed language (e.g., checking the simple property that all the a's are generated within 5 units of time, in any timed string that is accepted).

5.3 Timed ω-Grammar for GRC

In this section we show the timed context-free ω-grammar for the *controller* component of the GRC system, defined in Section 4.3.1, with two tracks. The behavior of the controller can be expressed graphically via the automaton in Figure 5.1.

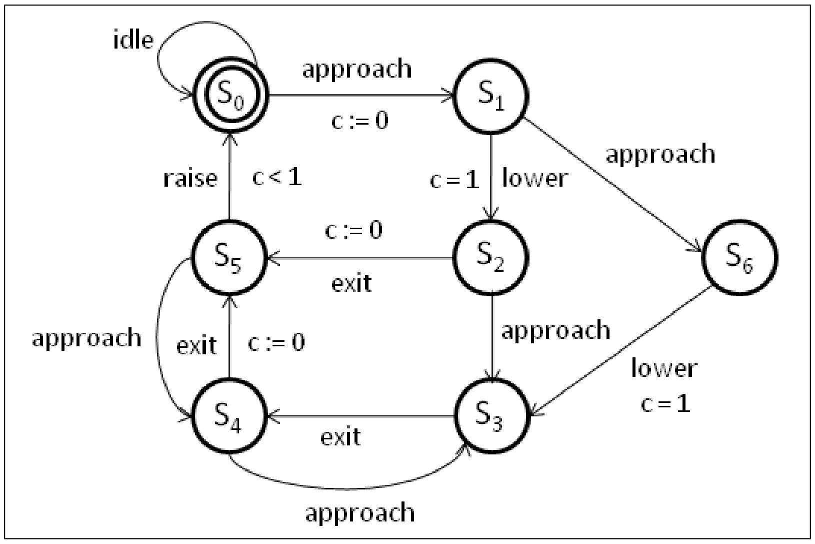

Figure 5.1. The controller for the GRC with two tracks

The context-free ω-grammar describing the behavior of *controller*, equivalent to the automaton in Figure 5.1, is presented as follows.

$C \rightarrow approach\ \{c := 0\}\ L\ exit\ \{c := 0\}\ raise\ \{c < 1\}\ C$

$C \rightarrow approach\ \{c := 0\}\ L\ N\ exit\ \{c := 0\}\ raise\ \{c < 1\}\ C$

$L \rightarrow lower\ \{c < 1\}$

$L \rightarrow approach\ lower\ \{c < 1\}\ exit$

$N \rightarrow approach\ exit$

$N \rightarrow approach\ exit\ N$

$N \rightarrow exit\ approach$

$N \rightarrow exit\ approach\ N$

The logic programming rendering of this ω-TCFG is presented in Table 5.2.

Table 5.2. The Timed ω-Grammar for the *controller* of The GRC with Two Tracks, Implemented in Coinductive CLP(R)

```
c(W, Ti, Wn, To) -->
  [(approach, W)], {{Ti = W, W1 > W}}, l(W1, Ti, W2, To1), {{W3 > W2}},
  [(exit, W3)], {{To2 = W3, W4 > W3}}, [(raise, W4)],
  {{W4 - To2 <1, W5 > W4}}, c(W5, W5, Wn, To).

c(W, Ti, Wn, To) -->
  [(approach, W)], {{Ti = W, W1 > W}}, l(W1, Ti, W2, To1), {W3 > W2}},
  n(W3, To1, W4, To2), {{W5 > W4}}, [(exit, W5)], {{To3 = W5, T6 > W5}},
  [(raise, T6)], {{T6 - To3 < 1, T7 > T6}}, c(T7, T7, Wn, To).

l(W, Ti, Wn, To) -->
  [(lower, W)], {{W - Ti < 1, To = Ti, Wn = W}}.

l(W, Ti, Wn, To) -->
  [(approach, W)], {{W1 > W}}, [(lower, W1)], {{W1 - Ti < 1, Wn > W1}},
  [(exit, Wn)], {{To = Ti}}.

n(W, Ti, Wn, To) -->
  [(approach, W)], {{Wn > W}}, [(exit, Wn)], {{To = Ti}}.

n(W, Ti, Wn, To) -->
  [(approach, W)], {{W1 > W}}, [(exit, W1)], {{W2 > W1}},
  n(W2, Ti, Wn, To).

n(W, Ti, Wn, To) -->
  [(exit, W)], {{Wn > W}}, [(approach, Wn)], {{To = Ti}}.

n(W, Ti, Wn, To) -->
  [(exit, W)], {{W1 > W}}, [(approach, W1)], {{W2 > W1}},
  n(W2, Ti, Wn, To).
```

The ω-TCFGs for *gate* and *track* components of the GRC problem along with their corresponding logic programs can be generated in a similar manner.

5.4 Conclusions

In this chapter we proposed timed grammars as a simple and natural way of describing timed languages. We extended context-free grammars with clocks and clock expressions. The resulting grammars, called timed CFGs, are a means of describing complex languages consisting of timed words, where a timed word is a symbol from the alphabet of the language the grammar generates, paired with the time that symbol was seen. As a result, timed CFGs are suitable for specifying systems whose behavior can be described by recursive structures with time constraints. We have developed a system for generating parsers for timed context-free ω-grammars using the DCG facility of logic programming coupled with coinductive CLP(R).

CHAPTER 6
TIMED π-CALCULUS

6.1 Introduction

The π-calculus was introduced by Milner et al. (Milner et al., 1992) with the aim of modeling concurrent/mobile processes. The π-calculus provides a conceptual framework for describing systems whose components interact with each other. It contains an algebraic language for descriptions of processes in terms of the communication actions they can perform. Theoretically, the π-calculus can model mobility, concurrency and message exchange between processes as well as infinite computation (through the infinite replication operator '!').

In many cases, processes run on controllers that control physical devices; therefore, they have to deal with physical quantities such as time, distance, pressure, acceleration, etc. Examples include communicating controller systems in cars (Anti-lock Brake System, Cruise Controllers, Collision Avoidance, etc.), automated manufacturing, smart homes, etc. Properties of such systems, which are termed cyber-physical systems (Lee, 2008, Gupta, 2006b), cannot be fully expressed within π-calculus. In a real-time/cyber-physical system the correctness of the system's behavior depends not only on the tasks that the system is designed to perform, but also on the time instants at which these tasks are performed. Real-time systems can be described as systems that must respond to externally generated signals or inputs within specified time limits; the future behavior of such systems depends on the times at which the external signals are received. This class of systems includes many safety critical systems such as those found in robotics and flight control. A real-time system is often composed of multiple mobile components that are working concurrently within some time constraints. While π-calculus can handle mobility and concurrency, it is not equipped to model real-time systems or cyber-physical systems (CPS) (Lee, 2008, Gupta, 2006b) and support reasoning about their behavior related to time and other physical quantities.

Several extensions of π-calculus with time have been proposed to overcome this problem (Berger, 2004, Lee and Zic, 2002, Ciobanu and Prisacariu, 2006, Degano et al., 1996); all these approaches discretize time rather than represent it faithfully as a continuous quantity.

In this chapter we consider the extension of π-calculus with continuous time by adding finitely many real-valued clocks and assigning time-stamps to actions. The resulting formalism can be used for describing concurrent real-time systems and CPS and reasoning about their behaviors[1]. For simplicity, the behavior of a real-time system is understood as a sequence (finite or infinite) of timed events, not states.

We develop an executable operational semantics of π-calculus in logic programming (LP) in which concurrency is modeled by *coroutining*, and (rational) infinite computation by *coinduction* (Simon et al., 2007, Gupta et al., 2007). This operational semantics is extended with continuous real-time, which we model with *constraint logic programming over reals* (Jaffar and Maher, 1994). The executable operational semantics directly leads to an implementation of the timed π-calculus. Note that there is past work on developing LP-based executable operational semantics of the π-calculus (but not timed π-calculus) (Yang et al., 2004), but it is unable to model infinite processes and infinite replication. In our implementation we are using coinductive logic programming, a more recently developed concept (Simon et al., 2007, Gupta et al., 2007), which allows such modeling.

Thus, our contributions are twofold: (i) we extend the π-calculus with real-time clocks: in contrast to other extensions, in our work the notion of time and clocks is adopted directly from the well-understood formalism of timed automata; (ii) we show how an executable operational semantics of the π-calculus can be elegantly realized through logic programming extended with coinduction and coroutining: by adding constraint logic programming over

[1]While we only focus on extending the π-calculus with continuous time, our method serves as a model for extending the π-calculus with other continuous quantities. An instance of this, though not in the context of π-calculus, can be found in (Saeedloei and Gupta, 2011a).

reals to the mix we allow this executable operational semantics to faithfully capture real-time behavior. The executable semantics allows us to prove behavioral and timing properties of systems modeled as timed π-calculus processes.

6.2 Timed π-calculus

6.2.1 Design Decisions

The future behavior of a real-time system depends not only on the contents of a received input/message (externally generated signal), but also on the time-stamp of the message: for future transitions to take place the time-stamp should satisfy certain time constraints specified by the system. To meet this requirement, we introduce (local) *clocks* and *clock operations*. We augment the original messages of π-calculus with two arguments: the first argument is the time-stamp of the message, while the second argument is the local clock that is used to generate the time-stamp. As in π-calculus, timed π-calculus processes use names (including clock names) to interact, and pass names to one another. These processes are identical to processes in π-calculus except that they have access to clocks which they can manipulate. Note that all the clocks are local clocks; however, their scope is changed as they are sent among processes. This will become clear when we explain how clock passing is performed in Section 6.2.5

When a process outputs a message through a channel, it also sends the time-stamp of the message and the clock[2] that is used to generate the time-stamp. Thus, messages are represented by triples of the form $\langle m, t_m, c\rangle$, where m is the message, t_m is the time-stamp on m, and c is the clock that is used to generate t_m. Note that it is important for the process to send its clock that is used to generate the time-stamp of the message, because the time-stamp of the incoming message in conjunction with the clock received is used by the receiving process to reason about timing requirements of the system and delays.

[2]Later when writing the operational semantics of timed π-calculus, we assume that all processes have access to the same wall clock. However, in reality it might not be the case; therefore, extra information has to be sent in the form of clocks.

We assume an infinite set $\mathcal{N}$ of names (channel names and names passing through channels), an infinite set Γ of clock names (disjoint from $\mathcal{N}$) and an infinite set Θ of variables representing time-stamps (disjoint from $\mathcal{N}$ and Γ). We use x, y, z, u, v as variables ranging over names and c, d, e, f as variables ranging over clock names. We use t_x, t_y, t_z, t_u, t_v and sometimes t, t' to represent time-stamps. Inspired by the notion of name transmission in π-calculus, we can treat time-stamps and clocks just as other names and transmit them through/with channels. Just as channel transmission results in dynamic configuration of processes, clock and time-stamp transmission can result in dynamic temporal behavior of processes. Note that all the clocks advance at the same rate. Our notion of real-time and clocks is adopted directly from *timed automata* (Alur and Dill, 1994).

6.2.2 Clocks and Clock Operations

The syntax of the timed π-calculus relies on a set of real-valued clocks and also on clock operations. A clock can be set to zero simultaneously with any transition (transitions are defined formally in Section 6.2.4). At any instant, the reading of a clock is equal to the time that has elapsed since the last time the clock was reset. We only consider non-Zeno behaviors, that is, only a finite number of transitions can happen within a finite amount of time. We consider two types of clock operations: resetting a clock and checking satisfiability of a clock constraint. Clock resets, denoted by γ, are used to remember the time at which particular actions in the system have taken place. Clock constraints indicate the timing constraints between various actions that appear in the system. We allow two forms of constraints: the first form compares a clock value with a real valued constant; the second form compares the difference of a clock value and a time-stamp with a real valued constant. A clock constraint, denoted by δ, is defined by the following syntactic rules, in which c is a clock name in Γ, r is a constant in $\mathbf{R}^+$ and t is a time-stamp variable in Θ. ϵ represents an empty clock constraint.

$$\delta ::= (c \sim r)\delta \mid (c - t \sim r)\delta \mid (t - c \sim r)\delta \mid \epsilon$$

$$\sim ::= < \mid > \mid \leq \mid \geq \mid =$$

There are two ways to measure the passing of time while checking for a clock constraint. It can be measured and reasoned about against (i) the last time a clock was reset: e.g., a constraint $(c < 2)$ on sending message m indicates that m must be sent out within two units of time since the clock c was reset, or (ii) the last time a clock c was reset in conjunction with a time-stamp t of an arriving or a sent message. Note that in this case, the time-stamp t must be generated by clock c. For instance, suppose that a process P sends two consecutive messages that are two units of time apart; if the time-stamp of the first message, generated by clock c is t_1, then the expression $c - t_1 = 2$ can be used to express this constraint.

Definition 6.2.1. *(Alur and Dill, 1994) A* clock interpretation $\mathcal{I}$ *for a set* Γ *of clocks assigns a real value to each clock; that is, it is a mapping from* Γ *to* $\mathbf{R}^+$*. We say that a clock interpretation* $\mathcal{I}$ *for* Γ *satisfies a clock constraint* δ *over* Γ *iff* δ *evaluates to true using the values given by* $\mathcal{I}$*. For* $t \in \mathbf{R}$*,* $I + t$ *denotes the clock interpretation which maps every clock* c *to the value* $I(c) + t$*. For* $\gamma \subseteq \Gamma$*,* $[\gamma \mapsto t]I$ *denotes the clock interpretation for* Γ *which assigns* t *to each* $c \in \gamma$*, and agrees with* I *over the rest of the clocks.*

6.2.3 Syntax

The set of timed π-calculus processes is defined as follows (P, P', M and M' range over timed π-calculus processes):

$$M ::= \delta\gamma\bar{x}\langle y, t_y, c\rangle.P \;\big|\; \delta\gamma x(\langle y, t_y, c\rangle).P \;\big|\; \delta\gamma\tau.P \;\big|\; 0 \;\big|\; M + M'$$

$$P ::= M \;\big|\; P \mid P' \;\big|\; !P \;\big|\; (z)\; P \;\big|\; [x = y]\, P$$

We use $\delta\gamma\bar{x}\langle y, t_y, c\rangle.P$ to stand for a process that sends the name y, time-stamp of y t_y, and its clock c via the channel x, and evolves to P, if the clock constraint δ is satisfied by the current value of clocks at the time of transition[3]. γ is an expression of the form $(c_1 := 0) \ldots (c_n := 0)$ which specifies the clocks to be reset with this transition. For example the expression $(c < 2)(c := 0)\bar{x}\langle y, t_y, c\rangle.P$ indicates that $\langle y, t_y, c\rangle$ must be sent out on channel

[3]We assume that the checking of clock constraints can be done instantaneously, because each constraint is a simple comparison. In practice a real-time system designer should account for the time it takes to check these constraints.

x within two units of time since the clock c was last reset, and moreover the clock c is reset when it is sent.

The expression $\delta\gamma x(\langle y, t_y, c\rangle).P$ stands for a process which is waiting for a message on channel x. When a message arrives, the process will behave like $P\{z/y, t_z/t_y, d/c\}$ (see the definition of substitution below) where z is the contents of the message; t_z is the time-stamp of the message; and d is the clock of the sending process that is used to generate t_z. Note that the time-stamp of the incoming message should satisfy the clock constraint expressed by δ, otherwise the process will become inactive (an inactive process is represented by 0). Again, γ specifies the clocks to be reset on performing the transition.

The expression $\delta\gamma\tau.P$ stands for a process that takes an internal action and evolves to P, and in doing so resets the clocks specified by γ, if the clock constraint δ is satisfied; otherwise it becomes inactive.

Process 0 is *inaction*; it is a process that does nothing. The operators $+$ and $|$ are used for nondeterministic *choice* and *composition* of processes, just as in π-calculus (Milner et al., 1992). The *replication* $!P$, can be thought of as an infinite composition $P \mid P \mid \ldots$, just as in π-calculus Milner et al. (1992). The *restriction* $(z)P$, behaves as P with z local to P. If $z \in \mathcal{N}$, z cannot be used as a channel over which to communicate with other processes or the environment. If $z \in \Gamma$, z is a local clock in P. $[x = y]\,P$ evolves to P if x and y are the same name. Note that *match* can be used for comparing two names, including clock names.

We call $\bar{x}\langle y, t_y, c\rangle$, $x(\langle y, t_y, c\rangle)$ and τ *prefixes*; in particular, $\bar{x}\langle y, t_y, c\rangle$ is an *output prefix* and $x(\langle y, t_y, c\rangle)$ is an *input prefix.* Analogous to π-calculus, restriction and input-prefix in timed π-calculus are name-binding operators: restriction specifies the scope of a name, while input-prefix acts as a place-holder for a name which may be received as input. In each process of the form $\delta\gamma x(\langle y, t_y, c\rangle).P$ the occurrences of y and c are binding occurrences, and the scope of the occurrences is P. Similarly, in a process of the form $(y)P$ the occurrence of y is a binding occurrence, and the scope of the occurrence is P.

Definition 6.2.2. *(Milner et al., 1992) An occurrence of y in a process is said to be* free *if it does not lie within the scope of a binding occurrence of y. An occurrence of a name in a* *process is said to be* bound *if it is not free. The set of bound names of P, $bn(P)$, contains all names which occur bound in P. The set of names occurring free in P is denoted $fn(P)$. We write $n(P)$ for the set $fn(P) \cup bn(P)$ of names of P.*

Definition 6.2.3. *(Milner et al., 1992) A substitution is a function θ from a set of names $\mathcal{N}$ to $\mathcal{N}$. If $x_i\theta = y_i$ for all i with $1 \leq i \leq n$ (and $x\theta = x$ for all other names x), we write $\{y_1/x_1, \ldots, y_n/x_n\}$ for θ.*

The effect of applying a substitution θ to a process P is to replace each free occurrence of each name x in P by $x\theta$, with change of bound names to avoid name capture (to preserve the distinction of bound names from the free names). Substitution for clock names and time-stamps can be defined similarly.

Example 6.2.1. *The expression $x(\langle m, t_m, c\rangle).(c - t_m > 5)\bar{y}\langle n, t_n, c\rangle$ represents a process that is waiting for a message on channel x. The process upon receiving a message m with time-stamp t_m and its accompanying clock c on channel x, sends a message n with time-stamp t_n on channel y, with the delay of at least 5 units of time since the time-stamp of m. The process will use the clock c to choose a time t_n on c such that $c - t_m > 5$.*

Example 6.2.2. *Consider a system which is composed of two processes P and Q that run in parallel. Moreover, there is a clock c that can be accessed by both P and Q which should be reset before the parallel execution begins. This scenario can be expressed by the expression $(c)(c := 0)\tau.(P \mid Q)$.*

Note that clock operations precede the prefixes and are used to express the timing requirements of real-time systems as well as the delays. Since we are associating time-stamps with all names, including channel names, on outputting names we are able to also reason about the time interval at which a particular channel is used. Note also that empty clock operations can be omitted.

6.2.4 Operational Semantics

A transition in the timed π-calculus is of the form $P_{(I,w)} \xrightarrow{\langle\delta,\alpha,\gamma\rangle} P'_{(I',w')}$. This transition means that if δ is satisfied by the current interpretation I, P evolves into P' at time w', and in doing so performs the action α and resets the clocks specified by γ. After this transition the new interpretation I' is $[\gamma \mapsto 0](I + w' - w)$ and the time w' at which action is performed is recorded. Note that w is the time of the last action before this transition. With abuse of notation, we have used γ as a set of clocks to be reset. We call the triple $\langle\delta, \alpha, \gamma\rangle$ a timed action, in which action α is defined by the following syntactic rule:

$$\alpha ::= \bar{x}\langle y, t, c\rangle \mid x(\langle y, t, c\rangle) \mid \tau \mid \bar{x}\langle (y), t, c\rangle$$

The first action is the *free output action* $\bar{x}\langle y, t, c\rangle$. This action is used for sending a name y, time-stamp of y, t, and the clock that is used to generate t via channel x. The process that gives rise to this action can be of the form $(c)\bar{x}\langle y, t, c\rangle.P$. In this action x and y are free and c is bound[4].

The second action is the *input action* $x(\langle y, t, c\rangle)$. This action is used for receiving any name z with its time-stamp t_z, and a clock d via x. y, t and c will be replaced by z, t_z and d in the receiving process. In this action x is free, while y and c are bound.

The third action is the *silent action* τ, which is used to express performing an internal action. Silent actions can naturally arise from processes of the form $\tau.P$, or from communications within a process (e.g., rule COM in Table 6.2).

The last action is *bound output action.* The expression $\bar{x}\langle (y), t, c\rangle$ is used by a process P for sending a private name y (y is bound in P). Bound output actions arise from free output actions which carry names out of their scope. The process that gives rise to this action can be of the form: $(c)(y)\bar{x}\langle y, t, c\rangle.P$. In this actions x is free, y and c are bound [5].

[4]The reason this action is called a free output action is because y is free. As we mentioned earlier all clocks are local clocks; therefore, clock names are not free names.

[5]Even though clock c is bound in P (it is a local clock), we don't put c within parentheses, as all clock names in timed π-calculus are local names.

Intuitively, a system specified by the set of timed π-calculus processes starts with an empty interpretation. As a process starts its execution, the set of its clocks are added to the clock interpretation. The renaming of local clock names is performed to avoid name clashes in the interpretation. All the clocks of a process are mapped to 0 initially. As time advances the value of all clocks advances, reflecting the elapsed time. The behavior of a system can be understood as a finite or infinite sequence of timed events of the form $(\alpha_1, w_1), (\alpha_2, w_2), \ldots$.

The operational semantics of timed π-calculus, represented in Table 6.1 and Table 6.2 is expressed as a set of transition rules which are labeled by timed actions.We use $fn(\alpha)$ for set of free names of α, $bn(\alpha)$ for set of bound names of α, and $n(\alpha)$ for the union of $fn(\alpha)$ and $bn(\alpha)$. In the formal exposition, we idealize reality and assume that actions are instantaneous; e.g., if a process Q is ready to send a name $\langle z, t_z, c\rangle$ via channel x, then $x(\langle y, t_y, d\rangle).P$ performs the input action immediately and becomes $P\{z/y, t_z/t_y, c/d\}$ in doing so. However, these assumptions do not put any constraints on the actual systems that are modeled using this formalism.

Table 6.1. Timed π-calculus Transition Rules

$$\text{OUT}\ \frac{(\delta\gamma\bar{x}\langle y,t,c\rangle.P)_{(I,w)} \xrightarrow{\langle\delta,\bar{x}\langle y,t,c\rangle,\gamma\rangle} P_{(I',w')}}{((c)\delta\gamma\bar{x}\langle y,t,c\rangle.P)_{(I,w)} \xrightarrow{\langle\delta,\bar{x}\langle y,t,d\rangle,\gamma\rangle} P\{d/c\}_{(I',w')}}\ d \notin fn((c)P)$$

$$\text{TAU}\ \frac{}{(\delta\gamma\tau.P)_{(I,w)} \xrightarrow{\langle\delta,\tau,\gamma\rangle} P_{(I',w')}}$$

$$\text{INP}\ \frac{}{(\delta\gamma x(\langle z,t_z,c\rangle).P)_{(I,w)} \xrightarrow{\langle\delta\{d/c\},x(\langle y,t,d\rangle),\gamma\{d/c\}\rangle} (P\{y/z,t/t_z,d/c\})_{(I',w')}}\ y,d \notin fn((z)P)$$

$$\text{MAT}\ \frac{P_{(I,w)} \xrightarrow{\langle\delta,\alpha,\gamma\rangle} P'_{(I',w')}}{[x=x]P_{(I,w)} \xrightarrow{\langle\delta,\alpha,\gamma\rangle} P'_{(I',w')}} \quad \text{SUM}\ \frac{P_{(I,w)} \xrightarrow{\langle\delta,\alpha,\gamma\rangle} P'_{(I',w')}}{(P+Q)_{(I,w)} \xrightarrow{\langle\delta,\alpha,\gamma\rangle} P'_{(I',w')}}$$

$$\text{PAR}\ \frac{P_{(I,w)} \xrightarrow{\langle\delta,\alpha,\gamma\rangle} P'_{(I',w')}}{(P \mid Q)_{(I,w)} \xrightarrow{\langle\delta,\alpha,\gamma\rangle} (P' \mid Q)_{(I',w')}}\ bn(\alpha) \cap fn(Q) = \emptyset$$

Table 6.2. ...Timed π-calculus Transition Rules, Continued

$$\text{OPEN I}\ \frac{P_{(I,w)} \xrightarrow{\langle\delta,\bar{x}\langle y,t,c\rangle,\gamma\rangle} P'_{(I',w')}}{(y)(c)P_{(I,w)} \xrightarrow{\langle\delta,\bar{x}\langle(u),t,d\rangle,\gamma\rangle} P'\{u/y,d/c\}_{(I',w')}}\ y \neq x \& u, d \notin fn((y)(c)P')$$

$$\text{OPEN II}\ \frac{P_{(I,w)} \xrightarrow{\langle\delta,\bar{x}\langle y,t,c\rangle,\gamma\rangle} P'_{(I',w')}}{(z)(c)P_{(I,w)} \xrightarrow{\langle\delta,\bar{x}\langle y,t,d\rangle,\gamma\rangle} (z)P'\{d/c\}_{(I',w')}}\ z \neq x, z \neq y, d \notin fn((c)P')$$

$$\text{COM}\ \frac{P_{(I,w)} \xrightarrow{\langle\delta,\bar{x}\langle z,t,c\rangle,\gamma\rangle} P'_{(I',w')}\ Q_{(I,w)} \xrightarrow{\langle\delta',x(\langle z,t,c\rangle),\gamma'\rangle} Q'_{(I',w')}}{(P \mid Q)_{(I,w)} \xrightarrow{\langle\delta\delta',\tau,\gamma\gamma'\rangle} (c)(P' \mid Q')_{(I',w')}}\ c \notin fn(Q)$$

$$\text{CLOSE}\ \frac{P_{(I,w)} \xrightarrow{\langle\delta,\bar{x}\langle(z),t,c\rangle,\gamma\rangle} P'_{(I',w')}\ Q_{(I,w)} \xrightarrow{\langle\delta',x(\langle z,t,c\rangle),\gamma'\rangle} Q'_{(I',w')}}{(P \mid Q)_{(I,w)} \xrightarrow{\langle\delta\delta',\tau,\gamma\gamma'\rangle} (z)(c)(P' \mid Q')_{(I',w')}}\ z, c \notin fn(Q)$$

$$\text{RES-INP}\ \frac{P_{(I,w)} \xrightarrow{\langle\delta,x(\langle y,t,c\rangle),\gamma\rangle} P'_{(I',w')}}{(z)P_{(I,w)} \xrightarrow{\langle\delta,x(\langle y,t,d\rangle),\gamma\rangle} (z)P'_{(I',w')}}\ z \neq x, z \neq y$$

$$\text{RES-TAU}\ \frac{P_{(I,w)} \xrightarrow{\langle\delta,\tau,\gamma\rangle} P'_{(I',w')}}{(z)P_{(I,w)} \xrightarrow{\langle\delta,\tau,\gamma\rangle} (z)P'_{(I',w')}}$$

$$\text{REP}\ \frac{P_{(I,w)} \xrightarrow{\langle\delta,\alpha,\gamma\rangle} P'_{(I',w')}}{!P_{(I,w)} \xrightarrow{\langle\delta,\alpha,\gamma\rangle} (P' \mid !P)_{(I',w')}}$$

$$\text{REP-COM}\ \frac{P_{(I,w)} \xrightarrow{\langle\delta,\bar{x}\langle z,t,c\rangle,\gamma\rangle} P'_{(I',w')}\ P_{(I,w)} \xrightarrow{\langle\delta',x(\langle z,t,c\rangle),\gamma'\rangle} P''_{(I',w')}}{!P_{(I,w)} \xrightarrow{\langle\delta\delta',\tau,\gamma\gamma'\rangle} (((c)(P' \mid P'')) \mid !P)_{(I',w')}}\ c \notin fn(P)$$

$$\text{REP-CLOSE}\ \frac{P_{(I,w)} \xrightarrow{\langle\delta,\bar{x}\langle(z),t,c\rangle,\gamma\rangle} P'_{(I',w')}\ P_{(I,w)} \xrightarrow{\langle\delta',x(\langle z,t,c\rangle),\gamma'\rangle} P''_{(I',w')}}{!P_{(I,w)} \xrightarrow{\langle\delta\delta',\tau,\gamma\gamma'\rangle} (((z)(c)(P' \mid P'')) \mid !P)_{(I',w')}}\ z, c \notin fn(P)$$

6.2.5 Passing Clocks

Link passing in timed π-calculus is handled in exactly the same manner as in π calculus, in the sense that a process P can send a public channel x to a process Q. However, if Q already has access to a private channel x before the transition, the latter must be renamed to avoid confusion: this is called scope intrusion (Milner et al., 1992). If P has a private link x that it sends to Q, the scope of restriction will be extended, this is called scope extrusion (Milner et al., 1992). In this case, if Q already has access to a public link x, then the name of the private link must be changed before the transition. Clock passing is handled similarly, in the sense that we must handle different patterns of occurrences of name-binding for clock names, i.e., input prefix and restriction, as well as their behavior in transitions. In particular there are two cases to be considered. Assume that $P'' = \delta\gamma\bar{y}\langle x, t_x, c\rangle.P'$ and $Q'' = \delta'\gamma' y(\langle z, t_z, d\rangle).Q'$.

- A process P has a clock c, and sends $\langle x, t_x, c\rangle$ to process Q. Thus $P = (c)P''$ and $Q = Q''$; the transition is

  $(P \mid Q)_{(I,w)} \xrightarrow{\langle\delta\delta',\tau,\gamma\gamma'\rangle} (c)(P' \mid Q'\{x/z, t_x/t_z, c/d\})_{(I',w')}$

  This is analogous to scope extrusion of private links in π-calculus.

- A process P sends $\langle x, t_x, c\rangle$ to process Q; however, Q already possesses a local clock c. The local clock in P must be renamed to avoid confusion. Thus $P = (c)P''$ and $Q = (c)Q''$; in this case the transition is

  $(P \mid Q)_{(I,w)} \xrightarrow{\langle\delta\delta',\tau,\gamma\gamma'\rangle} (c')(P'\{c'/c\} \mid (c)Q'\{x/z, t_x/t_z, c'/d\})_{(I',w')}$

Next we define the structural congruence for proposed timed π-calculus.

6.2.6 Structural Congruence

The notion of structural congruence for timed π-calculus processes is identical to that of original π-calculus (Milner et al., 1992). Two timed π-calculus processes are structurally congruent if they are identical up to structure. Structural congruence, $\equiv$, is the least equivalence relation preserved by the process constructs that satisfy the axioms in Table 6.3.

Table 6.3. Axioms of Structural Congruence

α-conversion: $P \equiv Q$ if Q can be obtained from P by a finite number of change of bound names.	
Parallel Composition:	*Restriction:*
$(P \mid Q) \mid R \;\equiv\; P \mid (Q \mid R)$	$(x)\;(y)\;P \;\equiv\; (y)\;(x)\;P$
$P \mid Q \;\equiv\; Q \mid P$	$(x)\;0 \;\equiv\; 0$
$P \mid 0 \;\equiv\; P$	
	Replication:
Summation:	$!P \equiv\; P \mid !P$
$(P+Q)+R \;\equiv\; P+(Q+R)$	
$P+Q \;\equiv\; Q+P$	*Scope Extension:*
$P+0 \;\equiv\; P$	$((y)P \mid Q) \;\equiv\; (y)(P \mid Q)$ if $y \notin fn(Q)$

6.2.7 Timed Bisimulation

We would like to identify two processes which cannot be distinguished by an observer. We assume that the observer is able to observe all kinds of actions and moreover, it can observe the time-stamps of the events.

Definition 6.2.4. *For two clock expressions $\delta\gamma$ and $\delta'\gamma'$, in which δ and δ' are clock constraints and γ and γ' are clock resets; we say $\delta\gamma \models_c \delta'\gamma'$ if there is a mapping μ that maps every occurrence of a clock c in δ to a clock c' in δ' and a clock d in γ to a clock d' in γ' and moreover, $\delta\mu \models \delta'$ in which $\delta\mu$ is a clock constraint obtained by applying μ to δ, and $\models$ is the standard entailment relation.*

Definition 6.2.5. *A binary relation $\mathcal{S}$ on timed π-calculus processes is a (strong) timed simulation if $P\mathcal{S}Q$ implies that:*

1. *If $P_{(I_1,w_1)} \xrightarrow{\langle\delta,\tau,\gamma\rangle} P'_{(I,w)}$, then for some δ', γ' and Q', $Q_{(I_2,w_2)} \xrightarrow{\langle\delta',\tau,\gamma'\rangle} Q'_{(I',w')}$, where $\delta\gamma \models_c \delta'\gamma'$, $\delta'\gamma' \models_c \delta\gamma$ and $P' \;\mathcal{S}\; Q'$,*

2. *If* $P_{(I_1,w_1)} \xrightarrow{\langle\delta,x(\langle y,t,c\rangle),\gamma\rangle} P'_{(I,w)}$ *and* $y,c \notin n(P,Q)$*, then for some* δ'*,* γ' *and* Q'*,* $Q_{(I_2,w_2)} \xrightarrow{\langle\delta',x(\langle y,t',c\rangle),\gamma'\rangle} Q'_{(I',w')}$*, where* $\delta\gamma \models_c \delta'\gamma'$*,* $\delta'\gamma' \models_c \delta\gamma$*, and for all* $\langle z,t_z,d\rangle$*,* $P'\{z/y,t_z/t,d/c\}\mathcal{S}Q'\{z/y,t_z/t',d/c\}$*,*

3. *If* $P_{(I_1,w_1)} \xrightarrow{\langle\delta,\bar{x}\langle(y),t,c\rangle,\gamma\rangle} P'_{(I,w)}$ *and* $y,c \notin n(P,Q)$*, then for some* δ'*,* γ' *and* Q'*,* $Q_{(I_2,w_2)} \xrightarrow{\langle\delta',\bar{x}\langle(y),t',c\rangle,\gamma'\rangle} Q'_{(I',w')}$*, where* $\delta\gamma \models_c \delta'\gamma'$*,* $\delta'\gamma' \models_c \delta\gamma$ *and* $P'\mathcal{S}Q'$*,*

4. *If* $P_{(I_1,w_1)} \xrightarrow{\langle\delta,\bar{x}\langle y,t,c\rangle,\gamma\rangle} P'_{(I,w)}$ *and* $c \notin n(P,Q)$*, then for some* Q'*,* $Q_{(I_2,w_2)} \xrightarrow{\langle\delta',\bar{x}\langle y,t',(c)\rangle,\gamma'\rangle} Q'_{(I',w')}$*, where* $\delta\gamma \models_c \delta'\gamma'$*,* $\delta'\gamma' \models_c \delta\gamma$ *and* $P'\mathcal{S}Q'$*.*

The relation $\mathcal{S}$ *is a (strong) timed bisimulation if both* $\mathcal{S}$ *and its inverse are timed simulation. The relation* $\dot{\sim}$*, (strong) bisimilarity, on timed processes is defined by* $P\dot{\sim}Q$ *if and only if there exists a timed bisimulation* $\mathcal{S}$ *such that* $P\mathcal{S}Q$*.*

Intuitively, two timed processes are strong timed bisimilar if they take the same set of actions and the time-stamps of the actions satisfy the same clock constraints.

Definition 6.2.6. *If* $x \neq y$*, then* $\delta\gamma(c)\bar{x}\langle(y),t,c\rangle.P$ *means* $(y)(c)\ \delta\gamma\bar{x}\langle y,t,c\rangle.P$*, and the prefix* $\bar{x}\langle(y),t,c\rangle$ *is called a derived prefix.*

Analogous to strong bisimilarity in π-calculus, $\dot{\sim}$ is not in general preserved by substitutions of names. It follows that (strong) timed bisimilarity is not preserved by input prefix. As a result, (strong) timed bisimilarity is not a congruence. A collection of algebraic laws for (strong) timed bisimilarity, which are straightforward extension of algebraic laws for bisimilarity in π-calculus (Milner et al., 1992), is presented in Table 6.4. Note that ρ in proposition 5(d) denotes a prefix, including a derived prefix.

Proposition 7 *Expansion*

Let $P \equiv \sum_i \delta_i\gamma_i\rho_i.P_i$ and $Q \equiv \sum_j \eta_j\lambda_j\phi_j.Q_j$ where δ_i and η_j are constraints, γ_i and λ_j specify the set of clocks to be reset and ρ_i and ϕ_j are prefixes; $bn(\rho_i) \cap fn(Q) = \emptyset$ for all i, and $bn(\phi_j) \cap fn(P) = \emptyset$ for all j; then

$$P \mid Q \ \dot{\sim} \ \sum_i \sum_j E_i\gamma_i\rho_i.F_j\lambda_j\phi_j.(P_i \mid Q_j)+$$

Table 6.4. Timed Bisimilarity Algebraic Laws

Proposition 1 $\dot{\sim}$ is an equivalence relation.

Proposition 2.a If $P \dot{\sim} Q$ then
(a) $P + R \dot{\sim} Q + R$
(b) $P \mid R \dot{\sim} Q \mid R$
(c) $[x = y]P \dot{\sim} [x = y]Q$
(d) $(u)P \dot{\sim} (u)Q$
(e) $\delta\gamma\tau.P \dot{\sim} \delta'\gamma'\tau.Q$ $\delta\gamma \models_c \delta'\gamma'$, $\delta'\gamma' \models_c \delta\gamma$
(f) $(c)\delta\gamma\bar{x}\langle y, t_y, c\rangle.P \dot{\sim} (c)\delta'\gamma'\bar{x}\langle y, t_y, c\rangle.Q$ $\delta\gamma \models_c \delta'\gamma'$, $\delta'\gamma' \models_c \delta\gamma$

Proposition 3 *Match*
(a) $[x = y]P \dot{\sim} 0$ if $x \neq y$
(b) $[x = x]P \dot{\sim} P$

Proposition 2.b If for all $\langle v, t_v, d\rangle$ in which $v \in fn(P) \cup fn(Q) \cup \{y\}$, $P\{v/y, t_v/t_y, d/c\} \dot{\sim} Q\{v/y, t_v/t_y, d/c\}$ then
$\delta\gamma x(\langle y, t_y, c\rangle).P \dot{\sim} \delta'\gamma' x(\langle y, t_y, c\rangle).Q$ if $\delta\gamma \models_c \delta'\gamma'$, $\delta'\gamma' \models_c \delta\gamma$

Proposition 4 *Summation*
(a) $P + 0 \dot{\sim} P$
(b) $P + P \dot{\sim} P$
(c) $P + Q \dot{\sim} Q + P$
(d) $P + (Q + R) \dot{\sim} (P + Q) + R$

Proposition 5 *Restriction*
(a) $(y)\ P \dot{\sim} P\ y \notin fn(P)$
(b) $(x)\ (y)\ P \dot{\sim} (y)\ (x)\ P$
(c) $(y)\ (P + Q) \dot{\sim} (y)\ P + (y)\ Q$
(d) $(y)\ \delta\gamma\rho.P \dot{\sim} \delta\gamma\rho.(y)\ P\ y \notin n(\rho)$
(e) $(y)\ \delta\gamma\bar{y}(c)\langle x, t, c\rangle.P \dot{\sim} 0$
(f) $(y)\ \delta\gamma y(\langle x, t, c\rangle).P \dot{\sim} 0$
(g) $(y)\ \delta\gamma y(\langle (x), t, c\rangle).P \dot{\sim} 0$

Proposition 6 *Composition*
(a) $P \mid 0 \dot{\sim} P$
(b) $P_1 \mid P_2 \dot{\sim} P_2 \mid P_1$
(c) $(y)P_1 \mid P_2 \dot{\sim} (y)(P_1 \mid P_2)\ y \notin fn(P_2)$
(d) $(P_1 \mid P_2) \mid P_3 \dot{\sim} P_1 \mid (P_2 \mid P_3)$
(e) $(y)(P_1 \mid P_2) \dot{\sim} (y)P_1 \mid (y)P_2\ y \notin fn(P_1) \cap fn(P_2)$

$$\sum_j \sum_i G_j\lambda_j\phi_j.H_i\gamma_i\rho_i.(P_i \mid Q_j) + \sum_{\rho_i \texttt{comp} \phi_j} T\gamma_i\lambda_j\tau.R_{ij}$$

E_i and F_j are clock constraints that result from actions in which P_i precede Q_j. Clock expressions for E_is are defined in the fourth column of Table 6.5; while clock expressions for F_js are defined via expressions F'_j (defined in the fifth column of Table 6.5) as follows:

$$F_j = \begin{cases} F'_j & \text{if } \gamma_i = \emptyset \\ F'_j\{(b_j - t)/b_j\} & \text{if } \gamma_i \neq \emptyset \end{cases}$$

Similarly, G_j and H_i are clock constraints that result from actions in which Q_j precede P_i. Clock expressions for G_js are defined in the fourth column of Table 6.6; while clock expressions for H_is are defined via expressions H'_i (defined in the fifth column of Table 6.6) as follows:

$$H_i = \begin{cases} H'_i & \text{if } \lambda_j = \emptyset \\ H'_i\{(a_i - t)/a_i\} & \text{if } \lambda_j \neq \emptyset \end{cases}$$

The expression T represents the set of clock constraints corresponding to τ actions which is defined in the last column of Table 6.5. The relation ρ_i comp ϕ_j (ρ_i complements ϕ_j) holds in the following four cases, which also defines R_{ij}:

1. ρ_i is $\bar{x}\langle u, t, c\rangle$ and ϕ_j is $x(\langle v, t_v, d\rangle)$; then R_{ij} is $(e)(P_i\{e/c\}|Q_j\{u/v, t/t_v, e/d\}$, where e is not free in $(c)P_i$ or in $(d)Q_j$,

2. ρ_i is $\bar{x}\langle (u), t, c\rangle$ and ϕ_j is $x(\langle v, t_v, d\rangle)$; then R_{ij} is$(w)(e)(P_i\{w/u, e/c\}|Q_j \{w/v, t/t_v, e/d\}$, where w and e are not free in $(u)(c)P_i$ or in $(v)(d)Q_j$.

3. ρ_i is $x(\langle v, t_v, d\rangle)$ and ϕ_j is $\bar{x}\langle u, t, c\rangle$; then R_{ij} is $(e)(P_i\{u/v, t/t_v, e/d\}|Q_j \{e/c\}$, where e is not free in $(d)P_i$ or in $(c)Q_j$,

4. ρ_i is $x(\langle v, t_v, d\rangle)$ and ϕ_j is $\bar{x}\langle (u), t, c\rangle$; then R_{ij} is $(w)(e)(P_i\{w/v, t/t_v, e/d\}| Q_j\{w/u, e/c\}$, where w and e are not free in $(v)(d)P_i$ or in $(u)(c)Q_j$.

Note that all types of clock constraints that appear in clock operations can be simplified to the form$(c \sim r)$, since the time-stamps appearing in the clock constraints are instantiated at the time of inferring transitions. Therefore in Table 6.5 and Table 6.6 all the clock constraints are of the form $(c \sim r)$.

To illustrate this, the expansion theorem is presented next for a simple case in which all the clock constraints appearing in P and Q are of the form $(c < a_i)$ and $(c < b_j)$ respectively. This case corresponds to the first rows of Tables 6.5 and 6.6.

Table 6.5. Clock Constraints for Expansion Theorem (P_is proceed Q_js)

δ_i	η_j	condition	E_i	F'_j	T
$(c < a_i)$	$(c < b_j)$		$(c < min(a_i, b_j))$	$(c < b_j)$	$(c < min(a_i, b_j))$
$(c < a_i)$	$(c = b_j)$		$(c < min(a_i, b_j))$	$(c = b_j)$	$(c = b_j)$
$(c < a_i)$	$(c > b_j)$		$(c < a_i)$	$(c > b_j)$	$(c < a_i)(c > b_j)$
$(c < a_i)$	$(c \leq b_j)$		$(c < min(a_i, b_j))$	$(c \leq b_j)$	$(c < min(a_i, b_j))$
$(c < a_i)$	$(c \geq b_j)$		$(c < min(a_i, b_j))$	$(c \geq b_j)$	$(c < a_i)(c \geq b_j)$
$(c > a_i)$	$(c < b_j)$	$a_i < b_j$	$(c > a_i)(c < b_j)$	$(c < b_j)$	$(c > a_i)(c < b_j)$
$(c > a_i)$	$(c = b_j)$	$a_i < b_j$	$(c > a_i)(c < b_j)$	$(c = b_j)$	$(c = b_j)$
$(c > a_i)$	$(c > b_j)$		$(c > a_i)$	$(c > b_j)$	$(c > max(a_i, b_j))$
$(c > a_i)$	$(c \leq b_j)$	$a_i < b_j$	$(c > a_i)(c < b_j)$	$(c \leq b_j)$	$(c > a_i)(c \leq b_j)$
$(c > a_i)$	$(c \geq b_j)$	$a_i < b_j$ $a_i > b_j$	$(c > a_i)(c < b_j)$ $(c > a_i)$	$(c \geq b_j)$ $(c \geq a_i)$	$(c \geq b_j)$ $(c > a_i)$
$(c = a_i)$	$(c < b_j)$	$a_i < b_j$	$(c = a_i)$	$(c < b_j)$	$(c = a_i)$
$(c = a_i)$	$(c = b_j)$	$a_i < b_j$	$(c = a_i)$	$(c = b_j)$	$(c = a_i)$
$(c = a_i)$	$(c > b_j)$		$(c = a_i)$	$(c > b_j)$	$(c = a_i)$
$(c = a_i)$	$(c \leq b_j)$	$a_i < b_j$	$(c = a_i)$	$(c \leq b_j)$	$(c = a_i)$
$(c = a_i)$	$(c \geq b_j)$		$(c = a_i)$	$(c \geq max(a_i, b_j))$	$(c = a_i)$
$(c \leq a_i)$	$(c < b_j)$	$a_i < b_j$ $a_i > b_j$	$(c \leq a_i)$ $(c < b_j)$	$(c < b_j)$ $(c < b_j)$	$(c \leq a_i)$ $(c < b_j)$
$(c \leq a_i)$	$(c = b_j)$	$a_i < b_j$ $a_i > b_j$	$(c \leq a_i)$ $(c < b_j)$	$(c = b_j)$ $(c = b_j)$	$(c = b_j)$ $(c = b_j)$
$(c \leq a_i)$	$(c > b_j)$		$(c \leq a_i)$	$(c > b_j)$	$(c \leq a_i)(c > b_j)$
$(c \leq a_i)$	$(c \leq b_j)$	$a_i < b_j$ $a_i > b_j$	$(c \leq a_i)$ $(c < b_j)$	$(c \leq b_j)$ $(c \leq b_j)$	$(c \leq a_i)$ $(c \leq a_i)$
$(c \leq a_i)$	$(c \geq b_j)$		$(c \leq a_i)$	$(c \geq b_j)$	$(c \leq a_i)(c \geq b_j)$
$(c \geq a_i)$	$(c < b_j)$	$a_i < b_j$	$(c \geq a_i)$	$(c < b_j)$	$(c \geq a_i)(c < b_j)$
$(c \geq a_i)$	$(c = b_j)$	$a_i < b_j$	$(c \geq a_i)$	$(c = b_j)$	$(c = b_j)$
$(c \geq a_i)$	$(c > b_j)$	$a_i < b_j$ $a_i > b_j$	$(c \geq a_i)$ $(c \geq a_i)$	$(c > b_j)$ $(c > b_j)$	$(c > b_j)$ $(c \geq a_i)$
$(c \geq a_i)$	$(c \leq b_j)$	$a_i < b_j$	$(c \geq a_i)$	$(c \leq b_j)$	$(c \geq a_i)(c \leq b_j)$
$(c \geq a_i)$	$(c \geq b_j)$	$a_i < b_j$ $a_i > b_j$	$(c \geq a_i)$ $(c \geq a_i)$	$(c \geq b_j)$ $(c > b_j)$	$(c \geq b_j)$ $(c \geq b_j)$

Table 6.6. Clock Constraints for Expansion Theorem (Q_js proceed P_is)

η_j	δ_i	condition	G_j	H'_i
$(c < b_j)$	$(c < a_i)$		$(c < min(b_j, a_i))$	$(c < a_i)$
$(c < b_j)$	$(c = a_i)$		$(c < min(b_j, a_i))$	$(c = a_i)$
$(c < b_j)$	$(c > a_i)$		$(c < b_j)$	$(c > a_i)$
$(c < b_j)$	$(c \leq a_i)$		$(c < min(b_j, a_i))$	$(c \leq a_i)$
$(c < b_j)$	$(c \geq a_i)$		$(c < min(b_j, a_i))$	$(c \geq a_i)$
$(c > b_j)$	$(c < a_i)$	$b_j < a_i$	$(c > b_j)(c < a_i)$	$(c < a_i)$
$(c > b_j)$	$(c = a_i)$	$b_j < a_i$	$(c > b_j)(c < a_i)$	$(c = a_i)$
$(c > b_j)$	$(c > a_i)$		$(c > b_j)$	$(c > a_i)$
$(c > b_j)$	$(c \leq a_i)$	$b_j < a_i$	$(c > b_j)(c < a_i)$	$(c \leq a_i)$
$(c > b_j)$	$(c \geq a_i)$	$b_j < a_i$	$(c > b_j)(c < a_i)$	$(c \geq a_i)$
		$b_j > a_i$	$(c > b_j)$	$(c \geq b_j)$
$(c = b_j)$	$(c < a_i)$	$b_j < a_i$	$(c = b_j)$	$(c < a_i)$
$(c = b_j)$	$(c = a_i)$	$b_j < a_i$	$(c = b_j)$	$(c = a_i)$
$(c = b_j)$	$(c > a_i)$		$(c = b_j)$	$(c > a_i)$
$(c = b_j)$	$(c \leq a_i)$	$b_j < a_i$	$(c = b_j)$	$(c \leq a_i)$
$(c = b_j)$	$(c \geq a_i)$		$(c = b_j)$	$(c \geq max(b_j, a_i))$
$(c \leq b_j)$	$(c < a_i)$	$b_j < a_i$	$(c \leq b_j)$	$(c < a_i)$
		$b_j > a_i$	$(c < a_i)$	$(c < a_i)$
$(c \leq b_j)$	$(c = a_i)$	$b_j < a_i$	$(c \leq b_j)$	$(c = a_i)$
		$b_j > a_i$	$(c < a_i)$	$(c = a_i)$
$(c \leq b_j)$	$(c > a_i)$		$(c \leq b_j)$	$(c > a_i)$
$(c \leq b_j)$	$(c \leq a_i)$	$b_j < a_i$	$(c \leq b_j)$	$(c \leq a_i)$
		$b_j > a_i$	$(c < a_i)$	$(c \leq a_i)$
$(c \leq b_j)$	$(c \geq a_i)$		$(c \leq b_j)$	$(c \geq a_i)$
$(c \geq b_j)$	$(c < a_i)$	$b_j < a_i$	$(c \geq b_j)$	$(c < a_i)$
$(c \geq b_j)$	$(c = a_i)$	$b_j < a_i$	$(c \geq b_j)$	$(c = a_i)$
$(c \geq b_j)$	$(c > a_i)$	$b_j < a_i$	$(c \geq b_j)$	$(c > a_i)$
		$b_j > a_i$	$(c \geq b_j)$	$(c > a_i)$
$(c \geq b_j)$	$(c \leq a_i)$	$b_j < a_i$	$(c \geq b_j)$	$(c \leq a_i)$
$(c \geq b_j)$	$(c \geq a_i)$	$b_j < a_i$	$(c \geq b_j)$	$(c \geq a_i)$
		$b_j > a_i$	$(c \geq b_j)$	$(c > a_i)$

$$P \mid Q \;\dot{\sim}\; \sum_i \sum_j (c < \min(a_i, b_j))\gamma_i\rho_i.(c < b_j)\lambda_j\phi_j.(P_i \mid Q_j) +$$
$$\sum_j \sum_i (c < \min(a_i, b_j))\lambda_j\phi_j.(c < a_i)\gamma_i\rho_i.(P_i \mid Q_j) +$$
$$\sum_{\rho_i \,\texttt{comp}\, \phi_j} (c < \min(a_i, b_j))\gamma_i\lambda_j\tau.R_{ij}$$

Example 6.2.3. *Let* $P = (c < 1)\bar{x}\langle z, t_z, c\rangle$ *and* $Q = (c < 0.5)(c := 0)x(\langle w, t_w, c\rangle)$*, then*

$$P \mid Q \;\dot{\sim}\; (c < 0.5)\bar{x}\langle z, t_z, c\rangle.(c < 0.5)(c := 0)x(\langle w, t_w, c\rangle) +$$
$$(c < 0.5)(c := 0)x(\langle w, t_w, c\rangle).(c < 1 - t_w)\bar{x}\langle z, t_z, c\rangle + (c < 0.5)(c := 0)\tau$$

Proofs of above propositions are extensions of the proofs for untimed π-calculus processes (Milner et al., 1992) (these extensions take clocks into account) which are presented in Appendix D.

6.3 Operational Semantics in Logic Programming

For a complete encoding of the operational semantics of timed π-calculus, we must account for the facts that: (i) clock constraints are posed over continuous time, (ii) infinite computations are defined in timed π-calculus (and also π-calculus) through the infinite replication operator '!', and (iii) we are dealing with mobile concurrent processes. We have developed an operational semantics of timed π-calculus using logic programming extended with constraints over reals and coinduction, in which channels are modeled as streams. We briefly explain how all the three aspects are handled within our framework.

In our LP formulation of the operational semantics of timed π-calculus, clock expressions and time constraints are handled using *constraint logic programming over reals* (Jaffar and Maher, 1994), (rational) infinite computations [6] are handled using *coinductive logic programming* (Simon et al., 2007, Gupta et al., 2007), and finally concurrency is simulated by

[6]We handle only *rational* infinite computations, where finite number of finite behaviors are repeated infinitely often.

coroutining within logic programming computations[7]: scheduling a goal to be executed immediately after a variable is bound can be used to model the actions taken by processes as soon as a message is received in the channel specified by that variable.

Our operational semantics expressed as a coinductive coroutined constraint logic program can be regarded as an interpreter for timed π-calculus expressions, and can be used for modeling and verification of real-time systems. We will show the effectiveness of our framework by applying it to the GRC problem.

6.4 Example: The generalized railroad crossing (GRC)

The complete description of GRC problem was presented in Section 4.3, the summary of this description is as follows. The GRC problem (Heitmeyer and Lynch, 1994) describes a railroad crossing system with several tracks and an unspecified number of trains traveling through the tracks. The gate at the railroad crossing should be operated in a way that guarantees the *safety* and *utility* properties. The *safety* property stipulates that the gate must be down while there is a train in the crossing. The *utility* property states that the gate must be up (or going up) when there is no train in the crossing. The system is composed of three components: *train*, *controller* and *gate*, which communicate by sending and receiving signals. The controller at the railroad crossing might receive various signals from trains in different tracks. In order to avoid signals from different trains being mixed, each train communicates through a private channel with the controller. A new channel is established for each approaching train to the crossing area through which the communication between the train and the controller takes place. For simplicity of presentation we consider only one track in this example. The components of the problem are specified via three timed automata in Figure 6.1.

[7]Coroutining can be practically realized through delay/freeze construct supported in most Prolog systems.

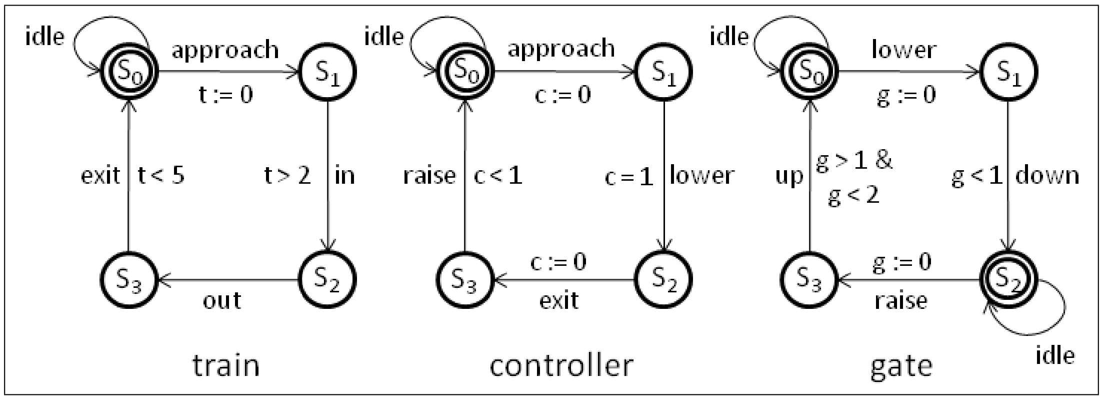

Figure 6.1. Timed automata for train, controller, and gate in GRC with one track

In our modeling of GRC in timed π-calculus, each component of the system is considered as a timed π-calculus process. The behavior of the system can be seen as a set of concurrent processes which is expressed graphically in Figure 6.2.

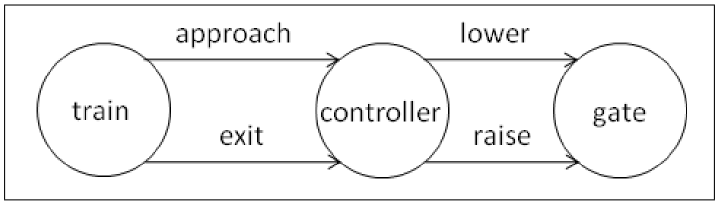

Figure 6.2. *train* | *controller* | *gate*

Note that the design of 1-track GRC shown in Figure 6.1 (originally from Alur and Dill (1994)) does not account for the delay between the sending of *approach* and *exit* signals by *train* and receiving them by *controller*. Similarly the delays between sending *lower* and *raise* by *controller* and receiving them by *gate* is not taken into account. In contrast, in our specification of GRC in timed π-calculus, we are considering the delays; therefore, all the time-related reasoning in the system is performed against *train*'s clock and the time-stamp of *approach* signal (sent by *train* to *controller*). The timed π-calculus expressions for processes of 1-track GRC are presented in Table 6.7.

Note that in the π-calculus expression for *train*, the two consecutive τ actions correspond to *train*'s internal actions *in* and *out*. Similarly, in the expression for *gate*, the two τ ac-

Table 6.7. The Timed π-Calculus Expressions For Components of GRC (With One Track)

$train \equiv$	$controller \equiv$
$!(ch)(t)\overline{ch1}\langle ch, t_c, t\rangle$	$!ch1(\langle y, t_y, d\rangle).$
$(t := 0)\overline{ch}\langle approach, t_a, t\rangle.$	$y(\langle x, t_x, c\rangle).$
$(t > 2)\tau.\tau.$	$([x = approach](c = 1)(c := 0)\overline{ch2}\langle lower, t_l, c\rangle+$
$(t < 5)\overline{ch}\langle exit, t_e, t\rangle)$	$[x = exit](c - t_x < 1)(c := 0)\overline{ch2}\langle raise, t_r, c\rangle)$
$gate \equiv$	
$!ch2(\langle x, t_x, g\rangle).$	
$([x = lower](g < 1)\tau + [x = raise](g > 1)(g < 2)\tau)$	
$main \equiv train \mid controller \mid gate$	

tions correspond to *gate*'s internal actions; the first τ represents *down*; while the second τ represents *up*.

The logic programming realization of timed π-calculus processes in 1-track GRC is presented in Table 6.8. For simplicity of presentation, we show the simplified version of this encoding; however, the complete code can be found in Saeedloei and Gupta (2011b) which can be executed in SWI Prolog.

The first argument of `train/5` is the list of timed events (a list of events with their timestamps) generated by *train*; while the second argument is the list of events sent to *controller*. The third argument is *train*'s current state. `W` is the current wall clock time and `Tc` is *train*'s clock. Similarly the first argument of `controller/3` is the list of timed events received from *train*, while the second argument is the list of events generated by *controller* and sent to *gate*. `Sc` is the current state of *controller*. Finally, the first argument of `gate/2` is the list of timed events received from *controller*, while the second argument is the current state of *gate*. Note that `train/5` is declared as coinductive only on the first three arguments. In other words, the current wall clock time, and *train*'s clock are non-coinductive arguments which will be ignored for coinductive success; however, the set of accumulated constraints on these two arguments have to be satisfied for coinductive success. `controller/3` and `gate/2` are declared as coinductive in all of their arguments.

Table 6.8. The Timed π-Calculus processes of GRC, Implemented in coinductive CLP(R)

```
:- coinductive(train(+, +, +, -, -)).
train(X, Y, Si, W, Tc) :-
  ( H = approach, {Tc2 = W}
  ; H = in, {W - Tc > 2, Tc2 = Tc}
  ; H = out, {Tc2 = Tc}
  ; H = exit, {W - Tc < 5, Tc2 = Tc} ),
  {W2 > W},
  t_trans(Si, H, So),
  freeze(X, train(Xs, Ys, So, W2, Tc2)),
  ((H = approach
   ;H = exit) ->
                Y = [(H,W)| Ys]
               ;Y = Ys ),
  X = [(H, W)| Xs].

:- coinductive(controller(+,+,+)).
controller([(H, W)| Xs], Y, Sc) :-
   freeze(Xs, controller(Xs, Ys, Sc3)),
   ( H = approach, M = lower, {W2 > W, W2 - W = 1}
   ; H = exit,     M = raise, {W2 > W, W2 - W < 1} ),
   c_trans(Sc,  H, Sc2),
   c_trans(Sc2, M, Sc3),
   Y = [(M, W2)| Ys].

:- coinductive(gate(+,+)).
gate([(H, W)| Xs], Sg) :-
  freeze(Xs, gate(Xs, Sg3)),
  ( H = lower, M = down, {W2 > W, W2 - W < 1}
  ; H = raise, M = up,   {W2 > W, W2 - W > 1, W2 - W < 2} ),
  g_trans(Sg,  H, Sg2),
  g_trans(Sg2, M, Sg3).

t-trans(s0,approach,s1).  c-trans(s0,approach,s1).  g-trans(s0,lower,s1).
t-trans(s1,in,      s2).  c-trans(s1,lower,   s2).  g-trans(s1,down, s2).
t-trans(s2,out,     s3).  c-trans(s2,exit,    s3).  g-trans(s2,raise,s3).
t-trans(s3,exit,    s0).  c-trans(s3,raise,   s0).  g-trans(s3,up,   s0).
```

Note that `t-trans/3, c-trans/3`, and `g-trans/3` specify the internal transitions of *train, controller*, and *gate* respectively. The entire system will wait for *train* to generate the initial signals and send them to *controller*; as soon as *controller* receives these signals (the first argument of `controller/3` gets bound), it will send appropriate signals to *gate*. This composition of three processes is realized by the expression:

```
freeze( A, (freeze(C, gate(C,s0)),
            controller(B,C,s0)) ),
train(A,B,s0,0,0).
```

Once the system is modeled as a coinductive coroutined CLP(R) program, the model can be used to verify interesting properties of the system by posing queries. Note that since the system generates the sequence of timed events as an output, the safety, utility, and other interesting properties of the system can be verified using simple queries in the exact same manner as it was performed in Section 4.3.2. We only discuss the liveness property here.

Checking liveness properties in general, includes essential algorithm for loop detection, which is computationally expensive. However, in our system loop detection can be performed elegantly using coinduction. The *liveness* property for GRC states that once the gate goes down, it will eventually goes up. To verify this, we negate the liveness property, using the predicate `not-live/1`, and look for the possibility of a situation in which *up* does not appear infinitely often in the accepting timed trace of the system.

```
not-live(R) :- main(R), co-not-member((up, _), R).
```

6.5 Conclusions and Related Work

Since the π-calculus was proposed by Milner et al. (Milner et al., 1992), many researchers have extended it for modeling distributed real-time systems. Berger has introduced timed π-calculus (π_t-calculus) (Berger, 2004), asynchronous π-calculus with timers and a notion of *discrete* time, locations, and message failure, and explored some of its basic properties.

Carlos Olarte has studied temporal CCP as a model of concurrency for mobile, timed reactive systems in his Ph.D thesis (Olate, 2009). He has developed a process calculus called universal temporal CCP. His work can be seen as adding mobile operation to the tcc. In utcc, like tcc, time is conceptually divided into time intervals (or time units); therefore it is discretized. Lee et al. (Lee and Zic, 2002) introduced another timed extension of π-calculus called real-time π-calculus (πRT-calculus). They have introduced the time-out operator and considered a global clock, single observer as part of their design, as is common in other (static) real-time process algebras. They have used the set of natural numbers as the time domain, i.e., time is *discrete* and is strictly increasing. Ciobanu et al. (Ciobanu and Prisacariu, 2006) have introduced and studied a model called timed distributed π-calculus in which they have considered timers for channels, by which they restrict access to channels. They use *decreasing* timers, and time is discretized in their approach also. Many other timed calculi have similar constructs which also discretize time (Degano et al., 1996, Laneve and Zavattaro, 2005, Mazzara, 2005). In summary, all these approaches share some common features; they use a *discrete* time-stepping function or timers to increase/decrease the time-stamps after every action (they assume that every action takes exactly one unit of time). *In contrast, our approach for extending π-calculus with time faithfully treats time as continuous.*

Rounds et al. (Rounds and Song, 2003) have proposed Φ-calculus as a hybrid extension of π-calculus, while we are focused on extending the π-calculus with real-time. They have proposed the notion of strong bisimilarty; however, their notion of bisimilarity does not take into account time or other continuous quantities. In contrast, we fully develop the notion of *(strong) timed bisimilarity.* We have also developed the executable operational semantics of timed π-calculus using logic programming which is not done in their work.

The work of Yi (Yi, 1991) shows how to introduce time into Milner's CCS to model real-time systems. An extra variable t is introduced which records the time delay before a message on some channel α is available, and also a timer for calculating delays. The idea is to use delay operators to suspend activities. In our opinion, it is much harder to specify real-time systems using delays. Our approach provides a more direct way of modeling time in π-calculus via stop watches, and also can be used to elegantly reason about delays.

In this chapter we extended the π-calculus with real time. The resulting formalism, timed π-calculus, is an expressive, natural model for describing real-time, mobile, concurrent processes. We proposed an operational semantics for timed π-calculus and developed the notion of timed bisimilarity. We also investigated the algebraic rules of timed bisimilarity. Most of the algebraic rules concerning bisimilarity in original π-calculus, holds for timed bisimilarity defined in our proposed timed π-calculus. We presented the outline of an implementation of the operational semantics of timed π-calculus; an implementation based on co-CLP extended with coroutining. Note that coinduction is used in order to handle the infinite behavior of processes, defined using the replication operator, '!'. To conclude, co-CLP extended with coroutining provides an expressive and easy-to-use framework for implementing the π-calculus extended with real-time via clocks, which then can be used for elegant modeling and verification of complex real-time systems and cyber-physical systems and for verifying their interesting properties.

CHAPTER 7
LOGIC PROGRAMMING FOUNDATIONS OF CPS

7.1 Introduction

Cyber-physical systems (CPS) are becoming ubiquitous. Almost every device today has a controller that reads inputs through sensors, does some processing and then performs actions through actuators. Examples include controller systems in cars (Anti-lock Brake System, Cruise Controllers, Collision Avoidance, etc.), automated manufacturing, smart homes, robots, etc. These controllers are discrete digital systems whose inputs are continuous physical quantities (e.g., time, distance, acceleration, temperature, etc.) and whose outputs control physical (analog) devices. Thus, cyber-physical systems involve both digital and analog data. In addition, CPS are assumed to run forever, and many CPS may run concurrently with each other (Lee, 2008, Gupta, 2006a). Due to the fundamentally discrete nature of computation, researchers have had difficulty dealing with continuous quantities in computations (typical approaches discretize continuous quantities, e.g., time). Likewise, modeling of perpetual computations is not straightforward (only recently, techniques such as coinductive logic programming (Simon et al., 2007, Gupta et al., 2007) have been introduced to operationally realize rational, infinite computations). Concurrency is reasonably well understood, but when combined with continuous quantities and with perpetual computations, CPS become extremely hard to model faithfully. In this chapter we develop techniques for faithfully modeling cyber-physical systems and verifying their properties. We consider communicative hybrid automata as the underlying model, and illustrate their use in specifying and verifying highly complex CPS. Our approach is based on using *logic programming* (*LP*) for modeling computations, *constraint logic programming* for modeling continuous physical quantities, *co-induction* for modeling perpetual execution and *coroutining* for modeling concurrency in CPS. CPS are thus represented as coroutined coinductive constraint logic programs which are subsequently used to elegantly verify cyber-physical properties of the system relating to safety, liveness and utility. These logic programs can also be used for automatically generating implementation code for the CPS.

To illustrate this, we consider the reactor temperature control system (X. Nicollin et al., 1992), and show how it can be naturally and elegantly modeled (and its properties verified) as a network of hybrid automata implemented as coroutined coinductive CLP(R) programs. Note that our framework for modeling and verifying CPS provides diagnostic information that can be used in design of the system. For instance, if our system fails to satisfy a safety requirement, it will generate a time trace of events as a proof of violation of the safety property. Another interesting feature of our framework is its ability to perform precise parametric analysis. The goal of parametric analysis is to determine necessary and sufficient constraints on parameters under which correctness requirements are met. Using our method we were able to compute exact bounds on parameters of the reactor temperature control system, thereby improving on the results of Henzinger and Ho (Henzinger and hsin Ho, 1995), who only compute them approximately.

Our contribution includes developing a practical and faithful realization of hybrid automata as coinductive coroutined constraint logic programs over reals. We identify the class of hybrid systems that can be modeled in our framework. What is noteworthy in our realization of hybrid automata is that, in contrast to other approaches, we do not discretize time (and other physical quantities): time is treated as a continuous quantity, with relationship between various time instances modeled as constraints over reals.

7.2 Motivation

CPS are highly complex systems for which today's computing and networking technologies do not provide adequate foundations. In fact, Edward Lee states (Lee, 2008):

> CPS are integrations of computation with physical processes. Embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computation and vice versa. In the physical world, the passage of time is inexorable and concurrency is intrinsic. Neither of these properties is present in today's computing and networking abstractions.

Lee goes on to argue that "the mismatch between these abstractions and properties of physical processes impede technical progress." Thus, according to Lee, a major challenge is to find the right abstractions for CPS. Similarly, Rajesh Gupta (Gupta, 2006a) urges researchers "to achieve the goal of semantic support for location and time at all levels," and address the following technical problems for CPS:

1. "How do we capture location (and timing) information into CPS models that allows for validation of the logical properties of a program against the constraints imposed by its physical (sensor) interaction."
2. "What are useful models for capturing faults and disconnections within the coupled physical-computational system? ..."
3. "What kind of properties can be verified, ..."
4. "What programming model is best suited for CPS applications ..."

Thus, the search is ongoing for a formalism that elegantly captures both the computational and the physical aspects of CPS, and that can handle their concurrent and non-terminating nature. Such a model (specification) should allow one to verify cyber-physical properties of the system, as well as automatically generate code (in a provably correct manner) that implements the system.

Verification and provably correct code generation are important, since errors in a cyber-physical system can mean the difference between life and death. For instance a car may have dozens of CPS (controllers) that are all networked, and making sure that they satisfy *safety* and *liveness* properties is important. For example, in a modern automobile every physical action (e.g., braking) is merely an instruction to a computer to carry out that action, the computer will in turn use sensors/actuators [CPS] to perform that action. Verifying the correctness of such systems therefore is doubly important (it may not be possible to stop a runaway car, for example, by putting it in neutral gear, because putting it in neutral gear is merely an instruction to the controller, which an incorrectly designed controller may simply ignore). Developing proper computational abstractions of CPS is extremely important: if they can't even be correctly modeled, then we can have no confidence in the correctness of the systems we develop.

7.3 Modeling CPS with Coroutined Coinductive CLP(R)

CPS have the following four characteristics (Lee, 2008, Gupta, 2006a): (i) they perform discrete computations, (ii) they deal with continuous quantities, (iii) they are concurrent, and (iv) they run forever. Due to fundamentally discrete nature of computation, researchers have had difficulty dealing with continuous quantities in computations. Likewise, modeling of perpetual computations is not straightforward. Concurrency is reasonably well understood, but when combined with continuous quantities and with perpetual computations, CPS become extremely hard to model faithfully. We show how all four aspects of CPS can be elegantly handled within LP extended with constraints over reals, coinduction, and coroutining.

We model CPS as communicative hybrid automata that control physical systems. A hybrid automaton is a finite-state automaton, extended with a set of real-valued variables. The *control locations (states)* of the automaton are labeled with evolution laws, which are specified as differential equations. The values of the variables change continuously with time according to the associated law at each location. If the value of a variable x does not change in a particular location, then $\dot{x} = 0$ and can be eliminated. Each location is also labeled with an invariant condition that must hold when the control resides at the location. The transitions of the automaton are labeled with guarded sets of assignments. A transition is enabled when the associated guard is true and its execution modifies the values of the variables according to the assignments. There is a labeling function that assigns a label (or synchronization letter) to each transition. There is also an initial condition that describes the set of possible values for the system when it starts in the initial location.

The reactor temperature control system is a traditional example of a cyber-physical system. The system consists of a reactor core and two control rods that control the temperature of the reactor core. The goal is to keep the temperature between temperatures θ_m and θ_M. If the temperature reaches θ_M, then it should be decreased by introducing one of the control rods into the reactor core. Figure 7.1 shows the hybrid system of this example. Variable θ measures the temperature, variables r_1 and r_2 measure the time that has elapsed since rod_1 and rod_2 were removed from the reactor core, respectively. Variables c_1 and c_2 are used to model two clocks of the core. Initially θ is θ_m degrees and both control rods are outside of the reactor core. In this case, θ rises according to the differential equation $\dot{\theta} = \frac{\theta}{10} - 50$, location *no_rod*. θ decreases according to the differential equations $\dot{\theta} = \frac{\theta}{10} - 56$ (location rod_1) and $\dot{\theta} = \frac{\theta}{10} - 60$ (location rod_2) depending on the control rod used. A control rod may be used again, if $T \geq 0$ units of time have elapsed since it was last removed. If θ cannot decrease because no control rod is available, then a shutdown of the reactor is necessary. A shutdown of the system should be prevented.

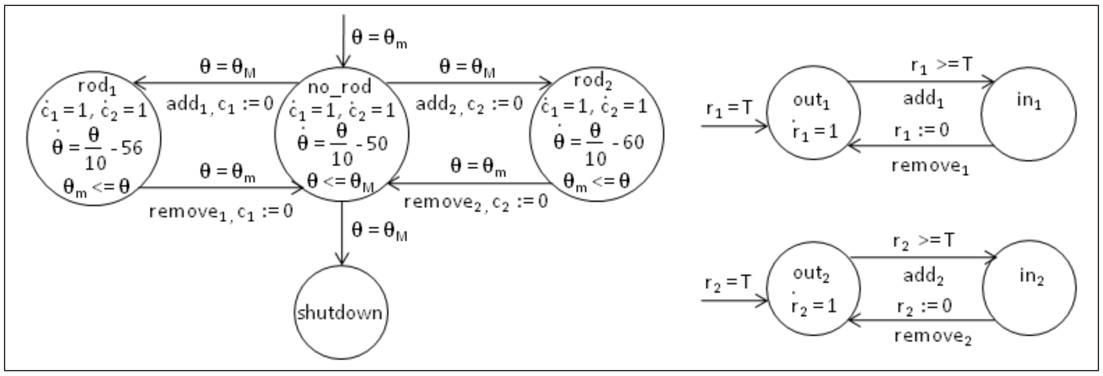

Figure 7.1. The Reactor Core, and Control Rod Automata

As we mentioned earlier, communicative hybrid automata constitute the foundations of cyber-physical systems. In our framework for modeling CPS, each hybrid automaton is modeled as a set of transition rules (one rule per transition in the automaton), where each rule is extended with constraints on time and other physical quantities. Therefore, hybrid automata are modeled as *logic programs* (Lloyd, 1987, Sterling and Shapiro, 1994), physical quantities are represented as continuous quantities (i.e., not discretized) and the constraints imposed on them by transitions are faithfully modeled with *constraint logic programming over reals* (Jaffar and Maher, 1994). By considering *coinductive logic programming* (Gupta et al., 2007), we are able to model the non-terminating aspect of hybrid automata, and finally concurrency between hybrid automata will be handled by allowing *coroutining* (realized via delay declarations of Prolog (Sterling and Shapiro, 1994) in our subsequent implementation) within logic programming computations. The coroutining facility of logic programming (in particular Prolog) is realized via built-in predicates such as *freeze*, *when*, etc. Coroutining deals with having Prolog goals scheduled for execution as soon as some conditions are fulfilled. In Prolog the most commonly used condition is the instantiation (binding) of a variable. Scheduling a goal to be executed immediately after a variable is bound can be used to model the transitions taken on synchronization letters.

Our logic programming realization of hybrid automata is illustrated by its application to the reactor temperature control system in the next section.

7.4 Application: Modeling the Reactor Temperature Control System

For modeling the reactor control system, we set $\theta_m = 510$, and $\theta_M = 550$. Note that these values have been used in literature such as Henzinger and hsin Ho (1995). A real-time system designer can choose other values for these parameters. Two clocks c_1 and c_2, are used by the core so that the time of transitions of the core can be remembered. c_i will be reset on transitions labeled by add_i and $remove_i$. This can be used for calculating the time elapsed in every control location of the core automaton. We solve the differential equations in each location and determine how long the system remains in this location and how the variables (temperature, and clocks) change during this time. Solving the differential equation in location *no_rod* results in: $\theta(t) = 10e^{(t/10)} + 500$. We determine at which time point we have to make transitions add_1 or add_2 by computing the time the reactor reaches the critical temperature θ_M, that is $\theta(t) = \theta_M$. Solving this equation we obtain the time spent in the *no_rod* location depending on the time point in which the location *no_rod* is entered. Similarly, solving the differential equations in locations rod_1 and rod_2 results in: $\theta(t) = -10e^{(t/10)} + 560$ and $\theta(t) = -50e^{(t/10)} + 600$, respectively. Again we can determine at which time point we have to make transitions $remove_1$ or $remove_2$ by computing the time the reactor reaches the critical temperature θ_m, that is $\theta(t) = \theta_m$. Solving this equation in control locations rod_1 and rod_2, we obtain the time spent in these locations, which correspond to the time for the reactor core to cool down in these locations. The logic programming representation corresponding to the hybrid automata in the reactor control system is presented in Table 7.1.

The first argument of the predicate `r1/7` (`r2/7`) is the current state (control location), the second argument is the synchronization letter which triggers the transition, while the third argument is the new control location the transition results in. W represents the wall clock time; the pair of arguments, Ti and To, represent the clock of the automaton. The last argument, T, is the parameter which determines the time that must elapse since the last time that a rod was removed from the reactor core before it can be reused again. The transitions of the core can be specified similarly by `c/11` rules. The first three arguments of `c/11` are similar to those of `r1/7` (`r2/7`). Pi is the temperature of the reactor core when the state is entered; while Po is the temperature at the time of transition. $Ti1, Ti2, To1$, and $To2$ are arguments that model two clocks in the core. The last argument of `c/11` is the flag that is used to determine which rod was used most recently. At state *no_rod*, when the core makes the transition `add1` or `add2`, the flag is used by the core to determine which clock to use to read the time at which the state *no_rod* is entered.

The rules for `r1/7` can be understood as follows: when the event `add1` occurs in location `out1`, a transition is made to location `in1`, if at least T units of time have elapsed since the

Table 7.1. The Logic Programming Realization of Transitions in *Rod1*, *Rod2*, and *Core*

```
r1(out1, add1,    in1,  W, Ti, To, T) :- {W - Ti >= T, To = Ti}.
r1(in1,  remove1, out1, W, Ti, To, T) :- {To = W}.

r2(out2, add2,    in2,  W, Ti, To, T) :- {W - Ti >= T, To = Ti}.
r2(in2,  remove2, out2, W, Ti, To, T) :- {To = W}.

c(norod, add1, rod1, Pi, Po, W, Ti1, Ti2, To1, To2, F) :-
  (F == 1 ->
           Ti = Ti1
          ;Ti = Ti2),
  {Pi < 550, Po = 550, exp(e, (W - Ti) / 10) = 5, To1 = W, To2 = Ti2}.

c(rod1, remove1, norod, Pi, Po, W, Ti1, Ti2, To1, To2, F) :-
  {Pi > 510, Po = 510, exp(e, (W - Ti1) / 10) = 5, To1 = W, To2 = Ti2}.

c(norod, add2, rod2, Pi, Po, W, Ti1, Ti2, To1, To2, F) :-
  (F == 1 ->
          Ti = Ti1
         ;Ti = Ti2),
  {Pi < 550, Po = 550, exp(e, (W - Ti) / 10) = 5, To1 = Ti1, To2 = W}.

c(rod2, remove2, norod, Pi, Po, W, Ti1, Ti2, To1, To2, F) :-
  {Pi > 510, Po = 510, exp(e, (W - Ti2) / 10) = 9/5, To1 = Ti1, To2 = W}.

c(norod, _, shutdown, Pi, Po, W, Ti1, Ti2, To1, To2, F) :-
  (F == 1 ->
          Ti = Ti1
         ;Ti = Ti2),
  {Pi < 550, Po = 550, exp(e, (W - Ti) / 10) = 5, To1 = Ti1, To2 = Ti2}.
```

resetting of r_1's clock. In location `in1`, when event `remove1` takes place, a transition is made to location `out1`, and the clock r_1 is reset. The resetting of the clock is denoted by setting the output argument To to W (the current wall clock time). The set of rules for `r2/7` and `c/11` can be understood similarly.

Once each hybrid automaton in the system is realized as a set of transition rules via logic programming, the coroutined, coinductive predicate `core/7`, realizes the core automaton, calling *c/11* repeatedly; while sending appropriate synchronization letters to rod_1 and rod_2. Coroutined, coinductive predicates `rod1/6` and `rod2/6` similarly call `r1/7` and `r2/7` respectively, on receiving these signals from the core. Note that Prolog's `freeze` builtin is used to model concurrency via coroutining.

Finally the `main/3` predicate represents the concurrent execution of all the components of the reactor control system. The first argument of `main/3` is the list of synchronization letters along with their time-stamps, which is generated by the core and sent to two rods (nondeterministically). The second argument is the initial wall clock time, and T is the parameter of the system explained earlier. The output of the system, S, is the timed trace of events, represented as a list of pairs in which the first element is the event and the second element is the time at which the event takes place, generated by the core. The condition that both rods are available initially, is specified using the set of constraints $\{W - Tr1 = T, W - Tr2 = T\}$.

Note that the coinductive termination of `core/7` will depend only on the first three arguments (signals sent, the state of *core* and the temperature); i.e., time, and flag will be ignored to check if `core` is cyclical. Similarly, the coinductive termination of `rod1/6` and `rod2/6` will depend on the first three arguments (signals received and states of rod1 and rod2); time and the parameter T will be ignored to check if `rod1`, and `rod2` are cyclical.

Once the entire system is modeled as a coinductive coroutined CLP(R) program, it can be used for finding the value of parameter T that guarantees the safety of the system. Calling `main`, with T as an unknown parameter, we obtain $T \leq 38.0666$, which is a necessary and sufficient condition on the parameter T that prevents the reactor from shutdown. The

Table 7.2. The Logic Programming Realization of Reactor Temperature Control System

```
:- coinductive(core(+,+,+,-,-,-,-)).
core(X, Si, Pi, W, Ti1, Ti2, Fi) :-
  (H = add1
  ;H = remove1
  ;H = add2
  ;H = remove2
  ;H = shutdown),
  {W2 > W},
  freeze(X,core(Xs, So, Po, W2, To1, To2, Fo)),
  c(Si, H, So, Pi, Po, W, Ti1, Ti2, To1, To2, Fi),
  ((H = add1
   ;H = remove1) -> Fo = 1
                 ;Fo = 2),
  ((H = add1
   ;H = remove1
   ;H = add2
   ;H=remove2) ->  X = [(H, W)| Xs]
                ;X = [(H, W)] ).

:- coinductive(rod1()+,+,+,-,-,-).
rod1([(H, W)| Xs], Si1, Si2, Ti1, Ti2, T) :-
  ((H = add1
   ;H = remove1) ->
         (H = add1 ->
              freeze(Xs, rod1(Xs, So1, Si2, To1, Ti2, T))
         ;    freeze(Xs, rod1(Xs, So1, Si2, To1, Ti2, T)
             ;           rod2(Xs, So1, Si2, To1, Ti2, T))),
  r1(Si1, H, So1, W, Ti1, To1, T)
  ;H = shutdown, {W - Ti1 < A, W - Ti2 < A}).

:- coinductive(rod2(+,+,+,-,-,-)).
rod2([(H, W)| Xs], Si1, Si2, Ti1, Ti2, T) :-
  ((H = add2
   ;H = remove2) ->
         (H = add2 ->
              freeze(Xs, rod2(Xs, Si1, So2, Ti1, To2, T))
         ;    freeze(Xs,rod1(Xs, Si1, So2, Ti1, To2, T)
             ;          rod2(Xs, Si1, So2, Ti1, To2, T))),
  r2(Si2, H, So2, T, Ti2, To2, T)
  ;H = shutdown, {W - Ti1 < A, W - Ti2 < A}).
```

Table 7.3. Concurrent Execution of *Core*, *Rode1*, and *Rod2*

```
main(S, W, T) :-
                {W - Tr1 = T, W - Tr2 = T},
                freeze(S, (rod1(S, s0, s0, Tr1, Tr2, T)
                         ;rod2(S, s0, s0, Tr1, Tr2, T))),
                core(S, s0, 510, W, Tc1, Tc2, 1).
```

Table 7.4. Verifying the Safety Property for The Reactor Temperature Control System

```
unsafe(S, W, T) :-
                   main(S, W, T),
                   member((shutdown, Ts), S).
```

verification requires 0.010 seconds on an Intel dual core 3.16 GHz processor with 4.00 GB of RAM. It is also worth noting that due to the elegance provided by logic programming, our entire model is expressed using approximately 120 lines of coinductive coroutined CLP(R) code.

Our logic programming realization of this system can also be used to verify interesting properties of the system by posing queries. Here we exploit the natural ability of logic/constraint programming systems to explore the entire state space of a program, merely by backtracking. To prove the *safety* property, we define the `unsafe/3` predicate which looks for an accepting timed trace that contains a *shutdown* event. Calling `unsafe/3` for any values of $W > 0$ and $T \leq 38.0666$ fails, which indicates the safety of the system.

It is interesting to note that using our system we were able to spot two errors in the paper of Urbina (Urbina, 1996). First, in the state *no_rod* of the hybrid automaton for the core, the temperature changes erroneously according to the differential equation $\dot{\theta} = \frac{\theta}{10} + 50$. Solving this differential equation results in $\theta(t) = 1010e^{(t/10)} - 500$. By setting $\theta(t) = 550$ in this equation, one can obtain $t = 0.388398$, which is the time that is spent by the core while in the state *no_rod* (the time that it takes for the reactor core to reach to the temperature 550, starting at temperature 510), which is obviously wrong. In contrast, the solution obtained by our system is $t = 16.0944$; this can be also verified if the equation is solved by hand. Second, the solutions for the differential equations in *no_rod*, rod_1, and rod_2 are given as $\theta_2 = \theta_1 - 500e^{(t_2-t_1)/10} + 500$, $\theta_2 = \theta_1 - 560e^{(t_2-t_1)/10} + 560$, and $\theta_2 = \theta_1 - 600e^{(t_2-t_1)/10} + 600$, respectively, where $t_2 \geq t_1$. Using these equations in our system resulted in no solutions. In other words these equations have solutions only when $t_2 < t_1$. To observe this, one can take for example the equation $\theta_2 = \theta_1 - 500e^{(t_2-t_1)/10} + 500$ for *no_rod*, and set $\theta_2 = 550$ and $\theta_1 = 510$ and verify that the solution for this equation will be $t_2 - t_1 = -0.833816$, which obviously does not satisfy the requirement $t_2 \geq t_1$.

Henzinger and Ho have also analyzed the reactor temperature control system with their symbolic model checker, HYTECH (Henzinger and hsin Ho, 1995). HYTECH analyzes a class of hybrid systems expressed via linear hybrid automata in which derivatives are constants. Since the hybrid automaton specifying the core in the reactor temperature control system is not linear, they have used two methods for converting it to a linear hybrid automaton. In the first method which is called the *rate translation*, they have approximated the derivative of the temperature in three locations of *core* by rate intervals of $[-5, -1]$, $[-9, -5]$ and $[1, 5]$ for locations rod_1, rod_2 and *no_rod*, respectively. Analyzing this converted hybrid automaton by HYTECH; they have reported $T < 20.44$ as a necessary and sufficient condition on the parameter T that prevents the reactor from shutdown. Using our system, we found out that this requirement on parameter T is indeed a sufficient condition, however, it is not a necessary condition, since for any value of T in which $T \leq 38.0666$ the system is still safe. Their second method for converting a nonlinear hybrid automaton to a linear hybrid automaton is called *clock translation* and it has two steps. In the first step, they have replaced the nonlinear variable of the core automaton by a clock; as a result θ is replaced by t_θ. In the second step, they have over approximated the resulting automaton by a linear automaton and obtained a linear hybrid automaton for the core that has the invariants $10t_\theta \leq 161$, $10t_\theta \leq 89$, and *true* for locations rod_1, rod_2 and *no_rod*, respectively. The original differential equations are also replaced by $\dot{t_\theta} = 1$ in all the locations of the core automaton. By analyzing this clock-translated core automaton in HYTECH, they have obtained $T < 37.8$ as a weaker condition on parameter T. This result is closer to the result generated by our system. However, these parameters are computed with exact precision in our approach, as our framework for modeling hybrid automata directly handles the physical quantities constrained by nonconstant derivatives, as long as the guards associated with transitions are of the form $x \sim a$, when $\sim \in \{=, <, >, \leq, \geq\}$, and a is a constant. In other words, we are not using any translation and approximation method for converting non-linear variables to linear variables; therefore, our system computes the parameter bounds exactly, and is free of possible imprecision that might be introduced if approximation methods are used. Note that one must use an appropriate solver in order to solve the set of constraints corresponding to the invariants, evolution laws, and guards on the transitions.

7.5 Conclusions and Related Work

In this chapter we presented a general framework for modeling/verifying cyber-physical systems. CPS are normally composed of set of processes that execute concurrently. The processes interact with each other through sending and receiving signals and run for ever. Communicative hybrid automata (Alur et al., 1992, 1995) constitute the foundations of cyber-physical systems. In our framework, hybrid automata are modeled as *logic programs* (Lloyd, 1987), physical quantities are faithfully represented as continuous quantities and the constraints imposed on them by CPS physical interactions are faithfully modeled with *constraint logic programming over reals.* By considering *coinductive logic programming* we are able to naturally model the non-terminating aspect of CPS. Finally, concurrency is handled by allowing *coroutining* within logic programming computations. As a result, CPS are modeled as coroutined, coinductive CLP(R) programs which can be used for verifying interesting properties of the system such as safety, utility, and liveness. Our approach can also be used to perform precise parametric analysis of CPS.

The reactor temperature control system has recently been adopted by researchers (R. A. Thacker et al., 2010) as an illustrative example in modeling and verifying CPS. Henzinger and Ho have analyzed this system with HYTECH; a symbolic model checker for linear hybrid automata (Henzinger and hsin Ho, 1995). They have used rate and clock translations for converting the nonlinear hybrid automaton to linear hybrid automaton. The parametric analysis of the reactor temperature control system by HYTECH, after rate translation, results in a very conservative constraint on parameter T. Alur et al. have presented a general framework for the formal specification and algorithmic analysis of hybrid systems (Alur et al., 1995). This work is also restricted to linear hybrid systems where all variables follow piecewise-linear trajectories. They have analyzed a variant of the reactor temperature control system using the KRONOS tool, another symbolic model checker for timed and hybrid automata. In the variant of the reactor system they have considered temperature rises and decreases at fixed rates, which is not faithful to the original problem. Back and Cerschi have modeled and verified a variant of the reactor temperature control system using continuous action systems (Cristina and Cerschi, 2000). In their variant of the system they have considered, again, temperature rises and decreases at fixed rates. The process of verifying safety property in their model is very lengthy and complex. They start by generating the state chart of the temperature control system to get a first approximation of the invariant for proving the safety property. Then, they keep adding information to the system's states in order to figure out an invariant strong enough to ensure safety.

We showed how the coinductive CLP(R)-based realization of hybrid automata along with coroutining can be used to elegantly and faithfully model the reactor temperature control system as well as verify its safety property. Our approach can handle non-linear variables directly, i.e., no approximation is needed. Using our model, we are able to determine the exact bound on the parameter T that guarantees the safety of the system. Our modeling of the system based on timed automata, hybrid automata, and coroutined coinductive CLP(R) is simpler, more elegant, and more precise than other approaches. In addition, unlike other realizations of timed-systems that discretize time, we treat time as a continuous quantity, and do not discretize it.

Timed concurrent constraint (TCC) programming (Saraswat et al., 1994) comes close to our work. Timed concurrent constraints have also been considered for verification (Falaschi and Villanueva, 2006). TCC does not consider perpetual computations and works only with least fixpoints of programs. Our work can be regarded as a practical realization of TCC as well as its extension with co-induction to handle perpetual computations.

To conclude, a combination of constraints over reals, coinduction, and coroutining provides an expressive, and easy-to-use formalism for modeling and analyzing cyber-physical systems modeled as communicative hybrid and timed automata. In fact, our framework is a general framework that can be applied to complex systems to handle any continuous quantity such as time, temperature, distance, pressure, etc. The logic programming based approach is simpler and more elegant than other approaches that have been proposed for this purpose. It is also more precise.

CHAPTER 8
CONCLUSIONS

We proposed a framework for modeling and verification of cyber-physical systems (including real-time systems and hybrid systems), a framework based on logic programming extended with constraints and coinduction. In doing so, we proposed a new programming paradigm, co-CLP, that merges constraint logic programming and co-LP. We proposed a new declarative semantics for co-CLP which differs from that of both constraint logic programming and coinductive logic programming. The traditional constraint logic programming (Jaffar and Maher, 1994), on the one hand, cannot handle infinite structures. The only domain considered in CLP is the domain of constraints; therefore, defining new function symbols and terms is not allowed. As a result, the terms and atoms that are used in a constraint logic program are limited to what is predefined in the structure, and they are interpreted in the domain of constraints. We allow not only the domain of constraints, but also the Herbrand universe. So, user-defined terms will be allowed. User-defined functions and predicates are given the standard interpretation in the Herbrand universe and the Herbrand base. However, constraints are interpreted using a predefined interpretation (provided by the domain of computation). On the other hand, the declarative semantics of co-LP does not allow constraints. The operational semantics proposed for co-CLP is based on the greatest fixed-point semantics and differs from that of constraint logic programming, as the operational semantics of CLP is based on the least fixed-point semantics. This new operational semantics takes constraints into account. Note that the operational semantics of co-LP cannot handle constraints.

We proposed techniques for including continuous time and other physical continuous quantities in computation.

These techniques include, first, a LP-based framework for modeling and verification of real-time systems. We employed our proposed co-CLP paradigm to develop a framework for modeling timed automata and pushdown timed automata, and used this logic programming realization to develop a framework for modeling and verification of real-time systems.

Second, we proposed timed ω-grammars as a simple and natural way of expressing real-time systems. The timed language recognized by the timed grammar expressing a real-time system is used to model the system as a timed trace of events in which the set of events in the system constitute the alphabet of the language. We used the co-CLP paradigm to develop effective and practical parsers for timed ω-languages.

Third, we extended the π-calculus with real-time and proposed an operational semantics, as well as a notion of timed bisimilarity for the resulting calculus, timed π-calculus. Timed π-calculus is an expressive language for describing concurrent, mobile, real-time systems. The replication operator of timed π-calculus (and also π-calculus) is used to express infinite computations and infinite behavior of systems. We employed co-CLP with the coroutining facility of logic programming to develop a framework for implementing the operational semantics of timed π-calculus. This logic program can be regarded as an interpreter for timed π-calculus expressions, and can be used for modeling and verification of real-time systems.

Finally, we identified four fundamental characteristics of cyber-physical systems, namely, performing discrete computations, dealing with physical continuous quantities, running forever, and being concurrent. We proposed a co-CLP-based framework for modeling and verification of CPS, in which discrete computation is handled in logic programming itself, physical continuous quantities are modeled within constraint logic programming, infinite behavior is modeled in coinductive logic programming, and the concurrent nature of CPS is handled by allowing coroutining within logic programming computations. This LP-based realization of CPS can be used for verifying properties of CPS.

In this dissertation we extended the operational semantics of co-CLP, in order to develop a practical, effective framework for modeling timed automata/PTA and real-time systems. However, we have not worked out the declarative semantics of this extension. As part of the future work, we would like to develop the declarative semantics of the extension of co-CLP, described in Section 4.

Including negation in co-CLP also constitutes part of our future work. We would like to extend the declarative and the operational semantics of co-CLP so that negative literals in the head and the body of clauses of a co-constraint logic program are allowed and have a well-defined meaning.

Future work also includes developing a fully automated interpreter for timed π-calculus based on logic programming and making it available to other researchers. We also plan to extend the timed π-calculus to include other continuous quantities and use it for modeling and reasoning about cyber-physical systems.

APPENDIX A
PROOF OF CORRECTNESS OF COINDUCTIVE CONSTRAINT LOGIC PROGRAMMING

Correctness of coinductive CLP and co-CLP is presented by proving the equivalence of the declarative semantics and operational semantics via soundness and completeness theorems. The correctness results presented here is restricted to those elements of the model that have idealized proofs. The proofs are straightforward extension of the proofs for correctness of coinductive LP and co-LP (Simon, 2006).

Lemma A.0.1. *If (a, E_1, C_1) has a successful derivation in a coinductive constraint logic program P over the constraint domain $\mathcal{D}$, with final state $(\emptyset, E_2, C_2)$, then (a, E_3, C_3) has a successful derivation in program P, where $E_2 \subseteq E_3$, $C_3 \models C_2$ and $\mathcal{D} \models C_3$.*

Proof: follows by the fact that in the sequence of states in a derivation, the system of equations monotonically increases, also C_3 is consistent and $C_3 \models C_2$.

Lemma A.0.2. *If atom a has a successful derivation in a coinductive constraint logic program P over the constraint domain $\mathcal{D}$, which* first *uses the clause $a' \leftarrow c, b_1, \ldots, b_n$, and transitions to a state $(node(a, [b_1, \ldots, b_n]), E, C)$, such that $E(a) = E(a')$ and $\mathcal{D} \models E(C) = E(c)$, then each (b_i, E, C) also has a successful derivation in program P.*

Proof: Assume T is the successful derivation tree for (a, E, C). A derivation for (b_i, E, C) can be obtained by taking each transition that modifies the subtree rooted at b_i in T. However, for transitions obtained by applying a coinductive hypothesis rule of the form $\eta(\pi, \pi')$, this will not work, since the parent a of b_i no longer exists. We consider two cases: (i) if $\pi = i.\pi_0$ and $\pi' = i.\pi_1$ for some integer i and paths π_0 and π_1, then we apply the transition rule $\eta(\pi_0, \pi_1)$ to the corresponding leaf to which the original derivation have applied $\eta(\pi, \pi')$. (ii) if $\pi = \epsilon$, a coinductive hypothesis cannot be applied to the corresponding leaf; however, the transitions of the entire original derivation of a can be recursively taken.

Lemma A.0.3. *If (a, E, C) has a successful derivation in a coinductive constraint logic program P over the constraint domain $\mathcal{D}$, which ends at state $(\emptyset, E', C')$, then $E'(E(a))$ is true in program P if $\mathcal{D} \models E'(E(C'))$.*

Proof: Let Q be the set of all groundings over the U_P of $E'(E(a))$ (note that the Herbrand universe contains the domain of computation D) such that $\mathcal{D} \models E'(E(C'))$. We show that $Q \subseteq gm(P, \mathcal{D})$.

Let $a' \in Q$, then there exist substitutions E_1, E_2, E_3, and set of consistent constraints C_2, C_3 such that $a' = E_1(E_2(E_3(a)))$, where E_1 is a grounding substitution for $E_2(E_3(a))$, (a, E_3, C_3) has a successful derivation ending in $(\emptyset, E_2, C_2)$, and $\mathcal{D} \models C_2$, $C_3 \models C_2$. Now assume $E = E_1 \cup E_2 \cup E_3$ and $C = C_2 \cup C_3$. By lemma A.0.1, (a, E, C) has a successful derivation. This derivation must begin with application of a definite clause rule of the form $a'' \leftarrow c, b_1, \ldots, b_n$, and result in the state $(node(a, [b_1, \ldots, b_n]), E, C)$, where $E(a) = E(a''), \mathcal{D} \models E(C) = E(c)$. By lemma A.0.2, each state (b_i, E, C) has a successful derivation. Let E' be a grounding substitution for the clause $E(a'' \leftarrow c, b_1, \ldots, b_n)$ and $C' = E(C)$, then $\mathcal{D} \models C'$. Let $E^* = E'(E(a'' \leftarrow c, b_1, \ldots, b_n))$, $E'(E(a'')) = a'$ and $E'' = E' \cup E$, then, $E^* = a' \leftarrow E''(C'), E''(b_1), \ldots, E''(b_n)$, $\mathcal{D} \models E''(C') = C''$. By lemma A.0.1, each (b_i, E'', C'') has a successful derivation, hence $E''(b_i) \in Q$. Therefore, $Q \subseteq gm(P, \mathcal{D})$.

Proof of Theorem 3.3.1: The proof follows by straightforward instantiation of lemma A.0.3.

Proof of Theorem 3.3.2: Let $a \in gm(P, \mathcal{D})$ have a rational idealized proof T. The derivation is constructed by a depth-first traversal of the idealized proof tree and recursively applying the clause corresponding to each node encountered to the corresponding leaf in the current state. The traversal stops at the root R of a subtree that is identical to a subtree rooted at a proper ancestor at depth n. Then the derivation applies a transition rule of the form $\eta(\pi, \pi')$ to the leaf corresponding to R in the current state, and finally the depth-first traversal continues traversing starting at a node in the idealized proof tree corresponding to some leaf in the current state of the derivation. The set of all subtrees of T is finite in cardinality. This implies that the maximum depth of the traversal in the idealized proof is finite. Moreover, all idealized proofs are finitely branching. This implies that the traversal always terminates. Therefore, the constructed derivation is finite.

We need to prove that the final state of the constructed derivation is a success state. We consider two cases that the traversal stops going deeper in the idealized proof tree: (i) the traversal reaches a leaf in the idealized proof tree, (ii) the traversal encounters a subtree identical to an ancestor subtree. In both cases, the derivation removes the corresponding leaf in the current state and maximum number of its ancestors such that the result is still a tree. Therefore, a leaf remains in the state only when its corresponding node in the proof tree has yet to be proved. Since every node in the idealized proof tree corresponding to a leaf in the state is traversed at some point, the final state's tree contains no leaves, and hence the final state has an empty tree, which is the definition of an accept state. Therefore, a has a successful derivation in program P.

Lemma A.0.4. *If (a, E_1, C_1) has a successful derivation in a co-constraint logic program P over the domain of constraints $\mathcal{D}$, with final state $(\emptyset, E_2, C_2)$, then (a, E_3, C_3) has a successful derivation in program P, where $\mathcal{D} \models C_3$, $E_2 \subseteq E_3, C_3 \models C_2$.*

Proof: follows by the fact that in the sequence of states in a derivation, the system of equations monotonically increases, also C_3 is consistent and $C_3 \models C_2$.

Lemma A.0.5. *If atom a has a successful derivation in a co-constraint logic program P over the domain of constraints $\mathcal{D}$, such that the* first *transition of this derivation uses the clause $a' \leftarrow c, b_1, \ldots, b_n$, the substitution obtained after this transition is E and the set of constraints is C, such that $E(a) = E(a')$ and $\mathcal{D} \models E(C) = E(c)$, then each (b_i, E, C) also has a successful derivation in program P.*

Proof: Assume T is the successful derivation tree for (a, E, C). A derivation for (b_i, E, C) can be obtained by taking each transition that modifies the subtree rooted at b_i in T.

However, for transitions obtained by applying a coinductive hypothesis rule of the form $\eta(\pi, \pi')$, this will not work, since the parent a of b_i no longer exists. We consider two cases: (i) if $\pi = i.\pi_0$ and $\pi' = i.\pi_1$ for some integer i and paths π_0 and π_1, then we apply the transition rule $\eta(\pi_0, \pi_1)$ to the corresponding leaf to which the original derivation have applied $\eta(\pi, \pi')$, (ii) if $\pi = \epsilon$, a coinductive hypothesis cannot be applied to the corresponding leaf; however, the transitions of the entire original derivation of a can be recursively taken.

Lemma A.0.6. *If (a, E, C) has a successful derivation in a co-constraint logic program P over the constraint domain $\mathcal{D}$, which ends at state $(\emptyset, E', C')$, then $E'(E(a))$ is true in program P if $\mathcal{D} \models E'(E(C'))$.*

Proof: The model of P consists of the union of the models of the strata of P. We show that if (a, E, C) has a successful derivation in program P ending at $(\emptyset, E', C')$, then all groundings of $E'(E(a))$ such that $\mathcal{D} \models E'(E(C'))$ are included in the model for the stratum in which a resides.

The proof proceeds by induction on the height of strata of P. Let Q_u be the set of all groundings over the U_P of $E'(E(a))$ (note that the Herbrand universe contains the domain of constraints $\mathcal{D}$) that satisfy $E'(E(C'))$ which are either in the same stratum u or some lower stratum. We prove by induction on the height of u that Q_u is contained in the model for u. We consider two cases:

(i) the stratum u is coinductive. We show that $Q_u \subseteq \nu T^{\mathcal{D}}_{u,P}$. Let $a' \in Q_u$, then there exist substitutions E_1, E_2, E_3, and set of consistent constraints C_2, C_3 such that $a' = E_1(E_2(E_3(a)))$, where E_1 is a grounding substitution for $E_2(E_3(a))$, (a, E_3, C_3) has a successful derivation ending in $(\emptyset, E_2, C_2)$, and $\mathcal{D} \models C_2$, $C_3 \models C_2$. Now assume $E = E_1 \cup E_2 \cup E_3$ and $C = C_2 \cup C_3$. By lemma A.0.4, (a, E, C) has a successful derivation. This derivation must begin with application of a definite clause rule of the form $a'' \leftarrow c, b_1, \ldots, b_n$, and result in the state $(node(a, [b_1, \ldots, b_n]), E, C)$, where $E(a) = E(a''), \mathcal{D} \models E(C) = E(c)$. By lemma A.0.5, each state (b_i, E, C) has a successful derivation. Let E' be a grounding substitution for the clause $E(a'' \leftarrow c, b_1, \ldots, b_n)$ and $C' = E(C)$, then $\mathcal{D} \models C'$. Let $E^* = E'(E(a'' \leftarrow$

$c, b_1, \ldots, b_n))$, $E'(E(a'')) = a'$ and $E'' = E' \cup E$, then, $E^* = a' \leftarrow E''(C'), E''(b_1), \ldots, E''(b_n)$, $\mathcal{D} \models E''(C') = C''$. By lemma A.0.4, each (b_i, E'', C'') has a successful derivation and and the stratification restriction implies that each $E''(b_i)$ is in a stratum equal to or lower than u. Hence $E''(b_i) \in Q_u$. Therefore, $Q_u \subseteq \nu T^{\mathcal{D}}_{u,P}$.

(ii) the stratum u is inductive. We show that $Q_u \subseteq \mu T^{\mathcal{D}}_{u,P}$. Let $a' \in Q_u$, then there exist substitutions E_1, E_2, E_3, and set of consistent constraints C_2, C_3 such that $a' = E_1(E_2(E_3(a)))$, where E_1 is a grounding substitution for $E_2(E_3(a))$, (a, E_3, C_3) has a successful derivation ending in $(\emptyset, E_2, C_2)$, and $\mathcal{D} \models C_2$, $C_3 \models C_2$. Now assume $E = E_1 \cup E_2 \cup E_3$ and $C = C_2 \cup C_3$. By lemma A.0.4 (a, E, C) has a successful derivation. We consider two cases: (i) if a' is in a lower stratum than u, then it follows by induction, (ii) a' does not occur in a stratum lower than u, i.e., a' is an inductive atom. The proof proceeds by induction on the length of the derivation a'. If the derivation consists of just one transition, then a' must unify with a fact $a \leftarrow c$ in program P such that $\mathcal{D} \models E(C) = E(c)$, therefore $a' \in \mu T^{\mathcal{D}}_{u,P}$, if $\mathcal{D} \models c$. If the derivation is of length $k > 1$, then the derivation begins with an application of a program clause $a'' \leftarrow c, b_1, \ldots, b_n$, where $E(a) = E(a''), \mathcal{D} \models E(C) = E(c)$. By lemma A.0.5 each state (b_i, E, C) has a successful derivation. Let E' be a grounding substitution for the clause $E(a'' \leftarrow c, b_1, \ldots, b_n)$ and $C' = E(C)$, then $\mathcal{D} \models C'$. Let $E^* = E'(E(a'' \leftarrow c, b_1, \ldots, b_n))$, $E'(E(a'')) = a'$ and $E'' = E' \cup E$, then, $E^* = a' \leftarrow E''(C'), E''(b_1), \ldots, E''(b_n)$, $\mathcal{D} \models E''(C') = C''$. By lemma A.0.4, each (b_i, E'', C'') has a successful derivation of length $k' < k$, and the stratification restriction implies that each $E''(b_i)$ is in a stratum equal to or lower than u. We consider two cases: (i) $E''(b_i)$ is in a stratum strictly lower than u, then by induction on strata $E''(b_i) \in \mu T^{\mathcal{D}}_{u,P}$. (ii) $E''(b_i)$ is not in a stratum lower than u, then since it has a derivation of length $k' < k$, by induction on the length of the derivation, $E''(b_i) \in \mu T^{\mathcal{D}}_{u,P}$. Therefore, $a' \in \mu T^{\mathcal{D}}_{u,P}$.

Proof of Theorem 3.4.1: If the goal $a_1, \ldots, a_n$ has a successful derivation in program P ending at state $(\emptyset, E, C)$, then each $E(a_i)$ independently has a successful derivation in program P if $\mathcal{D} \models E(C)$. By lemma A.0.6, each $E(a_i)$ is true in program P.

Proof of Theorem 3.4.2: We only consider the case when $n = 1$, i.e., the query is a single atom, since the case for $n = 0$ is trivial and the case for $n > 1$ follows by composing the individual derivations of each atom of the original query. Let $a \in M^{\mathcal{D}}(P)$ have a rational idealized proof T. The derivation is constructed by a depth-first traversal of the idealized proof tree and recursively applying the clause corresponding to each node encountered to the corresponding leaf in the current state. The traversal stops at the coinductive root $\pi = \pi'.\pi''$ of a subtree that is identical to a subtree rooted at a proper coinductive ancestor π'. Then the derivation applies a transition rule of the form $\eta(\pi', \pi)$ to the leaf corresponding to R in the current state, and finally the depth-first traversal continues traversing starting at a node in the idealized proof tree corresponding to some leaf in the current state of the derivation.

The set of all subtrees of T is finite in cardinality. The stratification restriction prevents a depth-first traversal from encountering the same subtree twice along the same path, if the subtree has an inductive atom at its root. Only subtrees rooted at coinductive atoms can repeat in such a fashion. These imply that the maximum depth of the traversal in the idealized proof is finite. Moreover, all idealized proofs are finitely branching. This implies that the traversal always terminates. Therefore, the constructed derivation is finite.

We need to prove that the final state of the constructed derivation is a success state. We consider two cases that the traversal stops going deeper in the idealized proof tree: (i) the traversal reaches a leaf in the idealized proof tree, (ii) the traversal encounters a subtree identical to an ancestor subtree. In both cases, the derivation removes the corresponding leaf in the current state and maximum number of its ancestors such that the result is still a tree. Therefore, a leaf remains in the state only when its corresponding node in the proof tree has yet to be traversed. Since every node in the idealized proof tree corresponding to a leaf in the state is traversed at some point, the final state's tree contains no leaves, and hence the final state has an empty forest, which is the definition of an accepting state. Therefore, a has a successful derivation in program P.

APPENDIX B
THE GENERAL METHOD OF CONVERTING TIMED AUTOMATA/PTA TO CO-CLP(R) PROGRAMS

Table B.1. The General Method of Converting Timed Automata/PTA to Co-CLP(R)

```
convert_pta :-
 transitions(Transitions),
 clocks(Clocks),
 create_clocks(Clocks, [], [], In, Out),
 open('file', append, Id),
 assert_transitions(Transitions, Clocks, In, Out, Id).

assert_transitions([H|T], Clocks, In, Out, Id) :-
 assert_trans(H, Clocks, In, Out, Id), nl(Id),
 assert_transitions(T, Clocks, In, Out, Id).

assert_transitions([], _Clocks, _In, _Out, Id) :- close(Id).

assert_trans((S1, Inp, S2, Act, Resets, Consts), Clocks, In, out, Id) :-
 assert_constraints(Consts, Resets, Clocks, Body),
 W = 'W', C = 'C',
 (Act = 'push(1)' ->
      Rule = ':-'(trans(S1, Inp, S2, W, In, Out, C, [ 1 | C ]), Body)
     ;Rule = ':-'(trans(S1, Inp, S2, W, In, Out, [ 1 | C ], C), Body)),
 write(Id, Rule).

assert_trans((S1, Inp, S2, Resets, Consts), Clocks, In, Out, Id) :-
 assert_constraints(Consts, Resets, Clocks, Body),
 W = 'W',
 Rule = ':-'(trans(S1, Inp, S2, W, In, Out), Body),
 write(Id, Rule).
```

Table B.2. ...The General Method of Converting Timed Automata/PTA to Co-CLP(R), Continued

```
assert_constraints([Const | Consts], Resets, Clocks, ','(A, As) ) :-
 atom_chars(Const, [Clock | Rest]),
 to_upper(Clock, Clock2),
 atom_char(ClockVar, Clock2),
 append(['{', ' ', 'W', ' ', -, ' ', ClockVar, 'i'], Rest, R1),
 append(R1, ['}'], R2),
 atom_chars(A, R2),
 assert_constraints(Consts, Resets, Clocks, As).

 assert_constraints([], Resets, Clocks, Consts) :-
 reset_clocks(Resets, Clocks, Consts).

reset_clocks(Resets, [Clock | Clocks], ','(A, As)) :-
 to_upper(Clock, Clock2),
 atom_char(ClockVar, Clock2),
 (member(Clock, Resets) ->
       atom_chars(A, ['{', ClockVar, 'o', =, 'W', '}'])
      ;atom_chars(A, ['{', ClockVar, 'o', =, ClockVar, 'i', '}'])),
 reset_clocks(Resets, Clocks, As).

reset_clocks(Resets, [Clock], A) :-
 to_upper(Clock, Clock2),
 atom_char(ClockVar, Clock2),
 (member(Clock, Resets) ->
       atom_chars(A, ['{', ClockVar, 'o', =, 'W', '}', '.'])
      ;atom_chars(A, ['{', ClockVar, 'o', =, ClockVar, 'i', '}', '.'])).
```

Table B.3. ...The General Method of Converting Timed Automata/PTA to Co-CLP(R), Continued

```
create_clocks([Clock | Clocks], P1, P2, In, Out) :-
 to_upper(Clock, Clock2),
 atom_char(ClockVar, Clock2),
 atom_chars(ClockIn, [ClockVar, 'i']),
 atom_chars(ClockOut, [ClockVar, 'o']),
 append(P1, [ClockIn], P11),
 append(P2, [ClockOut], P22),
 create_clocks(Clocks, P11, P22, In, Out).

create_clocks([], P1, P2, P1, P2).

create_driver(Id) :-
 StateIn = 'Si', StateOut = 'So',
 Inp1 = 'X',     InpRest = 'R',
 Wall = 'W',     WallNext = 'Wn',
 ClockIn = 'Ti', ClockOut = 'To',
 StackIn = 'Ci', StackOut = 'Co',
 OutputRest = 'S',
 atom_chars(Output, ['[', '(', Inp1, ',', Wall, ')', '|', OutputRest]),
 atom_chars(Inp, ['[', Inp1, '|', InpRest, ']']),
 Body1 = trans(StateIn, Inp1, StateOut, Wall, ClockIn, ClockOut, StackIn,
              StackOut),
 atom_chars(ClockAdvance, ['{', Wall, 'n', >, Wall, '}']),
 Body2 = driver(InpRest, StateOut, WallNext, ClockOut, StackOut,
               OutputRest),
 Body = ','(Body1, ','(ClockAdvance, Body2)),
 Rule = ':-'(driver(Inp, StateIn, Wall, ClockIn, StackIn, Output), Body),
 write(Id, Rule).
```

APPENDIX C
EQUIVALENCE OF PUSHDOWN TIMED AUTOMATA AND TIMED CONTEXT-FREE GRAMMARS

Theorem C.0.7. *A language is timed context-free if and only if some pushdown timed automaton recognizes it.*

Lemma C.0.8. *If a language is timed context-free, then some pushdown timed automaton recognizes it.*

Let A be a timed context-free language, then there is a timed context-free grammar G, generating it. We show how to convert G into an equivalent pushdown timed automaton, which we call P. We construct a pushdown timed automaton P to simulate leftmost derivations in G. Therefore, transitions of P will correspond to productions in G; and, computations of P will simulate leftmost derivations in G.

We give the formal details of the construction of the pushdown timed automaton P. Lets say that $P = (\Sigma, \Gamma, \delta, Q, Q_0, F, C)$, in which Σ is a finite alphabet, Γ is the stack alphabet including ϵ, δ is the set of transitions, Q is the (finite) set of states, Q_0 is the set of start states, F is the set of final state, and C is the set of clocks. To make the construction clearer we use shorthand notation for the transition function. This notation enables us to write an entire string on the stack in one step of the machine.

Let q and r be states of the pushdown timed automaton, with a clock expression e associated with the transition from q to r; and let a be in Σ_ϵ, t_a be the time-stamp of a, s be in Γ_ϵ. We want the pushdown timed automaton to go from q to r when it reads a and pops s. Furthermore we want it to push the entire string $u = u_1 \dots u_l$ on the stack at the same time. We implement this action by introducing new states $q_l, \dots, q_{l-1}$ and setting the transition function as follows

$$\delta(q,(a,t_a),s) \text{ to contain } (q1,e,u_l),$$
$$\delta(q_1,\epsilon,\epsilon) = \{(q_2,\epsilon,u_{l-1})\},$$
$$\delta(q_2,\epsilon,\epsilon) = \{(q_3,\epsilon,u_{l-2})\},$$
$$.$$
$$.$$
$$.$$
$$\delta(q_{l-1},\epsilon,\epsilon) = \{(r,\epsilon,u_1)\}.$$

We use the notation $(r,e,u) \in \delta(q,(a,t_a),s)$, which can be understood as follows: if q is the state of the automaton, a is the next input symbol with its time-stamp t_a, and s is the symbol on the top of the stack, the pushdown timed automaton may read the a and pop the s, then push the string u onto the stack and go on to the state r; while e is the (possibly empty) clock expression associated with this transition. We use ϵ for the empty clock expression. Figure C.1 shows this implementation.

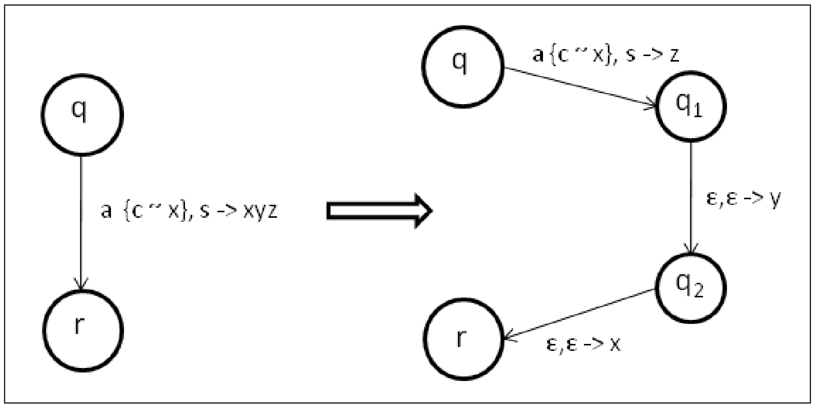

Figure C.1. Implementing the shorthand $(r,\{c \sim x\},xyz) \in \delta(q,(a,t_a),s)$

The states of P are $Q = \{q_{start}, q_{loop}, q_{accept}\} \cup E$, where E is the set of new states we need for implementing the shorthand. q_{start} is the start state and q_{accept} is the only accept state.

The transition function is defined as follows. First, we initialize the stack to contain the symbols \$, and the start variable S: $\delta(q_{start}, \epsilon, \epsilon) = (q_{loop}, \epsilon, S\$)$. Then we put in transitions as follows. We consider five cases:

Case I: The top of the stack contains a variable A without any clock constraint attached to it. Let $\delta(q_{loop}, \epsilon, A) = \{(q_{loop}, \epsilon, w)|$ where $A \mapsto w$ is a rule in $R\}$.

Case II: The top of the stack contains a variable A with the clock expression e associated with it. Let $A \mapsto w$ be a production rule in grammar G. We consider two subcases as follows.

Case II-a: w does not contain any terminal symbol. Let $\delta(q_{loop}, \epsilon, Ae) = \{(q_{loop}, \epsilon, we)\}$.

Case II-b: w contains at least one terminal symbol. Then w is of the form $w_1 T w_2$ in which T is a terminal symbol, and w_1 and w_2 are two possibly empty strings such that w_2 does not contain any terminal symbol. Let $\delta(q_{loop}, \epsilon, Ae) = \{(q_{loop}, \epsilon, w_1 T e w_2)\}$. In other words, the clock expression attached to a variable symbol A, corresponds to the last terminal symbol in the string that is reduced to A.

Note that if two sets of clock expressions, γ and σ, appear next to each other during a transition, they are replaced by $\gamma \cup \sigma$.

Case III: The top of the stack contains a terminal a which doesn't have any constraint associated with it. Let $\delta(q_{loop}, (a, t_a), a) = \{(q_{loop}, \epsilon, \epsilon)\}$.

Case IV: The top of the stack contains a terminal a with a clock constraint e associated with it. Let $\delta(q_{loop}, (a, t_a), ae) = \{(q_{loop}, e, \epsilon)\}$.

Case V: The empty stack marker \$ is on the top of the stack. Let $\delta(q_{loop}, \epsilon, \$) = \{(q_{accept}, \epsilon, \epsilon)\}$.

The state diagram of P is shown in Figure C.2. This completes the proof of Lemma C.0.8.

Lemma C.0.9. *If a pushdown timed automaton recognizes some language, then it is timed context-free.*

Given a pushdown timed automaton P, we want to construct a timed CFG G that generates all the strings that P accepts. For each pair of states p and q in P, G will have a variable

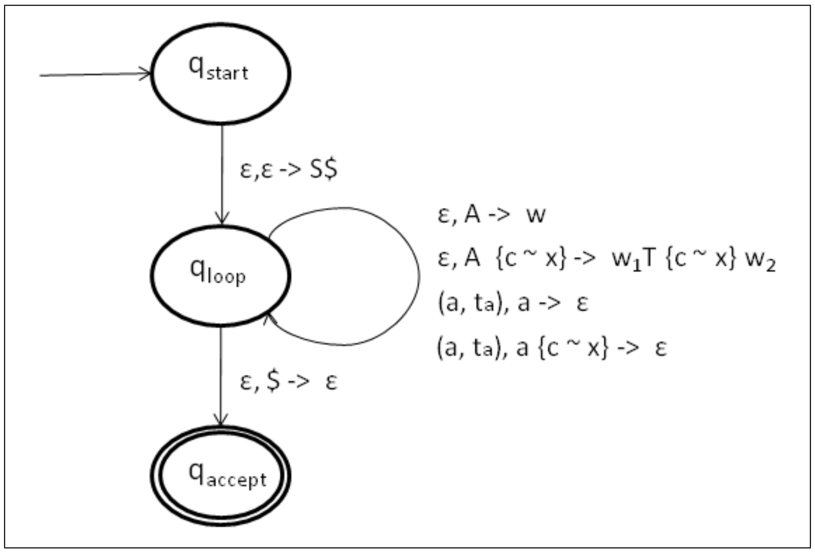

Figure C.2. State diagram of P

(non-terminal symbol), A_{pq}. This variable will generate all the strings that can take P from p with an empty stack to q with an empty stack. We would like to simulate the computations on P by derivations in G. To facilitate discussions we normalize transitions/computations in P such that:

1. P has a single final state q_{accept}
2. P accepts with an empty stack
3. Every move of P is either a push operation ($\epsilon \mapsto x, x \in \Gamma$), or a pop operation ($x \mapsto \epsilon, x \in \Gamma$).

To get feature 1, we create a distinguished final state q_{accept}, and add an ϵ transition from each original final state q to q_{accept} ($(q_{accept}, \epsilon, \epsilon) \in \delta(q, \epsilon, \epsilon)$). In q_{accept}, P simply pops any stack symbol, i.e., $(q_{accept}, \epsilon, \epsilon) \in \delta(q_{accept}, \epsilon, x)$. To obtain feature 3, we replace each transition that involve both push and pop with a two transition sequence that goes through a new state. We also replace each transition that neither pops nor pushes with a two transition sequence that pushes then pops an arbitrary stack symbol. We substitute every transition $(r, e, z) \in \delta(q, (a, t_a), x)$, with two transitions $(s, e, \epsilon) \in \delta(q, (a, t_a), x)$ and $(r, \epsilon, z) \in \delta(s, \epsilon, \epsilon)$, and every transition $(r, e, \epsilon) \in \delta(q, (a, t_a), \epsilon)$, with two transitions $(s, e, y) \in \delta(q, (a, t_a), \epsilon)$ and $(r, \epsilon, \epsilon) \in \delta(s, \epsilon, y)$.

Assume that $P = (\Sigma, \Gamma, \delta, Q, q_0, q_{accept}, C)$. We construct G as follows. The variables of G are $\{A_{pq} | p, q \in Q\}$. The start variable is $A_{q_0 q_{accept}}$. The grammar G's rules are constructed as follows.

- For each $p, q, r, s \in Q, t \in \Gamma$, $a, b \in \Sigma_\epsilon$, and e_1 and e_2 being clock expressions, if $\delta(p, (a, t_a), \epsilon)$ contains (r, e_1, t) and $\delta(s, (b, t_b), t)$ contains (q, e_2, ϵ), put the rule $A_{pq} \mapsto ae_1 A_{rs} be_2$ in G. Note that if there is no clock expressions associated with transitions, they can be omitted from the grammars.
- For each $p, q, r \in Q$, put the rule $A_{pq} \mapsto A_{pr} A_{rq}$ in G.
- For each $p \in Q$, put the rule $A_{pp} \mapsto \epsilon$ in G.

Next we prove that A_{pq} generates string x iff x can bring P from p with empty stack to q with empty stack.

Claim C.0.1. *If A_{pq} generates x, then x can bring P from p with empty stack to q with empty stack.*

Proof. We prove this by induction on the number of steps in the derivation of x starting from A_{pq}.

Basis: The derivation has one step.

The rule that was used in the derivation has no variables in its RHS. Therefore, the rule is of the form: $A_{pp} \mapsto \epsilon$. Clearly, the claim holds here.

Induction step: Assume that the claim is true for derivations of length at most $k, k \geq 1$. We prove that it is true for derivations of length $k + 1$.

Suppose that $A_{pq} \overset{*}{\Rightarrow} x$ with $k + 1$ steps. We consider two cases:

Case I: The first step in this derivation is $A_{pq} \Rightarrow ae_1 A_{rs} be_2$. Consider the portion y of x that A_{rs} generates, so $x = (a, t_a)y(b, t_b)$. Because $A_{rs} \overset{*}{\Rightarrow} y$ with k steps, P can go from r

on empty stack to s on empty stack (by induction hypothesis). Since $A_{pq} \mapsto ae_1A_{rs}be_2$ is a rule in G, $\delta(p,(a,t_a),\epsilon)$ contains (r,e_1,t) and $\delta(s,(b,t_b),t)$ contains (q,e_2,ϵ), where t is some stack symbol. Therefore, if P starts at p with an empty stack, it can go to state r and push t onto the stack, after reading (a,t_a). After reading string y, P will go to state s and leave t on the stack. After reading (b,t_b) it can go to state q and pop t off the stack. Therefore x can bring it from p with empty stack to q with empty stack.

Case II: The first step of derivation is $A_{pq} \Rightarrow A_{pr}A_{rq}$. Consider the portions y and z of x that are generated by A_{pr} and A_{rq}, respectively; so $x = yz$. Since $A_{pr} \overset{*}{\Rightarrow} y$ in at most k steps and $A_{rq} \overset{*}{\Rightarrow} z$ in at most k steps, by induction hypothesis y can bring P from p with empty stack to r with empty stack. Similarly, z can bring P from r with empty stack to q with empty stack. Therefore, x can bring it from p with empty stack to q with empty stack. □

Claim C.0.2. *If x can bring P from p with empty stack to q with empty stack, A_{pq} generates x.*

Proof. We prove this by induction on the number of steps in the computation of P that goes from p with empty stack to q with empty stack, on input string x.

Basis: The computation has zero steps.

The computation starts and ends at the same state, say p. We must show that $A_{pp} \overset{*}{\Rightarrow} x$. Since there are only zero steps, $x = \epsilon$. By construction, G has the rule $A_{pp} \mapsto \epsilon$. This proofs the basis.

Induction step: Assume that the claim is true for computations of length at most $k, k \geq 0$. We prove that it is true for computations of length $k+1$.

Suppose that P has a computation in which, x brings p with empty stack to q with empty stack in $k+1$ steps. We consider two cases:

Case I: The stack is empty only at the beginning and end of the computation. Assume that t is the stack symbol that is pushed into the stack at the first move. Then, t must be the symbol that is popped at the last move. Let (a,t_a) and (b,t_b) be the input reads in the first and last moves, respectively. Furthermore, let r be the state after the first move,

e_1 be the clock expression that is associated with this move, s be the state before the last move, and e_2 be the clock expression associated with this move. Then $\delta(p, (a, t_a), \epsilon)$ contains (r, e_1, t) and $\delta(s, (b, t_b), t)$ contains (q, e_2, ϵ). Therefore, the rule $A_{pq} \mapsto ae_1A_{rs}be_2$ is in G.

Let y be the portion of x without a and b (and clock expressions attached to them), so $x = ae_1ybe_2$. Input y can bring P from r to s without touching the symbol t that is on the stack and so P can go from r with an empty stack to s with an empty stack on input y. Therefore the computation on y has $(k + 1) - 2 = k - 1$ steps. By induction hypothesis $A_{rs} \overset{*}{\Rightarrow} y$. Hence $A_{pq} \overset{*}{\Rightarrow} x$.

Case II: The stack becomes empty elsewhere. Let r be a state at which the stack becomes empty other than at the beginning or end of the computation on x. Then the portions of the computation from p to r and from r to q each contain at most k steps. Assume that y and z are the input reads during the first and second portions, respectively. By induction hypothesis, $A_{pr} \overset{*}{\Rightarrow} y$ and $A_{rq} \overset{*}{\Rightarrow} z$. Since rule $A_{pq} \mapsto A_{pr}A_{rq}$ is in G, $A_{pq} \overset{*}{\Rightarrow} x$. $\square$

APPENDIX D
PROOFS OF ALGEBRAIC LAWS FOR TIMED BISIMULARITY

In this section we present the proofs of the propositions stated in Section 6.2.7. Proofs are extensions of the proofs for untimed π-calculus processes (Milner et al., 1992). These extensions take clocks into account. We first give a series of fundamental lemmas and definitions which will be used in later results. We use κ to denote a timed action $\langle \delta, \alpha, \gamma \rangle$ in this section. We also use the symbol $\equiv_\alpha$ to denote the relation of α-convertibility on processes defined in the standard way. (The subscript α here bears no relation to the actions α defined earlier in the paper.)

Lemma D.0.10. *If $P_{(I,w)} \xrightarrow{\langle \delta,\alpha,\gamma \rangle} P'_{(I',w')}$, then (i) $fn(\alpha) \subseteq fn(P)$ and (ii) $fn(P') \subseteq fn(P) \cup bn(\alpha)$.*

Proof: The proof is by induction on depth of inference. We consider each transition rule as the last rule applied in the inference of the antecedent $P_{(I,w)} \xrightarrow{\langle \delta,\alpha,\gamma \rangle} P'_{(I',w')}$. We consider two cases only.

(INP) Then $\alpha = x(y, t_y, c)$ and $P \equiv \delta\gamma x(z, t_z, d).P_1$ with $y, c \notin fn((z)(d)P_1)$ and $P' \equiv P_1\{y/z, t_y/t_z, c/d\}$, then (i) holds and (ii) $fn(P') \subseteq (fn(P_1) - \{z, d\}) \cup \{y, c\} \subseteq fn(P) \cup \{y, c\}$.

(CLOSE) Then $\alpha = \tau$ and $P \equiv P_1 \mid P_2$ with $P_{1(I,w)} \xrightarrow{\langle \delta,\bar{x}\langle (y),t,c\rangle,\gamma \rangle} P'_{1(I',w')}$, $P_{2(I,w)} \xrightarrow{\langle \delta',x(\langle y,t,c\rangle),\gamma' \rangle} P'_{2(I',w')}$ and $P' \equiv (y)(c)(P'_1 \mid P'_2)$, obviously (i) holds, and $fn(P'_1) \subseteq fn(P_1) \cup \{y, c\}$ and $fn(P'_2) \subseteq fn(P_2) \cup \{y, c\}$, so $fn(P') = (fn(P'_1) \cup fn(P'_2)) - \{y, c\} \subseteq fn(P)$.

Definition D.0.1. *In the following lemmas the phrase:*
if $P_{(I,w)} \xrightarrow{\kappa} P'_{(I',w')}$ then equally $Q_{(I,w)} \xrightarrow{\kappa'} Q'_{(I',w')}$ means that if $P_{(I,w)} \xrightarrow{\kappa} P'_{(I',w')}$ is inferred from the transition rules then, by an inference of no greater depth, $Q_{(I,w)} \xrightarrow{\kappa'} Q'_{(I',w')}$.

Lemma D.0.11. *Suppose that $P_{(I,w)} \xrightarrow{\langle \delta,\alpha,\gamma \rangle} P'_{(I',w')}$ in which $\alpha = x(\langle y, t, c \rangle)$ or $\alpha = \bar{x}\langle (y), t, c \rangle$, if $z \notin n(P)$, then equally for some $P'' \equiv_\alpha P'\{z/y\}$, $P_{(I,w)} \xrightarrow{\langle \delta,x(\langle z,t,c\rangle),\gamma \rangle} P''_{(I',w')}$ or $P'' \equiv_\alpha$*

$P'\{z/y\}$, $P_{(I,w)} \xrightarrow{\langle \delta, \bar{x}\langle (z),t,c\rangle, \gamma\rangle} P''_{(I',w')}$ respectively. Similarly, if $P_{(I,w)} \xrightarrow{\langle \delta,\alpha,\gamma\rangle} P'_{(I',w')}$ in which $\alpha = x(\langle y,t,c\rangle)$ or $\alpha = \bar{x}\langle (y),t,c\rangle$, if $d \notin n(P)$, then equally for some $P'' \equiv_\alpha P'\{d/c\}$, $P_{(I,w)} \xrightarrow{\langle \delta, x(\langle y,t,d\rangle),\gamma\rangle} P''_{(I',w')}$ or $P'' \equiv_\alpha P'\{d/c\}$, $P_{(I,w)} \xrightarrow{\langle \delta,\bar{x}\langle (y),t,d\rangle,\gamma\rangle} P''_{(I',w')}$ respectively.

Proof: By induction on depth of inference.

Definition D.0.2. *If κ is a timed action and θ is a substitution then $\kappa\theta$ is defined as follows:*

$$\langle \delta, \tau, \gamma\rangle\theta = \langle \delta\theta, \tau, \gamma\theta\rangle$$

$$\langle \delta, x(\langle y,t,c\rangle), \gamma\rangle\theta = \langle \delta[c \to c]\theta, x\theta(\langle y, t\theta, c\rangle), \gamma[c \to c]\theta\rangle$$

$$\langle \delta, \bar{x}\langle y,t,c\rangle, \gamma\rangle\theta = \langle \delta[c \to c]\theta, \bar{x}\theta\langle y\theta, t\theta, c\rangle, \gamma[c \to c]\theta\rangle$$

$$\langle \delta, \bar{x}\langle (y),t,c\rangle, \gamma\rangle\theta = \langle \delta[c \to c]\theta, \bar{x}\theta\langle (y), t\theta, c\rangle, \gamma[c \to c]\theta\rangle$$

Note that $[c \mapsto d]\theta$ denotes the substitution which assigns d to c and agrees with θ over the rest of clocks.

Lemma D.0.12. *If $P\{u/z\}_{(I,w)} \xrightarrow{\langle \delta,\alpha,\gamma\rangle} P'_{(I',w')}$ where $u \notin fn(P)$ and $bn(\alpha) \cap fn(P,u) = \emptyset$, then equally for some Q and β with $Q\{u/z\} \equiv_\alpha P'$ and $\beta\theta = \alpha$, $P_{(I,w)} \xrightarrow{\langle \delta,\beta,\gamma\rangle} Q_{(I',w')}$.*

Proof: By induction on depth of inference.

Lemma D.0.13. *If $P \dot{\sim} Q$ and $u \notin fn(P,Q)$, then $P\{u/z\} \dot{\sim} Q\{u/z\}$.*

Proof: Let $\mathcal{S} = \bigcup_{n<u} \mathcal{S}_n$ where

$$\mathcal{S}_0 = \dot{\sim}$$

$$\mathcal{S}_{n+1} = \{(P\{u/z\}, Q\{u/z\}) | P\mathcal{S}_n Q, u \notin fn(P,Q)\}$$

We show that $\mathcal{S}$ is a strong timed bisimulation. We prove by induction on n that if $P\mathcal{S}_n Q$ then

1. If $P_{(I_1,w_1)} \xrightarrow{\langle \delta,\tau,\gamma\rangle} P'_{(I,w)}$, then for some δ', γ' and Q', $Q_{(I_2,w_2)} \xrightarrow{\langle \delta',\tau,\gamma'\rangle} Q'_{(I',w')}$, where $\delta\gamma \models_c \delta'\gamma'$, $\delta'\gamma' \models_c \delta\gamma$ and $P'\mathcal{S}Q'$,

2. If $P_{(I_1,w_1)} \xrightarrow{\langle \delta, x(\langle y,t,c\rangle), \gamma\rangle} P'_{(I,w)}$ and $y, c \notin n(P,Q)$, then for some δ', γ' and Q', $Q_{(I_2,w_2)} \xrightarrow{\langle \delta', x(\langle y,t',c\rangle), \gamma'\rangle} Q'_{(I',w')}$, where $\delta\gamma \models_c \delta'\gamma'$, $\delta'\gamma' \models_c \delta\gamma$ and for all $\langle v, t_v, d\rangle$, $P'\{v/y, t_v/t, d/c\}\mathcal{S}Q'\{v/y, t_v/t', d/c\}$,

3. If $P_{(I_1,w_1)} \xrightarrow{\langle \delta, \bar{x}\langle (y),t,c\rangle, \gamma\rangle} P'_{(I,w)}$ and $y, c \notin n(P,Q)$, then for some δ', γ' and Q', $Q_{(I_2,w_2)} \xrightarrow{\langle \delta', \bar{x}\langle (y),t',c\rangle, \gamma'\rangle} Q'_{(I',w')}$, where $\delta\gamma \models_c \delta'\gamma'$, $\delta'\gamma' \models_c \delta\gamma$ and $P'\mathcal{S}Q'$,

4. If $P_{(I_1,w_1)} \xrightarrow{\langle \delta, \bar{x}\langle y,t,c\rangle, \gamma\rangle} P'_{(I,w)}$ and $c \notin n(P,Q)$, then for some δ', γ' and Q', $Q_{(I_2,w_2)} \xrightarrow{\langle \delta', \bar{x}\langle y,t',c\rangle, \gamma'\rangle} Q'_{(I',w')}$, where $\delta\gamma \models_c \delta'\gamma'$ and $\delta'\gamma' \models_c \delta\gamma$ and $P'\mathcal{S}Q'$.

If $n = 0$ then obviously 1,2,3 and 4 hold since $\mathcal{S}_0 = \dot{\sim}$.

Suppose $n > 0$ and $P\theta\mathcal{S}_n Q\theta$ where $P\mathcal{S}_{n-1}Q$ and $\theta = \{u/z\}$ where $u \in \mathcal{N}$ and $u \notin fn(P,Q)$. We consider only 3.

Suppose that $P\theta_{(I_1,w_1)} \xrightarrow{\langle \delta, \bar{x}\langle (y),t,c\rangle, \gamma\rangle} P'_{(I,w)}$, where $y, c \notin fn(P\theta, Q\theta)$. Choose $y', c' \notin n(P,Q,u,z)$. Then $P\theta_{(I_1,w_1)} \xrightarrow{\langle \delta, \bar{x}\langle (y'),t,c'\rangle, \gamma\rangle} P''_{(I,w)}$, where $P'' \equiv P'\{y'/y, c'/c\}$. Hence by Lemma D.0.12 for some P'' and x' with $P'''\theta \equiv P''$ and $x'\theta = x$, $P_{(I_1,w_1)} \xrightarrow{\langle \delta, \bar{x'}\langle (y'),t,c'\rangle, \gamma\rangle} P'''_{(I,w)}$. Since $P\mathcal{S}_{n-1}Q$ and $y', c' \notin n(P,Q)$ for some Q''', $Q_{(I_2,w_2)} \xrightarrow{\langle \delta', \bar{x'}\langle (y'),t',c'\rangle, \gamma'\rangle} Q'''_{(I',w')}$, where $\delta\gamma \models_c \delta'\gamma'$ and $\delta'\gamma' \models_c \delta\gamma$ and $P'''\mathcal{S}Q'''$. Hence $Q\theta_{(I_2,w_2)} \xrightarrow{\langle \delta', \bar{x}\langle (y'),t',c'\rangle, \gamma'\rangle} Q''_{(I',w')}$, where $Q'' \equiv Q'\theta$, and so $Q\theta_{(I_2,w_2)} \xrightarrow{\langle \delta', \bar{x}\langle (y),t',c\rangle, \gamma'\rangle} Q'_{(I',w')}$, where $Q' \equiv Q''\{y/y', c/c'\}$. Then

$P' \equiv P'''\{u/z\}\{y/y'\}\{c/c'\}\mathcal{S}\ Q'''\{u/z\}\{y/y'\}\{c/c'\}$ since $y, c \notin fn(P'''\{u/z\}, Q'''\{u/z\})$ $\equiv Q'$

Note that Lemma D.0.13 holds if u is a clock name. In other words, if $P\dot{\sim}Q$ and $d \in \Gamma$, $d \notin fn(P,Q)$, then $P\{d/c\}\dot{\sim}Q\{d/c\}$. The proof is similar to the proof that we presented for $y \in \mathcal{N}$ and is eliminated.

Definition D.0.3. *A relation $\mathcal{S}$ is a strong timed simulation up to restriction iff whenever $P\mathcal{S}Q$ then*

1. If $u \in \mathcal{N}, u \notin fn(P,Q)$ then $P\{u/z\}\mathcal{S}Q\{u/z\}$, and

2. (a) *If* $P_{(I_1,w_1)} \xrightarrow{\langle\delta,\tau,\gamma\rangle} P'_{(I,w)}$, *then for some* δ', γ' *and* Q', $Q_{(I_2,w_2)} \xrightarrow{\langle\delta',\tau,\gamma'\rangle} Q'_{(I',w')}$, *where* $\delta\gamma \models_c \delta'\gamma'$, $\delta'\gamma' \models_c \delta\gamma$ *and either* $P'\mathcal{S}Q'$ *or for some* P'', Q'' *and* u, $P' \equiv (u)P''$, $Q' \equiv (u)Q''$ *and* $P''\mathcal{S}Q''$,

   (b) *If* $P_{(I_1,w_1)} \xrightarrow{\langle\delta,x(\langle y,t,c\rangle)\rangle} P'_{(I,w)}$ *and* $y, c \notin n(P,Q)$, *then for some* δ', γ' *and* Q', $Q_{(I_2,w_2)} \xrightarrow{\langle\delta',x(\langle y,t',c\rangle),\gamma'\rangle} Q'_{(I',w')}$, *where* $\delta\gamma \models_c \delta'\gamma'$, $\delta'\gamma' \models_c \delta\gamma$ *and for all* $\langle v, t_v, d\rangle$, $P'\{v/y, t_v/t, d/c\}\mathcal{S}Q'\{v/y, t_v/t', d/c\}$,

   (c) *If* $P_{(I_1,w_1)} \xrightarrow{\langle\delta,\bar{x}\langle(y),t,c\rangle,\gamma\rangle} P'_{(I,w)}$ *and* $y, c \notin n(P,Q)$, *then for some* δ', γ' *and* Q', $Q_{(I_2,w_2)} \xrightarrow{\langle\delta',\bar{x}\langle(y),t',c\rangle,\gamma'\rangle} Q'_{(I',w')}$, *where* $\delta\gamma \models_c \delta'\gamma'$, $\delta'\gamma' \models_c \delta\gamma$ *and* $P'\mathcal{S}Q'$,

   (d) *If* $P_{(I_1,w_1)} \xrightarrow{\langle\delta,\bar{x}\langle y,t,c\rangle,\gamma\rangle} P'_{(I,w)}$ *and* $c \notin n(P,Q)$, *then for some* δ', γ' *and* Q', $Q_{(I_2,w_2)} \xrightarrow{\langle\delta',\bar{x}\langle y,t',c\rangle,\gamma'\rangle} Q'_{(I',w')}$, *where* $\delta\gamma \models_c \delta'\gamma'$, $\delta'\gamma' \models_c \delta\gamma$ *and* $P'\mathcal{S}Q'$.

A relation $\mathcal{S}$ *is a strong timed bisimulation up to restriction iff both* $\mathcal{S}$ *and its inverse are strong timed simulation up to restriction.*

Lemma D.0.14. *If* $\mathcal{S}$ *is a strong timed bisimulation up to restriction then* $\mathcal{S} \subseteq \dot{\sim}$.

Proof: Let $\mathcal{S}^* = \bigcup_{n<r} \mathcal{S}_n$ where

$$\mathcal{S}_0 = \mathcal{S}$$

$$\mathcal{S}_{n+1} = \{((r)P, (r)Q) | P\mathcal{S}_nQ\}$$

We show that $\mathcal{S}^*$ is a strong timed bisimulation. We show that by induction on n, if $P\mathcal{S}_nQ$ and $r \notin fn(P,Q)$, then $P\{r/z\}\mathcal{S}_nQ\{r/z\}$.

For $n = 0$ this is trivial.

Suppose $n > 0$ and $(v)P\mathcal{S}_n(v)Q$ where $P\mathcal{S}_{n-1}Q$ and $r \notin fn((v)P,(v)Q)$. Then $((v)P)\{r/z\} \equiv ((u)P)\{u/v\}\{r/z\}$ and $((v)Q)\{r/z\} \equiv ((u)Q)\{u/v\}\{r/z\}$ where $u \notin fn((v)P,(v)Q,r)$ and $u\{r/z\} = u$, so $((v)P)\{r/z\}\mathcal{S}_n((v)Q)\{r/z\}$.

We show by induction on n that if $P\mathcal{S}_nQ$ then

1. If $P_{(I_1,w_1)} \xrightarrow{\langle\delta,\tau,\gamma\rangle} P'_{(I,w)}$, then for some δ', γ' and Q', $Q_{(I_2,w_2)} \xrightarrow{\langle\delta',\tau,\gamma'\rangle} Q'_{(I',w')}$, where $\delta\gamma \models_c \delta'\gamma'$, $\delta'\gamma' \models_c \delta\gamma$ and $P'\mathcal{S}^*Q'$,

2. If $P_{(I_1,w_1)} \xrightarrow{\langle\delta,x(\langle y,t,c\rangle)} P'_{(I,w)}$ and $y, c \notin n(P,Q)$, then for some δ', γ' and Q', $Q_{(I_2,w_2)} \xrightarrow{\langle\delta',x(\langle y,t',c\rangle),\gamma'\rangle} Q'_{(I',w')}$, where $\delta\gamma \models_c \delta'\gamma'$, $\delta'\gamma' \models_c \delta\gamma$ and for all $\langle v, t_v, d\rangle$, $P'\{v/y, t_v/t, d/c\}\mathcal{S}^*Q'\{v/y, t_v/t', d/c\}$,

3. If $P_{(I_1,w_1)} \xrightarrow{\langle\delta,\bar{x}\langle(y),t,c\rangle,\gamma\rangle} P'_{(I,w)}$ and $y, c \notin n(P,Q)$, then for some δ', γ' and Q', $Q_{(I_2,w_2)} \xrightarrow{\langle\delta',\bar{x}\langle(y),t',c\rangle,\gamma'\rangle} Q'_{(I',w')}$, where $\delta\gamma \models_c \delta'\gamma'$, $\delta'\gamma' \models_c \delta\gamma$ and $P'\mathcal{S}^*Q'$,

4. If $P_{(I_1,w_1)} \xrightarrow{\langle\delta,\bar{x}\langle y,t,c\rangle,\gamma\rangle} P'_{(I,w)}$ and $c \notin n(P,Q)$, then for some δ', γ' and Q', $Q_{(I_2,w_2)} \xrightarrow{\langle\delta',\bar{x}\langle y,t',c\rangle,\gamma'\rangle} Q'_{(I',w')}$, where $\delta\gamma \models_c \delta'\gamma'$, $\delta'\gamma' \models_c \delta\gamma$ and $P'\mathcal{S}^*Q'$.

For $n = 0$ it follows from the fact that $\mathcal{S}_0$ is a strong timed bisimulation up to restriction and the definition of $\mathcal{S}^*$. The remaining details are omitted.

Proof of Proposition 1: Reflexivity and symmetry are obvious. For transitivity we show that $\dot{\sim}\dot{\sim}$ is a strong timed bisimulation. The proof proceeds by a case analysis on timed actions. We consider only one case.

Suppose that $y \notin n(P,R)$ and $P_{(I_1,w_1)} \xrightarrow{\langle\delta,x(\langle y,t,c\rangle),\gamma\rangle} P'_{(I,w)}$. Let $z \notin n(P,Q,R)$, then $P_{(I_1,w_1)} \xrightarrow{\langle\delta,x(\langle z,t,c\rangle),\gamma\rangle} P''_{(I,w)}$, where $P'' \equiv P'\{z/y\}$, so for some δ', γ' and Q',
$Q_{(I_2,w_2)} \xrightarrow{\langle\delta',x(\langle z,t',c\rangle),\gamma'\rangle} Q'_{(I',w')}$, where $\delta\gamma \models_c \delta'\gamma'$, $\delta'\gamma' \models_c \delta\gamma$ and for all u, $P''\{u/z\}\dot{\sim}Q'\{u/z\}$. Hence for some δ'', γ'' and R', $R_{(I_3,w_3)} \xrightarrow{\langle\delta'',x(\langle z,t'',c\rangle),\gamma''\rangle} R'_{(I'',w'')}$, where $\delta'\gamma' \models_c \delta''\gamma''$, $\delta''\gamma'' \models_c \delta'\gamma'$ and for all u, $Q'\{u/z\}\dot{\sim}R'\{u/z\}$. Then $R_{(I_3,w_3)} \xrightarrow{\langle\delta'',x(\langle y,t'',c\rangle),\gamma''\rangle} R''_{(I'',W'')}$, where $R'' \equiv R'\{y/z\}$ and for all u, $P'\{u/y\}\dot{\sim}\dot{\sim}R''\{u/y\}$.

Proof of Proposition 2.a:

(a) $\{(P + R, Q + R) \mid P\dot{\sim}Q\} \cup \dot{\sim}$ is a strong timed bisimulation.

(b) Let $\mathcal{S} = \{(P \mid R, Q \mid R) \mid P\dot{\sim}Q\}$. We show that $\mathcal{S}$ is a strong timed bisimulation up to restriction. If $P\dot{\sim}Q$ and $u \in \mathcal{N}$, $u \notin fn(P,Q)$ then by Lemma D.0.14, $P\{u/z\}\dot{\sim}Q\{u/z\}$ and so $(P|R)\{u/z\}\mathcal{S}(Q|R)\{u/z\}$. It is straightforward to check that the transitions corresponding to rules: PAR, COM, CLOSE, REP, REP-COM and REP-CLOSE hold.

(c) $\{([x=y]P,[x=y]Q) \mid P\dot{\sim}Q\} \cup \dot{\sim}$ is a strong timed bisimulation.

(d) It follows from Lemma D.0.13 that $\dot{\sim}$ is a strong timed bisimulation up to restriction. Therefore, by the proof of Lemma D.0.14, if $P\dot{\sim}Q$ then $(u)P\dot{\sim}(u)Q$.

(e) $\{(\delta\gamma\tau.P,\delta'\gamma'\tau.Q) \mid P\dot{\sim}Q, \delta\gamma \models_c \delta'\gamma', \delta'\gamma' \models_c \delta\gamma\} \cup \dot{\sim}$ is a strong timed bisimulation.

(f) $\{((c)\delta\gamma\bar{x}\langle y,t_y,c\rangle.P,(c)\delta'\gamma'\bar{x}\langle y,t_y,c\rangle.Q) \mid P\dot{\sim}Q, \delta\gamma \models_c \delta'\gamma', \delta'\gamma' \models_c \delta\gamma\} \cup \dot{\sim}$ is a strong timed bisimulation.

Proof of Proposition 2.b: Note that $\{(\delta\gamma x(\langle y,t_y,c\rangle).P,\delta'\gamma' x(\langle y,t_y,c\rangle).Q) \mid$ for all $\langle u,t_u,e\rangle$ in which $u \in fn(P,Q,y)$, $P\{u/y,t_u/t_y,e/c\}\dot{\sim}Q\{u/y,t_u/t_y,e/c\}$ and $\delta\gamma \models_c \delta'\gamma', \delta'\gamma' \models_c \delta\gamma\}$ is a strong timed bisimulation.

Proof of Proposition 3: We can prove the following relations to be strong timed bisimulations:

$\mathcal{S}_a = \{([x=y]P,0) \mid P \text{ is a process}, x \neq y\}$

$\mathcal{S}_b = \{([x=x]P,P) \mid P \text{ is a process}\} \cup \mathbf{Id}$

where **Id** is the identity on processes.

Proof of Proposition 4: The following relations are easily seen to be strong timed bisimulations:

$\mathcal{S}_a = \{(P+0,P) \mid P \text{ is a process}\} \cup \mathbf{Id}$

$\mathcal{S}_b = \{(P+P,P) \mid P \text{ is a process}\} \cup \mathbf{Id}$

$\mathcal{S}_c = \{(P+Q,Q+P) \mid P,Q \text{ are processes}\} \cup \mathbf{Id}$

$\mathcal{S}_d = \{(P+(Q+R),(P+Q)+R) \mid P,Q,R \text{ are processes}\} \cup \mathbf{Id}$

Proof of Proposition 5: It is straightforward to show that the following relations are strong timed bisimulations:

$\mathcal{S}_a = \{((y)P,P) \mid P \text{ is a process}, y \notin fn(P)\}$

$\mathcal{S}_b = \{((x)(y)P, (y)(x)P) \mid P \text{ is a process}\} \cup \mathbf{Id}$

$\mathcal{S}_c = \{((y)(P+Q), (y)P + (y)Q) \mid P, Q \text{ are processes}\} \cup \mathbf{Id}$

$\mathcal{S}_d = \{((y)\delta\gamma\rho.P, \delta\gamma\rho.(y)P) \mid P \text{ is a process, } y \notin n(\rho)\} \cup \mathbf{Id}$

$\mathcal{S}_e = \{((y)(c)\delta\gamma\bar{y}\langle x,t,c\rangle.P, 0) \mid P \text{ is a process}\}$

$\mathcal{S}_f = \{((y)\delta\gamma y(\langle x,t,c\rangle).P, 0) \mid P \text{ is a process}\}$

$\mathcal{S}_g = \{((y)\delta\gamma y(\langle (x),t,c\rangle).P, 0) \mid P \text{ is a process}\}$

Proof of Proposition 6: The proofs of Proposition 6 (a) and Proposition 6 (b) are straightforward.

Proof of Proposition 6 (c): We use the idea of a *strong timed bisimulation up to* $\dot{\sim}$ *and restriction.* First, we introduce the following concept.

Definition D.0.4. *A relation* $\mathcal{S}$ *is a strong timed simulation up to* $\dot{\sim}$ *iff whenever* $P\mathcal{S}Q$ *then*

1. *If* $P_{(I_1,w_1)} \xrightarrow{\langle\delta,\tau,\gamma\rangle} P'_{(I,w)}$*, then for some* δ'*,* γ' *and* Q'*,* $Q_{(I_2,w_2)} \xrightarrow{\langle\delta',\tau,\gamma'\rangle} Q'_{(I',w')}$*, where* $\delta\gamma \models_c \delta'\gamma'$*,* $\delta'\gamma' \models_c \delta\gamma$ *and* $P'\dot{\sim}\mathcal{S}\dot{\sim}Q'$*,*

2. *If* $P_{(I_1,w_1)} \xrightarrow{\langle\delta,x(\langle y,t,c\rangle),\gamma\rangle} P'_{(I,w)}$ *and* $y, c \notin n(P,Q)$*, then for some* δ'*,* γ' *and* Q'*,* $Q_{(I_2,w_2)} \xrightarrow{\langle\delta',x(\langle y,t',c\rangle),\gamma'\rangle} Q'_{(I',w')}$*, where* $\delta\gamma \models_c \delta'\gamma'$*,* $\delta'\gamma' \models_c \delta\gamma$*, and for all* $\langle v,t_v,d\rangle$*,* $P'\{v/y,t_v/t,d/c\}\dot{\sim}\mathcal{S}\dot{\sim}Q'\{v/y,t_v/t',d/c\}$*,*

3. *If* $P_{(I_1,w_1)} \xrightarrow{\langle\delta,\bar{x}\langle (y),t,c\rangle,\gamma\rangle} P'_{(I,w)}$ *and* $y, c \notin n(P,Q)$*, then for some* δ'*,* γ' *and* Q'*,* $Q_{(I_2,w_2)} \xrightarrow{\langle\delta',\bar{x}\langle (y),t',c\rangle,\gamma'\rangle} Q'_{(I',w')}$*, where* $\delta\gamma \models_c \delta'\gamma'$*,* $\delta'\gamma' \models_c \delta\gamma$ *and* $P'\dot{\sim}\mathcal{S}\dot{\sim}Q'$*,*

4. *If* $P_{(I_1,w_1)} \xrightarrow{\langle\delta,\bar{x}\langle y,t,c\rangle,\gamma\rangle} P'_{(I,w)}$ *and* $c \notin n(P,Q)$*, then for some* δ'*,* γ' *and* Q'*,* $Q_{(I_2,w_2)} \xrightarrow{\langle\delta',\bar{x}\langle y,t',c\rangle,\gamma'\rangle} Q'_{(I',w')}$*, where* $\delta\gamma \models_c \delta'\gamma'$*,* $\delta'\gamma' \models_c \delta\gamma$ *and* $P'\dot{\sim}\mathcal{S}\dot{\sim}Q'$*.*

$\mathcal{S}$ *is a strong timed bisimulation up to* $\dot{\sim}$ *iff both* $\mathcal{S}$ *and its inverse are strong timed simulations up to* $\dot{\sim}$*.*

Lemma D.0.15. *If* $\mathcal{S}$ *is a strong timed bisimulation up to* $\dot{\sim}$ *then* $\mathcal{S} \subseteq \dot{\sim}$*.*

Proof: Let $\mathcal{S}^* = \bigcup_{n<u} \mathcal{S}_n$ where

$$\mathcal{S}_0 = \dot{\sim}\mathcal{S}\dot{\sim}$$

$$\mathcal{S}_{n+1} = \{(P\{u/z\}, Q\{u/z\}) \mid P\mathcal{S}_n Q, u \notin fn(P, Q)\}$$

The proof is very similar to the proof of Lemma D.0.13 and is omitted.

Definition D.0.5. *A relation $\mathcal{S}$ is a strong timed simulation up to $\dot{\sim}$ and restriction iff whenever $P\mathcal{S}Q$ then*

1. *If $u \in \mathcal{N}, u \notin fn(P, Q)$ then $P\{u/z\}\mathcal{S}Q\{u/z\}$,*
2. *If $P_{(I_1,w_1)} \xrightarrow{\langle\delta,\tau,\gamma\rangle} P'_{(I,w)}$, then for some δ', γ' and Q', $Q_{(I_2,w_2)} \xrightarrow{\langle\delta',\tau,\gamma'\rangle} Q'_{(I',w')}$, where $\delta\gamma \models_c \delta'\gamma'$, $\delta'\gamma' \models_c \delta\gamma$ and either $P'\dot{\sim}\mathcal{S}\dot{\sim}Q'$ or for some P'', Q'' and u, $P'\dot{\sim}(u)P''$, $Q'\dot{\sim}(u)Q''$ and $P''\mathcal{S}Q''$,*
3. *If $P_{(I_1,w_1)} \xrightarrow{\langle\delta,x(\langle y,t,c\rangle),\gamma\rangle} P'_{(I,w)}$ and $y, c \notin n(P, Q)$, then for some δ', γ' and Q', $Q_{(I_2,w_2)} \xrightarrow{\langle\delta',x(\langle y,t',c\rangle),\gamma'\rangle} Q'_{(I',w')}$, where $\delta\gamma \models_c \delta'\gamma'$, $\delta'\gamma' \models_c \delta\gamma$ and for all $\langle u, t_u, d\rangle$, $P'\{u/y, t_u/t, d/c\}\dot{\sim}\mathcal{S}\dot{\sim}Q'\{u/y, t_u/t', d/c\}$,*
4. *If $P_{(I_1,w_1)} \xrightarrow{\langle\delta,\bar{x}\langle(y),t,c\rangle),\gamma\rangle} P'_{(I,w)}$ and $y, c \notin n(P, Q)$, then for some δ', γ' and Q', $Q_{(I_2,w_2)} \xrightarrow{\langle\delta',\bar{x}\langle(y),t',c\rangle,\gamma'\rangle} Q'_{(I',w')}$, where $\delta\gamma \models_c \delta'\gamma'$, $\delta'\gamma' \models_c \delta\gamma$ and $P'\dot{\sim}\mathcal{S}\dot{\sim}Q'$,*
5. *If $P_{(I_1,w_1)} \xrightarrow{\langle\delta,\bar{x}\langle y,t,c\rangle),\gamma\rangle} P'_{(I,w)}$ and $c \notin n(P, Q)$, then for some δ', γ' and Q', $Q_{(I_2,w_2)} \xrightarrow{\langle\delta',\bar{x}\langle y,t',c\rangle,\gamma'\rangle} Q'_{(I',w')}$, where $\delta\gamma \models_c \delta'\gamma'$, $\delta'\gamma' \models_c \delta\gamma$ and $P'\dot{\sim}\mathcal{S}\dot{\sim}Q'$.*

$\mathcal{S}$ is a strong timed bisimulation up to $\dot{\sim}$ and restriction iff both $\mathcal{S}$ and its inverse are strong timed simulations up to $\dot{\sim}$ and restriction.

Lemma D.0.16. *If $\mathcal{S}$ is a strong timed bisimulation up to $\dot{\sim}$ and restriction then $\mathcal{S} \subseteq \dot{\sim}$.*

Proof: Let $\mathcal{S}^* = \bigcup_{n<u} \mathcal{S}_n$ where

$$\mathcal{S}_0 = \dot{\sim}\mathcal{S}\dot{\sim}$$

$$\mathcal{S}_{n+1} = \dot{\sim}\{((u)P, (u)Q) | P\mathcal{S}_n Q\}\dot{\sim}$$

Then by an argument similar to that in the proof of Lemma D.0.14 it can be shown that $\mathcal{S}^*$ is a strong timed bisimulation. We omit the details.

We next continue proving Proposition 6 (c). We show that the relation

$$\mathcal{S} = \{((y)P_1 \mid P_2, (y)(P_1 \mid P_2)) \mid P_1, P_2 \text{ are processes}, y \notin fn(P_2)\} \ \cup \mathbf{Id}$$

is a strong timed bisimulation up to $\dot{\sim}$ and restriction. We show that for each P and Q such that $P\mathcal{S}Q$ and each transition $P_{(I_1,w_1)} \xrightarrow{\kappa} P'_{(I,w)}$, there exists a simulating transition $Q_{(I_2,w_2)} \xrightarrow{\kappa'} Q'_{(I',w')}$ satisfying the requirements of a strong timed simulation up to $\dot{\sim}$ and restriction, and vice versa. If $P \equiv Q$ this is trivial, so we assume that $P \equiv (y)P_1 \mid P_2$, $Q \equiv (y)(P_1 \mid P_2)$, and $y \notin fn(P_2)$.

We show that there always exists an appropriate transition $(y)(P_1 \mid P_2)_{(I_2,w_2)} \xrightarrow{\kappa'} Q'_{(I',w')}$ by a case analysis on how the transition $((y)P_1 \mid P_2)_{(I_1,w_1)} \xrightarrow{\kappa} P'_{(I,w)}$ is derived, and vice versa. We present only one case. For each case the derivations of transitions from P and Q are presented in the following way:

$$\frac{\vdots}{((y)P_1 \mid P_2)_{(I_1,w_1)} \xrightarrow{\kappa} P'_{(I,w)}}$$

$$\Updownarrow$$

$$\frac{\vdots}{(y)(P_1 \mid P_2)_{(I_2,w_2)} \xrightarrow{\kappa'} Q'_{(I',w')}}$$

We then have to prove three things:

($\Downarrow$): that the premises of the upper derivation imply the premises of the lower derivation;

($\Uparrow$): conversely that the premises of the lower derivation imply the premises of the upper derivation;

($\mathcal{S}$): that the derivatives P' and Q' satisfy the requirement of a strong timed bisimulation up to $\dot{\sim}$ and restriction.

Note that by the definition of strong timed simulation we only have to consider $\kappa = \langle\delta, \alpha, \gamma\rangle$ such that $y \notin bn(\alpha)$, since y occurs in the processes P and Q.

Case:

$$\text{COM: }\dfrac{\text{RES-INP: }\dfrac{P_{1(I_1,w_1)} \xrightarrow{\langle\delta,x(\langle z,t,c\rangle),\gamma\rangle} P'_{1(I,w)} \quad y \neq x, y \neq z}{(y)P_{1(I_1,w_1)} \xrightarrow{\langle\delta,x(\langle z,t,c\rangle),\gamma\rangle} (y)P'_{1(I,w)}} \qquad P_{2(I_1,w_1)} \xrightarrow{\langle\delta',\bar{x}\langle z,t,c\rangle,\gamma'\rangle} P'_{2(I,w)}}{((y)P_1 \mid P_2)_{(I_1,w_1)} \xrightarrow{\langle\delta\delta',\tau,\gamma\gamma'\rangle} (c)((y)P'_1 \mid P'_2)_{(I,w)}}$$

$$\Updownarrow$$

$$\text{RES-TAU: }\dfrac{\text{COM: }\dfrac{P_{1(I_1,w_1)} \xrightarrow{\langle\delta,x(\langle z,t,c\rangle),\gamma\rangle} P'_{1(I,w)} \qquad P_{2(I_1,w_1)} \xrightarrow{\langle\delta',\bar{x}\langle z,t,c\rangle,\gamma'\rangle} P'_{2(I,w)}}{(P_1|P_2)_{(I_1,w_1)} \xrightarrow{\langle\delta\delta',\tau,\gamma\gamma'\rangle} (c)(P'_1 \mid P'_2)_{(I,w)}}}{(y)(P_1 \mid P_2)_{(I_1,w_1)} \xrightarrow{\langle\delta\delta',\tau,\gamma\gamma'\rangle} (y)(c)(P'_1 \mid P'_2)_{(I,w)}}$$

($\Downarrow$): Trivial.

($\Uparrow$): From $y \notin fn(P_2)$ and Lemma D.0.10 we get that $y \neq x$. We cannot prove that $y \neq z$, but if $y = z$ then we use a fresh z' instead of z to get a simulating transition as follows: from Lemma D.0.11 we get that $P_{1(I_1,w_1)} \xrightarrow{\langle\delta,x(\langle z',t,c\rangle),\gamma\rangle} P'_1\{z'/y\}_{(I,w)}$. The simulating transition then is :

$$((y)P_1|P_2)_{(I_1,w_1)} \xrightarrow{\langle\delta\delta',\tau,\gamma\gamma'\rangle} (c)((y)P'_1\{z'/y\}|P'_2)_{(I,w)} \ (*)$$

($\mathcal{S}$): From Lemma D.0.10 with $y \notin fn(P_2)$ we get that $y \notin fn(P'_2)$, so

$$(c)((y)P'_1 \mid P'_2) \ \mathcal{S} \ (c)(y)(P'_1 \mid P'_2)$$

For the simulating transition (*) we know that $y = z$, since z' is chosen fresh, it follows that

$$(c)((y)P_1\{z'/y\}|P'_2) \ \equiv \ (c)((y)P'_1|P'_2) \ \mathcal{S} \ (c)(y)(P'_1|P'_2)$$

Proof of Proposition 6 (d): We will show that the relation

$$\mathcal{S} = \{((P_1 \mid P_2) \mid P_3, P_1 \mid (P_2 \mid P_3)) \mid P_1, P_2, P_3 \text{ are processes}\}$$

is a strong timed bisimulation up to $\dot{\sim}$ and restriction. For each P and Q such that $P\mathcal{S}Q$ and each transition $P_{(I_1,w_1)} \xrightarrow{\kappa} P'_{(I,w)}$ we must find a simulating transition $Q_{(I_2,w_2)} \xrightarrow{\kappa'} Q'_{(I',w')}$ satisfying the requirements of a strong timed simulation up to $\dot{\sim}$ and restriction, and vice versa.

The proof that an appropriate transition $Q_{(I_2,w_2)} \xrightarrow{\kappa'} Q'_{(I',w')}$ can be found, is by a case analysis on how the transition $P_{(I_1,w_1)} \xrightarrow{\kappa} P'_{(I,w)}$ is derived, and vice versa. One sample case is presented here.

Case:

$$\text{PAR: }\dfrac{\text{CLOSE: }\dfrac{P_{1(I_1,w_1)} \xrightarrow{\langle\delta,\bar{x}\langle(z),t,c\rangle,\gamma\rangle} P'_{1(I,w)} \qquad P_{2(I_1,w_1)} \xrightarrow{\langle\delta',x(\langle z,t,c\rangle),\gamma'\rangle} P'_{2(I,w)}}{(P_1 \mid P_2)_{(I_1,w_1)} \xrightarrow{\langle\delta\delta',\tau,\gamma\gamma'\rangle} (z)(c)(P'_1 \mid P'_2)_{(I,w)}}}{((P_1 \mid P_2) \mid P_3)_{(I_1,w_1)} \xrightarrow{\langle\delta\delta',\tau,\gamma\gamma'\rangle} ((z)(c)(P'_1 \mid P'_2) \mid P_3)_{(I,w)}}$$

$$\Downarrow$$

$$\text{CLOSE: }\dfrac{\text{PAR: }\dfrac{P_{2(I_1,w_1)} \xrightarrow{\langle\delta',x(\langle z',t,c\rangle),\gamma'\rangle} P'_2\{z'/z\}_{(I,w)} \quad z' \notin fn(P_3)}{(P_2 \mid P_3)_{(I_1,w_1)} \xrightarrow{\langle\delta',x(\langle z',t,c\rangle),\gamma'\rangle} (P'_2\{z'/z\} \mid P_3)_{(I,w)}} \qquad P_{1(I_1,w_1)} \xrightarrow{\langle\delta,\bar{x}\langle(z'),t,c\rangle,\gamma\rangle} P'_1\{z'/z\}_{(I,w)}}{(P_1 \mid (P_2 \mid P_3))_{(I_1,w_1)} \xrightarrow{\langle\delta\delta',\tau,\gamma\gamma'\rangle} (z')(c)(P'_1\{z'/z\} \mid (P'_2\{z'/z\} \mid P_3))_{(I,w)}}$$

($\Downarrow$): Using Lemma D.0.11 there exists a fresh z' such that $P_{1(I_1,w_1)} \xrightarrow{\langle\delta,\bar{x}\langle(z'),t,c\rangle,\gamma\rangle} P'_1\{z'/z\}_{(I,w)}$ and $P_{2(I_1,w_1)} \xrightarrow{\langle\delta',x(\langle z',t,c\rangle),\gamma'\rangle} P'_2\{z'/z\}_{(I,w)}$.

($\mathcal{S}$): Since z' is a fresh name, by alpha-converting z to z' and then using the Proposition 6 (c) we obtain

$$(z)(c)(P'_1 \mid P'_2) \mid P_3 \;\equiv\; (z')(c)(P'_1\{z'/z\} \mid P'_2\{z'/z\}) \mid P_3 \;\dot{\sim}\; (z')(c)((P'_1\{z'/z\} \mid P'_2\{z'/z\}) \mid P_3)$$

It follows that

$$(P'_1\{z'/z\} \mid P'_2\{z'/z\}) \mid P_3 \;\mathcal{S}\; P'_1\{z'/z\} \mid (P'_2\{z'/z\} \mid P_3)$$

proof of Proposition 6 (e): If $y \notin fn(P_1) \cap fn(P_2)$, then y cannot be free in both P_1 and P_2. Assume that y is not free in P_2. Then by Propositions 5 (a) and 6 (c):

$$(y)(P_1 \mid P_2) \mathrel{\dot\sim} (y)P_1 \mid P_2 \mathrel{\dot\sim} (y)P_1 \mid (y)P_2$$

The situation when y is not free in P_1 is similar.

Proof of Proposition 8 *Expansion Theorem*: Write R for the right hand side of the equation. Define the relation $\mathcal{S}$ by

$$\mathcal{S} \;=\; \{(P \mid Q), R)\} \cup \mathbf{Id}$$

It can be easily seen that $\mathcal{S}$ is a timed bisimulation.

Note that the side conditions $bn(\rho_i) \cap fn(Q) = \emptyset$ and $bn(\phi_j) \cap fn(P) = \emptyset$ are important, otherwise a bound name in ρ_i (or ϕ_j) would bind names in Q (or P) in the right hand side but not in the left hand side.

REFERENCES

Aggarwal, S. and R. P. Kurshan (1983). Modelling elapsed time in protocol specification. In *Protocol Specification, Testing, and Verification*, pp. 51–62.

Alur, R. (1991). *Techniques for Automatic Verification of Real-Time Systems*. Ph. D. thesis, Stanford University.

Alur, R., C. Courcoubetis, and D. L. Dill (1990). Model-checking for real-time systems. In *LICS*, pp. 414–425.

Alur, R., C. Courcoubetis, T. A. Henzinger, and P.-H. Ho (1992). Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pp. 209–229.

Alur, R. and D. L. Dill (1990). Automata for modeling real-time systems. In *ICALP*, Volume 443 of *Lecture Notes in Computer Science*, pp. 322–335. Springer.

Alur, R. and D. L. Dill (1994). A theory of timed automata. *TCS 126*(2), 183–235.

Alur, R. and T. A. Henzinger (1990). Real-time logics: Complexity and expressiveness. In *LICS*, pp. 390–401.

Alur, R., T. A. Henzinger, and H. Wong-toi (1995). The algorithmic analysis of hybrid systems. *Theoretical Computer Science 138*, 3–34.

Alur, R., T. A. Henzinger, and H. Wong-toi (1997). Symbolic analysis of hybrid systems. *Theoretical Computer Science 138*, 3–34.

Barwise, J. and L. Moss (1996). *Vicious circles: on the mathematics of non-wellfounded phenomena*. Stanford, CA, USA: CSLI.

Behrmann, G., A. David, and K. G. Larsen (2004). A tutorial on uppaal. In *SFM*, pp. 200–236.

Berger, M. (2004). Towards abstractions for distributed systems. Technical report, Imperial College London.

Brzozowski, J. A. and C.-J. H. Seger (1991). Advances in asynchronous circuit theory; part ii: Bounded inertial delay models, mos circuit design techniques. *Bulletin of the European Association for Theo- retical Computer Science 43*(3), 199–263.

Ciobanu, G. and C. Prisacariu (2006). Timers for distributed systems. *Electr. Notes Theor. Comput. Sci. 164*(3), 81–99.

Cleaveland, R., P. M. Lewis, S. A. Smolka, and O. Sokolsky (1996). The concurrency factory: A development environment for concurrent systems. In *CAV*, pp. 398–401.

Cleaveland, R. and S. Sims (1996). The ncsu concurrency workbench. In *CAV*, pp. 394–397.

Colmerauer, A. (1982). Prolog and infinite trees. In K. L. Clark and S.-A. Tärnlund (Eds.), *Logic Programming*, pp. 231–251. London: Academic Press.

Cristina, R.-J. B. and C. Cerschi (2000). Modeling and verifying a temperature control system using continuous action systems. In *Proc. of the 5th Int. Workshop on FMICS*.

Dang, Z. (2001). Binary reachability analysis of pushdown timed automata with dense clocks. In *CAV '01*, London, UK, pp. 506–518. Springer-Verlag.

Dang, Z., O. H. Ibarra, T. Bultan, R. A. Kemmerer, and J. Su (2000). Binary reachability analysis of discrete pushdown timed automata. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, London, UK, pp. 69–84. Springer-Verlag.

Degano, P., J. vincent Loddo, and C. Priami (1996). Mobile processes with local clocks. In *LOMAPS*, pp. 296–319. Springer-Verlag.

Dill, D. L. (1990). Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, London, UK, pp. 197–212. Springer-Verlag.

Emerson, E. A., A. K. Mok, A. P. Sistla, and J. Srinivasan (1991). Quantitative temporal reasoning. In *CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification*, London, UK, pp. 136–145. Springer-Verlag.

Falaschi, M. and A. Villanueva (2006). Automatic verification of timed concurrent constraint programs. *TPLP 6*(3), 265–300.

Gupta, G., A. Bansal, R. Min, L. Simon, and A. Mallya (2007). Coinductive logic programming and its applications. In *ICLP*, pp. 27–44. Springer.

Gupta, G. and E. Pontelli (1997). A constraint-based approach for specification and verification of real-time systems. In *IEEE Real-Time Systems Symp*, pp. 230–239.

Gupta, R. (2006a). Programming models and methods for spatiotemporal actions and reasoning in cyber-physical systems. In *NSF Workshop on CPS*.

Gupta, R. (2006b). Programming models and methods for spatiotemporal actions and reasoning in cyber-physical systems. In *NSF Workshop on CPS*.

Heitmeyer, C. L. and N. A. Lynch (1994). The generalized railroad crossing: A case study in formal verification of real-time systems. In *IEEE RTSS*, pp. 120–131.

Henzinger, T. A. and P. hsin Ho (1995). Hytech: The cornell hybrid technology tool. In *Hybrid Systems II, LNCS 999*, pp. 265–293. Springer-Verlag.

Henzinger, T. A., X. Nicollin, J. Sifakis, and S. Yovine (1992). Symbolic model checking for real-time systems. *Information and Computation 111*, 394–406.

Jaffar, J. and J.-L. Lassez (1987). Constraint logic programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, New York, NY, USA, pp. 111–119. ACM.

Jaffar, J. and M. J. Maher (1994). Constraint logic programming: A survey. *J. Log. Program. 19/20*, 503–581.

Jaffar, J., A. E. Santosa, and R. Voicu (2004). A CLP proof method for timed automata. In *RTSS*, pp. 175–186.

Laneve, C. and G. Zavattaro (2005). Foundations of web transactions. pp. 282–298. Springer-Verlag.

Lee, E. A. (2008, May). Cyber-physical systems: Design challenges. In *ISORC*.

Lee, J. Y. and J. Zic (2002). On modeling real-time mobile processes. *Aust. Comput. Sci. Commun. 24*(1), 139–147.

Lloyd, J. W. (1987). *Foundations of logic programming / J.W. Lloyd* (2nd, extended ed.). Springer, Berlin, New York.

Mazzara, M. (2005). Timing issues in web services composition. In *EPEW/WS-FM*, pp. 287–302.

Merritt, M., F. Modugno, and M. R. Tuttle (1991). Time-constrained automata. In *CONCUR '91: 2nd International Conference on Concurrency Theory, volume 527 of Lecture Notes in Computer Science*, pp. 408–423. Springer-Verlag.

Milner, R., J. Parrow, and D. Walker (1992, September). A calculus of mobile processes, parts i and ii. *Inf. Comput. 100*(1), 1–77.

Olate, C. (2009). *Universal Temporal Concurrent Constraint Programming*. Ph. D. thesis, LIX, Ecole Polytechnique.

Puchol, C. (1995). A solution to the generalized railroad crossing problem in Esterel. Technical report, Dep. of Comp. Science, The University of Texas at Austin.

R. A. Thacker et al. (2010). Automatic abstraction for verification of cyber-physical systems. In *ICCPS*, pp. 12–21.

Rounds, W. C. and H. Song (2003). The phi-calculus: A language for distributed control of reconfigurable embedded systems. In *HSCC*, pp. 435–449.

Saeedloei, N. and G. Gupta (2010). Timed definite clause omega-grammars. In *ICLP (Technical Communications)*, pp. 212–221.

Saeedloei, N. and G. Gupta (2011a, June). A logic-based modeling and verification of CPS. *SIGBED Rev. 8*, 31–34.

Saeedloei, N. and G. Gupta (2011b). Logic based models for the reactor temperature control system and the grc.

Saraswat, V. A., R. Jagadeesan, and V. Gupta (1994). Foundations of timed concurrent constraint programming. In *LICS*, pp. 71–80.

Simon, L. (2006). *Coinductive Logic Programming.* Ph. D. thesis, University of Texas at Dallas, Richardson, Texas.

Simon, L., A. Bansal, A. Mallya, and G. Gupta (2007). Co-logic programming: Extending logic programming with coinduction. In *ICALP*, pp. 472–483.

Sipser, M. (1996). *Introduction to the Theory of Computation.* International Thomson Publishing.

Sterling, L. and E. Shapiro (1994). *The art of Prolog (2nd ed.): advanced programming techniques.* Cambridge, MA, USA: MIT Press.

Thomas, W. (1990). Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pp. 133–192. MIT Press.

Urbina, L. (1996). Analysis of hybrid systems in clp(r). In *CP*, pp. 451–467.

X. Nicollin et al. (1992). An approach to the description and analysis of hybrid systems. In *Hybrid Systems*, pp. 149–178.

Yang, J., A. K. Mok, and F. Wang (1993). Symbolic model checking for event-driven real-time systems. In *IEEE Real-Time Systems Symposium*, pp. 23–33.

Yang, P., C. R. Ramakrishnan, and S. A. Smolka (2004). A logical encoding of the pi-calculus: model checking mobile processes using tabled resolution. *STTT 6*(1), 38–66.

Yi, W. (1991). CCS + time = an interleaving model for real time systems. In *ICALP*, pp. 217–228. Springer-Verlag New York, Inc.

Yoneda, T., A. Shibayama, B.-H. Schlingloff, and E. M. Clarke (1993). Efficient verification of parallel real-time systems. In *CAV*, pp. 321–346.

VITA

Neda Saeedloei was born in Tehran, Iran in 1975 as a daughter of Mahin Tabbakhi and Rahim Saeedloei. After receiving a Bachelor of Science in Applied Mathematics form Sharif University of Technology in 1997, she worked at ECMS Co. for seven years. She received a Master of Science in Computer Science from University of Texas at Dallas in August 2007. In December 2011, she received her Doctorate degree in Computer Science from University of Texas at Dallas.