Preprint from Proceedings of GI/ITG/GMM-Workshop "Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen", Berlin, Germany, March 2009

Using Implications for Optimizing State Set Representations of Linear Hybrid Systems

Florian Pigorsch and Christoph Scholl
Albert-Ludwigs-Universität Freiburg, Institut für Informatik,
D-79110 Freiburg im Breisgau, Germany
Email:{pigorsch, scholl}@informatik.uni-freiburg.de

Abstract

In this paper we present an approach to exploit pre-calculated implication knowledge in the construction of LinAIGs which represent sets of states of Linear Hybrid Systems. Our method computes implications between linear constraints and uses this information to strengthen SAT-based equivalence checks which occur during the construction of the LinAIGs. The approach is evaluated on several hybrid model checking benchmarks where LinAIGs are used as the core data-structure of the model checker. The results show that the use of implications can significantly reduce the number of applications of expensive SAT-modulo-Theories (SMT) methods, and thus can accelerate LinAIG compaction methods which use equivalence checks.

I. Introduction

In this paper we present a method for optimizing state set representations which are used during the verification of Linear Hybrid Systems (LHAs) [5], [4]. Just as BDDs in the verification of discrete systems, e.g., the so-called LinAIGs form a symbolic representation for sets of states of Linear Hybrid Systems, which include both Boolean and real-valued state variables. LinAIGs represent arbitrary Boolean combinations (including conjunction, disjunction and negation) of Boolean variables and linear constraints. This is in contrast to approaches like [9], [8] where sets of states of LHAs are represented by *explicit* representations of discrete states in connection with sets of convex polyhedra (i.e. *conjunctions* of linear constraints), with one such set for each explicitly represented discrete state. The advantages of LinAIGs become evident in particular in the case of LHAs with *large discrete state spaces* which prohibit an explicit representation of discrete states. In [5], [4] LinAIGs were used in the context of model checking of CTL formulas by backward analysis.

LinAIGs are an extension of FRAIGs (Functionally Reduced AND-Inverter-Graphs) [10], [11] by linear constraints. Just as FRAIGs for Boolean functions they use a number of methods to keep the invariant that there is not any pair of equivalent nodes in the representation which represent the same function (up to complementation). In FRAIGs most *non*-equivalences of nodes are found by simulation, whereas SAT methods are used in order to prove equivalences. In the case of LinAIGs equivalence proofs may need SAT-modulo-Theories (SMT) solvers (more precisely SMT solvers for linear arithmetic) instead of SAT solvers, since Boolean functions were enhanced by linear constraints. Here we can make use of the tremendous advances of SMT solvers like Yices [6] or MathSAT [3], e.g., which were achieved during last years (see also [2]). In simpler cases however, equivalences can already be proven by (more efficient) SAT solvers just looking at the Boolean abstractions of linear constraints.

In this paper we make a detailed look into the question how the applicability of SAT solvers can be extended by inserting information about linear constraints into the SAT problems. The information which we insert is given by *implications* between linear constraints which can be easily pre-computed and inserted into the SAT problems in form of additional clauses. Since implications between linear constraints are transitive, we arrive at several options for inserting implications between the subsets of linear constraints contained in the equivalence problems at hand (as will be shown later on in the paper). Our realizations for these options make use of well-known results from graph theory.

First experimental results using benchmarks from industrial case studies show that our "layered approach" for keeping the representations as compact as possible is indeed successful. In most cases it is highly profitable to transport "easy-to-detect" information (in form of implications between linear constraints) from the real-valued domain into the Boolean domain. The experiments also suggest a clear tendency how graph theoretical operations like transitive closure and transitive reduction should be applied to the implications before inserting them into SAT problems.

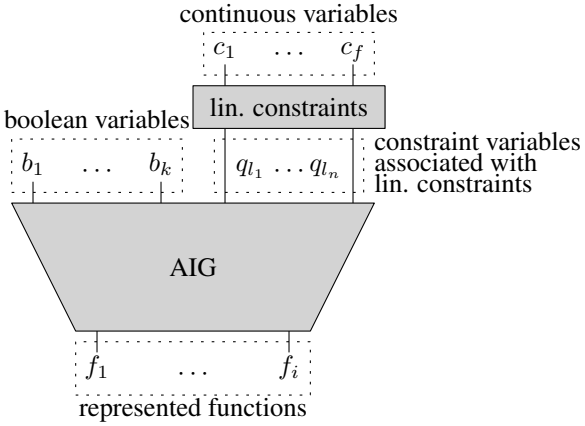


Fig. 1. The Structure of LinAIGs

The paper is organized as follows: In Sect. II we will give a short overview of the LinAIG data-structure including the main algorithm to maintain functional reduction of the structure. Sect. III focusses on implications, their computation and the graph based storage of implications. In Sect. IV we elaborate on how to exploit implications in SAT-based equivalence checks. After presenting experimental results in Sect. V, we conclude the paper in Sect. VI.

II. LinAIGs

Let $C = \{c_1, \ldots, c_f\}$ be a disjoint set of continuous variables ranging over the reals $\mathbb{R}$, and let $B = \{b_1, \ldots, b_k\}$ be a disjoint set of Boolean variables ranging over $\mathbb{B} = \{0, 1\}$. Let $\mathcal{L}(C) = \{\alpha_1 \cdot c_1 + \ldots \alpha_f \cdot c_f + \alpha_0 \leq 0 | \alpha_i \in \mathbb{Q} \wedge c_i \in C\}$ be the set of linear constraints over C.

$\mathcal{P}(C)$ is the set of all Boolean combinations of elements from $\mathcal{L}(C)$, a formula of $\mathcal{P}(C)$ represents a *non-convex polyhedron* over $\mathbb{R}^f$. We consider formulas from $\mathcal{P}(B, C)$, which is the set of all boolean combinations of boolean variables from B and linear constraints over C.

As a data-structure to represent formulas from $\mathcal{P}(B, C)$ we use LinAIGs [5], [4]. LinAIGs are And-Inverter Graphs (AIGs) enriched with linear constraints. The structure of LinAIGs is illustrated in Figure 1.

The main component of LinAIGs are And-Inverter Graphs, which basically are Boolean circuits solely consisting on AND gates and inverters. The second component of LinAIGs is a representation of the linear constraints $\mathcal{L}(C)$. The linear constraints are attached to the AIG by a set of new (Boolean) *constraint variables* $Q = \{q_1, \ldots, q_f\}$, where each occuring linear constraint l_i is uniquely encoded by some $q_i \in Q$.

In order to keep the LinAIGs representation compact, we maintain the *functional reduction* property, i. e. we avoid to represent equivalent (and antivalent) functions by different LinAIG nodes. Functionally reduced LinAIGs are called *semi-canonical*, since each node in a functionally reduced LinAIG represents a unique function (modulo complementation). This goal is achieved already during the construction of the LinAIG.

A. LinAIG Construction

Algorithm 1 shows the pseudo-code of the main LinAIG construction method computing the conjunction for two AIG nodes a and b. The algorithm extends the creation method of functionally reduced AIGs (FRAIGs) [10], [11] by adding SMT-based equivalence checks to find equivalences modulo linear constraints. The method produces a functionally reduced LinAIG by a layered application of various methods with increasing power and complexity:

First, trivial cases are checked, e. g. if operands a and b are equal (up to complementation) or one of the two operands is a constant. Then a *unique table* lookup is performed to find a node in the AIG which would be isomorphic to the inserted node.

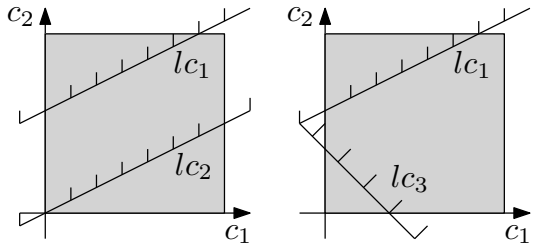
Next, equivalence checks are performed without interpreting constraint variables by their corresponding linear constraints. If two nodes can be proved to be equivalent already without interpreting constraint variables, then it is clear that they will remain equivalent when constraint variables are replaced by the corresponding linear constraints. These equivalence checks are reduced to SAT problems and in this way they provide a sound, but incomplete method for identifying already existing nodes representing the same function as the conjunction of a and b. However, just as for FRAIGs, not all nodes are considered as candidates for Boolean equivalence, but simulation is used to filter out easy-to-detect non-equivalences. I.e. candidate nodes for Boolean equivalence are collected by boolean simulation and only for candidate nodes with simulation results identical to those of the new AND node SAT-based functional equivalence checks are performed. Note that, due to the LinAIG's functional reduction property, at most one of the candidate node can be equivalent to the new node. If an equivalence is found, the candidate node is returned. For the SAT-based equivalence check of two nodes n_1 and n_2, we encode the *miter circuit* $n_1 \oplus n_2$ for the input cones of n_1 and n_2 as a CNF formula omitting the correspondence of constraint variables and linear constraints. If the SAT solver is able to find a satisfying assignment to the variables of the CNF, the two nodes n_1 and n_2 are proven to be non-equivalent (as long as the constraint variables are not replaced by the corresponding linear constraints) and in this case we learn the assignment to the Boolean input variables computed by the SAT solver as an additional Boolean simulation vector to refine future boolean simulations determining sets of candidate nodes.

Finally, exact equivalence checks are performed using an SMT solver taking linear constraints into account. Here filtering of candidates for equivalence is performed as well, but now (in order to avoid assignments to the constraint variables which are inconsistent wrt. the linear constraints) simulations with assignments to real variables $c_1, \ldots, c_f$ have to be used for proving non-equivalence. Finally, if no equivalent node was found, the new node is returned.

Algorithm 1 OperationAnd

1: **function** OPERATIONAND(a, b) ▷ Returns an AIG node equivalent to $a \wedge b$
2: $(trivial, trivial\_result) \leftarrow trivialCase(a, b)$ ▷ Checks for trivial cases, e. g.$a = b$
3: **if** $trivial$ **then**
4: **return** $trivial\_result$
5: $(exists\_isomorphic, isomorphic\_result) \leftarrow searchForIsomorphicNode(a, b)$ ▷ Checks if an isomorphic node exists
6: **if** $exists\_isomorphic$ **then**
7: **return** $isomorphic\_result$
8: $new \leftarrow newNode(a, b)$ ▷ Creates a new AIG node
9: **for** $c \in \{n \in AIG | bsim(n) = bsim(new)\}$ **do** ▷ Find candidates based on Boolean simulation
10: **if** $equivalentSAT(c, new)$ **then** ▷ Perform a SAT-based equivalence check
11: $delete(new)$
12: **return** c
13: **for** $c \in \{n \in AIG | rsim(n) = rsim(new)\}$ **do** ▷ Find candidates based on real valued simulation
14: **if** $equivalentSMT(c, new)$ **then** ▷ Perform an SMT-based equivalence check
15: $delete(new)$
16: **return** c
17: $insertIntoLinAIG(new)$
18: **return** new

Note that linear constraints are also subject to functional reduction, i. e. before inserting a new linear constraint into the LinAIG, we check if there already exists an equivalent (or antivalent) constraint. (If global variable bounds for variables c_i are given, then we call two linear constraints already as equivalent, if they do not differ in the subspace given by these bounds.) In such a case we avoid the creation of a new constraint variable and reuse the existing one.



(a) Implication between parallel constraints (b) Implication modulo bounds

Fig. 2. Examples of Implications

III. Implications

Since SMT solver calls may be expensive, we now look into the question how the (incomplete) SAT-based equivalence checks can be strengthened by inserting information about the linear constraint domain into SAT problems. The information which we use is given by *implications* between linear constraints which can be easily computed *a-priori*.

Definition 1 (Set of Implications) *Let C be a set of real-valued continuous variables, each variable $c_i \in C$ has upper and lower bounds $lb_i, ub_i \in \mathbb{R}$, such that $lb_i \leq c_i \leq ub_i$ (unbounded variables have infinite bounds). Let $L \subseteq \mathcal{L}(C)$ be a set of linear constraints over C.*

The set of implications $\mathcal{I}(L)$ *is defined as*

$$\mathcal{I}(L) := \{(l_1^{n_1}, l_2^{n_2}) \,|\, l_1, l_2 \in L,\, l_1 \neq l_2,\, n_1, n_2 \in \mathbb{B},\, \left(\bigwedge_{1\leq i\leq f} (lb_i \leq c_i \leq ub_i)\right) \rightarrow (l_1^{n_1} \rightarrow l_2^{n_2})\}$$

where l_i^0 denotes complementation, i. e. $l_i^1 := l_i$ and $l_i^0 := \overline{l_i}$.

Note that, according to this definition, $l_1^{n_1}$ already implies $l_2^{n_2}$, if the implication relation holds inside the subspace $\bigwedge_{1\leq i\leq f} (lb_i \leq c_i \leq ub_i)$ given by the global bounds on the real variables.

Due to efficiency reasons we only compute a subset I of $\mathcal{I}(L)$:

We compute implications between (parallel) linear constraints $\alpha_1 \cdot c_1 + \ldots + \alpha_f \cdot c_f + \alpha_0 \leq 0$ and $\alpha_1 \cdot c_1 + \ldots + \alpha_f \cdot c_f + \alpha_0' \leq 0$ where $\alpha_0 > \alpha_0'$ (and analogously between negations of linear constraints). It is easy to see that this is the only possible case of implications without taking the bounds of variables into account. An example is shown in Figure 2(a) where lc_1 implies lc_2.

Additionally we use an incomplete method to detect implications modulo global variable bounds, where a linear constraint $\alpha_1' \cdot c_1 + \ldots + \alpha_f' \cdot c_f + \alpha_0' \leq 0$ follows from $\alpha_1 \cdot c_1 + \ldots + \alpha_f \cdot c_f + \alpha_0 \leq 0$ within the variable bounds $lb_i \leq c_i \leq ub_i$. Figure 2(b) shows a situation where lc_1 implies the non-parallel lc_3 within the bounds (grey box).

A. Implication Graph

Let $L = \{lc_1, \ldots, lc_n\}$ be a set of linear constraints. The set $I \subseteq \mathcal{I}(L)$ of detected implication relations between pairs of (negated) linear constraints from L is represented in the *implication graph*, a skew-symmetric[1] directed graph $IG(L, I) = (V, E)$, where the set of vertices $V = \{v_{lc_1}, v_{\overline{lc_1}}, \ldots, v_{lc_n}, v_{\overline{lc_f}}\}$ contains two vertices for each linear constraint lc_i of L: the vertex v_{lc_i} for the non-negated constraint and $v_{\overline{lc_i}}$ for the negated constraint. For each detected implication $(lc_i \rightarrow lc_j) \in I$ we add the directed edge (v_{lc_i}, v_{lc_j}) and – since in this case $\overline{lc_j} \rightarrow \overline{lc_i}$ holds as well – its contrapositive $(v_{\overline{lc_j}}, v_{\overline{lc_j}})$ to the set of edges E (implications containing negations of linear constraints are handled analogously).

The implication graph for the set of linear constraints $\{lc_1, \ldots, lc_6\}$ and the implications $\{lc_1 \rightarrow lc_2, lc_2 \rightarrow lc_3, lc_2 \rightarrow \overline{lc_4}, lc_2 \rightarrow lc_5, lc_3 \rightarrow lc_5, lc_5 \rightarrow lc_6\}$ is shown in Figure 3.

[1]A directed graph is skew-symmetric iff it is isomorphic to the graph formed by reversing all of its edges. Here, the corresponding isomorphism σ is given by $\sigma(v_{lc_i}) = v_{\overline{lc_i}}$ and $\sigma(v_{\overline{lc_i}}) = v_{lc_i}$, for all $1 \leq i \leq n$.

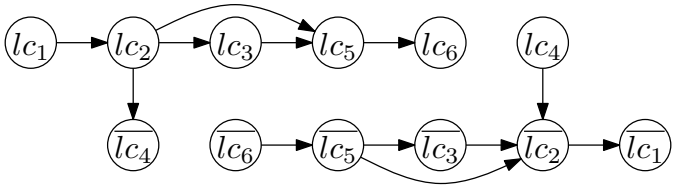


Fig. 3. An Implication Graph

Theorem 1 (Acyclicity of the Implication Graph) *Let $L \subseteq \mathcal{L}(C)$ be a set of linear constraints over C, containing pairwise non-equivalent and non-antivalent linear constraints only, and let I be a set of implications between (negations) of linear constraints from L.*

Then the implication graph $IG(L, I)$ is acyclic.

Proof: Assume that $IG(L, I)$ has a cycle $v_1, v_2, \ldots, v_{m-1}, v_m, v_1$. Due to the construction of the implication graph there exist cyclic implications between the corresponding linear constraints, and thus all constraints corresponding to the nodes on the cycle are pairwise equivalent or antivalent (depending on negations), which is a contradiction to the theorem's assumptions. ■

Since we detect and merge equivalent (and antivalent) linear constraints during construction of the LinAIG, the set of constraints contained in the LinAIG is free of equivalences (and antivalences). According to Theorem 1 an implication graph for that set of constraints occuring in a LinAIG is acyclic.

As we will describe in Sect. IV we add implications to SAT-based equivalence checks based on the *transitive closure* and the *transitive reduction* of the implication graph.

Definition 2 (Transitive Closure) *Let $G = (V, E)$ be a directed graph. The* transitive closure $G^+ = (V, E')$ *of G is a directed graph, such that $(v, w) \in E'$ if and only if there exists a directed path from v to w in G.*

The transitive closure of an acyclic graph can be computed in $O(|V| \cdot |E|)$ time.

Definition 3 (Transitive Reduction) *Let $G = (V, E)$ be a directed graph. The* transitive reduction $G^- = (V, E')$, $E' \subseteq E$, *of G is the smallest directed graph, such that there is a directed path from v to w in G^- if and only if there is a directed path from v to w in G.*

The transitive reduction of a graph G intuitively is the smallest subgraph of G that maintains all transitive relationships between vertices of G, i. e. if there exists a directed path $p = v, \ldots, w$ in G then there also exists a directed path $p' = v, \ldots, w$ in G^-.

As shown in [1] the transitive reduction of an acyclic, directed graph is uniquely determined and can be computed in $O(|V| \cdot |E|)$ time.

The algorithm for computing the transitive reduction of a directed, acyclic graph first traverses the graph in topological order and for each node $v \in V$ determines the set of nodes $reach_2(v)$ reachable from v with a path of length ≥ 2. In the second step all those edges $(v, w) \in E$ are deleted from the graph, where there exists a path from v to w with length ≥ 2, i. e. where $w \in reach_2(v)$.

The uniqueness of the transitive reduction can be shown by proving that if there exist two edges $(v, w), (v', w') \in E$ and two paths $p = v, \ldots, w$ and $q = v', \ldots w'$ with length ≥ 2, removing one of the edges does not prevent a removal of the second edge.

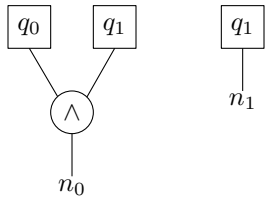
IV. Exploiting Implications in SAT Checks

A. A Motivating Example

Consider a pair of LinAIG nodes that are equivalent w. r. t. linear constraints.

The pure SAT-based equivalence check of the two LinAIG may find *spurious* satisfying assignments because the correspondence between constraint variables and linear constraints is not taken into account, and therefore it may classify the two nodes as *non-equivalent*.

Example 1 *Consider the following LinAIG nodes, where q_0 and q_1 are the constraint variables of $lc_0 : c_0 - 1 \leq 0$ and $lc_1 : c_0 \leq 0$ respectively.*



Checking the equivalence of n_0 and n_1 using a SAT solver will return the satisfying assignment $(q_0 = 0, q_1 = 1)$ and thus classify the nodes as non-equivalent.

On the other hand, it is clear that $(c_o - 1 \leq 0) \wedge (c_0 \leq 0) \equiv (c_0 \leq 0)$, i. e. an SMT-based equivalence check with full knowledge of linear constraints would classify the pair of nodes as *equivalent*, since all assignments to the constraint variables that may lead to a counter example are infeasible w. r. t. the corresponding linear constraints. In the example, the SMT check will detect the equivalence of n_0 and n_1.

However, it is possible to strengthen the SAT-based equivalence check procedure by adding limited knowledge about linear constraints, enabling the SAT solver to prove some of the equivalences that originally can only be found by full SMT checks. The knowledge that we use for this strengthening is a subset of the implications we compute a-priory by simple methods.

In the example, adding the implication $(lc_1 \rightarrow lc_0)$ in form of a binary clause expressing the implication in terms of constraint variables $(q_0 \vee \overline{q_1})$ to the CNF formula encoding the SAT-based equivalence check, allows the SAT solver to prove the equivalence of n_0 and n_1.

The exploitation of implications in SAT instances consists of two parts: (1) adding clauses to the SAT instances corresponding to implications, and (2) making simulation vectors, created from the satisfying assignments found by the SAT solver, consistent with the implications.

B. Adding Clauses

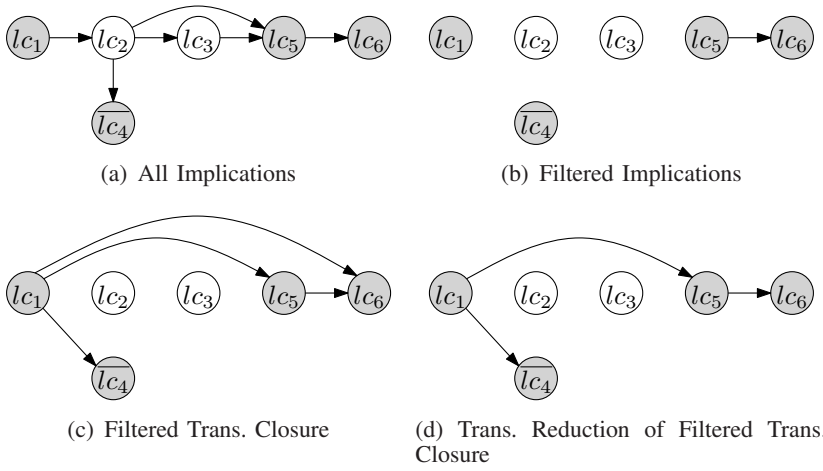
Let n_1 and n_2 be two LinAIG nodes, whose equivalence is to be checked by a SAT solver. Let L be the set of linear constraints occuring in the union of n_1's and n_2's cones. Let I be the set of computed implications between pairs of (negated) linear constraints occuring in the LinAIG.

By *adding a set of implications I' to a CNF* we mean that for any implication $(a \rightarrow b) \in I'$ we introduce a binary clause expressing the implication in terms of the corresponding constraint variables, e. g. for the implication $(lc_i \rightarrow lc_j)$ we add the clause $(\overline{q_i} \vee q_j)$ and for the implication $(lc_i \rightarrow \overline{lc_j})$ we add the clause $(\overline{q_i} \vee \overline{q_j})$.

There are several different strategies for selecting the set of implications to be added to the CNF of a SAT-based equivalence check. We will take a look on some of these strategies and discuss their effect on the SAT instance.

To support the discussion, we introduce an example on which we will apply the different strategies: Let $\{lc_1, lc_2, lc_3, lc_4, lc_5, lc_6\}$ be the linear constraints of a LinAIG, suppose we pre-computed the implications $I = \{lc_1 \rightarrow lc_2, lc_2 \rightarrow lc_3, lc_2 \rightarrow \overline{lc_4}, lc_2 \rightarrow lc_5, lc_3 \rightarrow lc_5, lc_5 \rightarrow lc_6\}$. Let $L = \{lc_1, lc_4, lc_5, lc_6\}$ be the constraints that are occuring in the cones of the two LinAIG nodes, whose equivalence is to be checked. Figure 4(a) shows one part of the skew-symmetric implication graph.

The simplest strategy to select a set of implications is to take *all* implications, irrespective of which constraint variables are occuring in the cones of the checked nodes. This method adds new variables to the SAT instance (exactly those constraint variables that are needed for the clausal representation of the implications but that do not occur in the cones). Additionally, since the set of all computed implications may contain redundant implications, redundant clauses may be added to the CNF of the SAT instance. Especially the size of small and easy SAT instances can be increased dramatically by this strategy. In the example variables for the constraints lc_2 and lc_3 would be added to the CNF, as well as a clause corresponding to the redundant implication $lc_2 \rightarrow lc_5$.



(a) All Implications

(b) Filtered Implications

(c) Filtered Trans. Closure

(d) Trans. Reduction of Filtered Trans. Closure

Fig. 4. Different Strategies for Selecting Implications

The introduction of new variables can be avoided by *filtering* the implications, such that only those implications are selected, whose both constraints are occuring in the cones to the tested nodes. The disadvantage of this strategy is shown in Figure 4(b): it is possible that some transitive implication relationships are omitted, leading to a suboptimal addition of implication knowledge and therefore resulting in a "weaker" SAT-check. In the example, the transitive implications $lc_1 \rightarrow \overline{lc_4}$ and $lc_1 \rightarrow lc_5$ are lost by simple filtering.

Omitting known transitive relationships between linear constraints occuring in the cones can be overcome by computing the *transitive closure* of the implication graph before filtering out implications concerning linear constraints not occuring in the cones. Again, the set of added implications is not minimal, since redundant implications may be added to the SAT instance. In our example, the redundant implication $lc_1 \rightarrow lc_6$ is added to the SAT instance (Figure 4(c)).

The minimized set of implications, in the sense that neither unneeded variables are introduced to the SAT instance nor redundant implications are added, while maintaining the transitive relationships, can be computed in a three staged approach: (1) compute the transitive closure of the implication graph, (2) restrict the resulting graph to the set of linear constraints occuring in the SAT instance, and (3) eliminate redundant implications by computing the *transitive reduction* of the restricted graph. The resulting set of implications for our example is shown in Figure 4(d).

In our experimental section we compare three of these strategies (adding all implications, adding the filtered transitive closure of the implications, and adding the minimized filtered transitive closure of the implications) against a LinAIG version that relies on SMT check only, as well as a version that performs SMT and SAT checks but completely omits implications.

Note that the transitive closure of the implication graph only needs to be recomputed if a new set of linear constraints is added to the LinAIG and some new implications are computed.

C. Making Simulation Vectors Consistent with Implications

To guide the process of functional reduction we use simulation by Boolean simulation vectors generated from earlier SAT-based equivalence checks. Since typically only a subset of the linear constraints existing in the LinAIG is involved in a SAT-based equivalence check, and, depending on the strategy, only a subset of all computed implications is added to the CNF, the satisfying assignment returned by the SAT solver may be incomplete, in the sense that is only assigns values to a subset of constraint variables. Such an incomplete assignment must be extended to a complete assignment before it can be used as a simulation vector. The extension is performed by assigning random Boolean values to the unassigned variables.

In case implications are used in the SAT-based equivalence checks, the completed simulation vector has to be consistent with the implications, i. e. the simulation vector must not assign values to constraint variables that are conflicting with an implication between the two variables. Inconsistent simulation

Model	PeakNodes	LCs	RVars	Implications	AndOps	Total Equiv.
3LP-cont-int-ED	12475	240	11	2985	276812	20765
3LP-cont-int-Sta	82171	546	11	15139	1009625	118689
3LP-cont	1271	118	8	412	16122	1222
3LP-disc-int-ED	3052	78	4	137	119453	21500
3LP-disc-int-Sta	3135	114	4	190	141955	29870
4LP-cont-int-ED	110115	391	11	7343	2897275	418935
4LP-cont-int-Sta	112349	829	11	26067	3400221	477556
4LP-cont	2582	231	8	1171	47917	4399
4LP-disc-int-ED	11020	111	4	197	160752	35729
4LP-disc-int-Sta	17936	202	4	335	196777	44995
xing-cont-int-ED	10614	1397	11	55671	412635	26467
xing-cont-int-const1	6122	2022	8	78662	141622	20376

TABLE I
COMMON MODEL CHECKING STATISTICS

vectors may prevent SAT-based equivalence checks for nodes, whose equivalence can be proven by SAT enriched with implications.

A simulation vector can be made consistent with the implications by traversing the implication graph's vertices in topological order, and for each outgoing edge $(v_{lc_i^{e_1}}, v_{lc_j^{e_j}})$ modifying the simulation vector's value for the constraint variable of the implied constraint according to the corresponding implication, e. g. if the edge is of the form (v_{lc_i}, v_{lc_j}), the simulation vector's value of the constraint variable c_j is set to 1 if c_i's value is 1, implications involving negated constraints are handled analogously.

Since all edges of the implication graph are processed exactly once, the algorithm is linear in the number of edges.

V. EXPERIMENTAL RESULTS

We implemented the proposed methods for exploiting implication knowledge in our LinAIG framework. We use the SAT solver MiniSAT [7] for the Boolean reasoning part and the SMT solver Yices [6] for performing SMT-based equivalence checks.

For demonstrating the effectiveness of the proposed method we ran the hybrid model checker FOMC [5], [4], which relies on LinAIGs as a symbolic data-structure for representing and manipulating state sets, on several hybrid model checking benchmarks from industrial case studies. Details on the model checking algorithm and the used benchmarks can be found in [5], [4].

We created five different versions of the LinAIG construction algorithm: the first version (*SMT only*) only uses SMT-based equivalence checks for detecting equivalent LinAIG nodes, the second version (*SMT, SAT*) additionally applies SAT checks omitting implications, the third variant (*SMT,SAT+Imp*) adds all computed implications to each SAT instance, the forth variant (*SMT,SAT+FTImp*) performs SMT- and SAT-based equivalence checks adding the filtered transitive closure of the implications to each SAT instance, and the fifth version (*SMT,SAT+RFTImp*) performs SMT- and SAT-based equivalence checks and adds the transitive reduction of filtered transitive closure of the implications.

All experiments were performed on an AMD Quadcore-Opteron with 2.3 GHz and 64 GB RAM running Linux. All reported run times are given in CPU seconds.

We assembled a table (Table I) that lists for all benchmarks some statistics common to the five LinAIG variants, like the number of linear constraints created during model checking, the number of continuous variables occuring, the number of computed implications, the peak number of LinAIG nodes, and the total number of detected equivalences.

As shown in Table I the number of linear constraints created during model checking ranges from 78 to 2022, up to 78662 implications are computed, and up to 477556 equivalent nodes are constructed during the procedure.

For each version we looked into the equivalence detection part of the algorithm. Table II shows the results of the different methods by counting the number of SAT-based equivalence checks and the number of equivalences detected by SAT (rows *SAT: chk/eq*), the number of SMT-based equivalence checks and the number of equivalences detected by SMT (rows *SMT: chk/eq*), the amounts of time

needed for SAT solving[2], implication handling[3], and SMT solving (rows *Time: SAT/Imp/SMT*) in CPU seconds, as well as the sum of the three times (rows *total eq chk time*), which summarizes the total amount of time needed for detecting equivalent LinAIG nodes.

By looking at the rows *SAT: chk/eq* and *SMT: chk/eq* it is easy to see that 25% (xing-cont-int-const1) up to 75% (3LP-cont-int-Sta) of all equivalences can already be detected by SAT-based equivalence checking (omitting implications), and thus do not have to be tested by SMT-based equivalence checks. By adding implications to the SAT checks, this ratio can be increased to values from 57% (xing-cont-int-ED) up to almost 100% (3LP-disc-int-ED, 3LP-disc-int-Sta, 4LP-disc-int-ED, 4LP-disc-int-Sta), further reducing the need to apply an SMT solver. This clearly indicates the usefulness of implications. As expected, the versions *SMT,SAT+Imp*, *SMT,SAT+FTImp*, and *SMT,SAT+RFTImp* show a common behavior w. r. t. the numbers of equivalences found by SAT, since in these versions the transitive relationships between linear constraints occuring in the SAT-checks are maintained. The versions mainly differ in the times used for SAT-based equivalence checks.

The experiments also show that the effort to minimize the set of added implications actually pays off: the SAT runtimes of *SMT,SAT+FTImp* are consistently smaller than or similar to the SAT times of *SMT,SAT+Imp*, while not changing the number of detected equivalences. Moreover, computing the transitive reduction (*SMT,SAT+RFTImp*) results in even smaller runtimes compared to *SMT,SAT+FTImp*.

On almost all tested benchmarks the variant *SMT,SAT+RFTImp* results in the smallest total amounts of time needed for checking the equivalence of LinAIG nodes (including SAT solving, handling of implications and SMT solving). An exception are the two *xing*-benchmarks, where the utilization of implications leads to a runtime increase. Detailed inspections of the model checking runs reveal the cause of the increase: most of the constraints created during model checking of these benchmarks are only temporary and become "garbage" soon in the sense that they do not occur in state set representations anymore. In the current implementation such constraints are remaining in the implication graph, negatively affecting the run time for handling implications. On the other hand SMT checks are not affected by unused linear constraints, since the SMT solver only deals with the constraints occurring in the tested cones. This problem can be eliminated by applying "garbage collection" for unused constraints, which will be added to our preliminary implementation in the near future.

In summary, the experimental results in Table II show that integrating easy-to-detect knowledge about linear constraints into SAT checks can indeed accelerate model checking of hybrid systems by successfully reducing the number of expensive SMT solver calls. Restricting the implications to those between linear constraints occurring in the problem at hand and further minimizing the number of these implications by transitive reduction turns out to be the best method among the alternatives investigated here.

VI. Conclusions

We presented an approach for beneficially exploiting implication knowledge in the LinAIG construction algorithm, especially focussing on "strengthening" SAT-based equivalence checks by different strategies for adding sets of implications. Our experimental results on hybrid model checking benchmarks from industrial case studies show that the number of applications of expensive SMT methods can substantially be reduced by integrating sets of implications, minimized by graph theoretical operations, into SAT-based equivalence checks, resulting in an accelerated LinAIG compaction.

In the future it will be interesting to use the implication knowledge also in other LinAIG optimization techniques, such as don't care based resynthesis of the underlying AIG structure.

References

[1] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM J. Comput.*, 1(2):131–137, 1972.

[2] C. Barrett, M. Deters, A. Oliveras, and A. Stump. Satisfiability Modulo Theories Competition (SMT-COMP) 2008: Rules and Precedures, 2008. http://smtcomp.org/rules08.pdf.

[2]The time for SAT solving includes the generation of the SAT instance and in particular the time needed for adding implications to the CNFs. Depending on the strategy adding implications involves filtering of the set of implications and computing the transitive reduction.

[3]The time for implication handling includes the pre-computation of implications, inserting implications into the implication graph, making simulation vectors consistent, and, depending on the strategy, computing the transitive closure of the implication graph.

Benchmark		SMT only	SMT, SAT	SMT, SAT+Imp	SMT, SAT+FTImp	SMT, SAT+RFTImp
3LP-cont	SAT: chk/eq	0/0	413/413	986/871	982/871	984/871
	SMT: chk/eq	1599/1222	1166/809	735/351	730/351	715/351
	Time: SAT/Impl/SMT	0.0/0.0/1.3	0.0/0.0/1.0	0.2/0.1/0.7	0.1/0.2/0.6	0.1/0.2/0.8
	Total eq chk time	1.3	1.0	**0.9**	**0.9**	1.0
3LP-cont-int-ED	SAT: chk/eq	0/0	16841/14691	22245/18823	22329/18823	22252/18823
	SMT: chk/eq	26810/20765	11710/6074	7534/1942	7800/1942	7465/1942
	Time: SAT/Impl/SMT	0.0/0.0/84.4	9.1/0.0/38.5	40.0/1.7/33.4	15.3/1.7/29.2	13.2/1.6/27.5
	Total eq chk time	84.4	47.6	75.1	46.1	**42.2**
3LP-cont-int-Sta	SAT: chk/eq	0/0	101498/89444	126916/107838	126758/107838	126740/107838
	SMT: chk/eq	161054/118689	69929/29245	52510/10851	52744/10851	54530/10851
	Time: SAT/Impl/SMT	0.0/0.0/2200.1	351.7/0.0/859.6	1212.4/20.7/684.9	423.9/19.5/614.9	371.7/18.7/640.4
	Total eq chk time	2200.1	1211.3	1918.0	1058.2	**1030.8**
3LP-disc-int-ED	SAT: chk/eq	0/0	6343/5596	22495/21486	22483/21486	22544/21486
	SMT: chk/eq	23176/21500	17576/15904	1648/14	1571/14	1627/14
	Time: SAT/Impl/SMT	0.0/0.0/30.5	1.2/0.0/23.4	5.5/0.1/4.9	5.3/0.1/4.6	4.9/0.2/4.6
	Total eq chk time	30.5	24.6	10.5	10.0	**9.7**
3LP-disc-int-Sta	SAT: chk/eq	0/0	8446/7833	30915/29795	30937/29795	30992/29795
	SMT: chk/eq	31909/29870	23951/22037	1847/75	1880/75	1904/75
	Time: SAT/Impl/SMT	0.0/0.0/36.4	1.3/0.0/27.6	7.7/0.2/6.3	5.8/0.3/5.8	5.2/0.2/5.8
	Total eq chk time	36.4	29.0	14.2	11.9	**11.2**
4LP-cont	SAT: chk/eq	0/0	1760/1749	3856/3538	3876/3538	3866/3538
	SMT: chk/eq	4916/4399	3147/2650	1314/861	1312/861	1329/861
	Time: SAT/Impl/SMT	0.0/0.0/4.3	0.2/0.0/2.9	2.0/0.6/1.3	0.8/0.5/1.3	0.5/0.5/1.2
	Total eq chk time	4.3	3.2	3.9	2.5	**2.2**
4LP-cont-int-ED	SAT: chk/eq	0/0	326283/250251	475441/379889	485840/379889	487117/379889
	SMT: chk/eq	845754/418935	587241/168684	447881/39046	461310/39046	449036/39046
	Time: SAT/Impl/SMT	0.0/0.0/15339.0	632.3/0.0/9689.7	2582.0/13.8/7350.2	1032.2/14.5/7351.8	933.8/14.5/7133.2
	Total eq chk time	15339.0	10321.9	9946.1	8398.5	**8081.4**
4LP-cont-int-Sta	SAT: chk/eq	0/0	326959/279822	493799/419812	491076/419812	491450/419812
	SMT: chk/eq	724255/477556	435623/197734	293920/57744	295138/57744	297881/57744
	Time: SAT/Impl/SMT	0.0/0.0/11615.2	421.0/0.0/6251.9	8206.5/44.8/4601.4	1362.5/43.5/4506.3	597.6/42.7/4296.5
	Total eq chk time	11615.2	6672.8	12852.7	5912.3	**4936.8**
4LP-disc-int-ED	SAT: chk/eq	0/0	25118/23947	37311/35697	37277/35697	37265/35697
	SMT: chk/eq	38448/35729	14211/11782	2471/32	2481/32	2422/32
	Time: SAT/Impl/SMT	0.0/0.0/123.7	10.3/0.0/54.3	23.9/0.4/13.5	25.6/0.3/11.0	22.1/0.3/11.0
	Total eq chk time	123.7	64.5	37.7	36.9	**33.4**
4LP-disc-int-Sta	SAT: chk/eq	0/0	32173/30755	46734/44715	46762/44715	46741/44715
	SMT: chk/eq	48142/44995	16983/14240	2913/280	2942/280	2863/280
	Time: SAT/Impl/SMT	0.0/0.0/243.5	23.5/0.0/83.3	46.2/0.5/19.4	50.4/0.5/17.8	44.2/0.5/19.3
	Total eq chk time	243.5	106.8	66.0	68.7	**64.0**
xing-cont-int-const1	SAT: chk/eq	0/0	5116/5098	14032/12011	14044/12011	14005/12011
	SMT: chk/eq	21769/20376	16640/15278	9629/8365	9608/8365	9581/8365
	Time: SAT/Impl/SMT	0.0/0.0/25.5	0.5/0.0/15.8	309.4/79.2/12.9	54.6/88.4/11.2	2.0/89.7/10.8
	Total eq chk time	25.5	**16.3**	401.5	154.1	102.5
xing-cont-int-ED	SAT: chk/eq	0/0	10547/9917	18071/15131	17978/15131	17981/15131
	SMT: chk/eq	29359/26467	19436/16550	14169/11336	14372/11336	14234/11336
	Time: SAT/Impl/SMT	0.0/0.0/67.8	4.2/0.0/39.1	280.0/36.8/26.8	42.5/41.5/28.5	6.0/40.5/24.4
	Total eq chk time	67.8	**43.3**	343.5	112.5	70.9

TABLE II
IMPACT OF IMPLICATIONS ON EQUIVALENCE CHECKS DURING MODEL-CHECKING

[3] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. Rossum, S. Schulz, and R. Sebastiani. MathSAT: Tight integration of SAT and mathematical decision procedures. *J. Autom. Reason.*, 35(1–3):265–293, 2005.

[4] W. Damm, S. Disch, H. Hungar, S. Jacobs, J. Pang, F. Pigorsch, C. Scholl, U. Waldmann, and B. Wirtz. Exact state set representations in the verification of linear hybrid systems with large discrete state space. In *5th International Symposium on Automated Technology for Verification and Analysis*, 2007, *LNCS 4762*, pp. 425–440. Springer.

[5] W. Damm, S. Disch, H. Hungar, J. Pang, F. Pigorsch, C. Scholl, U. Waldmann, and B. Wirtz. Automatic verification of hybrid systems with large discrete state space. In *4th Symposium on Automated Technology for Verification and Analysis*, 2006, *LNCS 4218*, pp. 276–291.

[6] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *18th Conference on Computer Aided Verification*, 2006, *LNCS 4144*, pp. 81–94. Springer.

[7] N. Eén and N. Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th InternationalConference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8,2003 Selected Revised Papers*, 2003, *LNCS 2919*, pp. 541–638. Springer.

[8] G. Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. In *8th Workshop on Hybrid Systems: Computation and Control*, 2005, *LNCS 3414*, pp. 258–273. Springer.

[9] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1(1–2):110–122, 1997.

[10] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton. FRAIGs: A unifying representation for logic synthesis and verification. Technical report, EECS Dept., UC Berkeley, 2005.

[11] F. Pigorsch, C. Scholl, and S. Disch. Advanced unbounded model checking by using AIGs, BDD sweeping and quantifier scheduling. In *6th Conference on Formal Methods in Computer Aided Design*, 2006, pp. 89–96. IEEE Press.