Analysis of Linear Hybrid Systems in CLP

Banda, Gourinath; Gallagher, John Patrick

Published in:
Lecture Notes in Computer Science

Publication date:
2009

Document Version
Publisher's PDF, also known as Version of record

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain.
- You may freely distribute the URL identifying the publication in the public portal.

Take down policy
If you believe that this document breaches copyright please contact rucforsk@ruc.dk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 14. Jun. 2020
Analysis of Linear Hybrid Systems in CLP *

Gourinath Banda and John P. Gallagher

Building 43.2, P.O. Box 260, Roskilde University, DK-4000 Denmark
Email: {gnbanda,jpg}@ruc.dk

Abstract. In this paper we present a procedure for representing the semantics of linear hybrid automata (LHAs) as constraint logic programs (CLP); flexible and accurate analysis and verification of LHAs can then be performed using generic CLP analysis and transformation tools. LHAs provide an expressive notation for specifying real-time systems. The main contributions are (i) a technique for capturing the reachable states of the continuously changing state variables of the LHA as CLP constraints; (ii) a way of representing events in the LHA as constraints in CLP, along with a product construction on the CLP representation including synchronisation on shared events; (iii) a framework in which various kinds of reasoning about an LHA can be flexibly performed by combining standard CLP transformation and analysis techniques. We give experimental results to support the usefulness of the approach and argue that we contribute to the general field of using static analysis tools for verification.

1 Introduction

In this paper we pursue the general goal of applying program analysis tools to system verification problems, and in particular to verification of real-time control systems. The core of this approach is the representation of a given system as a program, so that the semantics of the system is captured by the semantics of the programming language. Our choice of programming language is constraint logic programming (CLP) due to its declarative character, its dual logical and procedural semantics, integration with decision procedures for arithmetic constraints and the directness with which non-deterministic transition systems can be represented. Generic CLP analysis tools and semantics-preserving transformation tools are applied to the CLP representation, thus yielding information about the original system. This work continues and extends previous work in applying CLP to verification of real-time systems, especially [20, 13, 26].

We first present a procedure for representing the semantics of linear hybrid automata (LHAs) as CLP programs. LHAs provide an expressive notation for specifying continuously changing real-time systems. The standard logical model of the CLP program corresponding to an LHA captures (among other things) the reachable states of the continuously changing state variables of the LHA; previous related work on CLP models of continuous systems [20, 26] captured

* Work partly supported by the Danish Natural Science Research Council project SAFT: Static Analysis Using Finite Tree Automata.
only the transitions between control locations of continuous systems but not all states within a location. The translation from LHAs to CLP is extended to handle events as constraints; following this the CLP program corresponding to the product of LHAs (with synchronisation on shared events) is automatically constructed. We show that flexible and accurate analysis and verification of LHAs can be performed by generic CLP analysis tools coupled to a polyhedron library [5]. We also show how various integrity conditions on an LHA can be checked by querying its CLP representation. Finally, a path automaton is derived along with the analysis; this can be used to query properties and to generate paths that lead to given states.

In Section 2 the CLP representation of transition systems in general is reviewed. In Section 3 we define LHAs and then give a procedure for translating an LHA to a CLP program. Section 4 shows how standard CLP analysis tools based on abstract interpretation can be applied. In Section 5 we report the results of experiments. Section 6 concludes with a discussion of the results and related research.

2 Transition Systems

State transition systems can be conveniently represented as logic programs. The requirements of the representation are that the (possibly infinite) set of reachable states can be enumerated, that the values of state variables can be discrete or continuous, that transitions can be deterministic or non-deterministic, that traces or paths can be represented and that we can both reason forward from initial states or backwards from target states.

Various different CLP programs can be generated, providing a flexible approach to reasoning about a given transition system using a single semantic framework, namely, minimal models of CLP programs. Given a program \( P \), its least model is denoted \( M[P] \). In Section 6 we discuss the use of other semantics such as the greatest fixpoint semantics.

CLP Representation of Transition Systems A transition system is a triple \( \langle S, I, \Delta \rangle \) where \( S \) is a set of states, \( I \subseteq S \) is a set of initial states and \( \Delta \subseteq S \times S \) is a transition relation. The set \( S \) is usually of the form \( V_1 \times \cdots \times V_n \) where \( V_1, \ldots, V_n \) are sets of values of state variables. A run of the transition system is a sequence \( s_0, s_1, s_2, \ldots \) where \( \langle s_i, s_{i+1} \rangle \in \Delta, i \geq 0 \). A valid run is a run where \( s_0 \in I \). A reachable state is a state that appears in some valid run. A basic CLP representation is defined by a number of predicates, in particular init(\( S \)) and transition(\( S_1, S_2 \)) along with either rstate(\( S \)) or trace([\( S_k, \ldots, S_0 \)]). A program \( P \) is formed as the union of a set of clauses defining the transition relation transition(\( S_1, S_2 \)) and the initial states init(\( S \)) with the set of clauses in either Figure 1(i) or (ii). Figure 1(i) defines the set of reachable states; that is, \( s \) is reachable iff rstate(\( s \)) \( \in M[P] \) where \( s \) is the representation of state \( s \). Figure 1(ii) captures the set of valid run prefixes. \( s_0, s_1, s_2, \ldots \) is a valid run of the transition system iff for every non-empty finite prefix \( r_k \) of \( s_0, s_1, s_2, \ldots s_k \),
trace(rk) ∈ M[P], where rk is the representation of the reverse of prefix rk. Note that infinite runs are not captured using the least model semantics. The set of reachable states can be obtained directly from the set of valid run prefixes (since sk is reachable if s0, . . . , sk is the finite prefix of some valid run).

Alternatively, reasoning backwards from some state can be modelled. Figure 1(iii) gives the backwards reasoning version of Figure 1(i). Given a program P constructed according to Figure 1(iii), its least model captures the set of states s such that there is a run from s to a target (i.e. query) state. The predicate target(Sk) specifies the states from which backward reasoning starts. This is the style of representation used in [13], where it is pointed out that relation between forward and backward reasoning can be viewed as the special case of a query-answer transformation or so-called “magic-set” transformation [12]. Figure 1(iv) gives a further variation obtained by recording the dependencies on the target states; dqstate(S,Sk) ∈ M[P] iff a target state Sk is reachable from S. There are many other possible variants; system-specific behaviour is captured by the transition, init and target predicates while the definitions of rstate, qstate+ and so on capture various semantic views on the system within a single semantic framework. We can apply well-known semantics-preserving transformations such as unfolding and folding in order to gain precision and efficiency. For instance, the transition relation is usually unfolded away.

In Section 3 we show how to construct the transition relation and the init predicates for Linear Hybrid Automata. These definitions can then be combined with the clauses given in Figure 1.

3 Linear Hybrid Automata as CLP programs

Embedded systems are predominantly employed in control applications, which are hybrid in the sense that the system whose behaviour is being controlled has continuous dynamics changing in dense time while the digital controller has discrete dynamics. Hence their analysis requires modelling both discrete and continuous variables and the associated behaviours. The theory of hybrid automata [24] provides an expressive graphical notation and formalism featuring both discrete and continuous variables. We consider only systems whose variables change linearly with respect to time in this paper, captured by so-called Linear Hybrid Automata [2].
3.1 The language of Linear Hybrid Automata

Following [24], we formally define a linear hybrid automaton (LHA) as a 6-tuple \( \langle \text{Loc}, \text{Trans}, \text{Var}, \text{Init}, \text{Inv}, \text{D} \rangle \), with:

- A finite set \( \text{Loc} \) of locations also called control nodes, corresponding to control modes of a controller/plant.
- A finite set \( \text{Var} = \{x_1, x_2, \ldots, x_n\} \) of real valued variables, where \( n \) is the number of variables in the system. The state of the automaton is a tuple \( (l, X) \), where \( X \) is the valuation vector in \( \mathbb{R}^n \), giving the value of each variable. Associated with variables are two sets:
  - \( \dot{\text{Var}} = \{\dot{x}_1, \ldots, \dot{x}_n\} \), where \( \dot{x}_i \) represents the first derivative of variable \( x_i \) w.r.t time;
  - \( \text{Var}' = \{x'_1, \ldots, x'_n\} \), where \( x'_i \) represents \( x_i \) at the end of a transition.
- Three functions \( \text{Init}, \text{Inv} \) and \( \text{D} \) that assign to each location \( l \in \text{Loc} \) three predicates respectively: \( \text{Init}(l) \), \( \text{Inv}(l) \) and \( \text{D}(l) \). The free variables of \( \text{Init}(l) \) and \( \text{Inv}(l) \) range over \( \text{Var} \), while those of \( \text{D}(l) \) range over \( \text{Var} \cup \dot{\text{Var}} \). An automaton can start in a particular location \( l \) only if \( \text{Init}(l) \) holds. So long as it stays in the location \( l \), the system variables evolve as constrained by the predicate \( \text{D}(l) \) not violating the invariant \( \text{Inv}(l) \). The predicate \( \text{D}(l) \) constrains the rate of change of system variables.
- A set of discrete transitions \( \text{Trans} = \{\tau_1, \ldots, \tau_t\} \): \( \tau_k = \langle l, \gamma_k, \alpha_k, l' \rangle \) is a transition (i) uniquely identified by integer \( k \), \( 0 \leq k \leq t \); (ii) corresponding to a discrete jump from location \( l \) to location \( l' \); and (iii) guarded by a predicate \( \gamma_k \) and with actions constrained by \( \alpha_k \). The guard predicate \( \gamma_k \) and action predicate \( \alpha_k \) are both conjunctions of linear constraints whose free variables are from \( \text{Var} \) and \( \text{Var} \cup \dot{\text{Var}} \) respectively.

Figure 2 shows the LHA specification of a water-level monitor (taken from [21]).

3.2 LHA Semantics and Translation into CLP

LHA Semantics as a Transition System The semantics of LHAs can be formally described as consisting of runs of a labelled transition system (\( \text{LHA}_t \)), which we sketch here (full details are omitted due to lack of space).

A discrete transition is defined by \( (l, X) \rightarrow (l', X') \), where there exists a transition \( \tau = \langle k, l, \gamma, \alpha, l' \rangle \in \text{Trans} \) identified by \( k \); the guard predicate \( \gamma(k, l, l') \) holds at the valuation \( X \) in location \( l \) and \( k \) identifies the guarded transition; the associated action predicate \( \alpha(k, l, (l', X), (l', X')) \) holds at valuation \( X' \) with
which the transition ends entering the new location \( l' \) (\( k \) identifies the transition that triggers the action). If there are events in the system, the event associated with the transition labels the relation. Events are explained later in this section.

A delay transition is defined as: \( (l, X) \overset{\delta}{\rightarrow} (l^\delta, X^\delta) \) iff \( l = l^\delta \), where \( \delta \in \mathbb{R}_{\geq 0} \) is the duration of time passed staying in the location \( l \), during which the predicate \( \text{Inv}(l) \) continuously holds; \( X \) and \( X^\delta \) are the variable valuations in \( l \) such that \( D(l) \) and \( \text{Inv}(l) \), the predicates on location \( l \), hold. The predicate \( D(l) \) constrains the variable derivatives \( \dot{\text{Var}} \) such that \( X^\delta = X + \delta \cdot \dot{X} \).

Hence a delay transition of zero time-duration, during which the location and valuation remain unchanged is also a valid transition.

A run \( \sigma = s_0s_1s_2 \cdots \) is an infinite sequence of states \( (l, X) \in \text{Loc} \times \mathbb{R}^n \), where \( l \) is the location and \( X \) is the valuation. In a run \( \sigma \), the transition from state \( s_i \) to state \( s_{i+1} \) are related by either a delay transition or a discrete transition. As the domain of time is dense, the number of states possible via delay transitions becomes infinite following the infinitely fine granularity of time. Hence the delay transitions and their derived states are abstracted by the duration of time \( (\delta) \) spent in a location. Thus a run \( \sigma \) of an LHA is defined as \( \sigma = (l_0, X_0) \overset{\delta_0}{\rightarrow} (l_1, X_1) \overset{\delta_1}{\rightarrow} \cdots (l_j, X_j) \overset{\delta_j}{\rightarrow} \cdots (l_{j+k}, X_{j+k}) \), where \( \delta_j(j \geq 0) \) is the time spent in location \( l_j \) from valuation \( X_j \) until taking the discrete transition to location \( l_{j+1} \), when the guard \( \gamma_j \) holds. The new state \( (l_{j+1}, X_{j+1}) \) is entered with valuation \( X_{j+1} \) as constrained by \( \alpha_j \). Further \( \tau_j = (j, l_j, \gamma_j, \alpha_j, l_{j+1}) \in \text{Trans} \). Again during this time duration \( \delta_j \), the defined invariant \( \text{inv}(l_j) \) on \( l_j \) continues to hold, and the invariant \( \text{inv}(l_{j+1}) \) holds at valuation \( X_{j+1} \).

**LHA semantics in CLP** Table 1 shows a scheme for translating an LHA specification into CLP clauses. The transition predicate defined in the table is then used in the transition system clauses given in Figure 1. A linear constraint such as \( \text{Init}(l), \text{Inv}(l) \), etc. is represented as a CLP conjunction via \( \text{to_cLP}(\cdot) \).

The translation of LHAs is direct apart from the handling of the constraints on the derivatives on location \( l \), namely \( D(l) \) which is a conjunction of linear constraints on \( \text{Var} \). We add an explicit “time stamp” to a state, extending \( \text{Var}, \text{Var}' \) with time variables \( t, t' \) respectively giving \( \text{Var}_t, \text{Var}'_t \). The constraint \( D_t(l) \) is a conjunction of linear constraints on \( \text{Var}_t \cup \text{Var}'_t \), obtained by replacing each occurrence of \( \dot{x}_j \) in \( D(l) \) by \( (x_j' - x_j)/(t' - t) \) in \( D_t(l) \), where \( t', t \) represent the time stamps associated with \( x_j', \dot{x}_j \) respectively.

**Event Semantics in CLP** A system can be realized from two or more interacting LHAs. In such compound systems, parallel transitions from the individual LHAs rendezvous on events. Thus to model such systems the definition of an LHA is augmented with a finite set of events \( \Sigma = \{ \text{evt}_1, \ldots, \text{evt}_n \} \). The resulting LHA then is a seven-tuple \( \langle \text{Loc}, \text{Trans}, \text{Var}, \text{Init}, \text{Inv}, D, \Sigma \rangle \). Also associated with events is a partial function \( \text{event} : \text{Trans} \rightarrow \Sigma \), which labels (some) transitions with events.
LHA | CLP
---|---

**location l**  
**state variables** \(x_1, \ldots, x_n\)  
**state with time \(t\) and location \(l\)** \(S = [L, X_1, \ldots, X_n, T]\)

**state time**  
**state location**  
**temporal order on states**

\[\text{timeOf}(S, T) :- \text{lastElementOf}(S, T).\]
\[\text{locOf}(S, L) :- S = [L|\ldots].\]
\[\text{before}(S, S1) :- \text{timeOf}(S, T), \text{timeOf}(S1, T1), T < T1.\]

**Init(\(l\))**  
**Inv(\(l\))**  
**D(\(l\)) (using the derivative relation \(D(\(l\))\) explained in the text)**

\[\text{init}(S) :- \text{locOf}(S, L), \text{toClp}(\text{Init}(\(l\))).\]
\[\text{inv}(S) :- \text{locOf}(S, L), \text{toClp}(\text{Inv}(\(l\))).\]
\[\text{d}(S, S1) :- \text{locOf}(S, L), \text{timeOf}(S, T), \text{locOf}(S1, L), \text{timeOf}(S1, T1), \text{toClp}(\text{D}(\(l\))).\]

**LHA transition** \(\langle \(k, l, \gamma_k, \alpha_k, l'\)\)

\[\text{gamma}(K, L, S) :- \text{locOf}(S, L1), \text{toClp}(\gamma_k).\]
\[\text{alpha}(K, L, S1, S2) :- \text{locOf}(S1, L1), \text{locOf}(S2, L1), \text{toClp}(\alpha_k).\]

**delay transition**  
**discrete transition**

\[\text{transition}(S0, S1) :- \text{locOf}(S0, L0), \text{before}(S0, S1), \text{d}(S0, S1), \text{inv}(L0, S1).\]
\[\text{transition}(S0, S2) :- \text{locOf}(S0, L0), \text{before}(S0, S1), \text{d}(S0, S1), \text{gamma}(K, L0, S1), \text{alpha}(K, L0, S1, S2).\]

\[d([l, X^0_1, \ldots, X^0_n, Evt^0_1, \ldots, Evt^0_{ne}, T_0], [l, X^1_1, \ldots, X^1_n, Evt^1_1, \ldots, Evt^1_{ne}, T_1]) \leftarrow LC_{d \times t}, LC_{d}, X^1_1 = X^0_1 + d_{x_1} \times (T_1 - T_0), \ldots, X^1_n = X^0_n + d_{x_n} \times (T_1 - T_0), Evt^1_1 = 0, \ldots, Evt^1_{ne} = 0.\]

The translation of predicate \(\alpha\) to \text{alpha/4} is modified to encode the event notification as follows.

\[\text{alpha}(T, L_0, [L_1, X_1, \ldots, X_n, Evt_1, \ldots, Evt_{ne}], \ldots, [L_0, X_1', \ldots, X_n', Evt_1', \ldots, Evt_{ne}']) \leftarrow \text{LAC}_{1, \ldots, \text{LAC}_{na}}, E_{xe}.\]

**Table 1. Translation of LHAs to CLP**

In our framework we model event notification as constraints. To this end events are modelled as discrete state variables ranging over values \(\{0, 1\}\); these variables are initialized to 0; on an event-labelled transition the variable corresponding to the labelled event is set to value 1 to raise that event while the other event variables are reset to 0; on a transition not labelled with an event all event variables are reset to 0; the event variables remain constant within a location and thus at most one event variable can have value 1 at any time.

In the CLP translation the state vector of an LHA with events is given by \(S = [\text{Loc}, X_1, \ldots, X_n, Evt_1, \ldots, Evt_{ne}, T]\), where for \(j = 1\) to \(ne\) the variable \(Evt_j\) represents \(evt_j \in \Sigma\). The raising of event \(evt_{re}\) is modelled as a constraint \(E_{re} = (\forall_{i=1}^{ne} Evt_i = c_i)\) where the value \(c_i\) equals 1 if \(i = re\) or 0 otherwise. Similarly the constraint corresponding to no event raised is \(E_{none} = (\forall_{i=1}^{ne} Evt_i = 0)\).

We modify the previous translation of the derivative constraints \(D(l)\) to incorporate events, yielding the following definition of \(d/2\).

\[d([l, X^0_1, \ldots, X^0_n, Evt^0_1, \ldots, Evt^0_{ne}, T_0], [l, X^1_1, \ldots, X^1_n, Evt^1_1, \ldots, Evt^1_{ne}, T_1]) \leftarrow LC_{d \times t}, LC_{d}, X^1_1 = X^0_1 + d_{x_1} \times (T_1 - T_0), \ldots, X^1_n = X^0_n + d_{x_n} \times (T_1 - T_0), Evt^1_1 = 0, \ldots, Evt^1_{ne} = 0.\]
where if $Evt_{re}$ is the label on the transition $E_{xe} = E_{re}$ else $E_{xe} = E_{none}$ when there is no event label (where $E_{re}$ and $E_{none}$ are as defined above). The translation of the $\gamma$ and inv constraints is unaffected by the addition of events apart from the extension of the state vector with the event variables.

### 3.3 Parallel Composition of Linear Hybrid Automata

The discrete transitions of two automata $LHA_1$ and $LHA_2$ with events $\Sigma_1$ and $\Sigma_2$ respectively, synchronize on an event $evt$ as following:

- if $evt \in \Sigma_1 \cap \Sigma_2$, then the discrete transitions $\tau_i \in Trans_1$ and $\tau_j \in Trans_2$ labelled with the event $evt$ must synchronize;
- if $evt \notin \Sigma_1 \cap \Sigma_2$ but $evt \in \Sigma_1$, then the discrete transition $\tau_i \in Trans_1$ can occur simultaneously with a zero duration delay transition of $LHA_2$, and similarly if $evt \in \Sigma_2$.

Finally, a delay transition of $LHA_1$ with a duration $\delta$ must synchronize with a delay transition of $LHA_2$ of the same duration.

Synchronization is enforced by constructing a product of the associated labelled transition systems ($LHA_t$). In our framework, the product of two labelled transition systems is realized as the composition $\boxtimes$ of the corresponding CLP programs, which corresponds closely to the LHA product construction as defined in [24]. More efficient encodings have been investigated [28, 26]. $\boxtimes$ is defined as $CLP_1 \boxtimes CLP_2 = \{C_1 \boxtimes C_2 \mid C_1 \in CLP_1, C_2 \in CLP_2\}$ where $C_1 = p2(X) \leftarrow c_1(X, X'), p1(X')$ is a clause in $CLP_1$, $C_2 = q2(Y) \leftarrow c_2(Y, Y'), q1(Y')$ is a clause in $CLP_2$ and $C_1 \boxtimes C_2 = p2.q2(X, Y) \leftarrow c_1(X, X') \land c_2(Y, Y'), p1.q1(X' \cup Y')$. Here $p2.q2$ and $p1.q1$ are new predicates unique to the associated predicate pairs in the original programs. The notation $X$ above, where $X$ is a set, denotes a tuple of the elements of $X$ in some fixed order (e.g. alphabetical order).

The operation $\boxtimes$ is quadratic in that the number of clauses in the resultant $CLP_1 \boxtimes CLP_2$ equals $|CLP_1| \times |CLP_2|$. However with shared events many of them can be eliminated since their constraints are not consistent due to the event constraints. If there is a constraint $E = 1$ on a shared event variable $E$ in some transition, then it will only form a consistent product clause with other clauses with $E = 1$. Following this composition we successfully built product systems of: (i) a task scheduler (ii) a train gate controller (LHA) (iii) a train gate controller (TSA/TA), from their constituent automata.

### 3.4 Integrity Constraints on LHAs

The semantics of LHA given in Section 3 places certain restrictions on runs, in particular that the relevant invariant is satisfied so long as the automaton remains in a location. One approach to ensuring that the CLP program generates only valid runs is to build into the transitions all the necessary constraints. An alternative is to check statically that certain constraints on the CLP program
are satisfied. This enables us to generate a simpler CLP model than otherwise, omitting some “runtime” checks. These integrity checks also represent natural sanity checks on the LHA and checking them can locate specification errors. The conditions are as follows. (i) The invariants are convex (with respect to the given rates of change of the state variables). (ii) The invariants are satisfied when a location is entered via a transition from another location or by initial conditions. (iii) The enabling constraint ($\gamma$) on a transition out of location either implies the invariant on that location, or becomes true as soon as the invariant ceases to be true (e.g. the invariant might be $x < 10$ and the transition constraint $x = 10$, where $x$ increases with time). This condition should be checked because the language does permit discontiguous invariant and guard conditions which might result in a situation where the invariant becoming invalid with the outgoing discrete transition guard not yet enabled.

We check these by running queries on predicates representing the negation of the integrity conditions, which ought to fail if the conditions are met. For example nonconvex($L$) is defined as nonConvex($L$) ← locOf($S_0, L$), inv($L, S_0$), d($S_0, S_2$), inv($L, S_2$), d($S_0, S_1$), before($S_1, S_2$), negInv($L, S_1$). negInv($L, S_1$) is the negation of the invariant on location $L$ (the constraint language is closed under negation) and in general consists of several clauses since the negation of the invariant may be a disjunction.

We have noticed that condition (iii) above is violated in several LHAs in the literature. Typically, a transition that can fire “at any time” (perhaps triggered by an interrupt event) has $\gamma = true$. Hence this remains “firable” even when the invariant is false. If a violation occurs we can repair it by simply conjoining the invariant on the location to the transition constraint $\gamma$; this is the implicit intention (enforced by the LHA semantics) and achieves the required behaviour.

4 Analysis of the CLP representations

The concrete analysis problem is firstly to obtain an extensional, finite representation of the model; by extensional is meant a representation in which we can query and check model properties using simple constraint operations. In fact, we use CLP clauses in which the bodies consist only of constraints to represent the model (or an over-approximation of the model) of a CLP program. Thus checking of properties reduces in many cases to constraint solving [35].

Computing a Model The usual immediate-consequences operator $T_P$ for logic programs is modified for CLP programs [25]. The sequence $T_P^i(\emptyset)$, $i = 0, 1, 2, \ldots$ is an increasing (w.r.t. subset ordering) sequence of programs whose clauses consist of constrained facts, clauses of form $p(X) ← c(X)$ where $c(X)$ is a linear constraint. If the sequence stabilises with $T_P^i(\emptyset) = T_P^{i+1}(\emptyset)$ for some $i$, then $T_P^i(\emptyset)$ is the concrete model of $P$.

If the sequence does not stabilise (within some predefined resource limits) we are forced to compute an abstract model. There are various ways of doing this. Currently we use two abstractions: an abstraction defined in [13] and a
classical abstract interpretation based on convex polyhedral hulls with widening and narrowing operators [11]. More complex and precise abstractions, such as those based on the powerset domain consisting of finite sets of convex polyhedra could be used [4]. The resulting abstract model can be represented as a set of constrained facts, as in the concrete model (since a convex polyhedron can be represented as a linear constraint).

Checking Properties The concrete or abstract models can be used to check state safety properties. Suppose the constraint \( c(\bar{X}) \) defines some “bad” state. Then the query \( \neg c(\bar{X}), \text{rstate}(\bar{X}) \) is evaluated on the model. If this query has no solutions, the safety property is verified. Note that if the query succeeds, with an answer constraint, this means that the bad state is possibly reachable. The answer constraints yield a description of the state variables in the unsafe states.

An alternative approach to checking safety properties is to define the unsafe states as target states and compute the set of reaching states w.r.t. those target states, namely those states from which there exists a run to a target state. We then query this set to see whether any of the initial states are among them. If not, then the safety property is satisfied. This is the approach used in [13] for example. As before, we obtain in general an answer constraint. Suppose we obtain the answer that \( \text{qstate}(\bar{X}), c(\bar{X}) \) is a reaching state that is also an initial state. Then we can use this information to strengthen the conditions on the initial states; by adding extra checks to the initial states to ensure that the constraint \( \neg c(\bar{X}) \) holds, we can ensure satisfaction of the safety condition.

The CLP program constructed according to the schema summarised in Figure 1(iv) gives additional flexibility and prove certain simple path properties. In the backwards version, the model of the program captures dependencies between the target state and reaching states. This model can be used to answer queries such as “Is there a run from a given state \( s_1 \) to a target state \( s_2 \)?” or “Is it possible to reach state \( s_1 \) before state \( s_2 \)?” The answers to the queries, as before, could yield constraints giving conditional answers, with constraints linking the values of states in the start and end states.

Checking Path Properties One approach to checking properties of paths in the transition system is to use an explicit representation of the traces in the CLP program, using the scheme from Figure 1(ii) for example. However this can make the model of the program infinite even when the set of states is finite. Another approach is to build a path automaton while computing a model. A path automaton is a set of transitions of a tree grammar of the form \( f_i(v) \rightarrow v' \). This means that the state \( v' \) can be reached from state \( v \) via the \( i \)th transition. During the construction of the model we record which transitions can be applied in a state, where the states \( v_1, \ldots, v_k \) are identifiers for the constrained facts in the models (see above). With this automaton we can: (a) generate paths that reach some particular state; (b) check whether there is some “stuck” state which can be reached but from which no other state is reachable; (c) check regular
path properties using standard automata operations. Analysis techniques based on tree automata [16, 17] are applied for such analyses.

5 Experiments

In this section, experiments applying the CLP analysis tools to LHA systems are described. The tool-chain is as follows: \( \text{LHA}_{\text{spec}} \xrightarrow{\text{parse}} \text{CLP} \xrightarrow{\text{PE}} \text{CLP}_{\text{trans}} \xrightarrow{\text{prod}} \text{CLP}_{\text{trans}} \xrightarrow{\text{Reach}} \text{CLP}_{\text{model}} \xrightarrow{\text{FTA}} \text{CLP}_{\text{path}} \). Here \( \text{LHA}_{\text{spec}} \) represents LHA specifications; CLP represents arbitrary CLP programs; CLP\(_{\text{trans}}\) represents a subclass of CLP whose clauses are of the form \( p_i(\bar{X}) \leftarrow c(\bar{X}, \bar{X}') \), where \( \bar{X}, \bar{X}' \) are tuples of distinct variables, \( c(\bar{X}, \bar{X}') \) is a conjunction of constraints over the variables and \( p_i, p_j \) are non-constraint predicates (and \( p_j \) is possibly absent); CLP\(_{\text{model}}\) is a subclass of CLP\(_{\text{trans}}\) where the clause bodies consist only of constraints; CLP\(_{\text{path}}\) is a CLP program defining a finite tree automaton.

The \( \text{parse} \rightarrow \) step translates LHA specifications given in a simple source language which is simply a textual representation of the usual graphic specifications, into CLP according to the procedure defined in Section 3. In step \( \text{PE} \rightarrow \) the partial evaluator LOGEN [29] is applied in order to unfold the definitions of all the predicates except the state predicate \( \text{rstate} \) (or variations such as \( \text{qstate} \) or \( \text{dqstate} \)). Furthermore LOGEN filters are chosen to cause the single state argument to be replaced by a tuple of arguments for the state variables, with a separate state predicate for each location. The resulting programs are in the class CLP\(_{\text{trans}}\). In step \( \text{prod} \rightarrow \) the product of LHAs in CLP\(_{\text{trans}}\) form can be computed if necessary, yielding another CLP\(_{\text{trans}}\) program. The step \( \text{Reach} \rightarrow \) uses an analysis tool to compute (an approximation of) the least model of a CLP program, represented as a program in the class CLP\(_{\text{model}}\). This program can be used to check properties of the set of reachable states in the original LHA. Finally the step \( \text{FTA} \rightarrow \) derives a CLP program in the form of a finite automaton generating the set of possible paths in the preceding CLP\(_{\text{trans}}\) program. This program can be used to check path properties of the original LHA. Steps \( \text{parse} \rightarrow \) and \( \text{PE} \rightarrow \) are standard and are not discussed in detail here. Step \( \text{prod} \rightarrow \) is also a straightforward implementation of the definition of product. We focus on the analysis phases.

CLP Analysis Tools Our analysis tools are developed to analyse arbitrary CLP programs, in applications such as termination analysis and complexity analysis [7, 31, 10, 18]. We have not developed any analysis tools specifically for the CLP programs resulting from LHA translation. This is an important principle since CLP is a target representation for many different source languages; thus the same set of CLP analysis tools is applicable regardless of the source language. We have previously used similar tools to analyse PIC microprocessor code [23, 22]. We use an implementation in Ciao-Prolog that employs the Parma Polyhedra Library (PPL) [5] to handle the constraints. Note that this tool-set is a current snapshot; one of the key advantages of the approach is that improved CLP program
analyses can be incorporated as they become available. In particular we expect that analysis tools based on the powerset of convex polyhedra with widenings \[4\] will play an important role.

TP: This computes the model of a program using a least fixpoint iteration. Well-known optimisations such as the semi-naive technique are incorporated. If it terminates (within some predefined period) the computed model is precise (provided the input program contains only linear constraints – the analyser makes some safe linear over-approximation of non-linear constraints).

DP99: This computes an over-approximation of the least model; the technique was proposed by Delzanno and Podelski in \[13\] and used in their experiments. Each predicate with argument \(\bar{x}\) is approximated by a finite set of conjunctions of linear constraints over \(\bar{x}\). The “widening” (which as they point out does not in fact guarantee termination) on successive approximations \(F, F'\) returns each conjunction that is obtained from some conjunct \(c_1 \land \ldots \land c_n \in F'\) by removing all conjuncts \(c_i\) that are strictly entailed by some conjunct \(d_j\) of some “compatible” constrained atom \(d_1 \land \ldots \land d_m \in F\) where “compatible” means that \(c_1 \land \ldots \land d_m\) is satisfiable. It does terminate in some cases where TP does not.

For cases that do not terminate within some resource bound using one of the above two tools, we use the CHA tool. This is an abstract interpreter computing an over-approximation of the least model, based on convex polyhedral approximations. It computes one polyhedron for each predicate. Termination is guaranteed by one of the known widenings [21,5]. CHA incorporates state-of-the-art refinements such as an optional narrowing phase, “widening up-to” [21] and delayed widening. The tool is available on-line (http://saft.ruc.dk/CHA/).

In Table 2 we summarise the results of computing a model or approximate model for a number of examples. As discussed earlier, querying this model to check safety properties or dependencies is computationally straightforward using a constraint solver. The number of locations in the automaton is \(Q\) and number of discrete transitions is \(\Delta\). The number of clauses in the translated CLP programs includes the clauses for the delay transitions. For the FTA size, the table reports the size for the most precise analysis that terminated. Timings are given in seconds and the symbol \(\infty\) indicates failure to terminate within a time-out duration of 300 seconds.

Description of the Examples The Fischer Protocol, Water Level and Scheduler are taken from [21]; version (E) of the Scheduler is constructed using the product construction synchronised by events while the other one is specified directly as a single automaton. The Leaking Burner is taken from [1]; the Train Controller is specified in [2] including a number of state variables such as the distance of the train and the angle of the gate. [26] provide a simpler form as a timed automaton (TA) specifying only the events. The steam boiler problem is work in progress. The nine last examples are taken from [13]; these are specified as discrete automata (which can be modelled as special cases of LHAs)\(^1\). In all

\(^1\) We gratefully acknowledge the use of the examples from [13] which are available for download and were translated into standard CLP programs for our experiments.
of these examples we check the same properties as in the original references, though for the examples from [13] we cover only the safety properties. (We discuss analysis of liveness properties in Section 6).

Many of these examples can be analysed with $T_P$, and thus without using abstraction since the input programs contain only linear constraints. Though the number of states is infinite due to continuously changing state variables, the reachable states can often be captured by a finite set of constraints. We believe that this observation is related to the existence of a finite number of regions in timed automata [3]. We were only forced to use the convex polyhedral analyser for the Leaking Burner problem and the DP99 abstraction for a few of the discrete examples. We were able to verify some properties that could not be verified with convex hull analysis, such as the bound on the $k_1$ variable, which is the number of lower priority tasks waiting and/or preempted in the Scheduler example [21]. The time to compute a concrete model is of course often less than that needed for a convex polyhedral abstraction. However, even when a system can be analysed without abstraction, state-space explosion could force abstractions to be introduced; the Bakery series of examples indicates the exponential growth in the time to compute the model as more processes are introduced. In our experience the abstraction introduced in [13] (DP99 in Table 2) is of limited usefulness. We could find no examples apart from the ones in [13] in which a system failed to terminate in $T_P$ but terminated with DP99.

<table>
<thead>
<tr>
<th>Name</th>
<th>$Q$</th>
<th>$\Delta$</th>
<th>$CLP$</th>
<th>$T_P$ (secs.)</th>
<th>$DP99$ (secs.)</th>
<th>$CHA$ (secs.)</th>
<th>$FTA$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fischer Protocol</td>
<td>6</td>
<td>8</td>
<td>53</td>
<td>0.19</td>
<td>0.61</td>
<td>0.23</td>
<td>79</td>
</tr>
<tr>
<td>Leaking Burner</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>0.02</td>
<td>5</td>
</tr>
<tr>
<td>Scheduler</td>
<td>3</td>
<td>11</td>
<td>19</td>
<td>1.02</td>
<td>9.59</td>
<td>0.42</td>
<td>101</td>
</tr>
<tr>
<td>Scheduler (E)</td>
<td>3</td>
<td>11</td>
<td>15</td>
<td>0.94</td>
<td>40.16</td>
<td>0.63</td>
<td>93</td>
</tr>
<tr>
<td>Steam Boiler</td>
<td>10</td>
<td>23</td>
<td>166</td>
<td>0.50</td>
<td>0.74</td>
<td>0.88</td>
<td>146</td>
</tr>
<tr>
<td>Switch</td>
<td>2</td>
<td>2</td>
<td>15</td>
<td>0.03</td>
<td>0.08</td>
<td>0.04</td>
<td>33</td>
</tr>
<tr>
<td>Train Controller</td>
<td>3</td>
<td>8</td>
<td>26</td>
<td>0.15</td>
<td>1.44</td>
<td>0.15</td>
<td>57</td>
</tr>
<tr>
<td>Train Train</td>
<td>3</td>
<td>3</td>
<td>18</td>
<td>0.07</td>
<td>0.38</td>
<td>0.09</td>
<td>51</td>
</tr>
<tr>
<td>Train Gate</td>
<td>4</td>
<td>10</td>
<td>29</td>
<td>0.06</td>
<td>0.31</td>
<td>0.09</td>
<td>43</td>
</tr>
<tr>
<td>Train Controller System</td>
<td>14</td>
<td>20</td>
<td>143</td>
<td>4.54</td>
<td>$\infty$</td>
<td>8.00</td>
<td>324</td>
</tr>
<tr>
<td>Train Controller System (TA)</td>
<td>14</td>
<td>20</td>
<td>169</td>
<td>1.01</td>
<td>36.68</td>
<td>2.17</td>
<td>85</td>
</tr>
<tr>
<td>Water Level</td>
<td>4</td>
<td>4</td>
<td>27</td>
<td>0.04</td>
<td>1.26</td>
<td>0.16</td>
<td>56</td>
</tr>
<tr>
<td>Bakery 2</td>
<td>3</td>
<td>8</td>
<td>9</td>
<td>0.08</td>
<td>0.93</td>
<td>0.04</td>
<td>14</td>
</tr>
<tr>
<td>Bakery 3</td>
<td>3</td>
<td>21</td>
<td>24</td>
<td>3.02</td>
<td>127.72</td>
<td>0.07</td>
<td>134</td>
</tr>
<tr>
<td>Bakery 4</td>
<td>3</td>
<td>52</td>
<td>58</td>
<td>143.19</td>
<td>$\infty$</td>
<td>0.34</td>
<td>1456</td>
</tr>
<tr>
<td>Bbuffer 1</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>0.01</td>
<td>0.07</td>
<td>0.02</td>
<td>4</td>
</tr>
<tr>
<td>Bbuffer 2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>4</td>
</tr>
<tr>
<td>MutAst</td>
<td>7</td>
<td>20</td>
<td>21</td>
<td>0.27</td>
<td>0.60</td>
<td>0.09</td>
<td>30</td>
</tr>
<tr>
<td>Network</td>
<td>1</td>
<td>16</td>
<td>17</td>
<td>$\infty$</td>
<td>0.12</td>
<td>0.30</td>
<td>13</td>
</tr>
<tr>
<td>Ticket 2</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td>$\infty$</td>
<td>2.82</td>
<td>0.04</td>
<td>25</td>
</tr>
<tr>
<td>Ubuffer</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>$\infty$</td>
<td>0.24</td>
<td>0.05</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 2. Experimental Results
Nevertheless the ideas behind DP99 are valid; other approaches based on the powerset of convex polyhedra with true widenings [4] will replace DP99 in future experiments.

6 Related Work and Conclusions

The idea of modelling transition systems of various kinds as CLP programs goes back many years. Our work contributes to two areas, CLP modelling and CLP proof techniques. On the one hand we add to the literature on CLP modelling techniques in showing that the continuous semantics of LHAs can be captured in CLP programs. On the other we add to the existing literature showing that effective reasoning can be carried out on CLP programs, and display a family of different reasoning styles (e.g. forward, backwards, state dependencies) that can be generated from a single system specification and handled with a single set of analysis tools.

Though comparing various formalisms is not the aim of our work, it is noteworthy that LHAs are more expressive than other formalisms such as Timed Automata (TA) [3] or other finite automata as discussed in [9]. Consequently, from the modelling perspective, our framework has two advantages over the Uppaal [6] model checker or any other TA model checkers. Firstly, we can directly handle LHAs having multi-rate dynamics\(^2\), whereas Uppaal mandates that an LHA specification be compiled down into a TA specification before being verified. Secondly, Uppaal restricts the clock variables to be compared only with natural numbers, i.e. guards such as \(x > 1.1, 10 * x > 11\) or \(x > y\) are not permitted [27], while the CLP(Q) system permits us to handle such constraints. Such advantages in expressiveness might be balanced by extra complexity and possible non-termination; in the case of non-termination, we resort to abstraction. Furthermore, the state transition system CLP\(_{trans}\) (see Section 5) can be directly input to Uppaal after relatively minor syntactic changes. Thus, our approach can also be regarded as potentially providing a flexible LHA input interface incorporating abstraction for TA model checkers (among others). We experimented with Uppaal, in this manner, to verify the Water-level control system using CLP as an intermediate representation.

Gupta and Pontelli [20] and Jaffar et al. [26] describe schemes for modelling timed (safety) automata (TSAs) as CLP programs. Our work has similarities to both of these, but we go beyond them in following closely the standard semantics for LHAs, giving us confidence that the full semantics has been captured by our CLP programs. In particular the set of all reachable states, not only those occurring at transitions between locations as in the cited works, is captured. Delzanno and Podelski [13] develop techniques for modelling discrete transition systems which in our approach are special cases of hybrid systems.

Another direction we could have taken is to develop a direct translation of LHA semantics into CLP, without the intermediate representation as a transition

\(^2\) A timed automaton has variables called clocks that vary at a single rate i.e. \(\dot{x} = 1, \dot{y} = 1\), while an LHA can have variables that vary at different rates i.e. \(\dot{x} = 1, \dot{y} = 2\).
system. For example a “forward collecting semantics” for LHAs is given in [21], in the form of a recursive equation defining the set of reachable states for each location. It would be straightforward to represent this equation as a number of CLP clauses, whose least model was equivalent to the solution of the equation. A technique similar to our method of deriving the \( d \) predicate for the constraints on the derivatives could be used to model the operator \( S/D \) in [21] representing the change of state \( S \) with respect to derivative constraints \( D \). The approach using transition systems is a little more cumbersome but gives the added flexibility of reasoning forwards or backwards, adding traces, dependencies and so on as described in Section 2. The clauses we obtain for the forward transition system (after partial evaluating the \texttt{transition} predicate) are essentially what would be obtained from a direct translation of the recursive equation in [21]. The toolchain in Section 5 differs only in the choice of driver in the \textit{PE} step.

We focus on the application of standard analysis techniques using the bottom up least fixpoint semantics, but there are other approaches which are compatible with our representations. In [20] the CLP programs are run using the usual procedural semantics; this has obvious limitations as a proof technique. In [26] a method called “co-inductive” tabling is used. Path properties are expressed in [20] using CLP list programs; again this is adequate only for proofs requiring a finite CLP computation. [13] contains special procedures for proving a class of safety and liveness properties. Liveness properties require a greatest fixpoint computation (also used in [26] and [19]), which our toolset does not yet support. Our approach focusses on using standard CLP analysis techniques and abstractions based on over-approximations. However, the introduction of greatest fixpoint analysis engines into our framework is certainly possible and interesting.

A somewhat different but related CLP modelling and proof approach is followed in [8, 30, 32, 14, 15, 33, 34]. This is to encode a proof procedure for a modal logic such as CTL, \( \mu \)-calculus or related languages as a logic program, and then prove formulas in the language by running the interpreter (usually with tabling to ensure termination). The approach is of great interest but adapting it for abstraction of infinite state systems seems difficult since the proof procedures themselves are complex programs. The programs usually contain a predicate encoding the transitions of the system in which properties are to be proved, and thus could in principle be coupled to our translation. In this approach the full prover is run for each formula to be proved, whereas in ours an (abstract) model is computed once and then queried for difference properties. Our approach is somewhat more low-level as a proof technique but may offer more flexibility and scalability at least with current tools.

**Acknowledgements** We thank the LOPSTR’2008 referees and Julio Peralta for helpful comments and suggestions.

**References**


