CEGAR Based Bounded Model Checking of Discrete Time Hybrid Systems

Federico Mari and Enrico Tronci*

Dipartimento di Informatica, Università di Roma "La Sapienza",
Via Salaria 113, 00198 Roma, Italy
Tel.: +39 06 4991 8361; Fax: +39 06 8541 842
{mari,tronci}@di.uniroma1.it

Abstract. Many hybrid systems can be conveniently modeled as *Piecewise Affine Discrete Time Hybrid Systems* PA-DTHS. As well known *Bounded Model Checking* (BMC) for such systems comes down to solve a *Mixed Integer Linear Programming* (MILP) feasibility problem.

We present a SAT based BMC algorithm for automatic verification of PA-DTHSs. Using *Counterexample Guided Abstraction Refinement* (CEGAR) our algorithm *gradually* transforms a PA-DTHS verification problem into larger and larger SAT problems.

Our experimental results show that our approach can handle PA-DTHSs that are more then 50 times larger than those that can be handled using a MILP solver.

1 Introduction

Automatic analysis of *Hybrid Systems* poses formidable challenges both from a modeling as well as from a verification point of view. In fact the simultaneous presence of continuous and discrete variables may soon lead to *state explosion*, thus preventing completion of the verification process.

Many verification tools (*model checkers*) are available for automatic verification of hybrid systems. Here are a few examples. *Linear Hybrid Systems* (LHS) can be verified using HyTech [18,2,1]. If we restrict ourselves to LHSs in which all continuous variables have time derivative equal to 1 (clocks) then the UPPAAL [20,29] model checker can be used.

If we use a discrete model for time then we have *Discrete Time Hybrid Systems* (DTHSs). Many systems can be modeled or approximated using DTHSs. A model checker for (possibly nonlinear) DTHS is CMurphi [25,13,7].

Tools originally designed for hardware verification have also been used for hybrid systems verification. For example, in [28] SMV [23,26] has been used for verification of chemical processing systems.

By restricting our attention to linear DTHS we can design more efficient verification algorithms. *Piecewise Affine Discrete Time Hybrid Systems* (PA-DTHS) are an important subclass of DTHS. PA-DTHSs are DTHS whose behaviour can

* Corresponding author.

A. Bemporad, A. Bicchi, and G. Buttazzo (Eds.): HSCC 2007, LNCS 4416, pp. 399–412, 2007.

be defined using linear constraints involving real as well as discrete variables. An important subclass of PA-DTHSs are *Discrete Hybrid Automata* (DHA).

For DHA quite efficient *Mixed Integer Linear Programming* (MILP) based verification algorithms have been designed [4] and implemented within the symbolic DHA model checker HYSDEL [27,19].

SAT based *Bounded Model Checking* (BMC) [5,14] has turned out to be quite effective on hardware (e.g. see [21,31]), as well as on software systems (e.g. see [10,9]). Thus trying to use BMC in a DTHS setting is a quite natural step. For HYSDEL DHA this has been studied in [15].

Another interesting class of PA-DTHS is the one that can be handled by MathSAT [3,22]. SAT based *Counterexample Guided Abstraction Refinement* (CEGAR) [11] has also turned out to be a quite effective enhancement to BMC. This is the case for hardware verification (e.g. see [21]) as well as for software verification (e.g. see [17]). Our main contributions can be summarized as follows.

In Section 3 we define a quite large class of *Piecewise Affine* DTHS. Our class of DTHSs strictly contains those that can be handled by, e.g., UPPAAL, HYSDEL or MathSAT (Section 4).

In Section 5 we show how the BMC problem for a DTHS $\mathcal{H}$ in our class can be cast as an MILP feasibility problem $P_{\mathcal{H}}$. Of course, following [4], we could solve such MILP problem using a solver (e.g. GLPK [16], or CPLEX [12]). However for feasibility problems having many discrete variables (our target here) MILP solvers tend to be rather inefficient. For this reason our approach will be that of transforming our BMC problem for DTHSs into a SAT problem.

In Section 6 we give a *sound* and *complete* (up to ϵ) algorithm to transform our MILP problem $P_{\mathcal{H}}$ into a *Boolean Linear Programming* (BLP) problem $F^{\epsilon}_{\mathcal{H}}$. Our ϵ approximation of $P_{\mathcal{H}}$ only discretizes the continuous variables of $P_{\mathcal{H}}$. Effectiveness of our transformation rests on the fact that we *do not* discretize the real coefficients of the constraints in $P_{\mathcal{H}}$. Instead, for each contraint we generate a compact CNF (*conjunctive normal form*) representation of the set of assignments that falsify it.

In Section 7 we show how our BLP problem $F^{\epsilon}_{\mathcal{H}}$ can be effectively transformed into an equivalent SAT problem $B^{\epsilon}_{\mathcal{H}}$.

In Section 8, building on the transformation defined in Sections 7, we present a CEGAR based algorithm to solve our BLP problem $F^{\epsilon}_{\mathcal{H}}$ using a SAT solver. This yields a SAT based CEGAR BMC algorithm for our class of DTHSs. To the best of our knowledge for the class of systems we consider here no CEGAR BMC algorithms have been proposed in the literature. For example, our class of systems cannot be handled by the BMC algorithm proposed in [15].

In Section 9 we present experimental results showing effectiveness of the proposed approach. Given a DTHS $\mathcal{H}$ in our class we have essentially two ways of carrying out BMC: use an MILP solver to check feasibility of $P_{\mathcal{H}}$ *or* use our SAT based CEGAR approach. We present experimental results showing that using our SAT based CEGAR approach we can handle systems that are more than 50 times larger than those that can handled by an MILP solver.

2 Background

Notation 1. Let $X = [x_1, \ldots x_n]$, $Y = [y_1, \ldots y_n]$, be finite sequences (lists) of variables. By abuse of language we may regard sequences as set and we use $\cup$ to denote list concatenation. We denote with $f(X := Y)$ the expression $f([x_i := y_i | i = 1 \ldots n])$, that is the simultaneous substitution of the variables in X with those in Y. Moreover if X is clear form the context we just write $f(Y)$ for $f(X := Y)$.

Let $x \in X$, we denote with $\mathcal{D}_x$ the domain of x, that is the set on which x ranges. A *valuation* (over a list of variables X) is a function that maps each variable $x \in X$ to a value in $\mathcal{D}_x$. That is, a valuation is a point in $\times_{x \in X} \mathcal{D}_x$.

A *linear expression* over a list of variables X is a linear combination of variables in X with real coefficients. A *constraint* (over a list of variables X) is an expression of the form $\alpha \leq b$ where: α is a linear expression and b is a real constant. A *convex predicate* (over a list of variables X) is a finite conjunction of constraints. A *predicate* is defined as follows. A convex predicate is a predicate. If A and B are predicates then $(A \wedge B)$ is a predicate and $(A \vee B)$ is a predicate. As a *syntactic sugar*, if y is a discrete variable we will write $y < b$ for $y \leq b - 1$, $y > b$ for $y \geq b + 1$, $y \neq b$ for $((y \leq b - 1) \wedge (y \geq b + 1))$.

Classically a *Mixed Integer Linear Programming* (MILP) problem [8] is a linear optimization problem. However, in our context, we are only interested in finding *feasible* solutions. For this reason our definition of MILP does not include an objective function to be minimized.

Definition 1. *Let $M(X)$ be a convex predicate. The* Mixed Integer Linear Programming *(MILP) problem for $M(X)$ consists in finding a valuation V s.t.: 1. $M(V)$ holds 2. for all $x \in X$, $V(x) \in \mathcal{D}_x$. In other words we are looking for a satisfying assignment for the variables of M. An* Integer [Boolean] Linear Programming *(ILP [BLP]) problem is an MILP problem with only integer [boolean] variables.*

Usually an MILP problem P is represented as a list of constraints. That is $P = \{\sum_{j=1}^{n} a_{ij} x_j \leq b_j \mid i = 1, \ldots m \text{ and for } j = 1, \ldots n,\ x_j \in \mathcal{D}_{x_j}\}$. Using standard MILP techniques (e.g. [8]) it is possible to prove the following propositions.

Proposition 1. *Given a predicate $P(X)$ there exist a list Q of boolean variables and a convex predicate $L(Q, X)$ s.t. $\forall X$ $[P(X)$ iff $\exists Q L(Q, X)]$.*

Proposition 2. *Given an MILP problem P there exists an MILP problem Align(P) s.t. 1. P is feasible iff Align(P) is feasible; 2. All variables in Align(P) are nonnegative.*

Proposition 3. *Given an ILP problem P there exists a BLP problem Q s.t. 1. P is feasible iff Q is feasible; 2. All variables in Q are boolean.*

It can be easily shown [8] that, in general, MILP feasibility is an NP-complete problem. However there are quite effective *solvers* (e.g. CPLEX [12], GLPK [16]) that can handle non trivial MILP optimization problems.

Remark 1. Note however that MILP solvers are designed to solve optimization problems rather than feasibility problems. In particular the *branch-and-bound* heuristics typically implemented in MILP solvers are not effective on feasibility problems since there is no objective function for computing *the bound* in the branch-and-bound process.

For the above reason, state-of-the-art commercial MILP solvers like CPLEX perform as poorly as state-of-the-art open source MILP solvers like GLPK on MILP feasibility problems with many discrete variables (our case here). In fact, feasibility problems do not have an objective function and thus CPLEX sophisticated heuristics tend to be quite ineffective.

3 Piecewise Affine Discrete Time Hybrid Systems

In this section we introduce a class of *Piecewise Affine Discrete Time Hybrid Systems* (DTHSs for short).

Many classes of piecewise affine hybrid systems have been studied in the literature, e.g. [2,20]. The same holds true for piecewise affine discrete time hybrid systems, e.g. [4,27,15,3]. The class of systems we are considering is essentially the one used in [30].

Definition 2. *A* Discrete Time Hybrid System *(DTHS for short) is a 6-tuple* $\mathcal{H} = (Q, X, Init, Inv, r, R)$ *where:*

- $Q = [q_1, \ldots q_k]$ *is a finite sequence of* discrete *variables. Each variable* $q \in Q$ *ranges on a finite subset* $[l_q, u_q]$ *of the integers* $\mathbb{Z}$. *Thus* $\mathcal{D}_q = [l_q, u_q]$.
- $X = [x_1, \ldots x_n]$ *is a finite sequence of* real-valued *variables. Each variable* $x \in X$ *ranges on a bounded interval* $[l_x, u_x]$ *of the reals* $\mathbb{R}$. *Thus* $\mathcal{D}_x = [l_x, u_x]$. *Of course* Q *and* X *are disjoint lists.*
- $Init(Q, X)$ *is a* predicate *over* $Q \cup X$.
- $Inv(Q, X)$ *is a* predicate *over* $Q \cup X$.
- $r(Q, X, X')$ *is a* predicate *over* $Q \cup X \cup X'$, *where* $X' = [x'_1, \ldots x'_n]$.
- $R(Q, X, Q', X')$ *is a* predicate *over* $Q \cup X \cup Q' \cup X'$, *where* $Q' = [q'_1, \ldots q'_k]$.

As usual primed variables denote "next state" values. Usually, when modeling a DTHS, R is used to define *reset* transitions, that is $R(q, x, q', x')$ implies $q \neq q'$. This is also our modeling style. However, from a formal point of view, Definition 2 only requires that R defines next state values for discrete states.

The list of *state variables* S for the DTHS $\mathcal{H} = (Q, X, \text{Init}, \text{Inv}, r, R)$ is $S = Q \cup X$. A *state* for $\mathcal{H}$ is a valuation $s = (q, x)$ of S, where q is a valuation of Q and x is a valuation of X.

A *run* for the DTHS $\mathcal{H}$ is a sequence $(q(0), x(0))$, $(q(1), x(1))$, ... of states of $\mathcal{H}$ such that the following conditions are satisfied: 1. $\text{Init}(q(0), x(0)) \wedge \text{Inv}(q(0), x(0))$; 2. For all $k \geq 0$, $(R(q(k), x(k), q(k+1), x(k+1)) \vee (r(q(k), x(k), x(k+1)) \wedge q(k+1) = q(k))) \wedge \text{Inv}(q(k+1), x(k+1))$.

If $\pi = (q(0), x(0)), (q(1), x(1)), \ldots$ is a run of $\mathcal{H}$ we denote with $\pi(k)$ the k-th element of π. That is $\pi(k) = (q(k), x(k))$.

A state (q, x) of $\mathcal{H} = (Q, X, \text{Init}, \text{Inv}, r, R)$ is *k-reachable* if there exists a run π of $\mathcal{H}$ and there exists a $t \leq k$ s.t. $\pi(t) = (q, x)$.

In this paper we focus on bounded model checking of safety properties. That is, our goal is to check that for system $\mathcal{H}$ no *error state* is k-reachable. To this end we need to define the set of error states. This can be easily done using a predicate. The above considerations lead to the following definition.

Definition 3. *Let $\mathcal{H} = (Q, X, Init, Inv, r, R)$ be a DHTS, $E(Q, X)$ be a predicate over $(Q \cup X)$ and k be a natural number. We say that the triple $(\mathcal{H}, E, k)$ is* safe *(or that $\mathcal{H}$ is k-safe w.r.t. E) iff there is no run π of $\mathcal{H}$ for which there exists a $t \leq k$ s.t. $\pi(t) = (q, x)$ and $E(q, x)$ holds (i.e. $E(q, x) = 1$). In other words, no k-reachable state of $\mathcal{H}$ satisfies E.*

4 An Example of DTHS

We give an example of DTHS that will be useful to clarify the class of systems we are targeting. Consider a system consisting of k water pumps and $n > k$ tanks. Pump i $(i = 1, \ldots k)$ has (discrete) position $p_i \in \{1, \ldots n\}$, where $p_i = j$ means that pump i is above tank j. Each pump moves forward $(w_i = 1)$ and backward $(w_i = 0)$ between positions 1 and n. Pump i starts from position i. Each pump must stay in a position for at least α time units and must leave the position after at most $\beta \geq \alpha$ time units.

To compute the amount of water in a tank it is useful to introduce a boolean variable $z_{i,j}$ s.t. $z_{i,j} = 1$ iff water tank i is getting water from pump j, that is $z_{i,j} = 1$ iff $p_j = i$. This can be defined with the predicate $Z_{i,j} = (z_{i,j} = 0 \vee p_j = i) \wedge (z_{i,j} = 1 \vee p_j \neq i)$.

Each tank may get water from any of the pumps. Water flows out from tank i from a sink that can be open $(u_i = 1)$ or closed $(u_i = 0)$. The dynamics of the water level x_i of tank i satisfies the following constraint: $P_i = (x_i' \leq x_i - \gamma u_i + \sum_{j=1}^{k} \eta z_{i,j}) \wedge (x_i' \geq x_i - \mu u_i + \sum_{j=1}^{k} \theta z_{i,j})$, where: γ, μ model, respectively, min and max flow of water out of tank i and η, θ model, respectively, max and min flow of water out of pump j.

Water demand is modeled as follows. A tank sink can stay open for at most ζ time units and can stay closed for at most ξ time units. Moreover, the number of open sinks is at most $\Lambda \leq n$ and at least $\Gamma \leq \Lambda \leq n$. That is, $\sum_{i=1}^{n} u_i \geq \Gamma$ and $\sum_{i=1}^{n} u_i \leq \Lambda$. From the above description we can define a DTHS $\mathcal{H}$.

We expect that each tank i, has *enough* water $(x_i \geq m)$, but not *too much* $(x_i \leq M)$. Thus the predicate E representing our *error condition* can be defined as follows: $\bigvee_{i=1}^{n}((x_i < m) \vee (x_i > M))$.

Given an horizon k, our goal is to check that the triple $(\mathcal{H}, E, k)$ is *safe*.

Remark 2. Our class of DTHSs cannot be handled using the UPPAAL model checker, since we are not restricted to clock variables (e.g. see [20]). For example, tank water levels (x_i) cannot be modeled as UPPAAL clocks.

Remark 3. Our class of DTHS cannot be handled with HYSDEL since in our invariant we may have constraints consisting of discrete state variables. Such

constraints are not handled by DHA (e.g. see [15]). For example our invariant constraints $\sum_{i=1}^{n} u_i \geq \Gamma$, $\sum_{i=1}^{n} u_i \leq \Lambda$ cannot be handled using HYSDEL. Moreover we can handle nondeterminism in the discrete time dynamics which cannot be modeled with the DHA in [15].

Remark 4. Our class of DTHS cannot be handled using the MathSAT tool since we have constraints involving continuous as well as discrete variables, whereas MathSAT only handles constraints built out of one type of variables, i.e. only continuous variables or only discrete variables (e.g. see [6]). For example x_i's constraints in $x_i' \leq x_i - \gamma u_i + \sum_{j=1}^{k} \eta z_{i,j}$, $x_i' \geq x_i - \mu u_i + \sum_{j=1}^{k} \theta z_{i,j}$ cannot be modeled using MathSat since they involve continuous (x_i) as well as discrete ($z_{i,j}$) variables.

5 BMC of DTHS as a MILP Problem

The BMC problem for DTHSs can be cast as an MILP problem. For DHA this has been shown in [4]. Along the same line we can show that the same holds for DTHS.

Theorem 1. *Let $\mathcal{H}$ = (Q, X, Init, Inv, r, R) be a DTHS, $E(Q, X)$ be an error condition for $\mathcal{H}$ and k be a natural number. Then there exists a convex predicate $M(Y, Q_0, X_0, \ldots Q_k, X_k)$ s.t.*

- *All variables in Y are booleans;*
- *($\mathcal{H}$, E, k) is safe iff the MILP problem $M(Y, Q_0, X_0, \ldots Q_k, X_k)$ does not have a solution*
- *Let $y, (q(0), x(0)), \ldots (q(k), x(k))$ be a solution to the MILP problem $M(Y, Q_0, X_0, \ldots Q_k, X_k)$. Then $\phi = (q(0), x(0)), \ldots (q(k), x(k))$ is a path in $\mathcal{H}$ containing an error state. That is, there is a $t \leq k$ s.t. $E(q(t), x(t))$.*

6 From MILP to BLP

A BMC problem for DTHSs can be transformed into an MILP problem (Theorem 1). In order to transform a BMC problem for DTHSs into a SAT problem, here we show how an MILP problem can be transformed into a *Boolean Linear Programming* (BLP) problem. We do this in three steps. First we transform our MILP problem P into a problem P_1 in which all variables are nonnegative (Proposition 2). Second, given $\epsilon > 0$, we transform problem P_1 into an *Integer Linear Programming* (ILP) problem P_2^{ϵ} (Proposition 4). Finally, we transform P_2^{ϵ} into a BLP problem F^{ϵ} (Proposition 3).

Approximating continuous variables with discrete ones generates discretization errors. To account for such errors we relax P constraints. Let P be an MILP problem and $\epsilon > 0$. We define the ϵ-relaxation P_{ϵ} of P which is obtained by replacing the rhs b of each constraint in P with $(b + \epsilon)$. Thus ϵ defines the relaxation we are willing to accept on P constraints.

The *size of a linear constraint* is the number of variables occurring in it. The *size of an MILP* problem P ($|P|$) is the sum of the sizes of its constraints. Because of lack of space we omit the proof of the following proposition.

Proposition 4. *Given $\epsilon > 0$ and an MILP problem P in which all variables are nonnegative there exists a linear time (in $|P|$) algorithm $\mathcal{Q}_\epsilon$ s.t. 1. $\mathcal{Q}_\epsilon(P)$ is an ILP problem. 2. If P is feasible then $\mathcal{Q}_\epsilon(P)$ is feasible. 3. If $\mathcal{Q}_\epsilon(P)$ is feasible then P_ϵ is feasible.*

Assembling the transformations in Propositions 2, 4, 3 we can constructively prove the following theorem.

Theorem 2. *Let P be an MILP problem and let $\epsilon > 0$. Then there exists a linear time (in $|P|$) algorithm $\mathcal{L}_\epsilon$ s.t. 1. $\mathcal{L}_\epsilon(P)$ is a BLP problem. 2. (*Soundness*) If P is feasible then $\mathcal{L}_\epsilon(P)$ is feasible. 3. (*Completeness*) If $\mathcal{L}_\epsilon(P)$ is feasible then P_ϵ is feasible. 4. If $\mathcal{L}_\epsilon(P)$ is feasible [infeasible] then $\forall \epsilon' \geq \epsilon$ [$\forall 0 \leq \epsilon' \leq \epsilon$] $\mathcal{L}_{\epsilon'}(P)$ is feasible [infeasible].*

Theorem 2 says that by using $\mathcal{L}_\epsilon(P)$ instead of P we never have false negatives. That is we never declare safe an unsafe system. Moreover we never have false positives using $\mathcal{L}_\epsilon(P)$ instead of P_ϵ. On the other hand, $\mathcal{L}_\epsilon(P)$ may be feasible and P may be infeasible. That is we may declare unsafe a safe system. Note however that, as usual, by making ϵ *small enough* we can make the difference between P and P_ϵ arbitrarily small, at the price of making $|\mathcal{L}_\epsilon(P)|$ grow. Infact, it can be shown that the number of boolean variables of $\mathcal{L}_\epsilon(P)$ coming from the discretization of continuous variables is proportional to $\log_2(\epsilon^{-1})$.

Theorem 3, which proof we omit because of lack of space, guarantees that we never run into an infinite sequence of false positives. That is by taking smaller and smaller values for ϵ eventually we have that $\mathcal{L}_\epsilon(P)$ is infeasible (and so is P by Theorem 2) or that $\mathcal{L}_\epsilon(P)$ is feasible and so is P.

Theorem 3. *If P is infeasible then there exists $\epsilon > 0$ s.t. $\mathcal{L}_\epsilon(P)$ is infeasible.*

7 From BLP to SAT

Using Theorems 1 and 2, given a *tolerance* ϵ we can transform a BMC problem for DTHS $\mathcal{H}$ into a BLP problem M. In this Section we show how M can be transformed into a SAT problem.

Of course we may get a similar result by discretizing the real-valued coefficients in the linear inequalities in M, implementing floating point arithmetic and then translating the all problem into SAT. This is the approach followed, e.g., in the CBMC model checker [9]. However, if we follow such an approach even small systems will result in huge *Conjunctive Normal Forms* (CNFs), e.g. see the CBMC manual in [9]. Hence here we follow a different approach.

We represent a constraint P in M with its non-satisfying assignments. This yields an often compact CNF representation for P. The details follow.

Let $P = \sum_{i=n}^{n} a_i x_i \leq b$ be a constraint in M. We are looking for a CNF F s.t. $F(X) = 1$ iff $P(X) = 1$. Let V be the set of assignments that make P

```
 1 int blp2sat(a, b) {m1 = 0; m2 = 0;
 2 k = index of first undefined value in x;
 3 if (k > n)
 4 {if (∑_{i=1}^{n} a[i]x[i] > b) {F = (F ∧ clause(x)); return 1; }
 5  else return 0;}
 6 if (min(a, x) > b) {F = (F ∧ clause(x)); return 1;}
 7 if (a[k] > 0) {x[k] = 1} else {x[k] = 0;}
 8 m1 = blp2sat(a, b);
 9 if (m1 > 0) { x[k] = 1 - x[k]; m2 = blp2sat(a, b); }
10 return (m1 + m2); }
```

Fig. 1. Sketch of algorithm BLP2SAT

false. That is, $V = \{x \mid P(x) = 0\}$. The characteristic function for V is: $\chi_V = \bigvee_{(v_1,\ldots v_n)\in V} \bigwedge_{i=1}^{n} x_i^{v_i}$, where $x_i^{v_i}$ is $(x_i = v_i)$. Thus $x_i^1 = x_i$, $x_i^0 = \bar{x_i}$.

Let $F(X) = \neg\chi_V(X)$. Then $P(X) = 1$ iff $F(X) = 1$. In fact $P(X) = 1$ iff $\chi_V(X) = 0$ iff $\neg\chi_V(X) = 1$. Now $F(X) = \neg\chi_V(X) = \wedge_{(v_1,\ldots v_n)\in V} \vee_{i=1}^{n} x_i^{\bar{v_i}}$. Thus $F(X)$ is a CNF s.t. $F(X) = 1$ iff $P(X) = 1$.

Using the above procedure for all constraints in M we can build a CNF formula W s.t. $M(X) = 1$ iff $W(X) = 1$.

Example 1. As an example, let $X = [x_1, x_2, x_3]$ and $P(X) = 3x_1 + 2x_2 + 4x_3 \leq 5$. Then $V = \{(0, 1, 1), (1, 0, 1), (1, 1, 1)\}$. Thus $F(X) = (x_1 \vee \bar{x_2} \vee \bar{x_3}) \wedge (\bar{x_1} \vee x_2 \vee \bar{x_3}) \wedge (\bar{x_1} \vee \bar{x_2} \vee \bar{x_3})$.

Note that we do not need to discretize the real valued coefficients in the linear inequalities of M. However, to make the above approach interesting we need to generate F in a time efficient way and in such a way that F is not *too large*.

Our idea to produce a *small* CNF F in a *fast* way is the following. Let $P = \sum_{i=n}^{n} a_i x_i \leq b$. Let $y = (y_1, \ldots y_n)$ be an assignment s.t. $P(y) = 0$. If $a_i > 0$ and $y_i = 0$ then also for $z = (y_1, \ldots y_{i-1}, 1, y_{i+1}, y_n)$ we have $P(z) = 0$. Let F be the CNF representing P. Then $F = \ldots \wedge (x_1^{y_1} \vee \ldots \vee x_i^0 \vee \ldots x_n^{y_n}) \wedge (x_1^{y_1} \vee \ldots \vee x_i^1 \vee \ldots x_n^{y_n}) \wedge \ldots = \ldots \wedge (x_1^{y_1} \vee \ldots \vee x_{i-1}^{y_{i-1}} \vee x_{i+1}^{y_{i+1}} \vee \ldots x_n^{y_n})$.

That is, the value of y_i matters only if it is 1. Analogously, if $a_i < 0$ the value of y_i matters only if it is 0.

More formally, we say that variable y_i is *relevant* in P for the assignment y iff $(a_i > 0 \wedge y_i = 1) \vee (a_i < 0 \wedge y_i = 0)$. We denote with $\Gamma(P, y)$ the list of variables relevant in P for assignment y.

With the above considerations we can write the CNF for a constraint P as follows: $F(X) = \bigwedge_{v:P(v)=0} \bigvee_{x\in\Gamma(P,v)} x^{\overline{v(x)}}$.

Example 2. A CNF for P as in Example 1 is: $F = (\bar{x_2} \vee \bar{x_3}) \wedge (\bar{x_1} \vee \bar{x_3})$.

The above considerations suggest that, if our goal is to find a non-satisfying assignment, then we should first try setting to 1 the variables with large positive coefficients and to 0 the variables with large (in modulo) negative coefficients.

To generate F in a fast way, as a first step we *reorder* variables $x_1, \ldots x_n$ so that x_i precedes x_j in the ordering iff one of the following conditions hold: 1.

$a_i < 0$ and $a_j < 0$ and $a_i < a_j$; 2. $a_i > 0$ and $a_j > 0$ and $a_i > a_j$; 3. $a_i < 0$ and $a_j > 0$.

Let v be a partial assignment for X. That is $\forall x \in X\ v \in \{0, 1, \perp\}$. Function clause($v$) returns the clause $\bigvee_{x:v(x)\in\{0,1\}} x^{\overline{v(x)}}$.

Putting all the above considerations together leads to the *search-and-prune* algorithm sketched in the algorithm `blp2sat()` in Figure 1. As we shall see, our experimental results show that the search-and-prune heuristic used in function `blp2sat()` often allows us to quickly generate compact CNF representations for a linear inequality $\sum_{i=1}^{n} a_i x_i \leq b$.

In the following we describe the algorithm `blp2sat()` in Figure 1. First of all, `blp2sat()` assumes that variables have already been ordered as explained above. Let $P = \sum_{i=n}^{n} a_i x_i \leq b$ our constraint. Function `blp2sat()` has two arguments: the array a s.t. $a[i] = a_i$ and the rhs of P, b. The global variable x in `blp2sat()` stores the partial assignments found during the computation.

Line 2 of `blp2sat()` in Figure 1 computes in variable k the index of the first undefined assignment. If (line 3) this index is greater than n (number of variables in P) then all variables have been assigned a value and we go to line 4. Since all variables are defined we can evaluate the lhs $\sum_{i=1}^{n} a[i]x[i]$ of P. If $\sum_{i=1}^{n} a[i]x[i] > b$ (line 4) then constraint P is not satisfied and x contains a non-satisfying assignment for P. In such a case we add to F the clause clause(x) (line 4) and return 1, since we added one clause to F. If $\sum_{i=1}^{n} a[i]x[i] \leq b$ then x is a satisfying assignment for P and thus we generate no clause and return 0 (line 5).

If $k \leq n$ then $x[k]$ has not been assigned a value. Line 6 checks if there is hope to complete x to a satisfying assignment. If this is not the case then we add clause(x) to F (line 6) and return 1. If x may be completed to a satisfying assignment we go to line 7 and choose the value of $x[k]$ so as to make the lhs of P as large as possible in an attempt to find a non-satisfying assignment. After that in line 8 we recursively call `blp2sat()` and get in $m1$ the number of clauses produced by the setting for $x[k]$ chosen in line 7.

If the choice of $x[k]$ in line 7 has not produced non-satisfying assignments ($m1 = 0$) then, a fortiori, also the other possible assignment in line 7 for $x[k]$ will not produce any non-satisfying assignment. In other words, if $m1 > 0$ we consider the assignment (1 - $x[k]$), line 9, else we do not need to. Finally, line 10, we return the total number of non-satisfying assignments produced. Note how the test in line 7 allows us to prune the search tree for non-satisfying assignments.

8 Solving an MILP Problem with a SAT Based CEGAR

Rather than transforming an MILP problem into a SAT problem in one big step we can use a *Counterexample Guided Abstraction Refinement* (CEGAR) approach to *gradually* transform an MILP problem into a SAT problem.

First of all, given a linear constraint P, we can modify `blp2sat()` in Figure 1 so that each time it is called it generates at most `MaxClause` new clauses for the linear constraint P. The idea is that we can often prove correctness (UNSAT) or

find an error (SAT) without generating all clauses for each constraint. This can be done by storing the *state* σ of the computation of `blp2sat()`. State σ summarizes all the information we need to save to safely *stop* and, above all, *resume*, the computation of `blp2sat()`. Initially the state is empty (meaning `blp2sat()` is at its start point). We call `cegar-blp2sat()` the version of `blp2sat()` thus modified.

In the following we describe the algorithm `cegar_milp2sat()` that *gradually* transforms an MILP problem into a SAT problem using a CEGAR based approach. A sketch of `cegar_milp2sat()` is in Figure 2.

First, for each constraint $(A[i], b[i])$ (that is $\sum_{j=1}^{n} A[i]_j x_j \leq b[i]$) we denote with σ_i the state of the computation of `blp2sat(`$A[i], b[i]$`)`. Initially σ_i is empty for $i = 1, \ldots n$. In the following lines refer to function `cegar_milp2sat()` in Figure 2.

The argument of `cegar_milp2sat()` is an MILP problem P, i.e. the pair (A, b).

Line 2 replaces P with Align(P) (see Proposition 2). Line 3, given the tolerance ϵ and using Proposition 4, computes the number of bits needed for x in order to achieve tolerance ϵ. Line 4 replaces in the MILP problem all continuous variables with discrete ones (Proposition 4). Line 5 transforms all discrete variables into boolean ones (Proposition 3). Line 6 initializes the state of the computation (i.e. clause generation) for each constraint.

Line 7, for each constraint i initializes the *history* $\gamma[i]$ of i. The history of constraint i records when i turned out to be false under a candidate solution ρ. More specifically. Assume we are at iteration t of the while loop starting at line 9. The history $\gamma[i]$ is a bitvector of size m that has a 1 in position $\gamma[i][j]$ iff at iteration $(t - (j - m + 1))$ constraint i was false under assignment ρ. Thus to update $\gamma[i]$ we simply shift it of 1 bit to the right and set $\gamma[i][m-1]$ to 1 if the constraint is not satisfied by ρ, to 0 else. In our implementation we have set $m = 8$.

Line 8 initializes to 1 (empty set) the generated CNF F. Line 9 begins the main loop of `cegar_milp2sat()`. Line 10 begins the CEGAR for loop. Lines 10-12 generate an approximation of our MILP problem. Namely, for each constraint i, in line 11 we order constraint variables as described in Section 7 and in line 12 we generate at most `MaxClause` clauses for constraint i with lhs coefficients $A[i]$ and rhs coefficient $b[i]$ (function `cegar_blp2sat()`). Line 13 calls the SAT solver on CNF F. We use ZChaff [24,32] here as a SAT solver. The result may be UNSAT or SAT with an assignment ρ.

If we get UNSAT we are done since the original problem is then also UNSAT (line 14). However if we get SAT (line 15) the assignment ρ may not be a satisfying assignment for MILP (A, b) since we only generated at most `MaxClause` clauses for each constraint. In this case we go to line 16.

In lines 17-21, for each constraint i we update its *history* $\gamma[i]$. This is done by shifting $\gamma[i]$ one bit to the right and by writing a 0 (1) in the MSB (*Most Significant Bit*) of $\gamma[i]$ if ρ does (does not) satisfy constraint i.

If at the end of the loop in lines 17-21 no constraint has been found to be false then ρ is a *real* counterexample and we return SAT (line 22).

Lines 23-25 of `cegar_milp2sat()` for each constraint i compute in w_i the number of clauses to be generated by `cegar_blp2sat(`$A[i]$`,` $b[i]$`)`. This number

```
 1 void cegar_milp2sat(A, b) {
 2 Align variables to Zero;
 3 Compute number of bits for continuous variables;
 4 Discretize Continuous variables;
 5 Transform discrete variables into boolean variables;
 6 Initialize computation state of cegar_blp2sat();
 7 Initialize history γ to all 0s;
 8 F = 1;
 9 while (1) {
10   foreach i {
11       Order variables accordingly to coefficients;
12       F = F ∧ cegar_blp2sat(A[i], b[i]); }
13   sol = call_SAT_solver(F);
14   if (sol == UNSAT) {return (UNSAT);}
15   else {ρ = decode_SAT_assignment();}
16   oksat = 1;
17   for each constraint i {
18         shift γ[i] 1 bit to the right;
19         if (ρ does not satisfy constraint i)
20            {oksat = 0; γ[i][m − 1] = 1;}
21         else {γ[i][m − 1] = 0;} }
22   if (oksat == 1) { return (SAT);}
23   for each constraint i {
24        compute clauses to be generated w_i using γ[i];
25        cegar_blp2sat(A[i], b[i]); }
26   let ρ′ be the assignment containing only the decision variables
      in ρ;
27   add to F the clause ¬ρ. }}
```

Fig. 2. Sketch of algorithm `cegar_milp2sat()`

$\Gamma = 1$	$\Lambda = 1$	$\gamma = 1$	$\mu = 2$	$\theta = 1$	$\eta = 2$	$\alpha = 2$	$\beta = 4$	$\zeta = 3$	$\xi = 2$	$m = 6$	$M = 24$

Fig. 3. Parameters for the water-tanks system in Section 4

is 0 if the constraint was true under the last found assignment ρ. Otherwise it is greater than 0 and depends on the history of constraint i. Intuitively, the more often *and* the more recently constraint i has turned out to be false, the larger the values of w_i. In any case $w_i \leq$ `MaxClause`. After computing w_i we call `cegar_blp2sat(`$A[i]$`,` $b[i]$`)`.

In line 26, from the SAT solver data structures we compute the *decision variables* of ρ. Let ρ' be the assignment containing only the decision variables in ρ. In line 27 we add to F the clause $\neg\rho'$.

Eventually we get UNSAT or a real (not spurious) satisfying assignment.

9 Experimental Results

To assess effectiveness of our approach we have implemented the proposed algorithms and compared their performance w.r.t. MILP based BMC verification.

Let ($\mathcal{H}$, E, k) be a BMC problem. By using Theorem 1 we can transform such BMC problem into an MILP problem P. We then have two choices: use an MILP solver to check feasibility of P or use our SAT-CEGAR approach (Section 8) to check feasibility of P. In this section we compare these two approaches.

		GLPK		SAT				SAT-CEGAR			
k	h	Output	Time (secs)	Output	Time (secs)	CL	Mem (MB)	Output	Time (secs)	CL	Mem (MB)
1	7	SAT	88.09	SAT	3.12	334279	30.59	SAT	4.63	330318	30.56
2	4	SAT	4.95	SAT	5.66	600433	60.64	SAT	6.28	482930	47.74
3	4	UNSAT	7256.42	UNSAT	13.63	1.55e+06	123.96	UNSAT	3.09	370931	30.937
3	23		OOT	UNSAT	1902.32	8.93e+06	837.253	UNSAT	21.43	2.13e+06	240.83
3	98		OOT				OOM	UNSAT	153.35	9.08e+06	839.641
4	9		OOT	UNSAT	125.28	8.53e+06	833.473	UNSAT	9.79	1.1e+06	120.756
4	71		OOT				OOM	UNSAT	144.17	8.65e+06	836.938
5	3		OOT	UNSAT	62.44	6.75e+06	627.688	UNSAT	4.07	473834	47.8307
5	54		OOT				OOM	UNSAT	139.58	8.52e+06	836.518
6	1		OOT	UNSAT	46.99	5.21e+06	487.896	UNSAT	171.903	3.87e+06	381.539
6	48		OOT				OOM	UNSAT	157.57	8.2e+06	643.253

Fig. 4. Experimental results for the system in Section 4 with parameters in Figure 3

As a SAT solver we use ZChaff [24,32] a well know open source SAT solver. As an MILP solver for comparison we use GLPK [16]. In view of Remark 1 using GLPK rather than CPLEX [12] for our feasibility problems does not change meaningfully the results.

We use the example in Section 4 to assess effectiveness of our approach. This is a parametric example with most of the features that make *life hard* for reachability analysis.

Let k be the number of pumps. The number of tanks n is set to $2k$ and the system parameters are those in Figure 3.

Figure 4 shows our results when using GLPK, SAT without CEGAR (i.e. with `MaxClause` set to ∞) and our SAT-CEGAR (Section 8) with `MaxClause` set to 100. We set $\epsilon = 0.1$, that is we accept a relaxation of 0.1 on P constraints (Theorem 2). Note that (Theorem 2) if we get UNSAT then the original problem is UNSAT and for each $0 \leq \epsilon' \leq \epsilon$ we would get the same result (UNSAT).

The meaning of the columns in Figure 4 is the following.

Column k gives the number of pumps. The number of tanks is then $n = 2k$. *Column h* gives the BMC horizon. *Column Output* gives the outcome of the verification process. Namely, SAT if an error has been found within horizon h, UNSAT else. *Column Time* gives the CPU time in seconds. We have set a time limit of 180 minutes (10800 seconds) for the verification process. If a process does not complete by such time limit we report *Out Of Time* (OOT) in column Time. *Column Mem* gives the RAM used by the process. We report OOM if a process runs out of memory (1GB of RAM in our case). *Column CL* gives the number of clauses generated.

In our experiments we used a Mac Mini (CPU PowerPC G4 1.5 GHz; L2 Cache 512 KB; RAM 1GB).

For each k in Figure 4 we show the first horizon h to which we find an error or the last horizon that we were able to handle before going out of time or out of RAM. From Figure 4 we can clearly see how SAT behaves better than an MILP solver (GLPK) and how our SAT-CEGAR approach behaves better than SAT.

Figure 6 gives some detail about the SAT-CEGAR computation time (y axes) as a function of the horizon (x axes) with $k = 3$ pumps. We see that for SAT-CEGAR computation times are almost a linear function of the horizon.

Figure 5 gives some information about the MILP problem generated from our BMC problems.

		MILP Problem				
k	h	Rows	Real Vars	Non-Bool Int Vars	Bool Vars	Non-zeros
1	7	965	40	8	219	1826
2	4	1413	50	10	406	2764
3	4	2751	75	15	927	5550
3	98	64979	1485	297	21795	133014
3	99	65641	1500	300	22017	134370
4	71	82087	1440	288	30732	172856
4	72	83241	1460	292	31164	175288
5	54	100759	1375	275	40585	217330
5	55	102621	1400	280	41335	221350
6	48	136309	1470	294	57714	299844
6	49	139143	1500	300	58914	306084

Fig. 5. MILP problems generated for the water-tanks system in Section 4 with parameters as in Figure 3

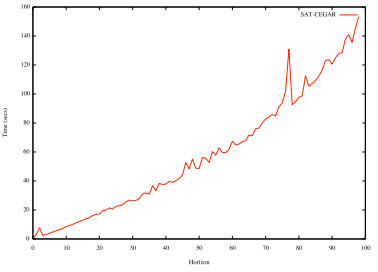

Fig. 6. SAT-CEGAR. Times for $k = 3$ and $n = 6$

Summing up, our experimental results show that our SAT-CEGAR approach can solve problems that are about 50 times larger than those that can be handled with an MILP solver. Namely, from Figure 4 we see that SAT-CEGAR can solve problems with $k = 6$ and $h = 48$ (i.e. 136309 linear constraints from Figure 5) whereas GLPK stops at $k = 3$, $h = 4$ (i.e. 2751 linear constraints from Figure 5).

10 Conclusions

We have presented a SAT based BMC algorithm for automatic verification of DTHSs. Using *Counterexample Guided Abstraction Refinement* (CEGAR) our algorithms *gradually* transforms a DTHS verification problem into larger and larger SAT problems.

Our experimental results show that our approach can handle DTHSs that are more then 50 times larger than those that can be handled using an MILP solver.

Acknowledgements. We are very grateful to HSCC'07 referees for their helpful comments on the submitted version of this paper.

References

1. url: `http://www.eecs.berkeley.edu/~tah/HyTech`.
2. R. Alur, T.A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. *IEEE Trans. on Software Engineering*, 22, 1996.
3. G. Audermand, M. Bozzano, A. Cimatti, and R. Sebastiani. Verifying industrial hybrid systems with mathsat. In *Proc. of the 2nd Int. Workshop on Bounded Model Checking*, 2004.
4. A. Bemporad and M. Morari. Verification of hybrid systems via mathematical programming. In *Proc. of: HSCC*, volume 1569 of *LNCS*. Springer, 1999.
5. A Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Proc. of TACAS*, volume 1579 of *LNCS*. Springer, 1999.
6. M. Bozzano, R. Bruttomesso, A. Cimatti, T. Juntilla, S. Ranise, P. van Rossum, and R. Sebastiani. Efficient satisfiability modulo theories via delayed theory combination. In *Proc. of CAV*, volume 3576 of *LNCS*, 2005.

7. url: http://www.dsi.uniroma1.it/~tronci/cached.murphi.html.
8. M. W. Carter and C. C. Price. *Operations Research - A Practical Introduction*. CRC Press, 2001.
9. url: http://www.cs.cmu.edu/~modelcheck/cbmc/.
10. Edmund Clarke and Daniel Kroening. Hardware verification using ANSI-C programs as a reference. In *Proc. of ASP-DAC*. IEEE Computer Society Press, 2003.
11. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proc. of CAV*, LNCS. Springer, 2000.
12. url: http://www.ilog.com/products/cplex/.
13. G. Della Penna, B. Intrigila, I. Melatti, E. Tronci, and M. Venturini Zilli. Exploiting transition locality in automatic verification of finite state concurrent systems. *International Journal of Software Tools for Technology Transfer (STTT)*, 6(4), 2004.
14. R. Raimi E. Clarke, A. Biere and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in system Design*, 19:7–34, July 2001.
15. N. Giorgetti, G. J. Pappas, and A. Bemporad. Bounded model checking of hybrid dynamical systems. In *Proc. of 44th IEEE Int Conf. CDC*, 2005.
16. url: http://www.gnu.org/software/glpk/glpk.html.
17. Anubhav Gupta and Ofer Strichman. Abstraction refinement for bounded model checking. In *CAV*, volume 3576 of *LNCS*, pages 112–124, 2005.
18. T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1, 1997.
19. url: http://control.ee.ethz.ch/~hybrid/hysdel.
20. Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal: Status and developments. In *CAV97*, number 1254 in LNCS. Springer–Verlag, 1997.
21. B. Li, C. Wang, and F. Somenzi. Abstraction refinement in symbolic model checking using satisfiability as the only decision procedure. *Software Tools for technology Transfer (STTT)*, 7(2):143–155, 2005.
22. url: http://mathsat.itc.it/.
23. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.
24. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *39th DAC*, 2001.
25. G. Della Penna, B. Intrigila, I. Melatti, M. Minichino, E. Ciancamerla, A. Parisse, E. Tronci, and M. V. Zilli. Automatic verification of a turbogas control system with the murphi verifier. In *Proc. of HSCC*, LNCS. Springer, 2003.
26. url: http://www.cs.cmu.edu/~modelcheck/.
27. F. D. Torrisi and A. Bemporad. Hysdel - a tool for generating computational hybrid models. *IEEE Trans. on Control Systems Technology*, 12(2):235–249, March 2004.
28. A. L. Turk, S. T. Probst, and G. J. Powers. Verification of real-time chemical processing systems. In *HRTS*, number 1201 in LNCS. Springer, 1997.
29. url: http://www.docs.uu.se/docs/rtmv/uppaal/.
30. R. Vidal, S. Schaffert, O. Shakernia, J. Lygeros, and S. Sastry. Decidable and semi-decidable controller synthesis for classes of discrete time hybrid systems. In *In Proc. of 40th IEEE CDC*, 2001.
31. url: http://vlsi.colorado.edu/~vis.
32. url: http://www.princeton.edu/~chaff/zchaff.html.