HySAT: An Efficient Proof Engine for Bounded Model Checking of Hybrid Systems[*]

Martin Fränzle and Christian Herde

Department of Computing Science, Research Group Hybrid Systems
Carl-von-Ossietzky Universität, D-26111 Oldenburg, Germany

{Fraenzle,Herde}@Informatik.Uni-Oldenburg.De

Abstract

In this paper we present HySAT, a bounded model checker for linear hybrid systems, incorporating a tight integration of a DPLL–based pseudo–Boolean SAT solver and a linear programming routine as core engine. In contrast to related tools like MathSAT, ICS, or CVC, our tool exploits the various optimizations that arise naturally in the bounded model checking context, e.g. isomorphic replication of learned conflict clauses or tailored decision strategies, and extends them to the hybrid domain. We demonstrate that those optimizations are crucial to the performance of the tool.

Keywords: verification, bounded model checking, hybrid systems, infinite-state systems, decision procedures, satisfiability.

1 Introduction

During the last ten years, formal verification of digital systems has evolved from an academic subject to an approach accepted by the industry, with dozens of commercial tools now available and used by major companies. Among the most successful methods in formal verification of discrete systems is bounded model checking (BMC), as suggested by Groote et al. in [22] and by Biere et al. in [8]. The idea of BMC is to encode the next–state relation of a system as a propositional formula, unroll this to some given finite depth k, and to augment it with a corresponding finite unravelling of the tableau of (the negation of) a temporal formula in order to obtain a propositional SAT problem which is satisfiable iff an error trace of length k exists. Enabled by the impressive gains

[*]This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS). See www.avacs.org for more information.

in performance of propositional SAT checkers in recent years, BMC can now be successfully applied even to very large finite-state designs.

Though originally formulated for discrete transition systems only, the basic idea of BMC to reduce the search for an error path to a satisfiability problem of a formula also applies to hybrid discrete–continuous systems. However, the BMC formulae arising from such systems are no longer purely propositional, but usually comprise complex Boolean combinations of arithmetic constraints over real-valued variables, thus entailing the need for new decision procedures to solve them.

Our tool HySAT provides a decision procedure that is tailored to fit the needs of BMC of infinite–state systems with piecewise linear variable updates, e.g. of linear hybrid automata. HySAT tightly integrates a state–of–the–art Davis–Putnam style SAT solver for pseudo–Boolean constraints with a linear programming routine, combining the virtues of both methods: Linear programming adds the capability of solving large conjunctive systems of linear inequalities over the reals, whereas the SAT solver accounts for fast Boolean search and efficient handling of disjunctions.

The idea to combine algorithms for SAT with decision procedures for conjunctions of numerical constraints in order to solve arbitrary Boolean combinations thereof has been pursued by several groups. A tight integration of a resolution based SAT checker with linear programming has first been proposed and successfully applied to planning problems by Wolfman and Weld [39]. More recently, Audemard et al. [2] have followed up with MathSAT, a tool combining SAT solving with a Bellman–Ford algorithm for difference logic constraints and a simplex algorithm for general linear constraints, used for applications in the context of temporal reasoning and model checking of timed automata. Tools supporting a more general class of formulae are CVC [5] and ICS [14], both integrating decision procedures for various theories, including Boolean logic, linear real arithmetic, uninterpreted function symbols, functional arrays, and abstract data types.

However, except for HySAT, all tools mentioned above lack some or all of the particular optimizations that arise naturally in the bounded model checking context. As observed by Strichman [34], BMC yields SAT instances that are highly symmetric as they comprise a k–fold unrolling of the systems transition relation. This special structure can be exploited to accelerate solving, e.g. by copying the explanation for a conflict which was encountered during the backtrack search performed by the SAT solver, to all isomorphic parts of the formula in order to prune similar conflicts from the search tree. This technique, in the following referred to as *isomorphy inference*, has been shown to yield considerable performance gains when performing BMC with propositional SAT engines. To the best of our knowledge, HySAT is the first solver that extends isomorphy inference accross transitions, as well as other domain–specific optimizations described in [34], to the hybrid domain. We will show that, compared to purely propositional BMC, similar or even higher performance gains can be accomplished within this context. The reason is that an inference step in the hybrid domain is computationally much more expensive than in propositional logic, as now richer logics have to be dealt with.

The paper is organized as follows. In the following two sections we explain the logical language solved by our SAT checker and review briefly how a linear hybrid automaton can be translated into a predicative formula suitable for bounded model checking. In section 4 we explain in detail the algorithmic ingredients of HySAT. In particular, we discuss the BMC–specific optimizations implemented in our tool. In section 5 we report some experimental results, and section 6 draws conclusions and describes directions for future research.

2 The logics

As we are aiming at automated state-exploratory analysis of linear hybrid automata [25, 24] without prior finite-state abstraction, HySAT addresses satisfiability problems in a two-sorted logics entailing Boolean-valued and real-valued variables. When encoding properties of linear hybrid automata, the Boolean variables are used for encoding the discrete state components, while the real variables represent the continuous state components.

The formulae are actually propositional, being conjunctions of *linear zero-one constraints* [19] (also known as *pseudo-Boolean constraints* [6]) for the Boolean part and of *guarded linear constraints* [39] for the real-valued part:

$$\begin{array}{rcl} \mathit{formula} & ::= & \{\mathit{clause} \wedge\}^{*}\, \mathit{clause} \\ \mathit{clause} & ::= & \mathit{linear\_ZO\_constraint} \mid \mathit{boolean\_var} \implies \mathit{linear\_constraint} \end{array}$$

Here, *linear_constraint* denotes a conjunction of linear inequalities over *real-valued* variables, i.e. the constraint part of an arbitrary linear program, while *linear_ZO_constraint* denotes a linear inequality over *Boolean-valued* variables. The reason for using linear zero-one constraint clauses instead of, e.g., disjunctive clauses (like in conjunctive normal forms) is that linear zero-one constraints are much more concise than disjunctive clauses and that we have a very efficient SAT solver —called "Goblin" [19]— for such constraint systems, yielding the base engine for HySAT.

2.1 Zero-one linear constraints

Rewriting arbitrary propositional formulae to conjunctive normal form (CNF) yields a worst-case exponential blowup in formula size if the number of propositional variables is to be preserved. To avoid this, all practical verification environments take advantage of satisfiability-preserving transformations that yield linear-size encodings through introduction of a linear number of auxiliary variables [36, 31, 37]. The price for introducing a linear number of auxiliary variables is, however, a worst-case exponential blow-up in the size of the search tree upon backtrack search. Yet, it has been observed that both causes of blow-up can often be avoided, as the Davis-Putnam-Loveland-Logemann search procedure for satisfying valuations generalizes smoothly to zero-one linear constraint systems (ZOLCS), which are the constraint parts of zero-one linear programs

[6, 38, 1, 19]. Zero-one linear constraint systems are expressive enough to facilitate a linear-size encoding of, e.g., gate-level netlists without use of auxiliary variables.

In a *zero-one linear constraint system* or *linear pseudo-Boolean constraint systems*, formulae are conjunctions of linear zero-one constraints. A *linear zero-one constraint* is of the form $a_1x_1+a_2x_2+\ldots a_nx_n \geq k$, where the x_i are *literals*, i.e. positive or negated *propositional variables*, the a_i are natural numbers, called the *weights* of the individual literals, and $k \in \mathbb{N}$ is the *threshold*.

Given a Boolean valuation of the propositional variables, a zero-one constraint is satisfied iff its left hand side evaluates to a value exceeding the threshold when the truth values `false` and `true` of the literals are identified with 0 and 1, respectively. Zero-one constraints can represent a wide class of monotonic Boolean functions, e.g. $1a+1b+1\overline{c}+1d \geq 1$ is equivalent to $a \vee b \vee \overline{c} \vee d$, $1a+1b+1\overline{c}+1d \geq 4$ is equivalent to $a \wedge b \wedge \overline{c} \wedge d$, and $1a+1b+3\overline{c}+1d \geq 3$ is equivalent to $c \implies (a \wedge b \wedge d)$. Consequently, ZOLCS can be exponentially more concise than CNF: a CNF expressing that at least n out of k variables should be true requires $\binom{n}{k}$ disjunctive clauses of length n each, i.e. is of size $O\left(\binom{n}{k}n\right)$, whereas the corresponding ZOLCS has size linear in k and logarithmic in n.

Formally, the syntax of linear zero-one constraints is

$$\begin{aligned}
\textit{linear\_ZO\_constraint} &::= \textit{linear\_term} \geq \textit{threshold}\\
\textit{linear\_term} &::= \{\textit{weight literal} +\}^* \textit{weight literal}\\
\textit{weight} &::\in \mathbb{N}\\
\textit{literal} &::= \textit{boolean\_var} \mid \overline{\textit{boolean\_var}}\\
\textit{boolean\_var} &::\in BV\\
\textit{threshold} &::\in \mathbb{N}
\end{aligned}$$

where BV is a countable set of Boolean variable names.
Zero-one constraints are interpreted over *Boolean valuations* $\sigma_{\mathbb{B}} : BV \xrightarrow{\text{total}} \mathbb{B}$ of the propositional variables. $\sigma_{\mathbb{B}}$ satisfies a constraint $a_1x_1+a_2x_2+\ldots a_nx_n \geq k$ iff $a_1\chi_{\sigma_{\mathbb{B}}}(x_1)+a_2\chi_{\sigma_{\mathbb{B}}}(x_2)+\ldots a_n\chi_{\sigma_{\mathbb{B}}}(x_n) \geq k$, where

$$\chi_{\sigma_{\mathbb{B}}}(x) = \begin{cases} 0 & \text{if } x \in V \text{ and } \sigma_{\mathbb{B}}(x) = \texttt{false},\\ 1 & \text{if } x \in V \text{ and } \sigma_{\mathbb{B}}(x) = \texttt{true},\\ 1-\chi_{\sigma_{\mathbb{B}}}(y) & \text{if } x \equiv \overline{y} \text{ for some } y \in V. \end{cases}$$

2.2 Guarded linear constraints

Zero-one constraints can only express constraints on Boolean variables. A second kind of clauses in our logics is *Boolean-guarded linear constraints* which express (linear) constraints between real-valued variables, as well as their interdependence with the Boolean valuation. A guarded linear constraint simply is an implication

$$\textit{boolean\_var} \implies \textit{linear\_constraint}$$

between a Boolean variable and a linear constraint over real-valued variables, i.e. a conjunction of linear inequations. Such a guarded linear constraint is

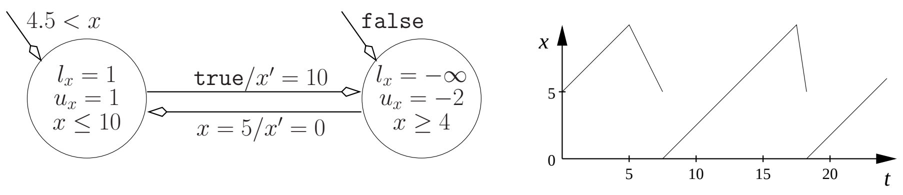


Figure 1: A linear hybrid automaton and a sample trajectory. l_x and u_x denote the lower and upper bounds on the slope of x in the corresponding states, while $x \leq 10$ and $x \geq 4$ are state invariants constraining x itself.

interpreted over a valuation $\sigma = (\sigma_{\mathbb{B}}, \sigma_{\mathbb{R}}) \in (BV \xrightarrow{\text{total}} \mathbb{B}) \times (RV \xrightarrow{\text{total}} \mathbb{R})$, where RV is the set of real variables occurring in linear constraints. The guarded linear constraint $v \implies c$ is satisfied by $\sigma = (\sigma_{\mathbb{B}}, \sigma_{\mathbb{R}})$ iff $\sigma_{\mathbb{R}}$ satisfies the linear constraint c or if $\sigma_{\mathbb{B}}(v) = \texttt{false}$.

2.3 Satisfaction of formulae

A *formula* ϕ is a conjunction of linear zero-one constraints and of guarded linear constraints and is thus interpreted over valuations

$$\sigma = (\sigma_{\mathbb{B}}, \sigma_{\mathbb{R}}) \in (BV \xrightarrow{\text{total}} \mathbb{B}) \times (RV \xrightarrow{\text{total}} \mathbb{R}) \ .$$

Obviously, ϕ is satisfied by $\sigma = (\sigma_{\mathbb{B}}, \sigma_{\mathbb{R}})$, denoted $\sigma \models \phi$, iff all linear zero-one constraints in ϕ are satisfied by $\sigma_{\mathbb{B}}$ and all guarded linear constraints in ϕ are satisfied by $(\sigma_{\mathbb{B}}, \sigma_{\mathbb{R}})$.

When solving satisfiability problems of formulae with Davis-Putnam-like procedures, we will build valuations incrementally such that we have to reason about *partial valuations* $\rho \in (BV \xrightarrow{\text{part.}} \mathbb{B}) \times (RV \xrightarrow{\text{part.}} \mathbb{R})$ of variables. We say that a variable $v \in BV \cup RV$ is *unassigned in* ρ iff $v \notin \text{dom}(\rho_{\mathbb{B}}) \cup \text{dom}(\rho_{\mathbb{R}})$. A partial valuation ρ is called *consistent for a formula* ϕ iff there exists a total extension $\sigma : (BV \xrightarrow{\text{total}} \mathbb{B}) \times (RV \xrightarrow{\text{total}} \mathbb{R})$ of ρ that satisfies ϕ. Otherwise, we call ρ *inconsistent for* ϕ. Furthermore, a partial valuation ρ is said to *satisfy* ϕ iff all its total extensions satisfy ϕ. As this definition of satisfaction agrees with the previous one on total valuations, we will use the same notation $\rho \models \phi$ for satisfaction by partial and by total valuations.

3 Predicative encoding of linear hybrid automata

A *linear hybrid automaton* $A = (\Sigma, T, R, inv, l, u, m, g, ass, init)$, as depicted in Fig. 1, consists of

- a finite set Σ of *locations*,
- a finite set T of *transitions*,
- a finite set R of continuous state components,

- a family $inv = (inv_\sigma)_{\sigma\in\Sigma}$ of *state invariants*, where each state invariant inv_σ is a linear predicate over R which constrains the valuations of the continuous state components when control resides in the discrete location σ,
- two families $l = (l_{\sigma,x})_{\sigma\in\Sigma,x\in R}$ and $u = (u_{\sigma,x})_{\sigma\in\Sigma,x\in R}$ assigning to each location $\sigma \in \Sigma$ and each continuous state component $x \in R$ the *minimum and maximum slope* of x while control resides in location σ. The individual $l_{\sigma,x}$ are constants in $\mathbb{Q} \cup \{-\infty\}$ and similarly $u_{\sigma,x} \in \mathbb{Q} \cup \{\infty\}$.
- a mapping $m : T \xrightarrow{\text{total}} \Sigma^2$ assigning to each transition the pair of source and sink state of the transition,
- a family $g = (g_t)_{t\in T}$ assigning to each transition a *transition guard* enabling that transition, where the transition guard is a linear predicate over R,
- a family $ass = (ass_t)_{t\in T}$ assigning to each transition a (possibly nondeterministic) *assignment* which is a linear predicate over R and R', where R' denotes primed variants of the state components in R. The interpretation is that undecorated state components $x \in R$ refer to the state immediately before the transition, while the primed variant $x' \in R'$ refers to the state immediately thereafter.
- a family $init = (init_\sigma)_{\sigma\in\Sigma}$ of *initial state predicates*, where each $init_\sigma$ is a linear predicate over R which constrains the valuations of the continuous state components when control resides *initially* in the discrete location σ.[1]

Hybrid automata engage in an alternation of continuous evolutions and discrete transitions. A *continuous evolution* of $A = (\Sigma, T, R, inv, l, u, m, g, ass, init)$ can be represented by a tuple $(\sigma, \vec{x}, \delta, \vec{x}')$ consisting of a discrete state $\sigma \in \Sigma$ the automaton resides in, a source continuous state $\vec{x} \in (R \xrightarrow{\text{total}} \mathbb{R})$ and a target continuous state $\vec{x}' \in (R \xrightarrow{\text{total}} \mathbb{R})$, as well as a duration $\delta \in \mathbb{R}_{\geq 0}$. Such a tuple is a continuous evolution of A iff for each $y \in R$ it holds that $\vec{x}'(y) \geq \vec{x}(y) + l_{\sigma_i,y} \cdot \delta$ and $\vec{x}'(y) \leq \vec{x}(y) + u_{\sigma_i,y} \cdot \delta$, and both $\vec{x}$ and $\vec{x}'$ satisfy inv_σ. Thus, δ represents the duration of A residing in state σ, and all continuous variables y evolve according to their slope bounds, and the invariant is true in the start and the end state (and thus, by convexity, in between). Similarly, an *immediate transition* can be represented by a tuple $(\sigma, \vec{x}, \sigma', \vec{x}')$ consisting of a discrete source state $\sigma \in \Sigma$ and a discrete target state σ', plus a continuous source state $\vec{x} \in (R \xrightarrow{\text{total}} \mathbb{R})$ and a continuous target state $\vec{x}' \in (R \xrightarrow{\text{total}} \mathbb{R})$. Such a tuple is an immediate transition iff there is a transition $t \in T$ with $m(t) = (\sigma, \sigma')$ such that $\vec{x}$ satisfies g_t and such that ass_t is satisfied if $\vec{x}$ is substituted for the variables in R and $\vec{x}'$ is substituted for the variables in R'.

A *run* $r = \langle(\sigma^0, \vec{x}^0, \delta^0, \vec{x}'^0), \ldots, (\sigma^n, \vec{x}^n, \delta^n, \vec{x}'^n)\rangle \in (\Sigma \times (R \xrightarrow{\text{total}} \mathbb{R}) \times \mathbb{R}_{\geq 0} \times (R \xrightarrow{\text{total}} \mathbb{R}))^*$ is a sequence of continuous evolutions of A linked by immediate

[1]A discrete location σ not to be taken initially takes the predicate $init_\sigma = \texttt{false}$.

transitions and grounded in a viable initial state. I.e., a run r satisfies the following properties:

- *Initialization:* $\vec{x}^0$ satisfies $init_{\sigma^0}$.
- *Progression by continuous evolution:* for all i, the tuple $(\sigma^i, \vec{x}^i, \delta^i, \vec{x}'^i)$ is a continuous evolution of A.
- *Progression by immediate transitions:* the tuple $(\sigma^i, \vec{x}'^i, \sigma^{i+1}, \vec{x}^{i+1})$ is an immediate transition of A for all $i < n$.

In order to perform *bounded model checking* (BMC) [8] with HySAT, i.e. checking of validity of temporal properties on finite unrollings of a transition system, we need to encode all runs of a given length $k \in \mathbb{N}$ in HySAT's logics. There are various ways of doing this, all with specific strengths and weaknesses. Yet all the reasonable ones share the property of featuring a plethora of structurally similar sub-formulae stemming from the iterated application of the transition relation and from the iterated continuous evolution in the k-fold unrolling. In order to exemplify this, we present here one particular form of such an unrolling which is very similar to the one used by Audemard et al. for MathSAT-based BMC of linear hybrid automata [3] and by Bemporad et al. for MILP-based BMC of linear hybrid automata [7].

Let $A = (\Sigma, T, R, inv, l, u, m, g, ass, init)$ be a linear hybrid automaton. In order to encode a transition sequence of A of some given length $k \in \mathbb{N}$, we proceed as follows:

1. For each discrete state $\sigma \in \Sigma$ we take $k+1$ Boolean variables σ^i, with $0 \le i \le k$. The value of σ^i encodes whether the automaton A is in state σ in step i. Here, we take "one-hot" encoding, i.e. $\sigma^i = \texttt{true}$ iff A is in state σ in step i. With one-hot encoding, there consequently is, for any $i \le k$, exactly one $\sigma \in \Sigma$ such that σ^i holds, which is enforced in the BMC formula by the $2k+2$ linear zero-one constraints

$$\bigwedge_{i=0}^{k} \left(\sum_{\sigma \in \Sigma} 1\sigma^i \le 1 \right) \wedge \bigwedge_{i=0}^{k} \left(\sum_{\sigma \in \Sigma} 1\overline{\sigma^i} \ge |\Sigma| - 1 \right)$$

2. For each transition $t \in T$ we take k Boolean variables t^i, with $1 \le i \le k$. The value of t^i encodes via one-hot encoding whether the ith move in the run is transition t. Wellformedness of the unrolling in the sense that exactly one transition is taken in each step is guaranteed by conjunctively adding the $2k$ linear zero-one constraints

$$\bigwedge_{i=1}^{k} \left(\sum_{t \in T} 1t^i \le 1 \right) \wedge \bigwedge_{i=1}^{k} \left(\sum_{t \in T} 1\overline{t^i} \ge |T| - 1 \right)$$

   to the formula.

3. For each continuous state component $x \in R$ we take $k+1$ real-valued variables x^i and another $k+1$ real-valued variables x'^i, with $i \leq k$. The value of x^i encodes the value of x immediately after the ith transition in the run, whereas x'^i represents the value immediately before transition $(i+1)$. For each $i \leq k$ we do, furthermore, take one real-valued variable δ^i representing the time spent in the ith state of the run. This allows us to formalize the *continuous evolutions* by conjoining the guarded linear constraint

$$\sigma^i \implies (x'^i \geq x^i + l_{\sigma,x}\delta^i \wedge x'^i \leq x^i + u_{\sigma,x}\delta^i)$$

for each $\sigma \in \Sigma$ and each $i \leq k$ to the formula.[2] Furthermore, we have to keep track of the *state invariants*, which are enforced by the guarded linear constraints

$$\sigma^i \implies (inv_{\sigma^i}[x_1^i, \ldots, x_n^i/x_1, \ldots, x_n] \wedge inv_{\sigma^i}[x_1'^i, \ldots, x_n'^i/x_1, \ldots, x_n]) \enspace ,$$

where $\{x_1, \ldots, x_n\} = R$.

4. The interplay between discrete states and transitions requires that t^i implies σ^{i-1} and $\widetilde{\sigma}^i$ for $(\sigma, \widetilde{\sigma}) = m(t)$. With linear zero-one constraints, this can be expressed by a single constraint

$$2\overline{t^i} + 1\sigma^{i-1} + 1\widetilde{\sigma}^i \geq 2$$

for each $t \in T$ and each $1 \leq i \leq k$. Furthermore, enabledness of the transition, i.e. validity of the *transition guard*, is enforced through the guarded linear constraint

$$t^{i+1} \implies g_t[x_1'^i, \ldots, x_n'^i/x_1, \ldots, x_n] \enspace .$$

Likewise, *assignments* are dealt with by

$$t^{i+1} \implies ass_t[x_1^i, \ldots, x_n^i/x_1, \ldots, x_n][x_1'^i, \ldots, x_n'^i/x_1', \ldots, x_n']$$

5. Finally, we have to add constraints describing the allowable initial states through the guarded linear constraint system

$$\bigwedge_{\sigma \in \Sigma} (\sigma^0 \implies init_\sigma)$$

Satisfying valuations of the formula thus obtained are in one-to-one correspondence to the runs of A of length k. As in BMC [8], satisfaction of temporal properties on all runs of depth k can thus be checked by adding to the formula the k-fold unrolling of a tableaux of the (negated) property, then checking the resulting formula for unsatisfiability. Using standard techniques from predicative semantics [23], the translation scheme can be extended to both shared variable

[2]If $l_{\sigma,x} = -\infty$ or $u_{\sigma,x} = \infty$, the corresponding part of the constraint is left out.

and synchronous message-passing parallelism, thereby yielding formulae of size linear in the number of parallel components.

Note that, except for step (5) of above encoding scheme, all steps generate multiple copies of the same basic formula, where the k or $k+1$ individual copies differ just in a consistent renaming of the variables. Therefore, a satisfiability checker tailored towards BMC of hybrid automata should exploit such isomorphies between subformulae for accelerating satisfiability checking, which is the distinguishing feature of HySAT. In order to simplify detection of isomorphic copies, HySAT is in fact fed with just a single copy of the transition and evolution predicates and performs the unrolling itself.

4 Ingredients of HySAT

The predicative encoding outlined above yields formulae which are Boolean combinations of linear arithmetic contraints. To deal with such formulae, HySAT's main components are

- the *solver core*, consisting of a tight integration of a SAT solver with a linear programming routine, described in section 4.1, and enhanced with domain-specific optimizations for BMC, as explained in section 4.2,
- an *API* to the solver core, providing methods for formula generation, simplification, common subexpression eliminiation, and for rewriting the resulting formula into a conjunctive form, namely a conjunction of zero-one linear constraints and guarded linear constraints, which is the input format of the solver core,
- a *frontend*, consisting of HySAT's input language and a bounded model checker, which performs the unwinding of the transition relation and controls the solver core via API calls.

To fit the needs of BMC, which involves checking the same system on different unrolling depths, the solver core and the API are designed to work in an incremental fashion in the sense that they allow to add (as well as delete) successively sets of constraints to (from) an existing problem and then redo the satisfiability check without starting SAT search from scratch each time.

4.1 Integration of DPLL-SAT and Linear Programming

Before addressing the integration of a propositional SAT solver with linear programming, we first briefly review some basics of the individual methods.

4.1.1 Boolean SAT

The best currently known procedures for deciding Boolean SAT problems implement variants of the classical Davis-Putnam-Loveland-Logemann (DPLL) procedure [13] and are based on backtracking in the space of partial value assignment. Given a Boolean formula Φ in conjunctive normal form (CNF) and a partial valuation ρ, which is empty at the start, the DPLL procedure incrementally extends ρ until either $\rho \models \phi$ holds or ρ turns out to be inconsistent for ϕ, in which case another extension is tried through backtracking. Extensions are constructed by performing *decision steps*, which entail selecting an unassigned variable "blindly" and assigning a truth-value to it, each followed by a *deduction phase*, involving the search for *propagating clauses* that require certain assignments in order to preserve their satisfiability, where execution of the implied assignments might cause the need for further such assignments, in this context also referred to as *implications*. However, deduction may also yield a *conflicting clause* which has all its literals assigned false, indicating the need for backtracking.

Like all pure backtracking algorithms, the classical DPLL procedure suffers from thrashing, i.e. repeated failure due to the same reason. To overcome this problem, modern SAT solvers implement a technique called *conflict-driven learning* [40], which attempts to derive sufficiently general reasons for conflicts being encountered and stores them for future guidance of the search. The standard scheme traces the reason back to a small (ideally minimal) number of assignments that triggered the particular conflict, and stores this reason by adding the negation of that assignment as as clause, termed *conflict clause*, to the clause database. Besides *learning*, state-of-the-art SAT solvers, as the one being integrated in HySAT, enhance the basic DPLL procedure by sophisticated heuristics for selecting the assignment performed at decision steps [27, 29], and add various algorithmic refinements, among them non-chronological backtracking [28, 29], random restarts [4] and lazy clause evaluation [29], to accelerate the proof search.

A pecularity of HySAT's SAT solver is its ability to directly handle linear zero-one constraint systems, a considerably more concise language than CNF.

DPLL on linear zero-one constraints. The DPLL procedure can easily be generalized from CNF to ZOLCS through adapting its deduction procedure to the following propagation rule for linear zero-one constraints: A zero-one constraint $\sum a_i x_i \geq k$ propagates a literal x_j iff setting this literal to false would make the constraint unsatisfiable, i.e. iff $(\sum a_i) - a_j < k$.

Note, that in contrast to a CNF clause, a zero-one constraint can propagate several literals simultaneously. As an example consider the constraint

$$5a + 3\overline{b} + 3c + 1d + 1e \geq 7$$

which propagates $\overline{b}$ and c immediately after setting a to false. Carrying out the assignments $b \mapsto \texttt{false}$ and $c \mapsto \texttt{true}$ reduces the constraint to

$$1d + 1e \geq 1$$

which shows that, as opposed to CNF-SAT, a zero-one constraint is not necessarily satisfied after propagation, and might thus become propagating more than once.

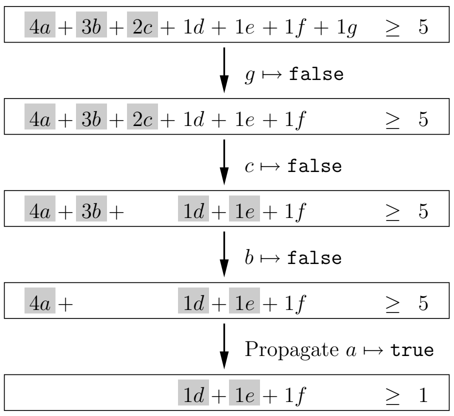

4a + 3b + 2c + 1d + 1e + 1f + 1g ≥ 5
g ↦ false
4a + 3b + 2c + 1d + 1e + 1f ≥ 5
c ↦ false
4a + 3b + 1d + 1e + 1f ≥ 5
b ↦ false
4a + 1d + 1e + 1f ≥ 5
Propagate a ↦ true
1d + 1e + 1f ≥ 1

Figure 2: Changes of the set of watched literals when successively setting g, c and b to false. Watched literals are marked with grey boxes.

Generalization of lazy clause evaluation. While the above generalization of the DPLL procedure to ZOLCS has already been proposed by Barth [6], its acceleration through lazy clause evaluation for zero-one constraints is a recent addition by Chai and Kuehlmann [10] and the authors of this paper [20].

A naive implementation of the deduction phase of DPLL would identify propagating clauses by visiting, after each assignment, *all* clauses containing the literal falsified by that assignment, as such clauses might have become propagating. The key idea of the enhanced algorithm is to watch only a subset of literals in each clause, and not to visit the clause when any other literal is assigned, as the watched set provides evidence for the clause to be non-propagating.

To apply lazy clause evaluation to zero-one constraints we have to determine a subset of unassigned literals from each constraint such that watching these literals is sufficient for detecting change of clause state from normal to propagating. Obviously, we are looking for minimal sets with this property in order to avoid unnecessary visits of constraints.

To this end, we arrange the literals of each constraint with respect to their weights, such that the literal with the largest weight is the leftmost one. Then we read the constraint from left to right and select the literals to be watched as follows:

1) The leftmost unassigned literal is selected.

2) The following literals are selected from the remaining unassigned ones until the sum of their weights, not including the weight of the leftmost unassigned literal, is greater than or equal to the constant on the righthand side of the constraint.

If a watched literal of a zero-one constraint is assigned false, our algorithm tries to re-establish a set of literals which is in accordance to rules 1) and 2).

This requires the search for a minimal set of literals which are either unassigned or true and whose weights sum up to a value that at least equals that of the watched literal which has been assigned false. If such a set exists then it is added to the set of watched literals to replace the one which has dropped out. If no such set exists then this indicates that the constraint has become a propagating one. The resulting propagations are determined by application of the propagation rule from the previous paragraph. Figure 2 illustrates the actions performed by the lazy clause-evaluation scheme by means of an example clause.

4.1.2 Linear programming

Linear programming deals with finding extremal values of a linear objective function when the variables are constrained by linear (in)equations, i.e. with problems that can be put in the general form

$$\begin{array}{rl} \text{maximize} & \vec{c}^T\vec{x} \\ \text{subject to} & \mathbf{A}\vec{x} \geq \vec{b} \end{array} \tag{1}$$

where $\vec{x}$ is the vector of variables to be solved for, and $\mathbf{A}$, $\vec{b}$ and $\vec{c}$ are given matrices or vectors of known coefficients. The linear expression $\vec{c}^T\vec{x}$ is called the *objective function*, (1) is referred to as a *linear program*.

HySAT uses LP as a black-box method to decide the feasibility of a set of linear constraints, i.e. to check whether for a given system of inequations $\mathbf{A}\vec{x} \geq \vec{b}$ the set of solutions $\{\vec{x} \in \mathbb{R} \mid \mathbf{A}\vec{x} \geq \vec{b}\}$ is non-empty, as well as a means for efficiently deriving explanations for infeasibility. The main reason for preferring LP over other methods of detecting feasibility of linear constraint systems (e.g., Fourier-Motzkin Elimination [18, 30, 9]) is that linear programming is known to be polynomial and scales extremely well in practice, even though the most frequently used codes are actually based on the non-polynomial Simplex method. Commercial codes like CPLEX tackle instances with more than 10^6 variables. In HySAT, however, we use the free LP library glpk[3] by Andrew Makhorin which provides a simplex solver, an interior point solver, and a solver supporting mixed integer linear programming (MILP), where some of the variables are required to be integer.

Checking feasibility of a system of linear inequations by linear programming is straightforward and requires only a hand-over of the unmodified or slightly modified (in case of strict inequations being entailed) linear constraint system to the LP, plus generation of a trivial (in case of only equations and non-strict inequations) or very simple objective function. To cope with systems containing *strict* inequations, which cannot be handled by LP directly, we use the standard trick of introducing a fresh slack variable ε and of replacing each strict inequation $\sum_{j=1}^{n} \mathbf{A}_{i,j}\vec{x}_j > \vec{b}_i$ by $\sum_{j=1}^{n} \mathbf{A}_{i,j}\vec{x}_j - \varepsilon \geq \vec{b}_i$. Instrumenting the resultant linear constraint system with the objective function ε to be maximized yields an LP which is feasible with strictly positive optimum iff the original constraint system is feasible.

[3]`http://www.gnu.org/software/glpk/glpk.html`

Extraction of explanations for infeasibility of a linear constraint system, on the other hand, can be performed by analyzing the solutions to adequately instrumented duals of the original constraint system. What we actually want to obtain is, in case that the original constraint system

$$C = \left(\begin{array}{lll} & \bigwedge_{i=1}^{k} & \sum_{j=1}^{n} \mathbf{A}_{i,j}\vec{x}_j \geq \vec{b}_i \\ \wedge & \bigwedge_{i=k+1}^{n} & \sum_{j=1}^{n} \mathbf{A}_{i,j}\vec{x}_j > \vec{b}_i \end{array} \right)$$

is infeasible, a subset $I \subseteq \{1, \ldots, n\}$ such that the subsystem $C|_I$ of the constraint system containing only the conjuncts from I also is infeasible, yet the subsystem is *irreducible* in the sense that any proper subset J of I designates a feasible system $C|_J$. Such an *irreducible infeasible subsystem* (IIS) is a prime implicant of all the possible reasons for failure of the constraint system C, and is thus a natural counterpart to the conflict clauses in the propositional setting as it prevents the proof search from visiting the same or related inconsistent constraint sets again. In case that the constraint system C contains only non-strict inequations (i.e., $k = n$), it is a well-known fact of linear programming (closely related to Farkas' Lemma) that extraction of irreducible infeasible subsystems can be reduced to finding extremal solutions of a dual system of linear inequations [21, 32]. We use the LP

$$\begin{array}{lrcl} \text{maximize} & \vec{w}^T\vec{y} & & \\ \text{subject to} & \mathbf{A}^T\vec{y} & = & 0 \\ & \vec{b}^T\vec{y} & = & 1 \\ & \vec{y} & \geq & 0 \\ \text{where} & \vec{w}_i = & \multicolumn{2}{l}{\begin{cases} -1 & \text{if } b_i \leq 0, \\ 0 & \text{if } b_i > 0 \end{cases}} \end{array}$$

where the objective function together with the choice of $\vec{w}$ guarantees boundedness such that an optimal solution exists whenever the LP is feasible. For such a solution, $I = \{i \mid \vec{y}_i \neq 0\}$ is an IIS. I.e., in case of systems containing only non-strict (in)-equations, we extract an IIS by just a single call to the LP procedure. In case $k < n$, we do first relax the strict inequations to non-strict ones, then search an IIS I of the relaxed system by solving the above dual system,[4] and finally apply a *deletion filter* [11, 12] to C_I (i.e., the reduction to I of the original constraint system entailing strict inequations) to further reduce I, if possible.[5] Such deletion filters entail multiple calls to the constraint solver in order to check for satisfiability of the constraint system with individual constraints being "switched off" (i.e., removed from I), thus requiring a worst-case linear number of calls to the LP. By applying deletion filters only to the (in

[4]Note that the relaxed system may turn out to be satisfiable. In this case, the following phase pursuing a deletion filter can be started on the full constraint system C instead of $C|_I$.

[5]Actually, application of the deletion filter is optional in HySAT, as the subsystems obtained from the first, LP-based phase are often tight enough (i.e., only marginally larger than the prime implicants) such that the overhead incurred from deletion-filtering is not amortized by the reduction in search space of the SAT procedure.

general, substantially) reduced subsystem C_I instead of the original system C, HySAT does, however, gain considerable performance compared to traditional deletion-filter methods.

4.1.3 Coupling SAT and LP

The basic idea of the integration is to guard each non-propositional constraint occuring in the input formula with a new Boolean variable and to pass the corresponding constraint to the linear programming routine whenever the SAT solver assigns that variable to true. In turn, constraints are removed from the LP-solver's database when their guard variables are unassigned again due to backtracking. The integration thus is an instance of the *lazy theorem proving* paradigm [16].

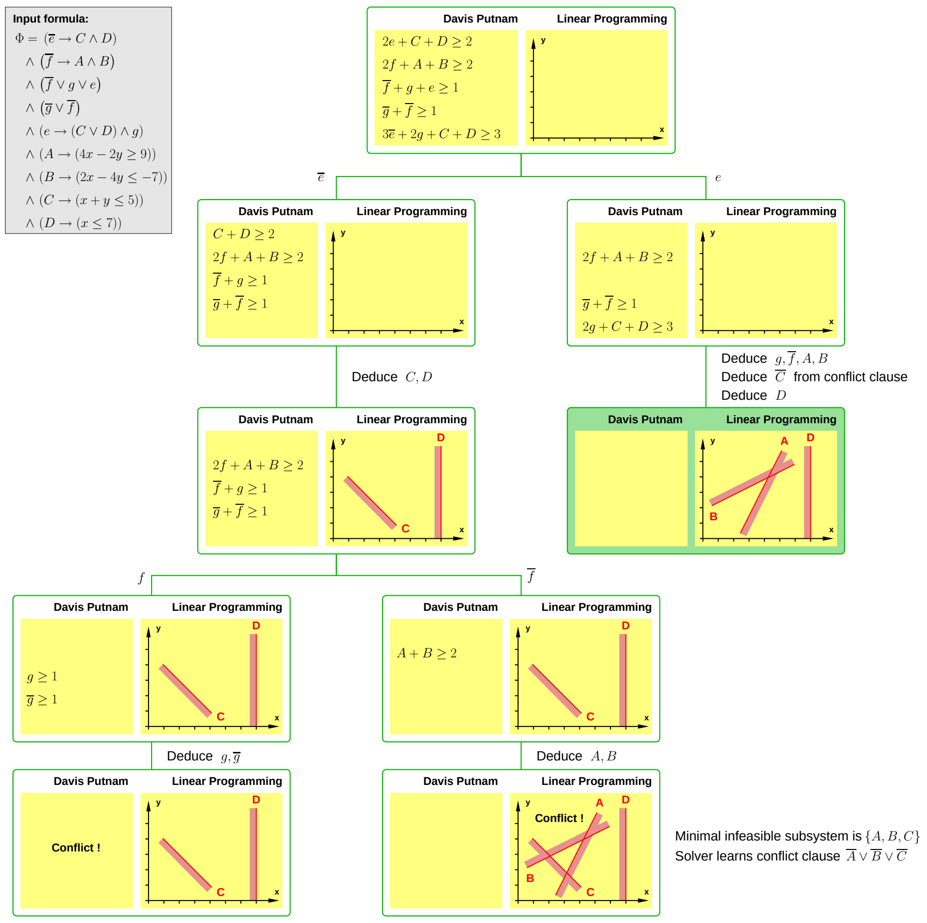
After each deduction phase in which no Boolean conflict was encountered, the SAT solver checks if new constraints have been added to the linear program since its last evaluation. If so, the linear programming routine is called to decide the feasibility of the set of constraints residing in its database. If the linear program turns out to be inconsistent, a conflict is reported to the SAT solver. Otherwise the SAT solver can proceed with the next decision step.

In case of a conflict, however, HySAT invokes a conflict-analysis routine that extracts an irreducible infeasible subsystem from the constraint set, as described in the previous section. The IIS, providing a minimal (however in general not unique) reason for the conflict, is communicated back to the SAT solver, which uses the guard variables of the linear constraints involved to construct a conflict clause which prevents that particular combination of constraints to be investigated again. The resulting interaction between DPLL proof search and feasibility check via LP is illustrated in Figure 3.

Besides learning from arithmetic conflicts, HySAT is also able to perform forward arithmetic inference, thereby deriving new arithmetic facts from feasible sets of linear constraints. Given a set $C = \{C_1, \dots, C_n\}$ of currently active arithmetic constraints, HySAT employs linear programming to determine for each continuous variable x occurring in C the minimum value x_{min} and the maximum value x_{max} consistent with $\bigwedge_{i=1}^{n} C_i$. If either of these values exists, HySAT adds the respective bound constraint, i.e. $x \geq x_{min}$ or $x \leq x_{max}$, guarded by a fresh boolean variable p, to its database, together with a propositional clause which is responsible for triggering the activation of the new constraint. To this end, the propositional clause is of form $p_{C_{i_1}} \wedge \ldots \wedge p_{C_{i_m}} \rightarrow p$, where the variables $p_{C_{i_j}}$ are the guard variables of a minimal set of constraints $C_{i_j} \in C$ whose conjunction implies the new bound constraint[6].

When learning a new bound constraint, HySAT also adds boolean clauses capturing all propositional dependencies between bound constraints concerning the same continuous variable, i.e. implicative dependencies between bounds as induced by the linear order on the reals. If the solver e.g. learns that in a certain

[6]It's noteworthy that the extraction of the minimal set $\{C_{i_j} \mid 1 \leq j \leq m\} \subseteq C$ does not entail any computational overhead, but is delivered by the LP solver as a byproduct of determining the bound on the respective continuous variable.



Input formula:
$\Phi = (\overline{e} \rightarrow C \wedge D)$
$\wedge\ (\overline{f} \rightarrow A \wedge B)$
$\wedge\ (\overline{f} \vee g \vee e)$
$\wedge\ (\overline{g} \vee \overline{f})$
$\wedge\ (e \rightarrow (C \vee D) \wedge g)$
$\wedge\ (A \rightarrow (4x - 2y \geq 9))$
$\wedge\ (B \rightarrow (2x - 4y \leq -7))$
$\wedge\ (C \rightarrow (x + y \leq 5))$
$\wedge\ (D \rightarrow (x \leq 7))$
Davis Putnam
Linear Programming
$2e + C + D \geq 2$
$2f + A + B \geq 2$
$\overline{f} + g + e \geq 1$
$\overline{g} + \overline{f} \geq 1$
$3\overline{e} + 2g + C + D \geq 3$
$\overline{e}$
e
$C + D \geq 2$
$2f + A + B \geq 2$
$\overline{f} + g \geq 1$
$\overline{g} + \overline{f} \geq 1$
$2f + A + B \geq 2$
$\overline{g} + \overline{f} \geq 1$
$2g + C + D \geq 3$
Deduce C, D
Deduce $g, \overline{f}, A, B$
Deduce $\overline{C}$ from conflict clause
Deduce D
$2f + A + B \geq 2$
$\overline{f} + g \geq 1$
$\overline{g} + \overline{f} \geq 1$
f
$\overline{f}$
$g \geq 1$
$\overline{g} \geq 1$
$A + B \geq 2$
Deduce $g, \overline{g}$
Deduce A, B
Conflict !
Conflict !
Minimal infeasible subsystem is $\{A, B, C\}$
Solver learns conflict clause $\overline{A} \vee \overline{B} \vee \overline{C}$
x
y
A
B
C
D

Figure 3: Backtrack-search tree arising in a tight integration of DPLL proof search with linear programming. x and y are real-valued, while e, f, g and A, B, C, D are Boolean. A, B, C, D are, furthermore, guard variables for arithmetic facts.

branch of the search tree $x \geq 5$ holds, it will therefore immediately exclude all combination of assignments to guard variables that would cause the activation of bound constraints $x \leq c$ with $c < 5$, thereby considerably pruning the search tree.

4.2 Optimizations for BMC

Compared to related tools like ICS which aim at being general-purpose decision procedures suitable for arbitrary formulae, HySAT's decision procedure has been tuned to exploit the unique characteristics of BMC formulae.

As observed by Strichman [34], the highly symmetric structure of the k-fold unrolling as shown in section 3 as well as the incremental nature of BMC can both be exploited for various optimizations in the underlying decision procedure. Currently, HySAT implements three optimizations which are described below.

4.2.1 Isomorphy inference

The learning scheme employed in propositional SAT solvers accounts for a substantial fraction of the solver's running time as it entails a non-trivial analysis of the implications that led to an inconsistent valuation. The creation of a conflict clause is in general even considerably more expensive in a combined solver like HySAT, as the analysis of a conflict involving non-propositional constraints requires the computationally expensive extraction of an IIS.

Isomorphy inference uses the (almost) symmetric structure of a BMC formula in order to add isomorphic copies of a conflict clause to the problem, thus multiplying the benefit taken from the time-consuming reasoning process which was required to derive the original conflict clause.

The concept is best illustrated using an example. Suppose that while solving a BMC instance the solver has encountered a conflict which yields the conflict clause $\mathcal{C}^0 = (\overline{x}_3^{j_1} \vee x_4^{j_2} \vee x_9^{j_3})$, relating three variables from cycles j_1, j_2 and j_3. The solver then not only adds $\mathcal{C}^0$ to ϕ^k, but also all possible clauses $\mathcal{C}^i = (\overline{x}_3^{j_1 \pm i} \vee x_4^{j_2 \pm i} \vee x_9^{j_3 \pm i})$, $i = 1, 2, \ldots$, obtained from $\mathcal{C}^0$ simply by index shifting.

Note, however, that BMC is not fully symmetric because of the initialization properties of runs (clause (5) of the translation scheme of section 3) and perhaps the verification goal. This implies that only conflict clauses inferred from facts which are independent from such asymmetric formula parts may be soundly replicated. Such dependency can be traced cheaply by marking initialization/goal predicates and dominantly inheriting such marks upon all inferences, inhibiting isomorphy inference whenever a mark is encountered.

4.2.2 Constraint sharing

When carrying out BMC incrementally for longer and longer unrollings, the consecutive formulae passed to the solver share a large number of clauses. Thus, when moving from the k-instance to the $(k+1)$-instance, we can simply conjoin the conflict clauses derived when solving k-instance to the formula for step $k+1$. However, this is only allowed for conflict clauses that were inferred from clauses which are common to both instances. We do currently decide this based on simple syntactic criteria, namely that the conflict clause was inferred purely from clauses stemming from the automaton. I.e. the inference may not involve the verification goal, which tends to become a weaker predicate on longer instances, as it usually entails reachability or recurrence. More elaborate schemes have, however, been investigated for propositional BMC in [26].

4.2.3 Tailored decision strategy

When applying general-purpose decision strategies to BMC formulae one can observe the phenomenon described in [34] that during the SAT search large sets of constraints belonging to distant cycles of the transition relation are being satisfied independently, until they finally turn out to be incompatible, often entailing the need for backtracking over long distances in the search tree.

In HySAT we adopt the solution proposed by Strichman [34] to avoid this problem: The heuristics of the SAT solver selects the decision variables in the natural order induced by the variable dependency graph of the BMC formula, i.e. either using a forward scheme, starting with variables from $\vec{x}^0$, then from $\vec{x}^1$, etc., or vice versa, engaging in a backward scheme. This allows conflicts to be detected and resolved more locally, speeding up the search, as witnessed by the results shown in figure 8.

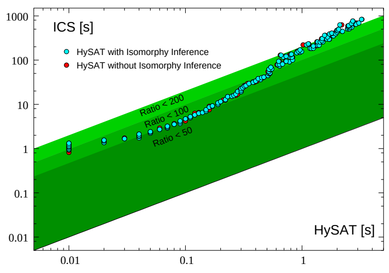
5 Benchmark results

For an evaluation of HySAT we conducted a series of experiments on BMC problems of hybrid automata in which we a) compared our tool with the ICS solver [14], and b) investigated the impact of the individual optimizations by comparing the computation times of our tool when running with and without the respective optimization beeing enabled. The unwindings fed to ICS were obtained through SRI's infinite-state BMC frontend to ICS as distributed in the SAL tool-set [15]. Our benchmarks are

- The *"leaking gas burner"* and *"water-level monitor"* included in the SAL distribution,
- An elastic approach to distance control of trains running on the same track, similar to the car platooning system used in the PATH project. Here, trains can accelerate or decelerate freely if they do not violate their mutual safety envelopes, yet an automatic speed control takes authority over a train if another train gets close, thereby controlling acceleration proportional (within physical limits) to the front and/or back proximity of the neighboring trains.
- A hybrid model of a car equipped with robotized five-speed transmission and a cruise control system which aims at maintaining a certain preset speed by actuating throttle and brake using two PI controllers. We adopted the model as reported in [35] and modified it by adding a realistic clutch behaviour in the initial acceleration phase.

The results of our experiments are shown in figures 4 – 8, with each data point representing a single BMC instance solved by two engines. Points lying on the diagonal, which is drawn as a solid line in all figures, indicate equal running times of both tools; points lying above (below) the diagonal represent instances that were solved faster by the engine whose running times can be read off from the x-axis (y-axis). Note the logarithmic scaling of the axes in figures 4 and 5.

It can be seen that the individual optimizations yield consistent performance benefits, with the merits becoming more evident with increasing unrolling depth, corresponding to computationally more costly SAT instances.

a) 

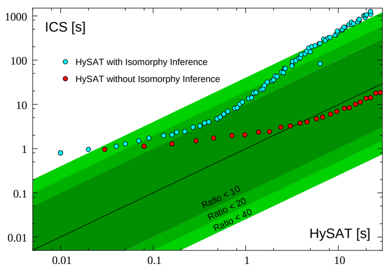
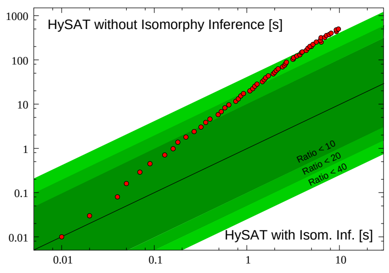
b) 

Figure 4: Performancs of HySAT relative to ICS: BMC times for a) gasburner model, b) water-level monitor.

a) 

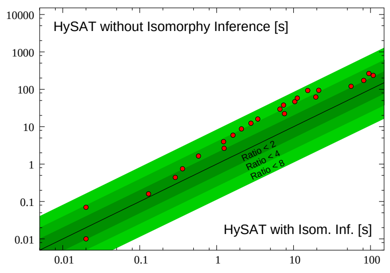
b) 

Figure 5: Impact of isomorphy inference: BMC times for a) water-level monitor, b) train distance control model, involving 5 trains.

This holds in particular for isomorphy inference, an exception being however the extremely deterministic gasburner model, see figure 5 a), where a strict state alternation is enforced by the discrete part such that learning of infeasible subsystems provides negligible extra information.

With respect to the decision strategy it turns out that there is no single optimal strategy. Depending on the specific shape of the initial state set and the target region, forward or backward strategies, though in general both better than the standard strategy, may be more beneficial. We are experimenting with randomized approaches to on-the-fly strategy switch to overcome the problem of selecting an appropriate strategy a priori.

6 Conclusion and further work

The benchmarks performed so far indicate a very competitive performance of HySAT when used for bounded model checking of linear hybrid systems. They do thus provide evidence for the effectiveness of HySAT's basic design decisions, which are

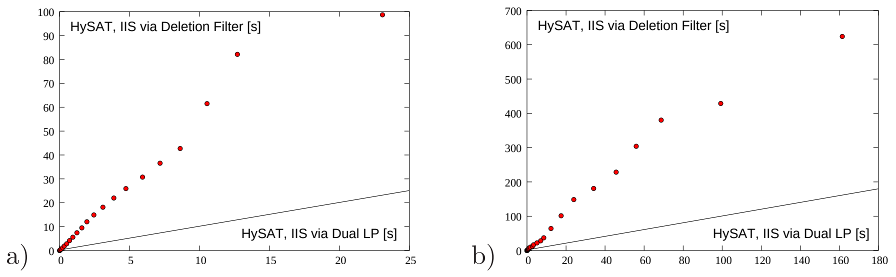


Figure 6: Comparison of deletion filter method for extraction of irreducible infeasible subsystems with method using the dual LP. Graphics show results for a) train distance control model, b) car model.

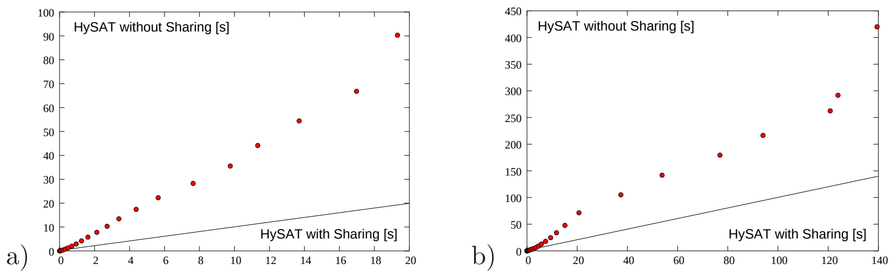
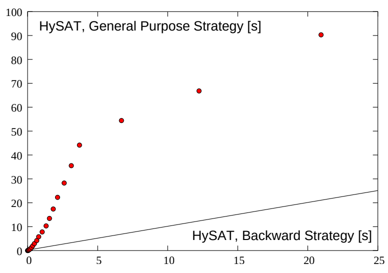


Figure 7: Impact of constraint sharing on BMC runtimes for a) train distance control model, b) car model.

1. the exploitation of structural properties of the formulae arising in bounded model checking and
2. the use of a non-clausal and thus more concise base logics.

With the current case studies, which are reachability properties in hybrid automata, measures of the first kind clearly have the predominant effect. Yet experiments with bounded model construction for the metric-time temporal logic Duration Calculus provide evidence that the conciseness gain from using linear zero-one constraint systems instead of CNF formulae will be essential to tractability once observers for metric-time temporal-logic formulae come into play [17].

HySAT's techniques for exploiting the particular structure of the verification conditions arising in bounded model checking (BMC) include inheritance of inference results along the temporal axis within an BMC instance, sharing of inference results across BMC instances, and decision heuristics in the SAT-solver that pay attention to the causal relationship between problem variables by doing chaining along the transition sequence. These algorithms have been inspired by similar optimizations developed by Strichman for finite-state BMC [34]; however such optimizations exhibit an even better payoff on the two-sorted

a) 

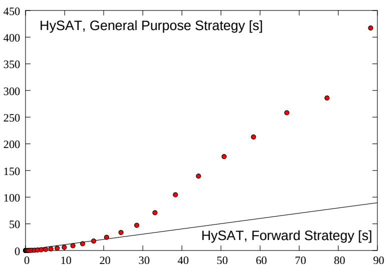
b) 

Figure 8: Impact of tailored decision strategies: a) For the train model the backward strategy clearly outperforms the general purpose decision strategy, whereas a forward scheme (not shown) slows down the solver. b) Conversely, for the car model the forward strategy is superior.

logics used here, as the price for copying inferences increases only marginally while the computational cost of the individual inference grows dramatically in the hybrid-state case. Consequently, the individual optimization yield speedups of up to, and sometimes even considerably exceeding, an order of magnitude.

An interesting aspect of isomorphically copying inference results, as in inheritance along the temporal axis or in sharing across BMC instances, is that even extremely costly inferences may amortize, provided that their results can be reused sufficiently often. Future versions of HySAT will thus incorporate more advanced —and computationally more costly— inference techniques combined with heuristics deciding when to use them. The rationale is here that a costly inference is more beneficial when there is sufficient chance for reuse of its result, i.e. when it is performed in early phases of the proof search, thus providing aggressive proof-tree pruning.

Another direction for future development is the extension of HySAT to non-linear arithmetic constraints, including transcendental functions, thereby extending the lazy theorem proving approach to undecidable domains which arise naturally in the verification of hybrid discrete-continuous systems. While undecidable theories are in general out-of-scope of the lazy theorem proving approach, as their processing requires extremely frequent calls to interactive theorem provers as subordinate procedures of the satisfiability solver, we exploit structural and topological properties of typical engineering problems to extend this approach far into undecidable arithmetic domains without entering into interactive verification schemes. Specifically addressing robust verification conditions, i.e. proof obligations that do not change their truth value under minor variation of the constants involved, we will integrate robust constraint solving procedures (e.g. those of [33]) with the lazy theorem proving paradigm. Given the ability of robust constraint solving to decide truth of robust formulae in real arithmetic including transcendental and other smooth functions, we thus obtain a fully symbolic procedures for the analysis of hybrid systems with large discrete state spaces and rich continuous dynamics.

Acknowledgements. The authors are grateful for the tight cooperation within the project area "Hybrid Systems" of the Transregional Research Action "AVACS" funded by the Deutsche Forschungsgemeinschaft. Special thanks go to Bernd Becker, Erika Ábrahám, Felix Klaedtke, and Jan-Georg Smaus for their kind hospitality and for many fruitful discussions during research visits to Freiburg, as well as to Fritz Eisenbrand and Markus Behle for their help in finding an efficient solution to the IIS extraction problem.

References

[1] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Generic ILP versus specialized 0-1 ILP: An update. In *Proc. ACM/IEEE Intl. Conf. Comp.-Aided Design (ICCAD)*, pages 450–457, Nov. 2002.

[2] G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowics, and R. Sebastiani. A SAT-based approach for solving formulas over boolean and linear mathematical propositions. In A. Voronkov, editor, *Proc. of the 18th International Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Artificial Intelligence*, pages 193–208. Springer-Verlag, 2002.

[3] G. Audemard, M. Bozzano, A. Cimatti, and R. Sebastiani. Verifying industrial hybrid systems with MathSAT. *ENTCS*, 89(4), 2004.

[4] L. Baptista, I. Lynce, and J. Marques-Silva. Complete search restart strategies for satisfiability. In *Proc. of the IJCAI'01 Workshop on Stochastic Search Algorithms (IJCAI-SSA)*, August 2001.

[5] C. Barrett, D. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *14th International Conference on Computer-Aided Verification*, 2002.

[6] P. Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-boolean optimization. Technical Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1995.

[7] A. Bemporad and M. Morari. Verification of hybrid systems via mathematical programming. In F. W. Vaandrager and J. H. van Schuppen, editors, *Hybrid Systems: Computation and Control (HSCC'99)*, volume 1569 of *Lecture Notes in Computer Science*, pages 31–45. Springer-Verlag, 1999.

[8] A. Biere, A. Cimatti, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[9] A. Bik and H. Wijshoff. Implementation of Fourier-Motzkin elimination. Technical Report TR94-42, Dpt. of Computer Sceince, University of Leiden, The Netherlands, 1994.

[10] D. Chai and A. Kuehlmann. A fast pseudo-boolean constraint solver. In *Proc. of the 40th Design Automation Conference (DAC 2003)*, pages 830–835, Anaheim (California, USA), June 2003. ACM.

[11] J. W. Chinneck. Finding a useful subset of constraints for analysis in an infeasible linear program. *INFORMS Journal on Computing*, 9(2):164–174, 1997.

[12] J. W. Chinneck and E. W. Dravnieks. Locating minimal infeasible constraint sets in linear programs. *ORSA Journal on Computing*, 3(2):157–168, 1991.

[13] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[14] L. de Moura, S. Owre, H. Ruess, J. Rushby, and N. Shankar. The ICS decision procedures for embedded deduction. In *2nd International Joint Conference on Automated Reasoning (IJCAR)*, volume 3097 of *Lecture Notes in Computer Science*, pages 218–222, Cork, Ireland, July 2004. Springer-Verlag.

[15] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In R. Alur and D. Peled, editors, *Computer-Aided Verification, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA, July 2004. Springer-Verlag.

[16] L. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Proceedings of the 18th International Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Computer Science*, pages 438–455. Springer-Verlag, July 2002.

[17] J. Enslev, A.-S. Nielsen, M. Fränzle, and M. R. Hansen. Bounded model construction for duration calculus. In N. Jones et al., editor, *Proceedings of the 17th Nordic Workshop on Programming Theory (NWPT 05).* Københavns Universitet, Oct. 2005.

[18] J. Fourier. Solution d'une qestion particulière du calcul des inégalités. *Nouveau Bulletin par la Société Philomathique des Paris*, pages 99–100, 1826.

[19] M. Fränzle and C. Herde. Efficient SAT engines for concise logics: Accelerating proof search for zero-one linear constraint systems. In M. Vardi and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2003)*, volume 2850 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2003.

[20] M. Fränzle and C. Herde. Efficient SAT engines for concise logics: Accelerating proof search for zero-one linear constraint systems. In A. V. Moshe Y. Vardi, editor, *Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2003)*, volume 2850 of *LNCS, subseries LNAI*, pages 302–316. Springer Verlag, 2003.

[21] J. Gleeson and J. Ryan. Identifying minimally infeasible subsystems of inequalities. *ORSA Journal on Computing*, 2(1):61–63, 1990.

[22] J. F. Groote, J. W. C. Koorn, and S. F. M. van Vlijmen. The safety guaranteeing system at station hoorn-kersenboogerd. In *Compass '95: 10th Annual Conference on Computer Assurance*, pages 57–68, Gaithersburg, Maryland, 1995. National Institute of Standards and Technology.

[23] E. C. R. Hehner. Predicative programming. *Communications of the ACM*, 27:134–151, 1984.

[24] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: The next generation. In *16th Annual IEEE Real-time Systems Symposium (RTSS 1995)*, pages 56–65. IEEE Computer Society Press, 1995.

[25] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing*, pages 373–382. ACM, 1995.

[26] H. Jin and F. Somenzi. An incremental algorithm to check satisfiability for bounded model checking. In A. Biere and O. Strichman, editors, *Preliminary Proceeding of BMC'04*. ETH Zürich, 2004.

[27] J. P. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Proc. of the 9th Portuguese Conference on Artificial Intelligence (EPIA)*, Sept. 1999.

[28] J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

[29] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. of the 38th Design Automation Conference (DAC'01)*, June 2001.

[30] T. S. Motzkin. *Beiträge zur Theorie der linearen Ungleichungen*. Doctoral dissertation, Universität Zürich, 1936.

[31] A. Nonnengart and C. Weidenbach. Computing small clause normal forms. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science B.V., 1999.

[32] M. E. Pfetsch. *The Maximum Feasible Subsystem Problem and Vertex-Facet Incidences of Polyhedra*. Doctoral dissertation, TU Berlin, 2002.

[33] S. Ratschan. Continuous first-order constraint satisfaction with equality and d isequality constraints. In P. van Hentenryck, editor, *Proc. 8th International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *Lecture Notes in Computer Science*, pages 680–685. Springer, 2002.

[34] O. Strichman. Tuning SAT checkers for bounded model checking. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification (CAV 2000)*, volume 1855 of *Lecture Notes in Computer Science*, pages 480–494. Springer-Verlag, 2000.

[35] F. D. Torrisi. *Modeling and Reach-Set Computation for Analysis and Optimal Control of Discrete Hybrid Automata*. Doctoral dissertation, ETH Zrich, 2003.

[36] G. Tseitin. On the complexity of derivations in propositional calculus. In A. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logics*, 1968.

[37] J. P. Warners. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters*, 68(2):63–69, 1998.

[38] J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A new incremental satisfiability engine. In *Proc. of the Design Automation Conference (DAC 2001)*, pages 542–545, Las Vegas (Nevada, USA), June 2001.

[39] S. A. Wolfman and D. S. Weld. The LPSAT engine & its application to resource planning. In T. Dean, editor, *Proc. 16th International Joint Conference on i Artificial Intelligence*, pages 310–315. Morgan Kaufmann Publishers, 1999.

[40] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proc. of the International Conference on Computer-Aided Design (ICCAD01)*, pages 279–285, Nov. 2001.