A Methodology for Modeling and Verification of Cyber-Physical Systems based on Logic Programming*

Neda Saeedloei
University of Minnesota Duluth
nsaeedlo@d.umn.edu

Gopal Gupta
University of Texas at Dallas
gupta@utdallas.edu

ABSTRACT

Model-based design and development has been applied successfully to design and development of complex systems, including safety critical systems. It is also a promising approach for designing cyber-physical systems (CPSs). In this paper we propose a methodology for model-based design of CPSs where, *logic programming* extended with *coinduction*, *constraints over reals*, and *coroutining* is used for modeling CPSs. This logic programming realization can be used for verifying interesting properties as well as generating implementations of CPSs. We use the reactor temperature control system as a running example to illustrate the various steps of our methodology. We present a model of the system using our framework and verify the safety property of the system. We also show how parametric analysis can be performed in our framework.

1. INTRODUCTION

Model-based design has been used as an effective approach for design, analysis, and verification of complex systems. The process of designing and developing a complex system can greatly benefit from building a formal model, i.e., one that is expressed in a formal language with well-defined semantics. The most important advantage is that, during the process of modeling, one obtains a deep insight into how the physical realisation of the system would behave in the real world.

Once a formal model is built, it can be used as a high-level "prototype": it can be experimented with and iteratively improved. (The cost of doing so is very significantly lower than that of experimenting with and improving a real implementation of the system.) Depending on how the model is constructed, such experimentation usually takes the form of either formally deriving logical conclusions, or of automatic simulation. The overall purpose is to obtain a validated model, i.e., one whose "behavior" does indeed comply with the desired behavior of the modeled system. The insight mentioned in the preceding paragraph arises out of the necessity of formulating very clear criteria for (or examples of) desirable and undesirable behavior.

Cyper-physical systems combine computational and physical elements in tight coordination. In addition to discrete computation, the presence of physical elements require CPSs to handle continuous quantities (e.g., time, distance, acceleration, temperature, etc.). Unlike embedded systems, CPSs are not stand-alone systems. Rather, they are a network of interacting and concurrently executing elements that have physical inputs and outputs. In addition, CPSs could possibly run forever [10, 18]. We consider communicating hybrid automata as the underlying model of CPSs and propose a framework based on logic programming (LP) extended with constraints, coinduction and coroutining for modeling them.

Recently, the authors have developed *coinductive constraint logic programming (co-CLP)* [28] as a combination of two powerful programming paradigms: constraint logic programming (CLP) [16, 15] and coinductive logic programming [8, 31, 30]. Co-CLP is a programming language that is suitable for modeling *infinite (rational)* behaviors of infinite objects with constraints imposed on them. In this paper we extend the efforts of Gupta et.al [9], and employ the principles of co-CLP [28] to propose a framework for modeling hybrid automata [14]. Then, we use the coroutining feature of logic programming to model communication among hybrid automata. Cyber-physical systems, modeled as communicating hybrid automata, are thus represented as coroutined coinductive constraint logic programs which are subsequently used to verify properties of the systems relating to safety, liveness and utility. Thus, our approach is based on using *logic programming* for modeling computations, *constraint logic programming* for modeling continuous physical quantities, *coinduction* for modeling possibly perpetual executions and *coroutining* for modeling concurrency in CPSs. What is noteworthy in our realization of CPSs is that, in contrast to other approaches, we do not discretize the physical quantities involved. Rather these physical quantities are assumed to range over continuous real values, with relationship between them modeled as constraints over reals.

We use the *reactor temperature control system* [33] as a running example to illustrate various aspects of modeling CPSs. The reactor temperature control system is a traditional example of a cyber-physical system which is also a safety critical system: it is, therefore, very important to ob-

*A preliminary version of this paper was published at ACM SIGBED Review - Work-in-Progress (WiP) Session of the 2nd International Conference on Cyber Physical Systems, 2011.

tain an accurate model of the system. Such a model is a timed model, as cyber-physical systems operate under real-time constraints. Our focus is to model the behavior of the system related to time and verify real-time properties.

Our framework for modeling and verifying CPSs provides diagnostic information that can be used in design of the system. For instance, if our system fails to satisfy the 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 [14].

In our previous work [27], we proposed a logic-based framework for modeling CPSs without presenting the necessary mathematical foundations for our approach. Having developed the required foundations by combining constraint logic programming and coinduction [28], in this paper we present the complete account of our approach for logic programming-based modeling of CPSs, for the first time.

We first present an overview of coinductive constraint logic programming. For more details on co-CLP, the reader is referred to [28].

1.1 co-CLP

Coinductive constraint logic programming [28] is a paradigm that combines constraint logic programming (CLP) [16, 15] and coinductive logic programming [8, 31, 30].

Constraint logic programming is a natural and expressive paradigm which combines two declarative paradigms: logic programming [20] and constraint solving. 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 objects and their properties; one needs to resort to greatest fixed-point semantics for that purpose.

Coinductive logic programming is an extension of logic programming which provides an elegant technique for finitely reasoning about infinite (rational) structures and their properties [8, 31, 30].

Coinductive LP 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, hybrid systems and cyber-physical systems [18, 11]. *Coinductive constraint logic programming* [28] is a paradigm that bridges this gap. It allows a special class of formulae (constraints) to be combined with traditional coinductive logic programming. We next present a brief overview of co-CLP.

Given a *signature* Σ, 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. It is assumed that Σ contains the binary predicate symbol $=$, interpreted as identity in D, and a special binary relation $=_u$ which is used for unification. Σ^D is an extension of Σ in which there is a constant for every element of D. Furthermore, Σ_P^D is the extension of Σ^D with the set of all user-defined functions and predicate symbols in program P. The set of predicate symbols defined by Σ is denoted by Π_c and the set of predicate symbols defined by program P is denoted by Π_P. Π_c and Π_P are disjoint.

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). Terms that are built upon variables and function symbols in Σ^D are denoted by T^D.

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 ensures 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.

A coinductive constraint logic program is composed of a collection of clauses of the form: $a :- c, b_1, \ldots, b_n.$ in which a, and $b_i, \leq i \leq n$, are user-defined atoms, while c is an arbitrary conjunction of constraints.

Consider the definition of a stream (list) of numbers given as program P1 below:

```
stream([H1, H2 | T]) :-
     number(H1), number(H2), {H2-H1 >= 3}, stream(T).
number(0).
number(s(N)) :- number(N).
```

In standard CLP the query `?- stream(X).` fails, since the model of P1 does not contain any instances of `stream/1`. However, the query `?- stream(X).` under the co-CLP interpretation of P1 should produce all infinite (rational) streams such as `X = [1, 5 | X]`, `X = [2, 6, 3, 7 | X]` , etc., as answers. The model of P1 does contain instances of `stream/1` (but proofs may be of infinite length).

Co-CLP allows programmers to manipulate infinite (rational) structures in the presence of constraints. As a result, unification must be extended and the "occurs check" removed: unification equations such as `X = [1 | X]` are allowed in co-CLP; in fact, such equations will be used to represent infinite (rational) structures in a finite manner.

From the semantic point of view, predicates and functions in Σ and constraints are interpreted using the predefined interpretation, i.e., the domain of computation $\mathcal{D}$. In particular, a constraint c is solvable if $\mathcal{D} \models \exists c$, where $\models$ is the standard entailment relation. A solution θ for c is a mapping from the variables in c to $\mathcal{D}$, such that $\mathcal{D} \models c\theta$. User-defined functions and predicates will be given the standard interpretation in the Herbrand universe and the Herbrand base [20].

From the procedural point of view, execution of a coinductive constraint logic 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. The operational semantics of co-CLP also relies on the *coinductive hypothesis rule*

[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.

and systematically computes elements of the *greatest fixed point* (*gfp*) of a program via backtracking. The coinductive hypothesis rule states that during execution, if the current resolvent R contains a call G' that unifies with an ancestor call G and the set of accumulated constraints are satisfied, then the call G' succeeds; the new resolvent is $R'\theta$ where $\theta = mgu(G, G')$[2] and R' is obtained by deleting G' from R. If $?{-}c_1,\ g_1, \ldots, g_n.$ is a goal, and $p{:}\ {-}c_2,\ b_1, \ldots, b_k.$ is a clause in the program, then the resolvent of the goal w.r.t. the given clause is

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

as long as $\mathcal{D} \models E(c_1 \wedge c_2)$ where E is the current system of equations (substitution) which includes unification of arguments of g_1 and p. $E(c_1 \wedge c_2)$ is the result of applying the substitution E on $c_1 \wedge c_2$. The constraint solver is used to test the validity of the condition on the constraints.

Regular constraint logic programming execution extended with the coinductive hypothesis rule is termed *co-constraint logic programming* (or co-CLP)[28]. 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. Even with the restriction to rational proofs, there are many applications of co-CLP. These include model checking, modeling ω-automata, etc.

2. A LP-BASED METHODOLOGY FOR MODELING CPSS

Cyber-physical systems have typically the following four characteristics [10, 18]: (i) they perform discrete computations, (ii) they deal with continuous quantities, (iii) they are concurrent, and (iv) they (possibly) run forever. Using the reactor temperature control system as an illustration, we next discuss the steps involved in modeling CPSs, and show how these characteristics are handled in our framework. We use the reactor temperature control system to illustrate various aspects of our framework.

The reactor temperature control system consists of a reactor core and two control rods whose task is to keep the temperature of the reactor core between two thresholds: 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 1 shows a model of the system based on hybrid automata. The hybrid automaton in left hand side models the core; while, the two hybrid automata in right hand side model two rods. In this model, θ represents the temperature, variables r_1 and r_2 measure the time that has elapsed since Rod1 and Rod2 were removed from the reactor core, respectively. Variables c_1 and c_2 are used to model the two clocks of the core. Initially θ is θ_m and both control rods are outside of the reactor core. This corresponds to state *no_rod*, in which θ rises according to the differential equation $\dot{\theta} = \frac{\theta}{10} - 50$. In states rod_1 and rod_2, the temperature decreases according to the differential equations $\dot{\theta} = \frac{\theta}{10} - 56$ and $\dot{\theta} = \frac{\theta}{10} - 60$, respectively. A control rod may be used again, if $T \geq 0$ units of time have elapsed since it was last removed. If θ cannot be decreased because no control rod is available, then shut down of the reactor is necessary. The goal of the system is to calculate parameter T such that the system is never shut down.

Obtaining accurate requirements of the system: The starting point in building a model of a system is the requirements. The requirements are usually expressed either in natural language or in some formal language.

It is a fact of life that complete requirements of systems often do not exist or, if they do exist, are incomplete, ambiguous or very low level. As a result, some parameters and functionalities might be missing and must be guessed. When dealing with CPSs, such incomplete requirements are inadequate, as CPSs are often safety-critical systems and unspecified behaviors and missing cases cannot be tolerated. Ambiguous requirements are also of limited use, as they are very hard to analyze and model. Finally, very low level requirements specifications are difficult to understand and reason about and hardly useful for modeling purposes. It is exactly the unavailability of satisfactory requirements specifications that makes modeling such an important step in the process of constructing a system.

In our framework, we assume that the requirements of the system are specified by hybrid automata. Hybrid automata have well-defined semantics and provide a high level, unambiguous framework for specifying systems requirements. Moreover, semantics of hybrid automata are defined over real-time state sequences, which makes them ideal for describing CPSs.

Modeling the environment and deciding the domains of physical quantities: Analyzing and modeling the environment in which a cyber-physical system operates is the next step in building a formal model of the system. This phase is particularly important, as CPSs constantly interact with their environment. This interaction is through feedback loops where, physical processes influence computations, and vice versa. The task of modeling the environment includes first, identifying such physical processes and second, specifying them formally.

The physical processes that are involved in CPSs are continuous in nature. Modeling such quantities as discrete variables could simplify the models, but it could adversely affect the accuracy of the model, and hence the correctness of the system. For instance, modeling time as a discrete quantity can result in a useful abstraction of the system and *possibly* a simpler model, one which can be verified reasonably fast using model checkers. But our studies show that a model based on the discrete model of time could behave differently than the one built based on the continuous model of time [25]. While time is just one example of a naturally continuous quantity, CPSs often deal with other such quantities, e.g., pressure, temperature, density, weight, etc. Modeling time and other physical quantities has been reported as one of the challenges in modeling CPSs [19, 11].

Coming back to our example of the reactor temperature control system, calculating the temperature is clearly central to the design and modeling of the system. Therefore, it is necessary to identify all the quantities that influence it. It is also important to model such quantities as continuous quantities, rather than discretizing them. In the reactor system the temperature is specified using differential equations over time. Therefore, "time" is the only quantity that influences the temperature.

In our framework based on constraint logic programming over reals (CLP(R)) [17] we model all physical continuous quantities as real-value variables and the relation between

[2] *mgu* is a shorthand for "most general unifier".

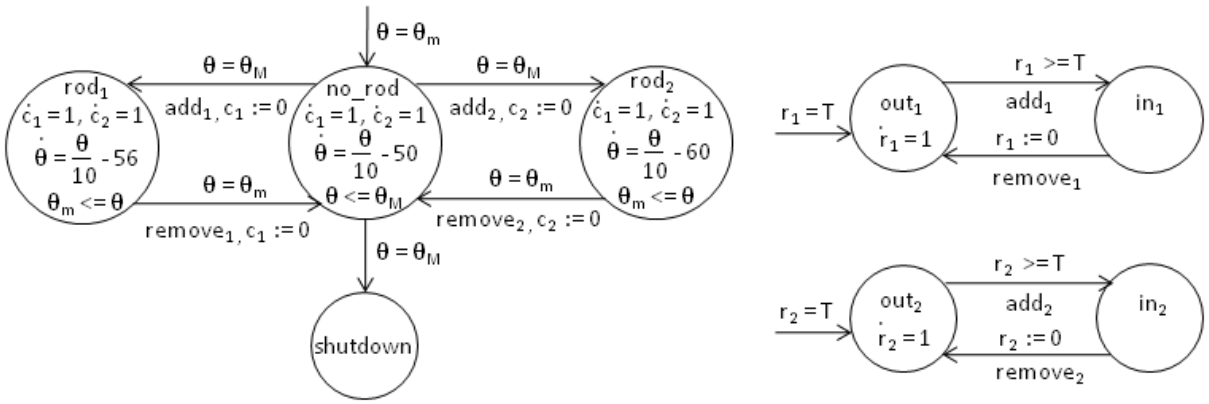

Figure 1: The Reactor Temperature Control System

them as constraints over reals. These variables take part in the internal computations, thereby updating the dynamics of the system. As such, these variables also represent the externally visible behavior of the system.

Modeling time: In a real-time or a cyber-physical system, the correctness of the system depends on not only the actions that the system takes, but also the times at which the actions are taken. Thus, the model of a cyber-physical system is a timed model and the properties that the system must satisfy are often behavioral properties with timing constraints. Therefore, being able to model continuous time and reason about it is an important step in the process of modeling CPSs. Our framework based on communicating hybrid automata uses real-valued clocks for modeling time, where timing requirements are expressed as constraints on clocks. Therefore, time is modeled as a continuous quantity and timing requirements are modeled as constraints over reals. Moreover, the following capabilities which are necessary for proper modeling of time are supported:

- modeling passage of time,
- remembering the time at which a particular event takes place,
- ability to measure the time that has elapsed since a particular event takes place,
- ability to specify the time requirements,
- providing the means to formally prove the satisfiability of timing requirements.

Later in this section we discuss our techniques for modeling these capabilities, in details.

Building the formal model of the system: A model of a cyber-physical system usually consists of two or more integrated sub-models: the model of the system to be constructed, and the model(s) of the relevant physical process(es). The sub-models affect each other through closed feedback loops: the physical processes influence the computations, and the results of the computations impact the physical processes.

Having (1) identified and modeled the environment, and (2) decided how to model time (and other physical quantities), we must now formally describe how the dynamics of the system change.

Cyber-physical systems are often very complex and analysis of their dynamics can be difficult. Apart from that, there are also difficulties specific to the task of building a model. First, the models can be only as good as the tools (and their underlying semantics) with which they are built: using tools that lack formal semantics [6, 3, 24] may result in models that are erroneous, ambiguous, incomplete, or nondeterministic. Second, different components of the system are often modeled with different models of computation: it is not always clear how to compose these into a single model [5] without losing correctness or introducing ambiguity.

Encoding hybrid automata as co-CLP programs: A hybrid automaton [1] is a tuple $H = \langle \Sigma, Q, V, C, E, A, I, Init \rangle$, where

- Σ is a finite alphabet or synchronization labels;
- Q is the (*finite*) set of locations;
- V is a finite set of real-valued variables. A valuation ν for the variables is a function that assigns a real-value $\nu(x) \in \mathbb{R}$ to each variable $x \in V$. We write $\mathcal{V}$ for the set of valuations;
- $C \subseteq V$ is a finite set of real-valued variables, called *clocks*;
- $E \subseteq Q \times \Sigma \times Q \times \phi(V) \times 2^C$ gives the set of transitions. An edge $\langle q, a, q', \delta, \gamma \rangle$ represents a transition from location q to location q' on synchronization label a. The transition is guarded by conjunctions of linear constraints δ; the set $\gamma \subseteq C$ gives the clocks to be reset with this transition. The variables in δ range over V;
- A is a labelling function that assigns to each location $q \in Q$ a set of flow conditions or evolution laws, which are specified as differential equations. The free variables of A range over variables in V and their first derivatives;
- I is a labeling function that assigns to each location $q \in Q$ an *invariant* $I(q) \subseteq \mathcal{V}$. The free variables of I range over variables in V;
- $Init$ is a function that assigns to each location a set of initial conditions. An automaton can start in a particular location q only if $Init(q)$ holds.

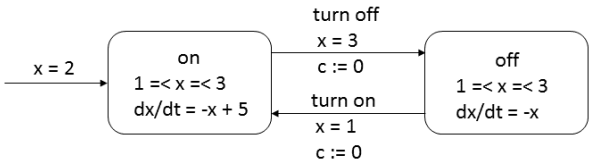

Figure 2: A Typical Thermostat

A state, s, is a pair (q, ν), where q is a location in Q and ν is a valuation in $\mathcal{V}$. A state of a hybrid automaton can change in two ways:

- By a *discrete* and *instantaneous* transition that changes both the control location and the values of the variables according to the transition relation;
- By a *time delay* that changes only the values of the variables according to the evolution laws of the current location.

The automaton may stay in a location as long as the location invariant is true; that is, some discrete transition must be taken before the invariant becomes false. Figure 2 shows a typical thermostat modeled as a hybrid automaton.

Our approach for modeling hybrid automata is a direct translation of transitions of hybrid automata to logic programming rules, where each rule is extended with a set of constraints. The general method of converting hybrid automata to coinductive constraint logic programs over reals is presented next. The hybrid automaton of Figure 2 is used to demonstrate the modeling technique. We define the following predicates to represent a hybrid automaton:

- `invariant/2` $\subset Q \times I$, which associates a set of invariants with states of a hybrid automaton;
- `evolution/2` $\subset Q \times A$, which associates a set of evolution laws with states of a hybrid automaton;
- `transitions/5` $\subset Q \times \Sigma \times Q \times \Delta \times \Gamma$, which represents the transitions of the hybrid automaton, where Δ is a list of linear constraints in $\phi(V)$ and Γ is a list of clock resets;
- `init/2` $\subset Q \times Init$, which associates a set of initial conditions with states of a hybrid automaton.

For instance, the hybrid automaton of Figure 2 is specified by the following logic programming facts:

```
invariant( on,  1 =< x =< 3 ).
invariant( off, 1 =< x =< 3 ).

evolution( on,  x' = -x+5 ).
evolution( off, x' = -x ).

transition( on,  turn-off, off, [x=3], [c:=0] ).
transition( off, turn-on,  on,  [x=1], [c:=0] ).

init( on,  [x=1] ).
init( off, [x=3] ).
```

Given this representation of a hybrid automaton, the method, first, generates a set of CLP(R) rules (one rule per transition of the automaton), where each rule is extended with constraints. Table 1 shows our notations for translating a hybrid automaton to a logic program. A pair of arguments

location q	Q
state variables $x_1, ..., x_n$	$X_1, ..., X_n$
clocks $c_1, ..., c_m$	$C_1, ..., C_m$
value of variable x_k before and after transitions, $k = 1, ..., n$	Xi_k, Xo_k
last (wall clock) time c_k was reset value of c_k after transitions, $k = 1, ..., m$	Ci_k Co_k
current time	W
resetting clock c_k, $k = 1, ..., m$	$Co_k = W$
clock constraint $c_k \sim \delta$	$W - Ci_k \sim \delta$

Table 1: Translating Hybrid Automata to CLP

`Xi` and `Xo` is used for every variable x of the automaton: `Xi` represents the value of x before the transition, and `Xo` is the value of x after the transition. The clock c is modeled using a pair of arguments: `Ci` and `Co`: `Ci` is used to remember the last (wall clock) time c was reset, while `Co` is used to pass on this clock's value to the next transition. Resetting c represented by $c := 0$, is modeled by the constraint `Co = W`, where `W` is the current wall clock time. A clock constraint of the form $c \sim a$, in which $\sim \in \{=, <, >, \leq, \geq\}$ and a is a constant, is modeled by the constraint `W - Ci` $\sim$ `a`, where `Ci` is the last (wall clock) time c was reset.

We assume the existence of a solver for differential equations: given a linear differential equation and an initial condition the solver finds the solution. Then, the predicate `clp/1` translates the resulted solution into a CLP constraint in a straightforward way.

`trans(` $Q_1, \Sigma, Q_2, W, [Xi_1, ..., Xi_n, Ci_1, ..., Ci_m],$
$[Xo_1, ..., Xo_n, Co_1, ..., Co_m]$ `) :-`
`transition(` $Q_1, \Sigma, Q_2, W, [X_1 = v_1, ..., X_k = v_k],$
$[C_1 := 0, ..., C_j := 0]$ `),`
`evolution(`Q_1`,` E_1`),`
`init(`Q_1`,` $Init_1$`),`
`solution(`E_1`,` $Init_1$`,` S_1`),`
`clp(`S_1`),`
`invariant(`Q_1`,` I_1`),`
$\{I_1\{Xi_1/x_1, ..., Xi_n/x_n\}\}$,
$\{Xo_1 = v_1, ..., Xo_k = v_k, Xo_{k+1} = Xi_{k+1}, ..., Xo_n = Xi_n\}$,
$\{Co_1 = W, ..., Co_j = W, Co_{j+1} = Ci_{j+1}, ..., Co_m = Ci_m\}$.

The body of each rule for `trans/6` is composed of invariants, evolution laws, assignments and guards, which are specified using CLP(R) constraints. The CLP(R) constraints are enclosed within the curly braces, as it is the convention in most Prolog systems. The first four arguments of `trans/6` are the current state, the input or the synchronization label, the resulting state, and the wall clock time, respectively. The last two arguments are the value of variables before and after the transition.

Using the definition of `trans/6` above, the transitions of the automaton in Figure 2 are generated as follows. The two equations $5 - 3/e^t$ and $3/e^t$ are solutions to the differential equations in states `on` and `off`, respectively.

```
trans(on, turn_off, off, W, [Xi, Ci], [Xo, Co]) :-
     {5 - 3/exp(e, (W-Ci)) = 3, Xo = 3, Co = W}.

trans(off, turn_on, on,  W, [Xi, Ci], [Xo, Co]) :-
     {3/exp(e, (W-Ci)) = 1,   Xo = 1, Co = W}.
```

Note that the set of constraints used for modeling clocks along with constraints corresponding to the invariants, evolution laws, and guards on the transitions are directly han-

dled in CLP(R)[3]. Therefore, transitions of hybrid automata are modeled as *logic programs* [20, 32], physical quantities are represented as continuous quantities (i.e., not discretized) and the constraints imposed on them by transitions are modeled with *constraint logic programming over reals* [16].

Using this approach, the set of transition rules for core, Rod1 and Rod2 of the reactor temperature control system can be obtained similarly:

```
rod1(out1,add1,   in1, W,[Ci],[Co]) :- {W-Ci>=T, Co=Ci}.
rod1(in1, remove1,out1,W,[Ci],[Co]) :- {Co=W}.

rod2(out2,add2,   in2, W,[Ci],[Co]) :- {W-Ci>=T, Co=Ci}.
rod2(in2, remove2,out2,W,[Ci],[Co]) :- {Co=W}.

core(norod,add1,rod1,W,[Pi,Ci1,Ci2],[Po,Co1,Co2],F) :-
 ( F==1 -> Ci=Ci1; Ci=Ci2 ),
 { Pi<550, 10*exp(e, (W-Ci)/10)=50,
   Po=550, Co1=W, Co2=Ci2 }.

core(rod1,remove1,norod,W,[Pi,Ci1,Ci2],[Po,Co1,Co2],F) :-
 { Pi>510, 10*exp(e, (W-Ci1)/10)=50,
   Po=510, Co1=W, Co2=Ci2 }.

core(norod,add2,rod2,W,[Pi,Ci1,Ci2],[Po,Co1,Co2],F) :-
 ( F==1 -> Ci=Ci1; Ci=Ci2 ),
 { Pi<550, 10*exp(e, (W-Ci)/10)=50,
   Po=550, Co1=Ci1, Co2=W }.

core(rod2,remove2,norod,W,[Pi,Ci1,Ci2],[Po,Co1,Co2],F) :-
 { Pi>510, 50*exp(e, (W-Ci2)/10)=90,
   Po=510, Co1=Ci1, Co2=W }.

core(norod,_,shutdown,W,[Pi,Ci1,Ci2],[Po,Co1,Co2],F) :-
 ( F==1 -> Ci=Ci1; Ci=Ci2 ),
 { Pi<550, 10*exp(e, (W-Ci)/10)=50,
   Po=550, Co1=Ci1, Co2=Ci2 }.
```

Having modeled the transitions of an automaton as a CLP(R) program, the second step of our framework generates a *coinductive* predicate[4] that runs the automaton by calling transition rules of the automaton repeatedly, and advancing the wall clock time after each transition. If a system is composed of more than one automaton, the concurrent execution of all hybrid automata is modeled by allowing coroutining (realized via delay declarations of Prolog [32] in our subsequent implementation) within logic programming computations.

The coroutining facility of LP (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 symbols.

For instance, the coroutined coinductive predicate `runcore/7` which realizes the core automaton sends synchronization symbols to `runrod1/6` and `runrod2/6`. The coroutined coinductive predicates `runrod1/6` and `runrod2/6` will postpone running `rod1` and `rod2` until the synchronization symbols sent by `runcore/7` are received by them.

```
:- coinductive runcore(+, +, +, -, -, -, -).
runcore(Si, Pi, Fi, X, W, Ci1, Ci2) :-
  (H=add1; H=remove1; H=add2; H=remove2; H=shutdown),
  { W2>W },
  freeze(X, runcore(So, Po, Fo, Xs, W2, Co1, Co2)),
  core(Si, H, So, W, [Pi,Ci1,Ci2], [Po,Co1,Co2], 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 runrod1(+, +, +, -, -, -).
runrod1(Si1, Si2, T, [(H, W)| Xs], Ci1, Ci2) :-
  (
    (H=add1; H=remove1)
  ->
    (  H=add1
    ->
       freeze(Xs, runrod1(So1, Si2, T, Xs, Co1, Ci2))
    ;
       freeze(Xs, runrod1(So1, Si2, T, Xs, Co1, Ci2)
                 ;runrod2(So1, Si2, T, Xs, Co1, Ci2))
    ),
    rod1(Si1, H, So1, W, [Ci1], [Co1])
  ;
    H=shutdown, { W-Ci1<T, W-Ci2<T}).

:- coinductive runrod2(+, +, +, -, -, -).
runrod2(Si1, Si2, T, [(H, W)| Xs], Ci1, Ci2) :-
  (
    ( H=add2; H=remove2 )
  ->
    (  H=add2
    ->
       freeze(Xs, runrod2(Si1, So2, T, Xs, Ci1, Co2))
    ;
       freeze(Xs, runrod1(Si1, So2, T, Xs, Ci1, Co2)
                 ;runrod2(Si1, So2, T, Xs, Ci1, Co2))
    ),
    rod2(Si2, H, So2, W, [Ci2], [Co2])
  ;
    H=shutdown, { W-Ci1<T, W-Ci2<T}).
```

Note that `runcore/7` is declared as coinductive only on the first three arguments [26]; i.e., time will be ignored to check if `runcore/7` is cyclical. Similarly, the coinductive termination of `runrod1/6` and `runrod2/6` will depend on the first three non-time-related arguments.

Finally the `main/3` predicate represents the concurrent execution of all the components of the system. The first argument of `main/3` is the list of synchronization symbols along with their time-stamps, which is generated by the core and sent to two rods (non-deterministically). The time-stamps are not concrete, but related by set of constraints. Therefore, the solutions that are obtained by our system are more general than what one might expect. The second argument is the initial wall clock time, and `T` is the parameter of the system explained earlier. The condition that both rods are available initially, is specified using the set of constraints `{W - Tr1 = T, W - Tr2 = T}`, where `Tr1` and `Tr2` are clocks of Rod1 and Rod2, respectively. Similarly, `Tc1` and `Tc2` are the two clocks of the reactor core.

```
main(S, W, T) :- { W - Tr1 = T, W - Tr2 = T },
                 ( runrod1(s0, s0, T, S, Tr1, Tr2)
                  ;runrod2(s0, s0, T, S, Tr1, Tr2) ),
                 runcore(s0, 510, 1, S, W, Tc1, Tc2).
```

Verifying desired properties and parametric analysis: Once a model of a system is obtained, the next step is to formally

[3]In general, one must use an appropriate solver in order to solve the set of constraints; in our framework we use CLP(R).

[4]The outline of this system is presented in [26].

prove that the model satisfies properties of interest. For cyber-physical systems, these properties are often behavioral properties which are related to time (e.g., liveness).

To prove the properties, we need support (e.g., a model checker or a theorem prover) for formal verification of such properties. This support can be directly integrated into the modeling tool, or provided by separate verification tools: in the latter case they must be able to act on input expressed in our modeling language. Since logic programming is grounded in theorem proving, it can be used directly both as a specification language and also as a theorem prover.

Once a cyber-physical system is modeled as a coroutined co-CLP(R) program, the model can 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, via backtracking. 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 `Q` does not hold.

For the reactor temperature control system, the goal is to find the value of parameter `T` that guarantees the safety of the system, that is, the reactor is never shut down. 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 shut down. The verification requires 0.010 seconds on an Intel dual core 3.16 GHz processor with 4.00 GB of RAM.

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 (the initial wall clock time is zero) and any value of `T`$\leq$ 38.0666 fails, which indicates the safety of the system.

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

3. RELATED WORK

Model-based design and model-driven development [13] of complex real-time systems (and, more recently, CPSs) have long been very active areas of research. These efforts have resulted in many powerful tools and techniques for building formal models, validating the models, proving the properties of the models, and more recently automatically generating source code from the models.

Simulink, Stateflow, and MATLAB have been used widely for simulation and model-based design of embedded systems and complex systems. Simulink provides a graphical editor for specifying and modeling systems. It also supports simulation, automatic code generation, and continuous test and verification. Stateflow [22] allows the user to combine graphical and tabular representations, including state transition diagrams, flow charts, state transition tables, and truth tables, in order to model how the system reacts to events, time-based conditions, and external input signals. However, the fact that all these tools lack a formal semantics, makes the models built with them untrustworthy, especially for code synthesis purposes.

Ptolemy II [5, 6] is another tool used mainly in academia for modeling and simulation of heterogeneous systems. It uses an actor-oriented design approach to model components that communicate via ports. The interactions between actors are governed by a set of rules that are defined by various models of computation. These models of computation include discrete events, continuous time, finite state machines, synchronous reactive, etc. Ptolemy II modeling language also lacks a formal semantics. In particular, it is not clear how to combine the various models of computation to build a unified model.

UML 2 [3] is another tool, widely used for modeling software systems. SysML [24], an extension of a subset of UML, offers noteworthy improvements over UML and has many useful features for modeling CPSs. SysML's language is based on using three groups of diagrams: structure diagrams, dynamic diagrams and requirements diagrams, each of which may be composed of other diagrams such as internal block diagrams, behavior diagrams, sequence diagrams, activity diagrams, etc. Flow ports are used in SysML to represent what can go through a block (in and/or out), whether it is data, matter, or energy. While SysML defines the syntax of all these different diagrams, it does not provide a semantics for them. In other words, a SysML diagram could be interpreted differently by different users.

MARTE (Modeling and Analysis of Real Time and Embedded systems) [23, 21] is another tool for modeling and analyzing real-time and embedded systems. It provides facilities to annotate models with information required to perform specific analysis such as performance and schedulability analysis and also quantitative analysis. Unfortunately, MARTE also does not provide formal semantics for models.

Statecharts [12] has been also widely used as a behavioral modeling language for complex discrete-event systems and reactive systems. It provides a compact and expressive way of specifying complex systems. However, the lack of formal semantics makes it difficult to analyze the systems that are specified using this formalism.

Timed concurrent constraint (TCC) programming [29] comes close to our work. Timed concurrent constraint programming has been considered for verification [7]; however, perpetual computations cannot be considered in TCC, as it works with least fixed points of programs only. Our work can be regarded as a practical realization of TCC as well as its extension with coinduction to handle perpetual computations.

HyTech is a symbolic model checker for linear hybrid automata [14], where physical quantities exhibit constant derivatives. When dealing with non-linear hybrid automata, two methods: *rate translation* and *clock translation*, are employed by HyTech for *approximating* the non-linear hybrid automata with linear hybrid automata.

Henzinger and Ho have also analyzed the reactor temperature control system with HyTech. Since the hybrid automaton for the core involves non-constant derivatives, *rate translation* and *clock translation* are employed to convert it to a linear hybrid automaton. Using *rate translation*, the derivative of the temperature is approximated by rate intervals of $[-5,-1]$, $[-9,-5]$ and $[1,5]$ for states rod_1, rod_2 and *no_rod*, respectively. The analysis of this converted hybrid automaton by HyTech results in `T`$<$ 20.44 as a necessary and sufficient condition on parameter `T` to prevent the reactor from shut down. 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` where `T`$\leq$ 38.0666 the system is still safe.

Clock translation involves two steps: in the first step, the

non-linear variable θ is replaced by a clock variable t_θ; in the second step, the resulting automaton is *over approximated* by a linear automaton with the invariants $10t_\theta \leq 161$, $10t_\theta \leq 89$, and *true* in states rod_1, rod_2 and *no_rod*, respectively. The original differential equations are also replaced by $\dot{t_\theta} = 1$ in all the states of the core. The analysis of this converted hybrid automaton by HyTech results in T< 37.8 as a weaker condition on parameter T. This result is closer to the result generated by our system. However, in our approach, these parameters are computed with exact precision, as our framework for modeling hybrid automata directly handles the physical quantities constrained by non-constant derivatives. In other words, we are not using any translation and approximation method for converting non-linear derivatives to linear derivatives; therefore, our system computes the parameter bounds exactly, and is free of possible imprecision that might be introduced if approximation methods are used.

A variant of the reactor temperature control system is also analyzed using KRONOS [4], another symbolic model checker for timed and hybrid automata. In this variant of the system, temperature rises and decreases at fixed rates also; hence, it is not faithful to the original problem.

4. CONCLUSIONS

In this paper we presented a general framework for model-based design and verification of cyber-physical systems based on logic programming.

Cyber-physical systems are normally composed of set of processes that run concurrently. The processes interact with each other by sending and receiving signals and they possibly run for ever. We modeled CPSs as networks of hybrid automata [14, 1, 2], where each hybrid automaton is modeled as a logic program [20] extended with constraints, coroutining and coinduction. In our framework physical quantities are faithfully represented as continuous quantities and the constraints imposed on them by physical interactions are modeled with *constraint logic programming over reals*. By considering *coinductive logic programming* we are able to naturally model the non-terminating aspect of CPSs. Finally, concurrency is handled by allowing *coroutining* within logic programming computations.

We illustrated our approach by applying it to modeling and verification of the reactor temperature control system. We showed how our technique can be used for verifying the safety property of the system, as well as performing precise parametric analysis. In contrast to other approaches, our approach can handle non-constant derivatives directly, i.e., they do not need to be approximated (as long as a solver, such as the CLP(R) solver, is available that can handle them).

5. REFERENCES

[1] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229, 1992.

[2] R. Alur, T. A. Henzinger, and H. Wong-toi. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide, The (2Nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[4] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, pages 546–550, 1998.

[5] C. X. Brooks, E. A. Lee, and S. Tripakis. Exploring models of computation with ptolemy II. In *Proceedings of the 8th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2010, part of ESWeek '10 Sixth Embedded Systems Week, Scottsdale, AZ, USA, October 24-28, 2010*, pages 331–332, 2010.

[6] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. In *Proceedings of the IEEE*, pages 127–144, 2003.

[7] M. Falaschi and A. Villanueva. Automatic verification of timed concurrent constraint programs. *TPLP*, 6(3):265–300, 2006.

[8] G. Gupta, A. Bansal, R. Min, L. Simon, and A. Mallya. Coinductive logic programming and its applications. In *ICLP*, volume 4670 of *Lecture Notes in Computer Science*, pages 27–44. Springer, 2007.

[9] G. Gupta and E. Pontelli. A constraint-based approach for specification and verification of real-time systems. In *IEEE Real-Time Systems Symp*, pages 230–239, 1997.

[10] R. Gupta. Programming models and methods for spatiotemporal actions and reasoning in cyber-physical systems. In *NSF Workshop on CPS*, 2006.

[11] R. Gupta. Programming models and methods for spatiotemporal actions and reasoning in cyber-physical systems. In *NSF Workshop on CPS*, 2006.

[12] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, pages 231–274, 8 1987.

[13] C. Heitmeyer, M. Pickett, L. Breslow, D. W. Aha, J. G. Trafton, and E. I. Leonard. High assurance human-centric decision systems. In *ICSE-13 Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, San Francisco, CA, 2013. IEEE Press, IEEE Press.

[14] T. A. Henzinger and P. Hsin Ho. Hytech: The Cornell hybrid technology tool. In *Hybrid Systems*, volume 999 of *Lecture Notes in Computer Science*, pages 265–293. Springer-Verlag, 1994.

[15] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119, New York, NY, USA, 1987. ACM.

[16] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.

[17] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The clp(r) language and system. *ACM Trans. Program. Lang. Syst.*, 14(3):339–395, May 1992.

[18] E. A. Lee. Cyber physical systems: Design challenges. In *Proceedings of the 2008 11th IEEE Symposium on*

Object Oriented Real-Time Distributed Computing, ISORC '08, pages 363–369. IEEE Computer Society, 2008.

[19] E. A. Lee. Cyber physical systems: Design challenges. In *IEEE Symposium on Object Oriented Real-Time Distributed Computing*, ISORC '08, pages 363–369. IEEE Computer Society, 2008.

[20] J. W. Lloyd. *Foundations of logic programming / J.W. Lloyd*. Springer, Berlin, New York, 2nd, extended edition, 1987.

[21] F. Mallet and R. de Simone. Marte: A profile for rt/e systems modeling, analysis and simulation. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, pages 43:1–43:8, 2008.

[22] Mathworks. *Stateflow - Finite State Machine Concepts*. Mathworks, 2008.

[23] MG. Uml profile for marte: Modeling and analysis of real-time embedded systems, 2009.

[24] OMG. OMG Systems Modeling Language (OMG SysML), Version 1.3, 2012.

[25] N. Saeedloei. How the model of time affects model of a cyber-physical system. In *forthcoming*.

[26] N. Saeedloei. *Modeling and Verification of Real-Time and Cyber-Physical Systems*. PhD thesis, University of Texas at Dallas, Richardson, Texas, 2011.

[27] N. Saeedloei and G. Gupta. A logic-based modeling and verification of CPS. *SIGBED Review*, 8(2):31–34, 2011.

[28] N. Saeedloei and G. Gupta. Coinductive constraint logic programming. In *FLOPS*, volume 7294 of *Lecture Notes in Computer Science*, pages 243–259. Springer, 2012.

[29] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming. In *LICS*, pages 71–80, 1994.

[30] L. Simon. *Coinductive Logic Programming*. PhD thesis, University of Texas at Dallas, Richardson, Texas, 2006.

[31] L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-logic programming: Extending logic programming with coinduction. In *ICALP*, volume 4596 of *Lecture Notes in Computer Science*, pages 472–483, 2007.

[32] L. Sterling and E. Shapiro. *The art of Prolog (2nd ed.): advanced programming techniques*. MIT Press, Cambridge, MA, USA, 1994.

[33] X. Nicollin et al. An approach to the description and analysis of hybrid systems. In *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 149–178. Springer, 1992.