Science of Computer Programming 77 (2012) 1122–1150

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Exact and fully symbolic verification of linear hybrid automata with large discrete state spaces[☆,☆☆]

Werner Damm [a,b], Henning Dierks [c], Stefan Disch [d], Willem Hagemann [e], Florian Pigorsch [d], Christoph Scholl [d,*], Uwe Waldmann [e], Boris Wirtz [b]

[a] OFFIS e.V., Escherweg 2, 26121 Oldenburg, Germany
[b] Carl von Ossietzky Universität Oldenburg, Ammerländer Heerstraße 114-118, 26111 Oldenburg, Germany
[c] HAW Hamburg, Berliner Tor 5, 20099 Hamburg, Germany
[d] Albert Ludwigs Universität Freiburg, Georges Köhler Allee 51, 79110 Freiburg, Germany
[e] Max-Planck-Institut für Informatik, Campus E1.4, 66123 Saarbrücken, Germany

A R T I C L E   I N F O

Article history:
Available online 20 July 2011

Keywords:
Verification
Linear hybrid automata
Symbolic representations
Non-convex polyhedra
Redundancy elimination
Satisfiability Modulo Theories (SMT)

A B S T R A C T

We propose an improved symbolic algorithm for the verification of linear hybrid automata with large discrete state spaces (where an explicit representation of discrete states is difficult). Here both the discrete part and the continuous part of the hybrid state space are represented by one symbolic representation called LinAIGs. LinAIGs represent (possibly non-convex) polyhedra extended by Boolean variables. Key components of our method for state space traversal are *redundancy elimination* and *constraint minimization*: redundancy elimination eliminates so-called redundant linear constraints from LinAIG representations by a suitable exploitation of the capabilities of SMT (Satisfiability Modulo Theories) solvers. Constraint minimization optimizes polyhedra by exploiting the fact that states already reached in previous steps can be interpreted as “don’t cares” in the current step. Experimental results (including comparisons to the state-of-the-art model checkers PHAVer and RED) demonstrate the advantages of our approach.

1. Introduction

We target the verification of safety properties for embedded control applications in the transportation domain. A characteristic property of such applications is the presence of diagnostic and fault-tolerance measures integrated into the controller, which often drastically dominate the core control algorithms. Typically, the ratio of the total number of control states over the number of distinct modes governing the continuous evolution of system variables in closed-loop control grows exponentially when elaborating a nominal specification of controllers into a complete design model catering for non-nominal behavior [4]. Large discrete state spaces arise naturally in industrial hybrid systems, due to the need to represent discrete inputs (such as setting of control switches), counters, sanity checkbits, possibly multiple concurrent state machines,

☆ 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, http://www.avacs.org/).

☆☆ Parts of the results of this paper have been previously presented in Damm et al. (2006) [1], Damm et al. (2007) [2] and Scholl et al. (2009) [3].

* Corresponding author.
E-mail addresses: damm@offis.de (W. Damm), henning.dierks@haw-hamburg.de (H. Dierks), disch@informatik.uni-freiburg.de (S. Disch), hagemann@mpi-sb.mpg.de (W. Hagemann), pigorsch@informatik.uni-freiburg.de (F. Pigorsch), scholl@informatik.uni-freiburg.de (C. Scholl), uwe@mpi-inf.mpg.de (U. Waldmann), Boris.Wirtz@uni-oldenburg.de (B. Wirtz).

0167-6423/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2011.07.006

system-degradation modes, and finite switching variables, which typically jointly with properties of sensor values determine the selection of the relevant control laws. While the number of control laws is typically small (say, less than 50 per electronic control unit, respectively) even for industrial control units, the discrete state space is extremely large. As an example, an autopilot model recently analyzed [4] exhibited 250 discrete state variables — clearly emphasizing the need to employ symbolic methods for discrete state space representation.

Such applications are out of reach of existing hybrid verification tools such as CheckMate [5], HyTech [6], PHAVer [7], `d/dt` [8]. While their strength rests in being able to address complex dynamics, they do not scale in the discrete dimension, since modes – the only discrete states considered – are represented explicitly when performing reachability analysis. On the other hand, hardware verification tools such as SMV [9] and VIS [10] scale to extremely large discrete systems, but clearly fail to be applicable to systems with continuous dynamics. To prove the safety of such controllers, we must thus combine methods for analyzing the pure control part with state space exploration methods dealing with large discrete state spaces. Our paper closes this gap, in providing a fully symbolic approach for the verification of control loops with both continuous time and discrete time models of hybrid controllers. In that way we are able to verify models from different design phases: models used for control law design (typically carried out in Matlab/Simulink®[1]) and models used for automatic code generation (such as for embedded code generation with TargetLink®[2]).

A key characteristic of our approach is the use of *precise* abstractions: we use a novel fully symbolic predicate abstraction of hybrid state spaces in backward reachability analysis, and provide an increasingly powerful suite of optimization techniques ultimately allowing to mitigate both a blow up in the discrete state space *and* in the number of required predicates while *maintaining preciseness* of the abstraction. The approach thus allows both verification *and* falsification of safety requirements of hybrid controllers, and, additionally, avoids an outer counter-example guided abstraction refinement loop as required for imprecise abstractions.

Several insights have been instrumental in maintaining both compactness and exactness of symbolic representations of hybrid state spaces in backward model checking:

- We lift the SAT modulo theory approach [11,12] to symbolic model checking by providing an extension of And-Inverter-Graphs called LinAIGs. LinAIGs represent Boolean combinations of Boolean variables and dynamically computed linear predicates over real variables. We tightly integrate an SMT solver for identification of equivalent subgraphs into our symbolic model checking algorithm, both for obtaining functionally reduced LinAIGs and for fixed point detection. Based on representations of sets of states by LinAIGs we are able to perform an exact verification of safety requirements on (real-valued) plant and controller variables.
- We exploit the fact that in spite of the huge discrete state space the number of control laws tends to be rather small, so that we can use co-factoring on modes when symbolically evaluating the effect of flows using differential inclusions.
- We employ Loos–Weispfenning quantifier elimination [13] for backward reachability analysis along continuous evolutions. The Loos–Weispfenning method is especially suited for our LinAIG representations, since it reduces quantifier elimination to a series of substitutions which can be easily performed in the LinAIG environment without the need for conversions into disjunctive or conjunctive normal forms.
- We provide a new method for the detection and removal of redundant constraints in non-convex polyhedra to counteract the blow-up from such quantifier elimination steps. For the detection of redundant constraints we use incremental SMT solving [11,12]. Then redundant constraints are removed based on conflict clauses learnt from SMT calls and based on Craig interpolation [14–17].
- We use automatically derived Boolean invariants to further prune symbolic representations.
- We further optimize backward model checking by lifting the well-known "onion technique" from symbolic BDD based verification [18,19] to the LinAIG level. Here we use again Craig interpolation to derive an optimized representation between the "onion ring" – the set of states *newly* reached in the previous pre-image step – and the set of all states reached in the previous pre-image step.

Fully symbolic representations of state sets by BDDs have been originally introduced in the context of hardware verification by Burch et al. [20,9]. For real time system verification, BDDs in combination with clock difference diagrams (CDDs) have been used as symbolic representation, and were shown to be more efficient than difference bounded matrices (DBMs) [21]. Closer to our research is the work by Wang resulting in the tool RED [22]. Wang proposes HRDs (Hybrid Restriction Diagrams) as a BDD-like data-structure which is able to represent hybrid state spaces symbolically. To optimize paths in HRDs he proposes a normalization method called downward 2-redundant detection. Paths in HRDs correspond to convex polyhedra and downward 2-redundant detection is able to find only special types of redundant linear constraints in these convex polyhedra. Straightforward containment checking is a second method which detects whether a path in an HRD specifying a polyhedron is subsumed by another path [22]. Frehse [23] uses a complete method to eliminate redundant linear constraints from convex polyhedra. In contrast, our approach allows to handle *non-convex* polyhedra, and it is thus more general than the methods in [22,23], which only work for convex polyhedra. Other optimizations

[1] From The Mathworks Inc., www.mathworks.com.
[2] From dSpace GmbH, www.dSPACE.de.

which are addressed in [23,24] limit the degree of precision in coefficients of linear constraints or overapproximate convex polyhedra by dropping constraints. These methods can speed up fixpoint computations, but also can generate spurious counterexamples. In contrast, our approach maintains precise abstractions.

We demonstrate the relevance of our approach with benchmarks from the transportation and automation domain, such as an aircraft flap controller which is supposed to correct the pilot's commands on flap extension and retraction in order to avoid retractions or extensions which could cause de-stabilization of the aircraft. We provide an extensive comparison of our approach with the PHAVer tool [23] as a leading representative for engines geared towards analysis of complex dynamics, and show that for the targeted class of applications with a large number of discrete states our tool drastically outperforms PHAVer. Moreover, we also compare our approach to the RED tool [22] which provides symbolic representations of hybrid state spaces based on HRDs. The results clearly show the advantages of our approach which arise from a non-trivial interaction of a series of different optimization methods compressing symbolic state set representations.

The presented methods are orthogonal and may be combined with techniques to further enhance scalability and cover richer classes of dynamics, e.g., incorporating robustness [25,26] or slackness [27,28] in models allowing precise abstractions by finite grids under robustness or slackness assumptions, counterexample-guided abstraction refinement as in [29,30,4], and techniques such as hybridization [31] for approximate linearization of richer dynamics.

Our paper is structured as follows. In Section 2 we describe syntax and semantics of our version of linear hybrid automata, which differs from the classical definition in distinguishing between modes – governing the evolution of continuous variables – and discrete states (such as used for protocols, health monitoring, fault-tolerance, counters, etc.). We also present LinAIGs as a fully symbolic representation of the state space of our extended linear hybrid automata. Section 3 introduces the basic model checking algorithm. We exploit the structural characteristics of industrial applications, where the number of control laws is drastically smaller than the number of discrete states, by factoring our symbolic state space representation according to modes, performing exact continuous pre-image computations per mode, and then re-combining these to obtain the complete continuous pre-image. Both the continuous and discrete pre-image computations (along discrete transitions, creating substitution instances of linear constraints occurring in the image) typically entail a drastic blow up. A suite of optimization techniques counteracting this effect is presented in Sections 4–6. Each of these contains an elaboration of the approach and experimental data supporting the relevance of the presented heuristics. Section 4 discusses how to gradually increase the power of LinAIG compaction techniques at the price of increased algorithmic complexity from SAT based reasoning to the integration of information on linear constraints, and then to SMT based reasoning. Moreover, we show how automatically computed invariants on a Boolean abstraction of the model allow further optimizations in space and time complexity. Section 5 elaborates our key technique for detecting and eliminating redundant linear constraints in LinAIGs. In Section 6 we demonstrate how to find Craig interpolants between the "onion ring" and the set of all previously reached states for further reducing the number of linear constraints. Section 7 provides a comprehensive evaluation of all presented optimization techniques with a suite of benchmarks covering both discrete time and continuous time extended linear hybrid automata from aerospace and industrial control applications. This includes a comparison with PHAVer and RED. The conclusion summarizes the finding and gives suggestions for further research.

2. Preliminaries

2.1. System model

In this section we give a brief review of our system model. We consider linear hybrid automata extended with discrete states (LHA+Ds), that is, an extension of linear hybrid automata (LHA) [32] with a set of discrete variables. A similar definition (called "continuous-time hybrid systems") can be found in [2]. In the sequel, for brevity we will often just refer to "linear hybrid automata".

We assume disjoint sets of variables C, D, I and M. The elements of C are continuous variables. They are interpreted over the reals $\mathbb{R}$ and represent sensor values, actuator values, plant states, and other real-valued variables used for the modeling of control-laws and plant dynamics. The elements of D and I are discrete variables, where I will be used for inputs. For simplicity, we assume that they are of Boolean type and range over the domain $\mathbb{B} = \{0, 1\}$. Discrete variables represent states from state-machines, switches, counters, sanity bits of sensor values, etc.. In the same way we assume that modes are encoded by a set $\mathbf{M} \subseteq \{0, 1\}^l$ of Boolean vectors of some fixed length l, leading to a set M of l (Boolean) mode variables. The finite (and typically small) set of modes corresponds to discrete states of an LHA. A mode determines how the continuous valuation evolves over time as long as the system is in the given mode. Each mode defines an appropriate subset of possible slopes for each continuous variable; the restrictions to the slopes are expressed by linear inequalities on the derivatives of the continuous variables.

Valuations $\mathbf{d}$, $\mathbf{c}$, and $\mathbf{m}$ of the variables in D, C, and M, respectively, represent states $(\mathbf{d}, \mathbf{c}, \mathbf{m})$ of our automata. Sets of states can be represented symbolically using a suitable (quantifier-free) logic formula over $D \cup C \cup M$. Here we restrict terms over C to the class of linear terms of the form $\sum \alpha_i c_i + \alpha_0$ with $c_i \in C$ and rational constants α_i. Predicates are given by the set $\mathcal{L}(C)$ of linear constraints, they have the form $t \sim 0$, where $\sim \in \{=, <, \leq\}$ and t is a linear term. Finally, $\mathcal{P}(D, I, C, M)$ (resp. $\mathcal{P}(D, C, M)$) is the set of all Boolean combinations of variables from $D \cup I \cup M$ (resp. $D \cup M$) and linear constraints over C. As usual a formula $\xi(D, C, M) \in \mathcal{P}(D, C, M)$ represents the sets of states $(\mathbf{d}, \mathbf{c}, \mathbf{m})$ in which $\xi(\mathbf{d}, \mathbf{c}, \mathbf{m})$ is true.

The formal definition of a ct-LHA+D is as follows:

Definition 1 (*Syntax of a ct-LHA+D*). A *continuous-time linear hybrid automaton with discrete states* (ct-LHA+D) contains six components:

- $C = \{c_1, \dots, c_f\}$ is a finite set of continuous variables.
- $D = \{d_1, \dots, d_n\}$ is a finite set of discrete variables, $I = \{d_{n+1}, \dots, d_p\}$, $(p \geq n)$ is a finite (and possibly empty) set of discrete inputs.
- $M = \{m_1, \dots, m_l\}$ is a finite set of Boolean mode variables that are used to represent a finite set of modes $\mathbf{M} = \{\mathbf{m}_1, \dots, \mathbf{m}_k\} \subseteq \{0, 1\}^l$ using a suitable encoding. Each mode $\mathbf{m}_i$ is associated with a conjunction $W_i(\mathbf{v})$ of linear inequations $\bigwedge_j \mathbf{w}_{ij}\mathbf{v} \leq w_{ij}$ with $\mathbf{w}_{ij} \in \mathbb{R}^f$ and $w_{ij} \in \mathbb{R}$, where f is the number of continuous variables. The linear inequation system W_i describes the possible derivatives of the evolution of the continuous variables $(c_1, \dots, c_f)$, that is, during the mode $\mathbf{m}_i$ at any time t the derivative $\mathbf{v}(t) = (v_1(t), \dots, v_f(t))$ must satisfy W_i.
- GC is a global constraint given by a formula $\xi_{gc}(D, C, M) \in \mathcal{P}(D, C, M)$. The typical usage of GC is to specify lower and upper bounds for continuous variables in runs to be considered.
- $Init$ is a set of initial states, given by a formula $\xi_{init} \in \mathcal{P}(D, C, M)$.
- DT is the set of discrete transitions; each discrete transition is given as a guarded assignment ga_i $(i = 1, \dots, u$ and $u \geq 1)$ in the form

$$\begin{aligned} \xi_i \rightarrow \quad & (d_1, \dots, d_n) := (g_{i,1}, \dots, g_{i,n});\\ & (c_1, \dots, c_f) := (t_{i,1}, \dots, t_{i,f});\\ & (m_1, \dots, m_l) := \mathbf{m}_{j_i}, \end{aligned}$$

where $\xi_i \in \mathcal{P}(D, C, M)$, $g_{i,j} \in \mathcal{P}(D, I, C, M)$, $t_{i,j} \in \mathcal{L}(C)$ and $\mathbf{m}_{j_i} \in \mathbf{M}$.

The set DT of discrete transitions is partitioned into three disjoint sets DT^d, DT^{d2c}, and DT^{c2d}. DT^d contains the *purely discrete* transitions, DT^{d2c} contains the *discrete-to-continuous* transitions and DT^{c2d} the *continuous-to-discrete* transitions. We assume that input variables occur only in transitions from DT^{c2d}.

The guards of the transitions from DT^d (DT^{d2c}, DT^{c2d}) must be mutually exclusive, i.e., for guarded assignments ga_i, $ga_j \in DT^d$ ($\in DT^{d2c}$, $\in DT^{c2d}$) with $i \neq j$: $\xi_i \Rightarrow \neg\xi_j$. The guards of the transitions from DT^d and DT^{d2c} form complete case distinctions, i.e., the disjunction of all guards of transitions from DT^d (DT^{d2c}) is true. Every transition from DT^{c2d} is labeled as either urgent or non-urgent.

For each mode $\mathbf{m}_i$ its *boundary condition* β_i is given by the cofactor of the disjunction of all urgent discrete transition guards from DT^{c2d} w.r.t. $\mathbf{m}_i$.[3] For each valuation of variables in D, the boundary condition must be equivalent to a disjunction of non-strict ($\leq$) linear inequations. □

Compared to standard definitions of LHA [32], the transitions in DT^{c2d} correspond to "jumps" out of continuous flows. Since we allow a (possibly empty) series of purely discrete transitions after each jump, we additionally introduce discrete transitions in DT^d and discrete-to-continuous transitions in DT^{d2c} (which again lead to a continuous flow). Thus, the system evolves by alternating between continuous flows, in which time passes and only continuous variables are changed according to the differential equations associated with the currently active mode of the system, and sequences of discrete transitions, which – following the synchrony hypothesis [33] – happen in zero time.

The following example illustrates Definition 1. The model describes a simple flap controller. A pilot wants to move the flaps of his aircraft either to the flap angles *minangle* or to *maxangle* by selecting one of the flap positions *minpos* or *maxpos*. The flap controller reads the pilot's choice in a regular interval, and controls the actual flap movement by choosing one of the modes *extend*, *retract*, or *standstill*, in which the flap is either extended or retracted with a fixed rate *flaprate*, or is not moved at all. This is modeled by the ct-LHA+D $(C, D \cup I, M, GC, Init, DT^{c2d} \cup DT^{d2c})$:

- The set $C = \{clock, flapangle\}$ contains a clock variable, which controls the activation of the controller, and the current flap angle.
- The discrete variables and inputs are defined by the sets $D = \{desired\_flappos\}$ and $I = \{pilot\_selection\}$. The input *pilot_selection* describes the choice of the pilot, which is saved by the controller to the variable *desired_flappos*.
- There are three mode variables $M = \{extend, retract, standstill\}$, which (using one-hot encoding) span the three modes $\mathbf{M} = \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}$.
  The associated linear inequations are given by

$$\begin{aligned} W_{(1,0,0)} &= (deriv\_clock = 1 \wedge deriv\_flapangle = flaprate)\\ W_{(0,1,0)} &= (deriv\_clock = 1 \wedge deriv\_flapangle = -flaprate)\\ W_{(0,0,1)} &= (deriv\_clock = 1 \wedge deriv\_flapangle = 0) \end{aligned}$$

where the variables *deriv_clock* and *deriv_flapangle* represent the time derivatives of *clock* and *flapangle*, respectively, and $x = c$ is an abbreviation for $x \leq c \wedge x \geq c$.

[3] The cofactor is the partial evaluation of the disjunction w.r.t. $(m_1, \dots, m_l) = \mathbf{m}_i$. It does not depend on M anymore.

- The clock variable is reset if it reaches a value *maxclock*, and the flap can only move between *minangle* and *maxangle*, so $GC = (0 \leq clock \leq maxclock \wedge minangle \leq flapangle \leq maxangle)$.
- Initially, the clock is 0 and the flap is within its valid range: $Init = (clock = 0 \wedge minangle \leq flapangle \leq maxangle)$.
- Finally, the discrete transitions are given by the sets D^{c2d} and D^{d2c} (for this simple model, purely discrete transitions are not necessary). We use the convention that trivial assignments of the form $x := x$ are left out in the assignment part.

  The controller is activated if the current pilot's decision needs to be read, or if the flaps have reached an extremal position. Thus D^{c2d} is defined by the urgent transitions

$$\begin{array}{lcl} clock \geq clock\_max & \Longrightarrow & desired\_flappos := pilot\_selection; \\ & & clock := 0; \\ clock < clock\_max \wedge extend \wedge flapangle \geq maxangle & \Longrightarrow & ; \\ clock < clock\_max \wedge retract \wedge flapangle \leq minangle & \Longrightarrow & ; \end{array}$$

The boundary condition of a mode is computed by the cofactor of the disjunction of all urgent discrete transition guards from DT^{c2d} w.r.t. this mode. Thus, the boundary conditions are (equivalent to) $clock \geq clock\_max \vee flapangle \geq maxangle$ for mode (1, 0, 0), $clock \geq clock\_max \vee flapangle \leq minangle$ for mode (0, 1, 0), and $clock \geq clock\_max$ for mode (0, 0, 1).

In the DT^{d2c} transitions the next mode is selected:

$$\begin{array}{lcl} desired\_flappos = maxpos \wedge flapangle < maxangle & \Longrightarrow & (extend, retract, standstill) := (1, 0, 0); \\ desired\_flappos = maxpos \wedge flapangle \geq maxangle & \Longrightarrow & (extend, retract, standstill) := (0, 0, 1); \\ desired\_flappos = minpos \wedge flapangle > minangle & \Longrightarrow & (extend, retract, standstill) := (0, 1, 0); \\ desired\_flappos = minpos \wedge flapangle \leq minangle & \Longrightarrow & (extend, retract, standstill) := (0, 0, 1); \end{array}$$

The semantics of a ct-LHA+D is defined by specifying its trajectories:

Definition 2 (*Semantics of a ct-LHA+D*). • A *state* of a ct-LHA+D is a valuation $s = (\mathbf{d}, \mathbf{c}, \mathbf{m})$ of D, C and M.

- There is a continuous transition from a state $s^i = (\mathbf{d}^i, \mathbf{c}^i, \mathbf{m}^i)$ to a state $s^{i+1} = (\mathbf{d}^{i+1}, \mathbf{c}^{i+1}, \mathbf{m}^{i+1})$ if there exists a $\lambda \in \mathbb{R}_{\geq 0}$ and a function $\mathbf{v}$ from $\mathbb{R}$ to $\mathbb{R}^f$ such that the following conditions are satisfied:
  - $W(\mathbf{v}(t))$ holds for all t with $0 \leq t \leq \lambda$, where W is the linear inequation system associated with $\mathbf{m}^i$;
  - $(\mathbf{d}^{i+1}, \mathbf{c}^{i+1}, \mathbf{m}^{i+1}) = (\mathbf{d}^i, \mathbf{c}^i + \int_0^\lambda \mathbf{v}(t)\,dt, \mathbf{m}^i)$;
  - For every $0 \leq \lambda' < \lambda$, the state $(\mathbf{d}^i, \mathbf{c}^i + \int_0^{\lambda'} \mathbf{v}(t)\,dt, \mathbf{m}^i)$ satisfies GC and does not satisfy β_i (i.e., neither we violate the global constraints nor hit an urgent discrete transition guard along the way).

  We also say that s^{i+1} is a λ-time successor of s^i, written as $s^i \rightarrow^\lambda s^{i+1}$.
- A *trajectory* of a ct-LHA+D is a finite sequence of states $(s^i = (\mathbf{d}^i, \mathbf{c}^i, \mathbf{m}^i))_{0 \leq i \leq n}$ or an infinite sequence of states $(s^i = (\mathbf{d}^i, \mathbf{c}^i, \mathbf{m}^i))_{i \geq 0}$ where all states satisfy GC and one of the following conditions holds for each $i \geq 0$ (or $0 \leq i \leq n-1$):
  1. There is a continuous-to-discrete transition $ga_i \in DT^{c2d}$ from $s^i = (\mathbf{d}^i, \mathbf{c}^i, \mathbf{m}^i)$ to $s^{i+1} = (\mathbf{d}^{i+1}, \mathbf{c}^{i+1}, \mathbf{m}^{i+1})$, i.e., the guard ξ_i is true in $(\mathbf{d}^i, \mathbf{c}^i, \mathbf{m}^i)$ and there is a valuation $\mathbf{i}$ of the input variables such that the values in $(\mathbf{d}^{i+1}, \mathbf{c}^{i+1}, \mathbf{m}^{i+1})$ result from executing the assignments in ga_i. If $i > 0$, then the previous transition from s^{i-1} to s^i is a continuous transition.
  2. There is a discrete transition $ga_i \in DT^d$ from $s^i = (\mathbf{d}^i, \mathbf{c}^i, \mathbf{m}^i)$ to $s^{i+1} = (\mathbf{d}^{i+1}, \mathbf{c}^{i+1}, \mathbf{m}^{i+1})$, i.e., the guard ξ_i is true in $(\mathbf{d}^i, \mathbf{c}^i, \mathbf{m}^i)$ and the values in $(\mathbf{d}^{i+1}, \mathbf{c}^{i+1}, \mathbf{m}^{i+1})$ result from executing the assignments in ga_i. If $i > 0$, then the previous transition ga_{i-1} from s^{i-1} to s^i is a discrete transition in DT^d or a continuous-to-discrete transition in DT^{c2d}.
  3. There is a discrete-to-continuous transition $ga_i \in DT^{d2c}$ from $s^i = (\mathbf{d}^i, \mathbf{c}^i, \mathbf{m}^i)$ to $s^{i+1} = (\mathbf{d}^{i+1}, \mathbf{c}^{i+1}, \mathbf{m}^{i+1})$ (defined in the same way as given above). If $i > 0$, then the previous transition ga_{i-1} is in DT^d or in DT^{c2d}.
  4. There is a continuous transition from $s^i = (\mathbf{d}^i, \mathbf{c}^i, \mathbf{m}^i)$ to $s^{i+1} = (\mathbf{d}^{i+1}, \mathbf{c}^{i+1}, \mathbf{m}^{i+1})$ (i.e., $s^i \rightarrow^\lambda s^{i+1}$ for some $\lambda \in \mathbb{R}_{\geq 0}$). If $i > 0$, then the previous transition ga_{i-1} from s^{i-1} to s^i is in DT^{d2c}.
- A state $s' = (\mathbf{d}', \mathbf{c}', \mathbf{m}')$ is reachable from the state $s = (\mathbf{d}, \mathbf{c}, \mathbf{m})$, if there is a trajectory that starts from $s = (\mathbf{d}, \mathbf{c}, \mathbf{m})$ and ends in $s' = (\mathbf{d}', \mathbf{c}', \mathbf{m}')$. The reachable state set of a ct-LHA+D contains all states that are reachable from the initial states which satisfy ξ_{init}.

By the definition given above, a trajectory always contains subsequences of a continuous transition, followed by a continuous-to-discrete transition, followed by a (potentially empty) series of discrete transitions, followed by a discrete-to-continuous transition and so on. Trajectories may start with an arbitrary type of transition. Time passes only during continuous flows and continuous flows only change continuous variables in such a way that the derivative of the evolution satisfies the respective inequation system W. Discrete transitions happen in zero time, update both discrete and continuous variables, and finally select the next active mode. Transitions from DT^{c2d} are usually urgent in our applications, that is, they fire once they become enabled. Non-urgent transitions are also permitted though.

Our approach can be applied mutatis mutandis to discrete-time linear hybrid automata extended with discrete states (dt-LHA+D) where the sets DT^{d2c} and DT^{c2d} are empty and trajectories consist only of discrete transitions. In this case, input variables may occur in DT^d. Most of our considerations hold for both variants, and in this case, we will just refer to LHA+Ds.

In summary, our models are closed-loop models without continuous input variables, combining controller and its controlled plant, hence sensors and actuators are internal continuous variables. Interactions of the environment are only possible through discrete input variables, allowing, e.g., to select set-points, and to react to protocol messages.

Non-deterministic choices are also modeled using discrete input variables. Non-determinism in plant dynamics is modeled by choosing appropriate bounds on the derivatives.

A LHA+D may be accompanied by an *invariant Inv* given by a formula $\xi_{inv}(D, C, M) \in \mathcal{P}(D, C, M)$. Invariants characterize properties of the LHA+D that have already been proved – for instance by a separate run of the model checker – and can now be used in order to accelerate the current run of the model checker. They are thus a tool to modularize larger verification tasks. We emphasize the difference between invariants and global constraints. A global constraint *GC defines* that trajectories violating *GC* are irrelevant and should be ignored, for instance because they cannot correspond to actual behavior of the plant, or because some time bound is reached. An invariant *Inv postulates* that all trajectories starting from *Init* have the property *Inv* — if the invariant is unsound, that is, if there exists a trajectory violating *Inv*, the result of the model checker is unspecified.

2.2. Representation of state sets

Our goal is to check whether all states of a LHA+D reachable from *Init* are within a given set of (safe) states *Safe*. To establish this, a backward fixpoint computation is performed. We start with the set $\neg$*Safe* and repeatedly compute the pre-image until a fixpoint is reached or some initial state is reached during the backward analysis. In the latter case, a state outside of *Safe* is reachable.

For this fixpoint computation we need a compact representation of sets of states of LHA+Ds. Sets of states of LHA+Ds are represented by formulas from $\mathcal{P}(D, C, M)$ (which are Boolean combinations over D, M and linear constraints $\mathcal{L}(C)$) and for efficiently implementing such formulas we make use of a specific data structure called LinAIGs [1,2]. By using LinAIGs both the discrete part and the continuous part of the hybrid state space are represented by one symbolic representation.

Using efficient methods for keeping LinAIGs as compact as possible is a key point for our approach. This goal is achieved by a rather complex interaction of various methods. In this section we give a brief overview of the basic components of LinAIGs. In sections 4.1, 4.2, 5 and 6 we describe optimizations to increase the efficiency of the data structure.

Boolean part. The component of LinAIGs representing Boolean formulas consists of a variant of AIGs, the so-called Functionally Reduced AND-Inverter Graphs (FRAIGs) [34,35]. AIGs enjoy a widespread application in combinational equivalence checking and Bounded Model Checking (BMC). They are basically Boolean circuits consisting only of AND gates and inverters. In contrast to BDDs, they are not a canonical representation for Boolean functions, but they are “semi-canonical” in the sense that every node in the FRAIG represents a unique Boolean function. To achieve this goal several techniques like structural hashing, simulation[4] and SAT solving are used:

First, local transformation rules are used for node minimization. For instance, we apply structural hashing for identifying isomorphic AND nodes which have the same pairs of inputs.

Moreover, we maintain the so-called “functional reduction property”: Each node in the FRAIG represents a unique Boolean function. Using a SAT solver we check for equivalent nodes while constructing a FRAIG and we merge equivalent nodes immediately.[5]

Of course, checking each possible pair of nodes would be quite inefficient. However, *simulation* using test vectors of Boolean values restricts the number of candidates for SAT checks to a great extent. If for a given pair of nodes simulation is already able to prove non-equivalence (i.e., the simulated values are different for at least one test vector), the more time consuming SAT checks are not needed. The simulation vectors are initially random, but they are updated using feedback from satisfied SAT instances (i.e., from proofs of non-equivalence).

For the pure Boolean case, enhanced with other techniques such as quantifier scheduling, node selection heuristics and BDD sweeping, FRAIGs proved to be a promising alternative to BDDs in the context of symbolic model checking, replacing BDDs as a compact representation of large discrete state spaces [35]. Similar techniques have been successfully applied for satisfiability checking of quantified Boolean formulas (QSAT), too [36,37].

Continuous part. In LinAIGs, the FRAIG structure is enriched by linear constraints. We use a set of new (Boolean) *constraint variables* Q as additional inputs to the FRAIG. Every linear constraint $\ell_i \in \mathcal{L}(C)$ is connected to the Boolean part by some $q_{\ell_i} \in Q$. The constraints are of the form $\sum_{i=1}^{n} \alpha_i c_i + \alpha_0 \sim 0$ with rational constants α_j, real variables c_i, and $\sim \in \{=, <, \leq\}$. The structure of LinAIGs is illustrated in Fig. 1.

During our model checking algorithm we avoid introducing linear constraints which are equivalent to existing constraints. The restriction to *linear* constraints makes this task simple, since it reduces to the application of (straightforward) normalization rules.

2.3. Quantifier elimination for linear real arithmetic

For the computation of continuous steps based on state set representations given by LinAIGs (see Section 3.1.2) we need quantifier elimination for linear real arithmetic. For this we use the Loos–Weispfenning test point method [13,38], which replaces existentially quantified formulas by finite disjunctions using sets of symbolic substitutions.

4 In this context, simulation means the evaluation of FRAIG nodes for a set of given inputs; it does not refer to the simulation of controller models.

5 In the same way we prevent the situation that one node in a FRAIG represents the complement of the Boolean function represented by another node in the same FRAIG.

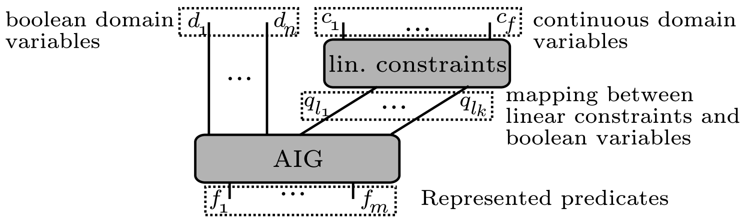


Fig. 1. Structure of LinAIGs.

The Loos–Weispfenning method is based on the following observation: Assume that a formula $\psi(x, \vec{y})$ is written as a positive boolean combination of linear constraints $x \sim_i t_i(\vec{y})$ and $0 \sim'_j t'_j(\vec{y})$, where $\sim_i, \sim'_j \in \{=, \neq, <, \leq, >, \geq\}$. Let us keep the values of $\vec{y}$ fixed for a moment. If the set of all x such that $\psi(x, \vec{y})$ holds is non-empty, then it can be written as a finite union of (possibly unbounded) intervals, whose boundaries are among the $t_i(\vec{y})$. To check whether $\exists x.\ \psi(x, \vec{y})$ holds, it is therefore sufficient to test $\psi(x, \vec{y})$ for either all upper or all lower boundaries of these intervals. The test values may include $+\infty$, $-\infty$, or a positive infinitesimal ε, but these can easily be eliminated from the substituted formula. For instance, if x is substituted by $t_j(\vec{y}) - \varepsilon$, then both the linear constraints $x \leq t_i(\vec{y})$ and $x < t_i(\vec{y})$ are turned into $t_j(\vec{y}) \leq t_i(\vec{y})$, and both $x \geq t_i(\vec{y})$ and $x > t_i(\vec{y})$ are turned into $t_j(\vec{y}) > t_i(\vec{y})$.

There are two possible sets of test points, depending on whether we consider upper or lower boundaries:

$$TP_1 = \{+\infty\} \cup \{\, t_i(\vec{y}) \mid \sim_i \in \{=, \leq\}\,\} \cup \{\, t_i(\vec{y}) - \varepsilon \mid \sim_i \in \{\neq, <\}\,\}$$
$$TP_2 = \{-\infty\} \cup \{\, t_i(\vec{y}) \mid \sim_i \in \{=, \geq\}\,\} \cup \{\, t_i(\vec{y}) + \varepsilon \mid \sim_i \in \{\neq, >\}\,\}.$$

Let TP be the smaller one of the two sets and let T be the set of all symbolic substitutions x/t for $t \in TP$. Then the formula $\exists x.\ \psi(x, \vec{y})$ can be replaced by an equivalent finite disjunction $\bigvee_{\sigma \in T} \psi(x, \vec{y})\sigma$. The size of TP is in general linear in the size of ψ, so the size of the resulting formula is quadratic in the size of ψ. This is independent of the Boolean structure of ψ — conversion to DNF is not required. On the other hand, if ψ is a disjunction $\bigvee \psi_i$, then the test point method can also be applied to each of the formulas ψ_i individually, leading to a smaller number of test points. Moreover, when the test point method transforms each ψ_i into a finite disjunction $\bigvee \psi_i^j$, then each ψ_i^j contains at most as many linear constraints as the original ψ_i, and only the length of the outer disjunction increases.

The Loos–Weispfenning method can easily be generalized to formulas that involve both linear constraints and Boolean variables. It can therefore work directly on the internal formula representation of LinAIGs — in contrast to the classic Fourier–Motzkin algorithm, there is no need for a separation of Boolean and numerical parts or for a costly CNF or DNF conversion before eliminating quantifiers. Moreover, the resulting formulas preserve most of the Boolean structure of the original ones: the method behaves largely like a generalized substitution.

3. Model checking of linear hybrid automata

3.1. Step computation

As already mentioned above, we check safety properties by using a backward fixpoint algorithm which computes the set of states from which states in $\neg Safe$ can be reached (and we check whether one of these states is contained in *Init*). For the fixpoint computation we need pre-image computations to evaluate both discrete and continuous transitions.[6]

3.1.1. Discrete steps

The step computation is similar for discrete, continuous-to-discrete and discrete-to-continuous transitions. For that reason we consider here only purely discrete transitions which are given by guarded assignments ga_i ($i = 1, \ldots, v$ and $v \geq 1$) and we just give a brief review of the step computation [1].

We differentiate between the discrete variables in $D \cup M$ and the constraint variables Q.

In a discrete transition the term $g_{i,j}(D, I, C, M)$ is assigned to the discrete variable $d_j \in D$ under condition $\xi_i(D, C, M)$. This translates to the following (logical) function, assigning a Boolean formula over $D \cup I \cup C \cup M$ to each $d_j \in D$:

$$pre(d_j) = \bigwedge_{i=0}^{v} \big(\xi_i(D, C, M) \ \Rightarrow\ g_{i,j}(D, I, C, M)\big)\,.$$

6 We have chosen the backward direction, because for discrete transitions the pre-image can be expressed essentially by a substitution (see Hoare's program logic [39]). By contrast, forward model checking makes use of the discrete image, and computing the latter with a LinAIG representation requires quantifier elimination.

Analogously, *pre* is defined for the variables in M. For the continuous part, $q_\ell \in Q$ is updated by

$$pre(q_\ell) = \bigwedge_{i=0}^{v} \big(\xi_i(D, C, M) \;\Rightarrow\; q_{\ell[c_1,\ldots,c_f/t_{i,1}(C),\ldots,t_{i,f}(C)]}\big)$$

that is, q_ℓ is replaced by a Boolean combination of Boolean and constraint variables, where $q_{\ell[c_1,\ldots,c_f/t_{i,1}(C),\ldots,t_{i,f}(C)]}$ is a (possibly new) constraint variable representing the linear constraint that results from ℓ by replacing every c_j by the term $t_{i,j}(C)$.

Finally, the set of states which can be reached by a backward step from a set of states described by ϕ is computed by substituting in parallel the pre-images for the respective variables.

$$Pre^d(\phi) = \phi[d/pre(d), d \in D \cup M]\,[q/pre(q), q \in Q].$$

Note that the correctness of the discrete step computation relies on the fact that the guards in the guarded assignments form a complete and disjoint case distinction (see Definition 1).

3.1.2. Continuous steps

In our system model, the time steps only concern the evolutions of continuous variables and leave the discrete part unchanged. For the symbolic treatment of continuous pre-image computations we exploit the fact that the number of modes (i.e., of distinct control laws) for a given control applications is drastically smaller than the number of discrete states and typically well below 100. This allows to factor our symbolic representation according to modes, and thus to perform a precise analysis of continuous pre-image computations for each mode individually. For each mode, the continuous unsafe pre-image Pre^c can be expressed as a formula with two quantified real variables (time) and one quantified function from $\mathbb{R}$ to $\mathbb{R}^f$ denoting the derivative of the continuous evolution at some time. We will show how to eliminate these quantifiers to arrive at a formula which can again be represented by a LinAIG.

Let $\phi(D, C, M)$ be a representation of a state set and let $\phi(D, Q, M)$ be its Boolean abstraction replacing linear constraints $\ell_i \in \mathcal{L}(C)$ by $q_{\ell_i} \in Q$. Each valuation $\mathbf{m}_i$ of the mode variables in M encodes a concrete mode with a boundary condition β_i and an inequation system $W_i(\mathbf{v}) \Leftrightarrow \bigwedge_j \mathbf{w}_{ij}\mathbf{v} \le w_{ij}$ characterizing the possible derivatives $\mathbf{v}$ of the continuous evolution. Let ϕ_i be the cofactor of ϕ w.r.t. mode $\mathbf{m}_i$. Thus we have $\phi \Leftrightarrow \bigvee_{i=1}^{k} \phi_i \wedge (m_1, \ldots, m_l) = \mathbf{m}_i$, where each ϕ_i is a Boolean formula over D and Q.[7] For each mode $\mathbf{m_i}$, we must now determine the set of all valuations for which there exists some (arbitrarily long) evolution that has a derivative satisfying the inequation system W_i and leads to a valuation satisfying ϕ_i and does not meet any point that satisfies the boundary condition β_i or violates the global constraints GC before.[8] We denote this set by $Pre^c(\phi_i, W_i, \beta_i)$. Logically, it can be described by the formula

$$\begin{aligned}
&\exists\lambda.\, \lambda \ge 0\\
&\quad \wedge \exists\mathbf{v}.\, (\forall t.\, W_i(\mathbf{v}(t)))\\
&\qquad\qquad \wedge \phi_i(\mathbf{c} + \textstyle\int_0^\lambda \mathbf{v}(t)\,\mathrm{d}t)\\
&\qquad\qquad \wedge GC(\mathbf{c} + \textstyle\int_0^\lambda \mathbf{v}(t)\,\mathrm{d}t)\\
&\qquad\qquad \wedge \forall\lambda'.\,(\lambda' < \lambda \wedge 0 \le \lambda') \to (\neg\beta_i(\mathbf{c} + \textstyle\int_0^{\lambda'} \mathbf{v}(t)\,\mathrm{d}t) \wedge GC(\mathbf{c} + \textstyle\int_0^{\lambda'} \mathbf{v}(t)\,\mathrm{d}t)).
\end{aligned}$$

Under the assumption that the set described by GC is convex, W_i is a conjunction of linear inequations, and β_i is equivalent to a disjunction of linear inequations for any valuation of the variables in D, we may replace without loss of generality the function $\mathbf{v}(t)$ by a constant $\mathbf{v}$; moreover one can replace the universal quantification over λ' by two test points, namely 0 and $\lambda - \varepsilon$, where the formula with ε represents the limit for $\varepsilon \to +0$. Using the fact that we are only interested in states satisfying GC, the formula can be simplified (modulo GC) to

$$\phi_i(\mathbf{c}) \vee \exists\lambda.\, \lambda > 0 \wedge \exists\mathbf{v}.\, W_i(\mathbf{v}) \wedge \phi_i(\mathbf{c} + \lambda\mathbf{v}) \wedge GC(\mathbf{c} + \lambda\mathbf{v}) \wedge \neg\beta_i(\mathbf{c}) \wedge \neg\beta_i(\mathbf{c} + (\lambda - \varepsilon)\mathbf{v}).$$

If β_i is a disjunction of linear constraints, one can show that $\neg\beta_i(\mathbf{c}) \wedge \neg\beta_i(\mathbf{c} + (\lambda - \varepsilon)\mathbf{v})$ is equivalent to $\neg\beta_i(\mathbf{c}) \wedge \neg\beta_i'(\mathbf{c} + \lambda\mathbf{v})$ where β_i' is the disjunction of linear constraints one obtains from β_i by replacing all non-strict inequalities ($\le$) by strict ones ($<$), or in other words, β_i without its boundary. To get rid of the non-linearity of quantified variables, we use the trick of Alur et al. [40] and replace the product $\lambda\mathbf{v}$ by a new vector $\mathbf{u}$. We obtain:

$$\phi_i(\mathbf{c}) \vee \exists\lambda.\, \lambda > 0 \wedge \exists\mathbf{u}.\, W_i'(\mathbf{u}, \lambda) \wedge \phi_i(\mathbf{c} + \mathbf{u}) \wedge GC(\mathbf{c} + \mathbf{u}) \wedge \beta_i(\mathbf{c}) \wedge \beta_i'(\mathbf{c} + \mathbf{u})$$

where the inequation system $W_i'(\mathbf{u}, \lambda)$ is given by $\bigwedge_j \mathbf{w}_{ij}\mathbf{u} \le w_{ij}\lambda$.

It remains to convert this formula over $\lambda, \mathbf{u} = (u_1, \ldots, u_f)$, C, and D into an equivalent formula over the original variables in C and D. This amounts to variable elimination for linear real arithmetic (with variables $u_1, \ldots, u_f$ and λ) and may be performed by the Loos–Weispfenning test point method already described in Section 2.3.

7 The variables in D are assumed to remain constant during mode $\mathbf{m}_i$, so Boolean expressions over D behave like propositional variables. For simplicity, we will ignore them in the rest of this section.

8 Recall that β_i is the cofactor of the disjunction of all *urgent* discrete transition guards w.r.t. $\mathbf{m}_i$; non-urgent transitions are ignored at this point.

3.2. Model checking algorithm

Using the pre-image computations described above, we can now define the model checking algorithm. Starting from a representation of the unsafe states, the backward reachability analysis alternates between series of discrete steps and continuous flows, where the latter are surrounded by continuous-to-discrete (c2d) and discrete-to-continuous (d2c) steps. Since input variables are read during c2d steps, these variables are existentially quantified in the results of c2d pre-image computations.[9] The iteration is performed until a global fixpoint is finally reached or until an initial state is reached.

```
begin
    $\phi_0^{d2c} := \neg safe$;
    $i := 0$;
    repeat
        $i := i + 1$;
        // Discrete fixed point iteration:
        $j := 0$;
        $\phi_0^d := \phi_{i-1}^{d2c}$;
        repeat
            $j := j + 1$;
            $\phi_j^d := (Pre^d(\phi_{j-1}^d \wedge GC)) \vee \phi_{i-1}^{d2c}$;
        until $GC \wedge \phi_j^d \wedge \neg\phi_{j-1}^d = 0$;
        $\phi_i^{dfp} := \phi_j^d$;
        // Evaluate c2d transitions:
        $\phi_i^{c2d} := \left(\exists d_{n+1}, \ldots d_p(Pre^{c2d}(\phi_i^{dfp} \wedge GC)) \wedge \bigvee_{j=1}^k \beta_j\right) \vee \neg safe$;
        // Evaluate continuous flow:
        $\phi_i^{flow} := \bigvee_{h=1}^k Pre^c(\phi_i^{c2d}|_{\overrightarrow{m}=\mathbf{m}_h}, W_i, \beta_i) \wedge (\overrightarrow{m} = \mathbf{m}_h)$;
        // Evaluate d2c transitions:
        $\phi_i^{d2c} := (Pre^{d2c}(\phi_i^{flow} \wedge GC)) \vee \neg safe$;
    until $GC \wedge \phi_i^{d2c} \wedge \neg\phi_{i-1}^{d2c} = 0$;
    if $GC \wedge (\phi_i^{dfp} \vee \phi_i^{c2d} \vee \phi_i^{flow} \vee \phi_i^{d2c}) \wedge init \neq 0$ then return $false$;
    return $true$;
```

4. Basic optimizations of LinAIG representations

4.1. LinAIG optimizations using information on linear constraints

In this section we describe optimizations to our LinAIG data structure which go beyond a separate treatment of the Boolean part and the continuous part (linear constraints). Of course, just keeping the Boolean part and the continuous part (linear constraints) of LinAIGs separate would lead to a loss of information. Since we would forget correlations between linear constraints, we would give up much of the potential for optimizing the representations. Moreover, we need this information when we have to check whether two sets of states are equivalent during the fixpoint check of the model checking procedure. As a simple example consider the two predicates $\phi_1 = (c_1 < 5)$ and $\phi_2 = (c_1 < 10) \wedge (c_1 < 5)$. If $c_1 < 5$ is represented by the Boolean constraint variable q_{ℓ_1} and $c < 10$ by variable q_{ℓ_2}, then the corresponding Boolean formulas q_{ℓ_1} and $q_{\ell_1} \wedge q_{\ell_2}$ are not equivalent, whereas ϕ_1 and ϕ_2 are certainly equivalent. Both as a means for further compaction of our representations and as a means for detecting fixpoints we need methods for transferring knowledge from the continuous part to the Boolean part. In the example above this may be the information that $q_{\ell_1} = 1$ and $q_{\ell_2} = 0$ can not be true at the same time or that ϕ_1 and ϕ_2 are equivalent when replacing boolean variables by the corresponding linear constraints.

4.1.1. Implication-based compaction

As a first method to transfer information from the continuous to the Boolean part we consider dependencies between linear constraints that are easy to detect a priori.

Computing implications between linear constraints. It is not known initially which dependencies are actually needed in the rest of the computation; for this reason we restrict to two simple cases: First, we compute unconditional implications

9 For discrete-time LHA+Ds, only discrete steps are performed and the existential quantification over input variables happens there.

between linear constraints $\alpha_1 c_1 + \cdots + \alpha_n c_n + \alpha_0 \leq 0$ and $\alpha_1 c_1 + \cdots + \alpha_n c_n + \alpha'_0 \leq 0$, where $\alpha_0 > \alpha'_0$ (and analogously implications involving negations of linear constraints). Second, we use a sound but incomplete method to detect implications which follow from global constraints. Here we restrict ourselves to global constraints of the form $l_i \leq c_i \leq u_i$ for the continuous variables. If $(\alpha_1 - \alpha'_1)b_1 + \cdots + (\alpha_n - \alpha'_n)b_n + \alpha_0 - \alpha'_0 \geq 0$, where $b_i = l_i$ if $\alpha'_i < \alpha_i$ and $b_i = u_i$ otherwise, then the linear constraint $\alpha'_1 c_1 + \cdots + \alpha'_n c_n + \alpha'_0 \leq 0$ follows from $\alpha_1 c_1 + \cdots + \alpha_n c_n + \alpha_0 \leq 0$ and the global lower and upper bounds $l_i \leq c_i \leq u_i$.

Using implications between linear constraints. Implications between linear constraints are used for compaction of the FRAIG part as follows:

Suppose we have found a pair of linear constraints ℓ_1 and ℓ_2 with $\ell_1 \Rightarrow \ell_2$, where ℓ_1 and ℓ_2 are represented by the constraint variables q_{ℓ_1} and q_{ℓ_2} in the Boolean part. Then we know that the combination of values $q_{\ell_1} = 1$ and $q_{\ell_2} = 0$ is inconsistent w.r.t. the continuous part, i.e., it will never be applied to inputs q_{ℓ_1} and q_{ℓ_2} of the Boolean part. We transfer this knowledge to the Boolean part by a modified behavior of the FRAIG package: first we adjust the simulation test vectors (over Boolean variables and constraint variables q_{ℓ_i}), such that they become consistent with the found implications (potentially leading to the fact that proofs of non-equivalence by simulation will not hold any longer for certain pairs of nodes). This is achieved by maintaining a directed graph $G_{impl} = (V, E)$ with $V = \{ q_{\ell_i} \mid \ell_i \in \mathcal{L}(C) \} \cup \{ \overline{q_{\ell_i}} \mid \ell_i \in \mathcal{L}(C) \}$ and $E = \{ (q_{\ell_{i_1}}, q_{\ell_{i_2}}) \mid \ell_{i_1} \Rightarrow \ell_{i_2},\ \ell_{i_1}, \ell_{i_2} \in \mathcal{L}(C) \} \cup \{ (\overline{q_{\ell_{i_2}}}, \overline{q_{\ell_{i_1}}}) \mid \ell_{i_1} \Rightarrow \ell_{i_2},\ \ell_{i_1}, \ell_{i_2} \in \mathcal{L}(C) \}$. The resulting graph is skew-symmetric[10] and acyclic, since a cycle in the graph would mean that certain linear constraints are equivalent (or antivalent), which is prevented by normalization during insertion of linear constraints. Thus we are able to adjust test vectors following a topological order in this graph G_{impl}: After adding newly found implications between linear constraints to G, we traverse the graph in topological order and at each vertex q_{ℓ_i} we modify the test vector values of all successors q_{ℓ_j} of q_{ℓ_i} such that they become consistent with the implication $\ell_i \Rightarrow \ell_j$.

Secondly, we make use of implications $\ell_{i_1} \Rightarrow \ell_{i_2}$ found between linear constraints during SAT checking. We introduce the implication $q_{\ell_{i_1}} \Rightarrow q_{\ell_{i_2}}$ as an additional binary clause in every SAT problem checking equivalence of two nodes depending on $q_{\ell_{i_1}}$ and $q_{\ell_{i_2}}$. In that way, non-equivalences of LinAIG nodes which are only caused by differences w.r.t. inconsistent input value combinations with $q_{\ell_{i_1}} = 1$ and $q_{\ell_{i_2}} = 0$ will be turned into equivalences, removing redundant nodes in the LinAIG.

Note that it is not necessary to add all existing implications to the SAT problems: Implications which follow from transitivity may be omitted. In fact, experimental results show that it pays off to add only a minimal number of clauses for implications. These clauses result from the transitive reduction of the acyclic directed graph G_{impl} [41].[11]

4.1.2. *Using a decision procedure for deciding equivalence*

In addition to the eager check for implications between linear constraints above, we use an SMT (SAT modulo theories) solver [11,12] as a decision procedure for the equivalence of nodes in LinAIGs (representing boolean combinations of linear constraints and boolean variables). The sub-LinAIGs rooted by two nodes which are to be compared are translated into the input format of the SMT solver[12] and the solver decides equivalence or non-equivalence. If two nodes are proven to be equivalent (taking the linear constraints into account), then these nodes can be merged, leading to a compaction of the representation (or even leading to the detection of a fixpoint in the model checking computation).

Remark 1. Note that it is also possible to merge nodes which are equivalent "modulo invariants" which have been proved separately (see page 1127). Assume that invariants *Inv* are described by the predicate ξ_{inv}. During our backward reachability analysis we consider sets ϕ of states from which the unsafe states are reachable. Since we are only interested in the question whether we can reach an unsafe state from the initial states, we may arbitrarily add states in $\neg Inv$ to ϕ or remove states in $\neg Inv$ from ϕ: By adding a state s in $\neg Inv$ to ϕ we never add a path from an initial state via s to an unsafe state. By removing a state s in $\neg Inv$ from ϕ we never remove an existing path from an initial state via s to an unsafe state. Both statements are true, because (by definition) states in $\neg Inv$ can never be reached from the initial states.

Thus, we are allowed to merge two nodes n_1 and n_2, if the represented predicates f_{n_1} and f_{n_2} are equivalent "modulo invariants", i.e., if $(f_{n_1} \oplus f_{n_2}) \wedge \xi_{inv}$ is unsatisfiable.[13]

Concerning node merging based on SMT solver applications we consider two possible strategies:

- A first variant we implemented is a *fully lazy* application of an SMT solver. The fully lazy variant invokes the SMT solver only when explicit equivalence checks and fixpoint checks are used in the model checking procedure.

[10] 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 $\sigma(q_{\ell_i}) = q_{\overline{\ell_i}}$ and $\sigma(q_{\overline{\ell_i}}) = q_{\ell_i}$, for all $\ell_i \in \mathcal{L}(C)$.

[11] The (unique) transitive reduction of a acyclic directed graph can be computed in $O(|V| \cdot |E|)$ [42].

[12] In our implementation we use Yices [11] for this task.

[13] A similar optimization can be performed w.r.t. the global constraints represented by *GC*, because in our model checking algorithm the (backward) reachable states are intersected with *GC* after each preimage step. However, it is not semantically sound to optimize $\phi \wedge GC$ "modulo *GC*" *before* preimage computation by substitutions; we have to be careful with the order of operations when performing optimizations modulo global constraints. Technical details are omitted here.

- On the other hand, it is possible to use an SMT solver in an *eager* manner whenever a new node is inserted into the LinAIG, just as SAT (together with simulation) is used in the FRAIG representation of the Boolean part. This leads to an LinAIG representation where different nodes always represent different predicates.

However, to avoid as many SMT checks as possible, we make use of the Boolean reasoning features offered by FRAIGs during insertion of nodes into the FRAIG part of the LinAIG. This leads to the following layered approach:

1. At first, the FRAIG package uses structural hashing for identifying an existing AND node which has the same pair of inputs. If such a node already exists, it is not necessary to insert a new (equivalent) node. Moreover, identification of identical nodes is assisted in the FRAIG package by (restricted) local transformation and normalization rules.
2. Second, it is checked whether there is already a node in the representation which represents the same Boolean function, when *constraint variables q_{ℓ_i} are not interpreted by their corresponding linear constraints ℓ*. When the node which is about to be inserted is compared to an existing node, we have to solve a Boolean problem with pure Boolean input variables and constraint variables q_{ℓ_i}. This Boolean problem is encoded as an input to a CNF-based (Boolean) SAT solver. If the SAT solver proves that two nodes are equivalent, it is clear that the nodes remain equivalent when constraint variables q_{ℓ_i} are interpreted by their corresponding linear constraints ℓ_i. Thus, in case of equivalence the existing node and the node to be inserted can be merged.[14] Note that the translation step into a SAT instance includes additional clauses for implications between linear constraints as described in Section 4.1.1.

   The set of candidate nodes for SAT checks is determined by simulation. The simulation assigns values to pure Boolean variables and constraint variables q_{ℓ_i}. SAT checks for proving equivalence need only be applied to pairs of nodes which show the same results for all simulation vectors used. (As already mentioned in Section 4.1.1 we use only simulation vectors which are consistent w.r.t.detected implications between linear constraints.)
3. Finally, an SMT solver is used for checking whether the node to be inserted represents a predicate which is already represented in the LinAIG. Similarly to the simulation approach for FRAIGs, the number of potential SMT-based equivalence checks is reduced based on simulation. We use simulation with test vectors as an incomplete but cheap method to show the *non-equivalence* of LinAIG nodes. However, note that for this purpose we can not use the same simulation vectors as we use for the pure FRAIG part of the LinAIG (assignments of values to pure boolean variables and constraint variables q_{ℓ_i} which are initially random, but are enhanced by counterexamples learnt from SAT applications later on), since these vectors are *not necessarily consistent w.r.t. the interpretation of constraint variables q_{ℓ_i} by their corresponding linear constraints ℓ_i*. If a proof of non-equivalence for two nodes is based on non-consistent simulation vectors, it may be incorrect. For this reason we use an appropriate set of test vectors in terms of real variables such that we can compute consistent Boolean valuations of linear constraints based on the real valued test vectors. These values, combined with assignments to the pure Boolean variables, may be used for proving non-equivalences of LinAIG nodes representing predicates over Boolean variables and linear constraints.

   At first, test vectors consist of arbitrary values for real-valued variables c_i (taking global constraints *GC* and invariants *Inv* into account, if they exist). Later on, we add test vectors learnt from successful applications of the SMT solver. If we are able to prove non-equivalence of two LinAIG nodes, the SMT solver returns an assignment to the Boolean variables and the real-valued variables (occurring in linear constraints) which witnesses a difference between the two corresponding formulas over Boolean variables and linear constraints. Based on the intuition that these assignments represent interesting corner cases for distinguishing between different predicates we learn the corresponding vectors for later applications of simulation. Our experimental results clearly demonstrate that this is a effective strategy for reducing the number of SMT checks in the future.

The details given above show that even in the eager variant SMT checks will not be used during node insertion, if we find an equivalent node based on pure Boolean reasoning in steps 1 and 2. Implications between linear constraints computed as given in Section 4.1.1 help in finding more equivalences by Boolean reasoning. Moreover, if it is proven by simulation that the new node is different from all existing nodes, then SMT checks can be avoided, too.

4.1.3. *Effect of LinAIG optimizations using information on linear constraints*

We use a simplified version of the flap controller case study presented in Section 7 to demonstrate the effect of our basic LinAIG optimizations using information on linear constraints. (In the simplified version the number of goal positions of the flap was reduced from 4 to 3.)

At first, we used a fully lazy application of an SMT solver (with SMT checks only for explicit equivalence checks and fixpoint checks in the model checking procedure) and omitted implication-based compaction. In the second experiment we used lazy SMT application together with implication-based compaction and finally we used eager SMT checking as described above. In Fig. 2 the evolution of the number of active LinAIG nodes over time is shown (dotted line without implication-based compaction, dashed line with implication-based compaction, and solid line for eager SMT application). Fig. 3 shows the same comparison regarding the number of active linear constraints. The figures show that implication-based compaction is indeed able to reduce the number of active LinAIG nodes and active linear constraints. With implication-based compaction

[14] Our FRAIG package does not necessarily choose the existing node, but selects the smallest of the two representations.

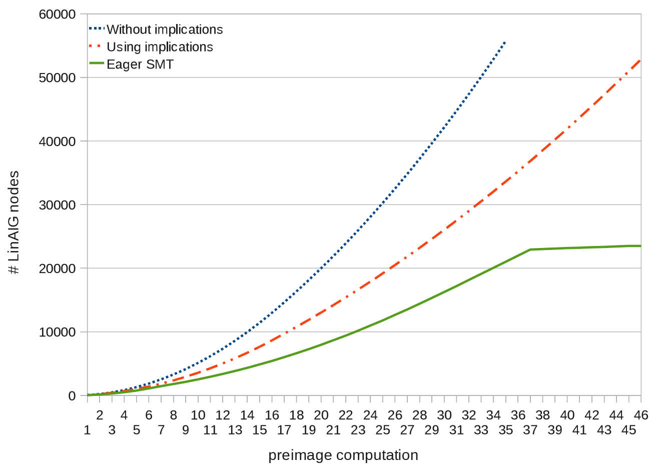


Fig. 2. Evolution of LinAIG nodes without implications, with implications, and with eager SMT checks.

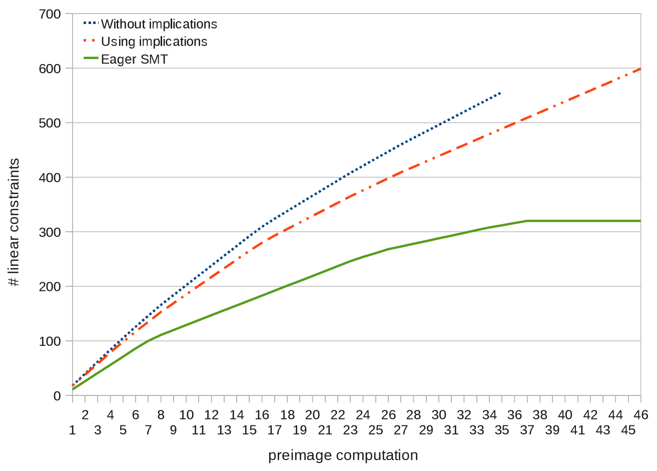


Fig. 3. Number of linear constraints without implications, with implications, and with eager SMT checks.

the model checking run was finished within 1 CPU hour whereas the version without implication-based compaction did not finish within the time limit of 3 CPU hours. For eager SMT application we additionally observe considerable improvements compared to the lazy version with implication-based compaction both w.r.t. the numbers of active LinAIG nodes and active linear constraints. The time for the complete model checking run was reduced from 1 CPU hour to 4 CPU minutes.

In the following we confine ourselves to the eager SMT application and provide a more detailed analysis demonstrating the effect of our layered approach using Boolean reasoning, simulation and SMT checks during node insertion.

For 92.1% of our attempts to insert a node into the LinAIG we obtained the result that there was a functionally equivalent node already in the representation. For the remaining cases when no equivalent node was found we were able to prove non-equivalence already by simulation with our (real-valued) test vectors in 97.4% of these cases; only in 2.6% of these non-equivalent cases did we have to use an SMT check, because the test vectors were not yet strong enough.

When we consider the cases when a functionally equivalent node was found during node insertion, we also observe that in almost all cases an SMT check was unnecessary. In 99.97% of these cases we were able to prove equivalence just by Boolean reasoning, i.e., by local reasoning like structural hashing or by SAT strengthened by implications.

If we consider only the cases when we tried to prove equivalence using Boolean reasoning, we can observe that 90.7% of the potential SAT checks were avoided by structural arguments and 7.8% by simulation (proving Boolean non-equivalence). Only in the remaining 1.5% of the cases were SAT checks performed (of which 89.1% were successful in proving Boolean equivalence). This again proves the importance of less expensive methods to filter out simple cases.

Altogether we can observe that by a sophisticated interaction between all components in our layered approach we can reduce the number of expensive SMT checks to a great extent, even if we use the eager version of SMT checking.

4.2. Optimization by Boolean invariants

In our experiments we made the observation that backward reachability analysis often visits a large number of discrete states (or even modes) which are not reachable from the initial states. For an extreme case consider a unary encoding of n

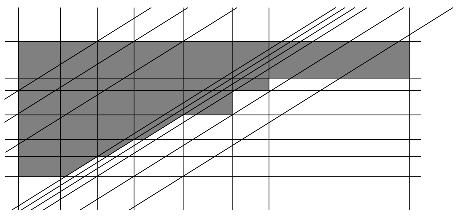


Fig. 4. Before redundancy removal.

discrete states: Backward reachability analysis starting from the unsafe states potentially has to traverse 2^n discrete states, since it is not clear in advance that at most n patterns of state bits can be reached from the initial state.

This is not really surprising and in fact forward and backward reachability analyses have "symmetrical disadvantages": A forward traversal usually visits large sets of states which do not have a connection to the unsafe states and a backward traversal usually visits large sets of states which do not have a connection to the initial states. (The most extreme case occurs when there is no path from the initial states to the unsafe states.)

In order to mitigate this problem we follow the idea of supporting backward analysis by information obtained from an approximate forward analysis. More precisely, we compute an overapproximation of the states reachable from the initial states based on a Boolean abstraction of our system model.

For Boolean abstraction, predicates $\xi \in \mathcal{P}(D, I, C, M)$ in the system model are replaced by their Boolean abstraction $ba(\xi)$: We assign a new Boolean input variable $d_j^{\ell_i}$ to each linear constraint ℓ_i in the system model. $ba(\xi)$ results from ξ by replacing each linear constraint ℓ_i occurring in ξ by $d_j^{\ell_i}$. Guarded assignments

$$\xi_i \to (d_1, \ldots, d_n) := (g_{i,1}, \ldots, g_{i,n});\ \ (c_1, \ldots, c_f) := (t_{i,1}, \ldots, t_{i,f});\ \ (m_1, \ldots, m_l) := \mathbf{m}_{j_i}$$

are replaced by

$$ba(\xi_i) \to (d_1, \ldots, d_n) := (ba(g_{i,1}), \ldots, ba(g_{i,n}));\ \ (m_1, \ldots, m_l) := \mathbf{m}_{j_i}.$$

The Boolean abstraction of the system model then results only from the boolean abstractions of discrete transitions, continuous transitions are neglected. If $d_{p+1}, \ldots, d_m$ are the new Boolean input variables assigned to linear constraints and if the initial states of the hybrid automaton are given by the predicate ξ_{init}, then the initial states of the Boolean abstraction are defined by $\exists d_{p+1}, \ldots, d_m ba(\xi_{init})$.

Now it is clear that the forward reachable states $fwreach_{ba}$ of the Boolean abstraction of an automaton overapproximate the forward reachable states $fwreach$ of the original automaton. In other words, $fwreach_{ba}$ is a (purely Boolean) invariant of the automaton. We compute $fwreach_{ba}$ by a standard symbolic forward model checker for discrete systems [10]. This invariant can then be used to optimize state set representations as described in Remark 1 on page 1131.

Effect of optimization by boolean invariants. In a first (and simpler) version of our flap controller case study (for details see Section 7) we used a model without error detection capabilities and a unary encoding of the states (4LP-noHM). The model checking time for this model was 177 CPU minutes (with all optimizations described in the previous section). Using the Boolean overapproximation of the forward reachable states we were able to reduce the run time to 122.7 CPU seconds. This is due to the fact that 75% of the discrete states were proved to be unreachable from the initial states. These unreachable states were used to optimize state sets during backward analysis.

5. Redundancy elimination

5.1. Motivation

In Sections 2.2 and 4.1 we already introduced several methods which turn LinAIGs into an efficient data structure for Boolean combinations of Boolean variables and linear constraints over real variables. However, especially in connection with the Loos–Weispfenning quantifier elimination used to compute continuous steps, one observes that the number of "redundant" linear constraints grows rapidly during the fixpoint iteration of the model checker. For illustration see Figs. 4 and 5, which show a typical example from a model checking run representing a small state set based on two real variables: Lines in Figs. 4 and 5 represent linear constraints, and the gray shaded area represents the space defined by some Boolean combination of these constraints. Whereas the representation depicted in Fig. 4 contains 24 linear constraints, a closer analysis shows that an optimized representation can be found using only 15 linear constraints as depicted in Fig. 5.

Removing such *redundant constraints* from our representations is a crucial task for the success of our methods. The motivation for this lies in the observation that for preimage computations the complexity of the result strongly depends on the number of linear constraints on which the original representation depends: Suppose the original representation depends on n linear constraints. The result of a discrete step may depend on $n \times v$ linear constraints in the worst case, if v is the number of guarded assignments in the discrete transition relation (see Section 3.1.1). Eliminating a single quantifier by the Loos–Weispfenning method used in continuous step computations may lead to a quadratic increase in the number of linear constraints in the result.

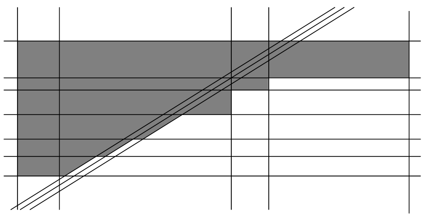


Fig. 5. After redundancy removal.

5.2. *Redundancy detection and removal for convex polyhedra*

It should be noted that, since we represent arbitrary Boolean combinations of linear constraints (and Boolean variables), removing redundant linear constraints is not as straightforward as for other approaches such as [32,24], which represent sets of convex polyhedra, i.e., sets of conjunctions $\ell_1 \wedge \cdots \wedge \ell_n$ of linear constraints. If one is restricted to convex polyhedra, the question whether a linear constraint ℓ_1 is redundant in the representation reduces to the question whether $\ell_2 \wedge \cdots \wedge \ell_n$ represents the same polyhedron as $\ell_1 \wedge \cdots \wedge \ell_n$, or equivalently, whether $\neg\ell_1 \wedge \ell_2 \wedge \cdots \wedge \ell_n$ represents the empty set. This question can simply be answered by a linear program solver.

5.3. *Redundancy detection for* LinAIGs

Redundancy of linear constraints is defined as follows:

Definition 3 (*Redundancy of Linear Constraints*)**.** Let F be a Boolean function, let $d_1, \ldots, d_n$ be Boolean variables and let $\ell_1, \ldots, \ell_k$ be linear constraints over real-valued variables $C = \{c_1, \ldots, c_f\}$. The linear constraints $\ell_1, \ldots, \ell_r$ $(1 \le r \le k)$ are called *redundant* in the representation of $F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ iff there is a Boolean function G with the property that $F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ and $G(d_1, \ldots, d_n, \ell_{r+1}, \ldots, \ell_k)$ represent the same predicates.

Our check for redundancy is based on the following theorem:

Theorem 1 (*Redundancy Check*)**.** *For all* $1 \le i \le k$ *let* ℓ_i *be a linear constraint over real-valued variables* $\{c_1, \ldots, c_f\}$ *and* ℓ'_i, exactly the same *linear constraint as* ℓ_i*, but now over a* disjoint copy $\{c'_1, \ldots, c'_f\}$ *of the variables. Let* $\equiv$ *denote Boolean equivalence. The linear constraints* $\ell_1, \ldots, \ell_r$ $(1 \le r \le k)$ *are* redundant *in the representation of* $F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ *if and only if the predicate*

$$\big(F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k) \wedge \neg F(d_1, \ldots, d_n, \ell'_1, \ldots, \ell'_k)\big) \wedge \bigwedge_{i=r+1}^{k} (\ell_i \equiv \ell'_i) \tag{1}$$

is not satisfiable by any assignment of Boolean values to $d_1, \ldots, d_n$ *and real values to the variables* $c_1, \ldots, c_f$ *and* $c'_1, \ldots, c'_f$*.*

Note that the check from Theorem 1 can be performed by a (conventional) SMT solver (e.g. [11,12]).

At first, we give a proof for the only-if-part of Theorem 1.

Proof of Theorem 1 (*Only-if-part*)**.** Let us assume that the predicate from formula (1) is satisfiable and under this assumption we prove that it cannot be the case that all linear constraints $\ell_1, \ldots, \ell_r$ are redundant, i.e., there is no Boolean function G such that $F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ and $G(d_1, \ldots, d_n, \ell_{r+1}, \ldots, \ell_k)$ represent the same predicates.

Consider some satisfying assignment to the predicate from formula (1) as follows: For the real variables $c_1 := v_{c_1}, \ldots, c_f := v_{c_f}$ with $(v_{c_1}, \ldots, v_{c_f}) \in \mathbb{R}^f$, for the copied real variables $c'_1 := v_{c'_1}, \ldots, c'_f := v_{c'_f}$ with $(v_{c'_1}, \ldots, v_{c'_f}) \in \mathbb{R}^f$, and for the Boolean variables $d_1 := v_{d_1}, \ldots, d_n := v_{d_n}$ with $(v_{d_1}, \ldots, v_{d_n}) \in \{0, 1\}^n$.

This satisfying assignment implies a corresponding truth assignment to the linear constraints by $\ell_i(v_{c_1}, \ldots, v_{c_f}) = v_{\ell_i}$ $(1 \le i \le k)$ with $v_{\ell_i} \in \{0, 1\}$ and to the copied linear constraints by $\ell'_i(v_{c'_1}, \ldots, v_{c'_f}) = v_{\ell'_i}$ $(1 \le i \le k)$ with $v_{\ell'_i} \in \{0, 1\}$.

Since the assignment satisfies formula (1), it holds that

$$\begin{array}{llll} F(v_{d_1}, \ldots, v_{d_n}, v_{\ell_1}, \ldots, v_{\ell_k}) = 1, & \text{(a)} & F(v_{d_1}, \ldots, v_{d_n}, v_{\ell'_1}, \ldots, v_{\ell'_k}) = 0, & \text{(b)} \\ \quad v_{\ell_i} = v_{\ell'_i} \text{ for all } r+1 \le i \le k. & \text{(c)} & & \end{array}$$

Then $F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k) = G(d_1, \ldots, d_n, \ell_{r+1}, \ldots, \ell_k)$ would imply $G(v_{d_1}, \ldots, v_{d_n}, v_{\ell_{r+1}}, \ldots, v_{\ell_k}) = 1$ because of (a) and $G(v_{d_1}, \ldots, v_{d_n}, v_{\ell'_{r+1}}, \ldots, v_{\ell'_k}) = 0$ because of (b). However, since $v_{\ell_i} = v_{\ell'_i}$ for all $r+1 \le i \le k$ (c), this is a contradiction. $\square$

A constructive proof for the if-part of Theorem 1 is given in Section 5.4.

Overall algorithm for redundancy detection. Now we can present our overall algorithm detecting a maximal set of linear constraints which can be removed from the representation *at the same time*. We start with a small example demonstrating the effect that it is not enough to consider redundancy of single linear constraints and to construct larger sets of redundant constraints simply as unions of smaller sets.

Example 1. Consider the predicate $F(c_1, c_2) = (c_1 \geq 0) \wedge (c_2 \geq 0) \wedge \neg(c_1 + c_2 \leq 0) \wedge \neg(2c_1 + c_2 \leq 0)$. It is easy to see that both the third and the forth linear constraint in the conjunction have the effect of "removing the value $(c_1, c_2) = (0, 0)$ from the predicate $F'(c_1, c_2) = (c_1 \geq 0) \wedge (c_2 \geq 0)$". Therefore both $\ell_3 = (c_1 + c_2 \leq 0)$ and $\ell_4 = (2c_1 + c_2 \leq 0)$ are obviously redundant linear constraints in F. However, it is also easy to see that ℓ_3 and ℓ_4 are not redundant in the representation of F *at the same time*, i.e., only $\neg(c_1 + c_2 \leq 0)$ *or* $\neg(2c_1 + c_2 \leq 0)$ can be omitted in the representation for F.

This observation motivates the following overall algorithm to detect a maximal set of redundant linear constraints:

```
Input  : Predicate $F(d_1, \dots, d_n, \ell_1, \dots, \ell_k)$
Output: $S$: Maximal set of redundant linear constraints
begin
    $S := \emptyset$;
    for $i := 1$ to $k$ do
        if redundant($F, S \cup \{\ell_i\}$) then $S := S \cup \{\ell_i\}$;
    return $S$;
```

redundant$(F, S \cup \{l_i\})$ implements the check from Theorem 1 by using an SMT solver. It is important to note that the k SMT problems to be solved in the above loop share almost all of their clauses. For that reason we make use of an *incremental* SMT solver to solve this series of problems. An incremental SMT solver is able to profit from the similarity of the problems by transferring learned knowledge from one SMT solver call to the next (by means of learned conflict clauses).

5.4. Removal of redundant linear constraints

Suppose that formula (1) of Theorem 1 is unsatisfiable. Now we are looking for an efficient procedure to compute a Boolean function G such that $G(d_1, \dots, d_n, \ell_{r+1}, \dots, \ell_k)$ and $F(d_1, \dots, d_n, \ell_1, \dots, \ell_k)$ represent the same predicates. The construction of an appropriate function G provides a constructive proof for the if-part of Theorem 1. Obviously, the Boolean functions F and G do not need to be identical in order to achieve this objective given above; they are allowed to differ for "inconsistent" arguments which can not be produced by evaluating the linear constraints with real values. The set of these arguments is described by the following "don't care set" dc_{inc}:

Definition 4. The *don't care set* dc_{inc} *induced by linear constraints* $\ell_1, \dots, \ell_k$ is defined as

$$dc_{inc} := \{(v_{d_1}, \dots, v_{d_n}, v_{\ell_1}, \dots, v_{\ell_k}) \mid (v_{d_1}, \dots, v_{d_n}) \in \{0,1\}^n, (v_{\ell_1}, \dots, v_{\ell_k}) \in \{0,1\}^k \text{ and } \forall \mathbf{v_c} \in \mathbb{R}^f\ \exists 1 \leq i \leq k \text{ with } \ell_i(\mathbf{v_c}) \neq v_{\ell_i}\}. \quad (2)$$

As we will see in the following, it is possible to compute a function G as needed by making use of the don't care set dc_{inc}. However, an efficient realization would certainly need a compact representation of the don't care set dc_{inc}. Fortunately, a closer look at the problem reveals the following two interesting observations which turn our basic idea into a feasible approach:

1. In general, we do not need the complete set dc_{inc} for the computation of the Boolean function G.
2. A representation of a sufficient subset dc'_{inc} of dc_{inc} which is needed for removing the redundant constraints $\ell_1, \dots, \ell_r$ is already computed by an SMT solver when checking the satisfiability of formula (1) (if one assumes that the SMT solver uses the option of minimizing conflict clauses, as we will see later on).

In order to explain how an appropriate subset dc'_{inc} of dc_{inc} is computed by the SMT solver (when checking the satisfiability of formula (1)) we start with a brief review of the functionality of an SMT solver[15]:

An SMT solver introduces constraint variables q_{ℓ_i} for linear constraints ℓ_i (just as in LinAIGs as shown in Fig. 1). First, the SMT solver looks for satisfying assignments to the Boolean variables (including the constraint variables). Whenever the SMT solver detects a satisfying assignment to the Boolean variables, it checks whether the assignment to the constraint variables is consistent, i.e., whether it can be produced by replacing real-valued variables by reals in the linear constraints. This task is performed by a linear program solver. If the assignment is consistent, then the SMT solver has found a satisfying assignment, otherwise it continues searching for satisfying assignments to the Boolean variables. If some assignment $\epsilon_1, \dots, \epsilon_m$ to constraint variables $q_{\ell_{i_1}}, \dots, q_{\ell_{i_m}}$ was found to be inconsistent, then the Boolean "conflict clause" $(\neg q_{\ell_{i_1}}^{\epsilon_1} \vee \dots \vee \neg q_{\ell_{i_m}}^{\epsilon_m})$ is added to the set of clauses in the SMT solver to avoid running into the same conflict again.[16] The negation of this conflict clause describes a set of don't cares due to an inconsistency of linear constraints.

15 Here we refer to the *lazy* approach to SMT solving, see [43], e.g., for an overview.

16 By definition $q_{\ell_{i_j}}^1 := q_{\ell_{i_j}}$ and $q_{\ell_{i_j}}^0 := \neg q_{\ell_{i_j}}$.

Now consider formula (1), which has to be solved by an SMT solver, and suppose that the solver introduces Boolean constraint variables q_{ℓ_i} for linear constraints ℓ_i and $q_{\ell'_i}$ for ℓ'_i $(1 \leq i \leq k)$. Whenever there is some satisfying assignment to Boolean variables (including constraint variables) in the SMT solver, it will be necessarily shown to be inconsistent, since formula (1) is unsatisfiable.

In order to define an appropriate function G we introduce the concept of so-called orbits: For an arbitrary value $(v_{d_1}, \ldots, v_{d_n}, v_{\ell_{r+1}}, \ldots, v_{\ell_k}) \in \{0, 1\}^{n+k-r}$ the corresponding orbit is defined by

$$orbit(v_{d_1}, \ldots, v_{d_n}, v_{\ell_{r+1}}, \ldots, v_{\ell_n}) := \{(v_{d_1}, \ldots, v_{d_n}, v_{\ell_1}, \ldots, v_{\ell_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_n}) \mid (v_{\ell_1}, \ldots, v_{\ell_r}) \in \{0, 1\}^r\}.$$

The following essential observation results from the unsatisfiability of formula (1):

If some orbit $orbit(v_{d_1}, \ldots, v_{d_n}, v_{\ell_{r+1}}, \ldots, v_{\ell_k})$ contains two different elements $v^{(1)} := (v_{d_1}, \ldots, v_{d_n}, v_{\ell_1}, \ldots, v_{\ell_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_k})$ and $v^{(2)} := (v_{d_1}, \ldots, v_{d_n}, v_{\ell'_1}, \ldots, v_{\ell'_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_k})$ with $F(v^{(1)}) \neq F(v^{(2)})$, then

(a) $v^{(1)} \in dc_{inc}$ or $v^{(2)} \in dc_{inc}$ and
(b) the SMT solver detects and records this don't care when solving formula (1).

In order to show fact (a), we consider the following assignment to Boolean variables and Boolean abstraction variables in formula (1): Let $d_1 := v_{d_1}, \ldots, d_n := v_{d_n}, q_{\ell_1} := v_{\ell_1}, \ldots, q_{\ell_r} := v_{\ell_r}, q_{\ell'_1} := v_{\ell'_1}, \ldots, q_{\ell'_r} := v_{\ell'_r}, q_{\ell_{r+1}} := q_{\ell'_{r+1}} := v_{\ell_{r+1}}, \ldots, q_{\ell_k} := q_{\ell'_k} := v_{\ell_k}$. (Thus $v^{(1)}$ is assigned to the variables $d_1, \ldots, d_n, q_{\ell_1}, \ldots, q_{\ell_k}$ and $v^{(2)}$ to the variables $d_1, \ldots, d_n, q_{\ell'_1}, \ldots, q_{\ell'_k}$.) It is easy to see that this assignment satisfies the Boolean abstraction of formula (1). Since formula (1) is unsatisfiable, the assignment has to be inconsistent w.r.t. the interpretation of constraint variables by linear constraints. So there must be an inconsistency in the truth assignment to some linear constraints $\ell_1, \ldots, \ell_k, \ell'_1, \ldots, \ell'_k$. Since the linear constraints ℓ_i and ℓ'_i are based on disjoint sets of real variables $C = \{c_1, \ldots, c_f\}$ and $C' = \{c'_1, \ldots, c'_f\}$, already the partial assignment to $\ell_1, \ldots, \ell_k$ or the partial assignment to $\ell'_1, \ldots, \ell'_k$ has to be inconsistent, i.e., $v^{(1)} \in dc_{inc}$ or $v^{(2)} \in dc_{inc}$.

Fact (b) follows from the simple observation that the SMT solver has to detect and record the inconsistency of the assignment mentioned above in order to prove the unsatisfiability of formula (1) and with minimization of conflict clauses it detects only conflicts which are confined either to $\ell_1, \ldots, \ell_k$ or to $\ell'_1, \ldots, \ell'_k$.[17]

Altogether this means that the elements of some $orbit(v_{d_1}, \ldots, v_{d_n}, v_{\ell_{r+1}}, \ldots, v_{\ell_k})$ which are not in the subset dc'_{inc} of dc_{inc} computed by the SMT solver are either all mapped by F to 0 or are all mapped by F to 1. Thus, we can define an appropriate function G by the don't care assignment as follows:

1. If $orbit(v_{d_1}, \ldots, v_{d_n}, v_{\ell_{r+1}}, \ldots, v_{\ell_k}) \subseteq dc'_{inc}$, then $G(v_{d_1}, \ldots, v_{d_n}, v_{\ell_{r+1}}, \ldots, v_{\ell_k})$ is chosen arbitrarily.
2. Otherwise $G(v_{d_1}, \ldots, v_{d_n}, v_{\ell_{r+1}}, \ldots, v_{\ell_k}) = \delta$ with $F(orbit(v_{d_1}, \ldots, v_{d_n}, v_{\ell_{r+1}}, \ldots, v_{\ell_k}) \setminus dc'_{inc}) = \{\delta\}, \delta \in \{0, 1\}$.

It is easy to see that G does not depend on variables $q_{\ell_1}, \ldots, q_{\ell_r}$ and that G is well-defined (this follows from $|F(orbit(v_{d_1}, \ldots, v_{d_n}, v_{\ell_{r+1}}, \ldots, v_{\ell_k}) \setminus dc'_{inc})| = 1$), i.e., G is a possible solution according to Definition 3. This consideration also provides a proof for the "if-part" of Theorem 1.

A predicate DC'_{inc} which describes the don't cares in dc'_{inc} may be extracted from the SMT solver as a disjunction of negated conflict clauses which record inconsistencies between linear constraints.

Note that according to case 1 of the definition of G there may be several possible choices fulfilling the definition of G.

Redundancy removal by existential quantification. A straightforward way of computing an appropriate function G relies on existential quantification:

- At first by $G' = F \wedge \neg DC'_{inc}$ all don't cares represented by DC'_{inc} are mapped to the function value 0.
- Secondly, we perform an existential quantification of the variables $q_{\ell_1}, \ldots, q_{\ell_r}$ in G': $G = \exists q_{\ell_1}, \ldots, q_{\ell_r} G'$. This existential quantification maps all elements of an orbit $orbit(v_{d_1}, \ldots, v_{d_n}, v_{\ell_{r+1}}, \ldots, v_{\ell_k})$ to 1, whenever the orbit contains an element ϵ with $dc'_{inc}(\epsilon) = 0$ and $F(\epsilon) = 1$. Since due to the argumentation above there is no other element δ in such an orbit with $dc_{inc}(\delta) = 0$ and $F(\delta) = 0$, G eventually differs from F only for don't cares defined by DC'_{inc} and it certainly does not depend on variables $q_{\ell_1}, \ldots, q_{\ell_r}$, i.e., existential quantification computes one possible solution for G according to Definition 3 (more precisely it computes exactly the solution for G which maps a minimum number of elements of $\{0, 1\}^{n+k-r}$ to 1).

Redundancy removal with Craig interpolants. Although our implementation of LinAIGs supports the quantification of Boolean variables by a series of methods like functional reduction (see Section 2.2), quantifier scheduling, BDD sweeping and node selection heuristics (see [35]), there remains the risk of doubling the representation size by quantifying a single Boolean variable.[18] Therefore the computation of G by $G = \exists q_{\ell_1}, \ldots, q_{\ell_r} G'$ as shown above may potentially lead to large LinAIG representations (although it reduces the number of linear constraints).

[17] For our purposes, it does not matter whether the inconsistency is given in terms of linear constraints $\ell_1, \ldots, \ell_k$ or $\ell'_1, \ldots, \ell'_k$. We are only interested in assignments of Boolean values to linear constraints leading to inconsistencies; of course, the set of all inconsistencies is the same both for $\ell_1, \ldots, \ell_k$ and their copies $\ell'_1, \ldots, \ell'_k$.

[18] Basically, existential quantification of a Boolean variable is reduced to a disjunction of both cofactors w.r.t. 0 and w.r.t. 1.

On the other hand, this choice for G is only one of many other possible choices. Motivated by these facts we looked for an alternative solution. Here we present a solution which needs only one application of Craig interpolation [14,15] instead of a series of existential quantifications of Boolean variables.

Recently, Craig interpolation was applied by McMillan for generating an overapproximated image operator to be used in connection with Bounded Model Checking [16] and by Lee et al. for computing a so-called dependency function in logic synthesis algorithms [17]. According to [14,15], a Craig interpolant is computed for an *unsatisfiable* Boolean formula H in Conjunctive Normal Form (CNF) (i.e., for a conjunction of disjunctions of Boolean variables) which is partitioned into two parts A and B with $H = A \wedge B = 0$.

Note that (in contrast to McMillan's application [16]) Craig interpolation leads to an exact result (as one of several possible choices) in our context and not to an approximation.

Don't cares can be assigned arbitrarily in order to make our function G independent from $q_{\ell_1}, \ldots, q_{\ell_r}$, thus our task is to find a Boolean function $G(d_1, \ldots, d_n, q_{\ell_{r+1}}, \ldots, q_{\ell_k})$ with

$$(F \wedge \neg DC'_{inc})(d_1, \ldots, d_n, q_{\ell_1}, \ldots, q_{\ell_k}) \Longrightarrow G(d_1, \ldots, d_n, q_{\ell_{r+1}}, \ldots, q_{\ell_k}), \tag{3}$$

$$G(d_1, \ldots, d_n, q_{\ell_{r+1}}, \ldots, q_{\ell_k}) \Longrightarrow (F \vee DC'_{inc})(d_1, \ldots, d_n, q_{\ell_1}, \ldots, q_{\ell_k}). \tag{4}$$

Now let $A(d_1, \ldots, d_n, q_{\ell_1}, \ldots, q_{\ell_r}, q_{\ell_{r+1}}, \ldots, q_{\ell_k}, h_1, \ldots, h_l)$ represent the CNF for a Tseitin transformation [44] of $(F \wedge \neg DC'_{inc})(d_1, \ldots, d_n, q_{\ell_1}, \ldots, q_{\ell_r}, q_{\ell_{r+1}}, \ldots, q_{\ell_k})$ (with new auxiliary variables $h_1, \ldots, h_l$). Likewise, let $B(d_1, \ldots, d_n, q_{\ell'_1}, \ldots, q_{\ell'_r}, q_{\ell_{r+1}}, \ldots, q_{\ell_k}, h'_1, \ldots, h'_{l'})$ be the CNF for a Tseitin transformation of $(\neg F \wedge \neg DC'_{inc})(d_1, \ldots, d_n, q_{\ell'_1}, \ldots, q_{\ell'_r}, q_{\ell_{r+1}}, \ldots, q_{\ell_k})$ (with new auxiliary variables $h'_1, \ldots, h'_{l'}$).

Then A and B fulfill the precondition "$A \wedge B = 0$" for Craig interpolation [14,15]:

Suppose that there is a satisfying assignment to $A \wedge B$ given by $d_1 := v_{d_1}, \ldots, d_n := v_{d_n}, q_{\ell_1} := v_{\ell_1}, \ldots, q_{\ell_r} := v_{\ell_r}, q_{\ell'_1} := v_{\ell'_1}, \ldots, q_{\ell'_r} := v_{\ell'_r}, q_{\ell_{r+1}} := v_{\ell_{r+1}}, \ldots, q_{\ell_n} := v_{\ell_n}$, and the corresponding assignments to auxiliary variables $h_1, \ldots, h_l$ and $h'_1, \ldots, h'_{l'}$ which are implied by these assignments. According to the definition of A and B this would mean that the set $orbit(v_{d_1}, \ldots, v_{d_n}, v_{\ell_{r+1}}, \ldots, v_{\ell_k})$ would contain two elements $(v_{d_1}, \ldots, v_{d_n}, v_{\ell_1}, \ldots, v_{\ell_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_k})$ and $(v_{d_1}, \ldots, v_{d_n}, v_{\ell'_1}, \ldots, v_{\ell'_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_k})$ which do not belong to the don't care set dc'_{inc} and which fulfill $F(v_{d_1}, \ldots, v_{d_n}, v_{\ell_1}, \ldots, v_{\ell_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_k}) = 1$ and $F(v_{d_1}, \ldots, v_{d_n}, v_{\ell'_1}, \ldots, v_{\ell'_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_k}) = 0$. This is a contradiction to the property shown above that the elements of $orbit(v_{d_1}, \ldots, v_{d_n}, v_{\ell_{r+1}}, \ldots, v_{\ell_k})$ which are not in dc'_{inc} are either all mapped by F to 0 or are all mapped by F to 1.

A Craig interpolant G computed for A and B has the following properties [14,15]:

- It depends only on common variables $v_{d_1}, \ldots, v_{d_n}, q_{\ell_{r+1}}, \ldots, q_{\ell_k}$ of A and B,
- $A \Longrightarrow G$, i.e., G fulfills Eq. (3), and
- $G \wedge B$ is unsatisfiable, or equivalently, $G \Longrightarrow \neg B$, i.e., G fulfills Eq. (4).

This shows that a Craig interpolant for (A, B) is exactly one of the possible solutions for G which we were looking for. According to [15,16] a Craig interpolant can be computed in linear time based on a proof by resolution that a formula in CNF (in our case $A \wedge B$ as defined above) is unsatisfiable. Such proofs can be computed by any modern SAT solver with proof logging turned on.

5.5. Effect of redundancy removal

To demonstrate the effect of redundancy removal we consider the dam case study described in Section 7 (with parameters $t = 30, d = 50$). In Fig. 6 the evolution of the number of linear constraints is compared for the version without redundancy removal (line with squares), and for the version with redundancy removal (line with diamonds). The figure shows that redundancy removal is able to keep the growth of the number of linear constraints under control. The run time with redundancy removal by Craig interpolation is below 63 CPU minutes but model checking without redundancy removal does not finish within the CPU limit of 4 hours (with a large growth rate w.r.t. the number of linear constraints).

Using redundancy removal by existential quantification the number of linear constraints can be kept small as well, but the run times are much higher compared to Craig interpolation and therefore the model checking run does not finish within the CPU limit of 4 CPU hours. (Because the number of linear constraints does not differ between the two versions of redundancy removal, we omitted the version with existential quantification in the figure.)

6. Constraint minimization

6.1. Observations w.r.t. backward reachability analysis

According to Section 3 model checking is performed by a backward reachability analysis for LHA+Ds. Starting from a representation of the unsafe states, the backward reachability analysis performs an evaluation of a continuous flow, then

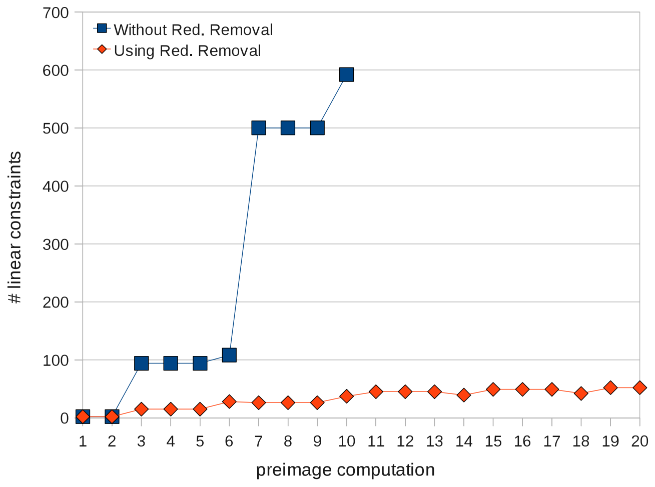


Fig. 6. Number of linear constraints without redundancy removal, and with redundancy removal by Craig interpolation.

an evaluation of a discrete-to-continuous step, a series of discrete steps until a local fixpoint is reached, a continuous-to-discrete step, then again a continuous flow, and so on. This iteration is performed until a global fixpoint is finally reached or until an initial state is reached.

In the following we present a further analysis and an improvement of the basic model checking algorithm. Since the basic ideas can also be explained using a simplified version of the fixpoint iteration, we make use of such a simplification for ease of explanation. In particular we assume here a fixpoint iteration just using discrete transitions. The simplified fixpoint iteration is given by Algorithm 1.

```
begin
    $\phi_0 := \neg safe$;  $i := 0$;
    if $\phi_0 \wedge init \neq 0$ then return $false$;
    repeat
        $i := i + 1$;  $\phi_i := \exists d_{n+1}, \dots d_p Pre^d(\phi_{i-1}) \vee \neg safe$;
        if $\phi_i \wedge init \neq 0$ then return $false$;
    until $\phi_i \wedge \neg\phi_{i-1} = 0$;
    return $true$;
```

Algorithm 1: Simplified reachability analysis assuming only discrete transitions.

The set ϕ_0 is initialized by the set of unsafe states. In the set ϕ_i we collect all states from which we can reach an unsafe state by a trajectory of length $\leq i$ and in that way we compute representations of larger and larger sets. If we use exactly Algorithm 1 we observe that we perform "duplicated work" in each step, since we start the preimage computation in line 3 also from states in ϕ_{i-1} which were already included in ϕ_{i-2} — and for these states we already performed a preimage computation.

This observation is not new, it had already been made when symbolic BDD based model checking was introduced for discrete systems in the late eighties [18,19]. A first idea for solving this problem is just to compute the preimage only for the "onion ring", i.e., for those states which were reached in the previous step and not before. This leads to Algorithm 2.

```
begin
    $\phi_0 := \neg safe$;  $\psi_0 := \neg safe$;  $i := 0$;
    if $\phi_0 \wedge init \neq 0$ then return $false$;
    repeat
        $i := i + 1$;  $\psi_i := \exists d_{n+1}, \dots d_p Pre^d(\psi_{i-1})$;
        $\psi_i := \psi_i \wedge \neg\phi_{i-1}$ ;                    // Newly reached states
        if $\psi_i \wedge init \neq 0$ then return $false$;
        $\phi_i := \phi_{i-1} \vee \psi_i$ ;                         // All reached states
    until $\phi_i \wedge \neg\phi_{i-1} = 0$;
    return $true$;
```

Algorithm 2: Changed reachability analysis.

However with symbolic representations it is by no means clear that the representation for a smaller set is indeed more compact and thus needs fewer resources while computing preimages. Consider the following example which illustrates this fact:

Example 2. We consider the set of unsafe states represented by the formula

$$\begin{aligned}\phi_0 = {} & (y > 1) \wedge (y < 33) \wedge (x > 4) \wedge (x < 36) \wedge (y - x < 5) \\ & \wedge [(y < 17) \vee (x > 16)] \wedge [(y < 21) \vee (x > 20)] \wedge [(y < 25) \vee (x > 24)] \wedge [(y < 29) \vee (x > 28)] \\ & \wedge [(x - y < 11) \vee (y > 7) \wedge (x < 22) \vee (y > 11) \wedge (x < 26) \vee (y > 15) \wedge (x < 30) \vee (y > 19) \wedge (x < 34)].\end{aligned}$$

ϕ_0 is illustrated in 7(a). With the assignment $x := x + 1, y := y - 1$ the preimage $Pre^d(\phi_0)$ results in

$$\begin{aligned}\phi_1 = {} & (y > 2) \wedge (y < 34) \wedge (x > 3) \wedge (x < 35) \wedge (y - x < 7) \\ & \wedge [(y < 18) \vee (x > 15)] \wedge [(y < 22) \vee (x > 19)] \wedge [(y < 26) \vee (x > 23)] \wedge [(y < 30) \vee (x > 27)] \\ & \wedge [(x - y < 9) \vee (y > 8) \wedge (x < 21) \vee (y > 12) \wedge (x < 25) \vee (y > 16) \wedge (x < 29) \vee (y > 20) \wedge (x < 33)].\end{aligned}$$

Both ϕ_0 and ϕ_1 depend on 22 linear constraints. If we compute a formula for the newly reached states by $\phi_1 \wedge \neg\phi_0$ as in Algorithm 2, we first obtain a representation depending on 44 constraints (and note that we obtain the same number of linear constraints, if we compute $\phi_1 \vee \neg safe$ as in line 3 of Algorithm 1). By removing redundant linear constraints from this representation of $\phi_1 \wedge \neg\phi_0$ we arrive at a representation depending on 24 linear constraints.[19] $\phi_1 \wedge \neg\phi_0$ is labeled "onion ring" in 7(a). Unfortunately, the representation of $\phi_1 \wedge \neg\phi_0$ is not simpler than that of ϕ_1, but more complicated. This shows that computing $\phi_1 \wedge \neg\phi_0$ is not necessarily superior to just using ϕ_1, if the state sets are represented symbolically.

Again, there is an idea from symbolic BDD based model checking for discrete systems which can help in this context: For the states in $\psi_i := \psi_i \wedge \neg\phi_{i-1}$ computed in line 4 of Algorithm 2 we have to compute the preimage in the next step. The states in ϕ_{i-1} on the other hand may or may not enter the next preimage computation (without changing the final set of reached states). That means that ψ_i can be optimized w.r.t. the "don't care set" ϕ_{i-1}. In [18,19] these don't cares were used in order to optimize the BDD representation of ψ_i by the *constrain* or *restrict* operation.

However it remains to question what an appropriate cost measure for optimizing our state set representations which are given by LinAIGs is. Here we propose to use the number of linear constraints the representation depends on. As already discussed in Section 5.1, removing redundant constraints is crucial for the success of our methods and we strongly prefer to compute preimages for state set representations which are optimized w.r.t. the number of linear constraints. By considering the set of already reached states as a "don't care set", we obtain additional degrees of freedom for keeping the number of linear constraints under control. Using such an optimization we will arrive at a modified version of Algorithm 2 where line 4 is replaced by $\psi_i := constraint\_minimization(\psi_i, \neg\phi_{i-1})$.

6.2. Making use of additional degrees of freedom

Before we look into the question of how to use don't cares in order to minimize the number of linear constraints, we have a look at Example 2 again:

Example 3. Now we interpret $\phi_0 = \neg safe$ as a don't care set and try to replace $\phi_1 = Pre^d(\neg safe)$ by a simpler representation just by changing ϕ_1 inside the don't care set. 7(b) gives a solution ("optimized shape") which depends only on 14 linear constraints.

In the remainder of this section we present how to compute such solutions by a suitable algorithm. Our methods generalize the approach for redundancy elimination from Section 5.

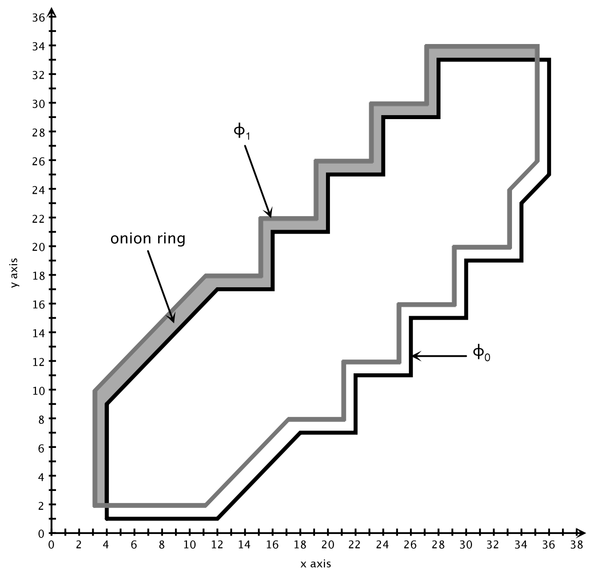
We start with a method to check whether a fixed set of linear constraints can be removed from a representation by using don't care conditions. In the following we assume a predicate $F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ which is to be optimized, a predicate $DC(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ for the don't care conditions, and a set $\ell_1, \ldots, \ell_r$ $(1 \le r \le k)$ of linear constraints which we would like to remove from F using don't care optimization.

Definition 5 (*DC-Removability of Linear Constraints*)**.** The linear constraints $\ell_1, \ldots, \ell_r$ $(1 \le r \le k)$ are called *DC-removable* from the representation of $F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ using don't cares from $DC(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ iff there is a Boolean function G with the property that $\neg DC(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k) \wedge F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ and $\neg DC(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k) \wedge G(d_1, \ldots, d_n, \ell_{r+1}, \ldots, \ell_k)$ represent the same predicates.[20]

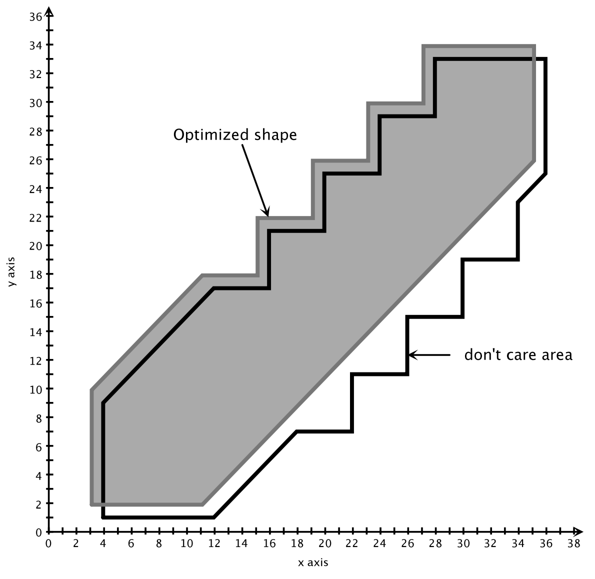
6.3. Checking DC-removability

The check whether a set of linear constraints is DC-removable is based on the following theorem, which generalizes Theorem 1:

[19] If $\phi_1 \wedge \neg\phi_0$ is represented by a LinAIG, the redundancy removal operation produces exactly such a representation with 24 linear constraints.
[20] This means that F and G are the same except for don't cares.



(a) Preimage computation.



(b) Don't care optimization.

Fig. 7. Motivating example for constraint minimization.

Theorem 2 (*DC-Removability Check*)**.** *For all* $1 \le i \le k$ *let* ℓ_i *be a linear constraint over real-valued variables* $\{c_1, \ldots, c_f\}$ *and* ℓ_i' exactly the same *linear constraint as* ℓ_i*, but now over a* disjoint copy $\{c_1', \ldots, c_f'\}$ *of the real-valued variables. Let* $\equiv$ *denote Boolean equivalence. The linear constraints* $\ell_1, \ldots, \ell_r$ $(1 \le r \le k)$ *are* DC-removable *from* $F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ *using don't cares from* $DC(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ *if and only if the predicate*

$$F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k) \wedge \neg F(d_1, \ldots, d_n, \ell_1', \ldots, \ell_k') \wedge \neg DC(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k) \wedge \neg DC(d_1, \ldots, d_n, \ell_1', \ldots, \ell_k') \wedge \bigwedge_{i=r+1}^{k} (\ell_i \equiv \ell_i') \tag{5}$$

is not satisfiable by any assignment of Boolean values to $d_1, \ldots, d_n$ *and real values to the variables* $c_1, \ldots, c_f$ *and* $c_1', \ldots, c_f'$*.*

Proof of Theorem 2 (*Only-if-part*)**.** The proof of the only-if-part generalizes the corresponding part of the proof for Theorem 1. We assume that the predicate from formula (5) is satisfiable and under this assumption we prove that it cannot be the case that all linear constraints $\ell_1, \ldots, \ell_r$ are DC-removable, i.e., there is no Boolean function G such that

$\neg DC(d_1, \dots, d_n, \ell_1, \dots, \ell_k) \wedge F(d_1, \dots, d_n, \ell_1, \dots, \ell_k)$ and $\neg DC(d_1, \dots, d_n, \ell_1, \dots, \ell_k) \wedge G(d_1, \dots, d_n, \ell_{r+1}, \dots, \ell_k)$ represent the same predicates.

As in the proof for Theorem 1 on page 1135 we assume a satisfying assignment to the predicate from formula (5). This satisfying assignment fulfills

$$\begin{array}{llll} F(v_{d_1}, \dots, v_{d_n}, v_{\ell_1}, \dots, v_{\ell_k}) = 1, & \text{(a)} & F(v_{d_1}, \dots, v_{d_n}, v_{\ell'_1}, \dots, v_{\ell'_k}) = 0, & \text{(b)} \\ v_{\ell_i} = v_{\ell'_i} \text{ for all } r+1 \le i \le k, & \text{(c)} & & \\ DC(v_{d_1}, \dots, v_{d_n}, v_{\ell_1}, \dots, v_{\ell_k}) = 0, & \text{(d)} & DC(v_{d_1}, \dots, v_{d_n}, v_{\ell'_1}, \dots, v_{\ell'_k}) = 0. & \text{(e)} \end{array}$$

Then $\neg DC(d_1, \dots, d_n, \ell_1, \dots, \ell_k) \wedge F(d_1, \dots, d_n, \ell_1, \dots, \ell_k) = \neg DC(d_1, \dots, d_n, \ell_1, \dots, \ell_k) \wedge G(d_1, \dots, d_n, \ell_{r+1}, \dots, \ell_k)$ would imply $G(v_{d_1}, \dots, v_{d_n}, v_{\ell_{r+1}}, \dots, v_{\ell_k}) = 1$ because of (a) and (d) and $G(v_{d_1}, \dots, v_{d_n}, v_{\ell'_{r+1}}, \dots, v_{\ell'_k}) = 0$ because of (b) and (e). However, since $v_{\ell_i} = v_{\ell'_i}$ for all $r+1 \le i \le k$ (c), this is a contradiction. □

Just as for the detection of a maximal set of redundant linear constraints, we can define an overall algorithm detecting a maximal set of linear constraints which are DC-removable *at the same time*. This algorithm is based on the check from Theorem 2 and makes use of an incremental SMT solver.

6.4. Computing an optimized representation

The ideas for a constructive proof for the if-part of Theorem 2 and thus a method to compute an appropriate function G as defined above are similar to the ideas behind the corresponding construction for the redundancy removal operation which was already described in Section 5.4. (However, in contrast to constraint minimization, redundancy removal does not change represented shapes at all.)

We assume that formula (5) from Theorem 2 is unsatisfiable and try to change $F(d_1, \dots, d_n, q_{\ell_1}, \dots, q_{\ell_k})$ in a way that the result G will be independent from $q_{\ell_1}, \dots, q_{\ell_r}$. In addition to the set dc_{inc} of don't cares due to inconsistent assignments to constraint variables (Definition 4) (or the subset $dc'_{inc} \subseteq dc_{inc}$ extracted from an SMT solver checking formula (5)), we can make use of a set dc of don't cares which results from the Boolean abstraction of the don't care predicate DC:

$$dc = \{(v_{d_1}, \dots, v_{d_n}, v_{\ell_1}, \dots, v_{\ell_k}) \mid (v_{d_1}, \dots, v_{d_n}) \in \{0,1\}^n \text{ and } \exists \mathbf{v_c} \in \mathbb{R}^f \text{ with } \ell_1(\mathbf{v_c}) = v_{\ell_1}, \dots, \ell_k(\mathbf{v_c}) = v_{\ell_k} \\ \text{and } DC(v_{d_1}, \dots, v_{d_n}, v_{\ell_1}, \dots, v_{\ell_k}) = 1\}.$$

Now we are looking for a Boolean function $G(d_1, \dots, d_n, q_{\ell_{r+1}}, \dots, q_{\ell_k})$ with

$$(F \wedge \neg DC \wedge \neg DC'_{inc})(d_1, \dots, d_n, q_{\ell_1}, \dots, q_{\ell_k}) \Longrightarrow G(d_1, \dots, d_n, q_{\ell_{r+1}}, \dots, q_{\ell_k}) \quad \text{and} \tag{6}$$

$$G(d_1, \dots, d_n, q_{\ell_{r+1}}, \dots, q_{\ell_k}) \Longrightarrow (F \vee DC \vee DC'_{inc})(d_1, \dots, d_n, q_{\ell_1}, \dots, q_{\ell_k}). \tag{7}$$

With an argument which is analogous to Section 5.4 the computation of an appropriate function G can be performed by Craig interpolation.

6.5. Effect of constraint minimization

To demonstrate the effect of constraint minimization we use again the dam case study described in Section 7 (with parameters $t = 30$, $d = 50$). The run time for model checking using redundancy removal as presented in the sections before is 62.2 CPU minutes. If we additionally use constraint minimization presented in this section, the run time reduces to 11.7 CPU minutes. Fig. 8 gives the explanation for this effect: Here we compare the number of linear constraints occurring in the predicates which go into the next preimage computation. We can observe that constraint minimization is indeed successful in minimizing the number of these linear constraints. In the beginning, the variants with and without constraint minimization do not differ much, but soon the gap between both methods becomes larger. Since our techniques such as Loos–Weispfenning quantifier elimination (followed by redundancy removal) are rather sensitive to the number of linear constraints in the representations, this explains the gain in performance for constraint minimization.

7. Experimental results

7.1. Experimental setup

To evaluate our approach, we ran experiments on an AMD Opteron with 2.3 GHz and 4 GB RAM for each single experiment. Run times are always given in CPU seconds. The experiments had a timeout of 10,800 CPU seconds (3 h).

We implemented the ideas presented in the sections before in our model checker called FOMC. In FOMC we use two SMT solvers. Yices [11] is used for all equivalence checks and redundancy detection and MathSAT [12] is used for the generation and extraction of conflict clauses depending on constraint variables ("theory lemmas", see Section 5.4). For Craig interpolation based on proofs of unsatisfiability we use a modified (single-threaded) version of the SAT solver Mira [45]. Additionally we use the CLN [46] library to implement exact arithmetic. Inside the Boolean optimization routines the SAT solver MiniSat [47] and the BDD package CUDD [48] are used.

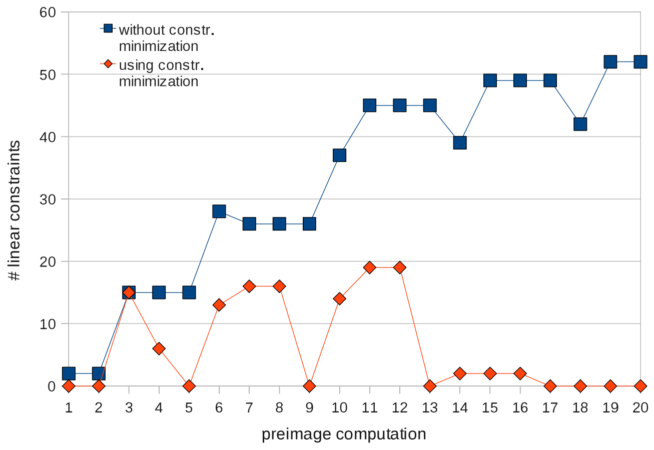


Fig. 8. Evolution of linear constraints with/without constraint minimization.

Table 1
Discrete-time flap controller case study with 3 lever positions.

Configuration	3LP-noHM				3LP-I				3LP-II			
	Runtime	# Steps	Nodes	LC	Runtime	# Steps	Nodes	LC	Runtime	# Steps	Nodes	LC
basic version	TO	(36)			TO	(7)			TO	(5)		
+ implications	3618.4	46	53,601	604	891.3	14	23,231	227	TO	(13)		
+ eager SMT	225.0	46	24,815	332	274.8	14	10,223	156	790.6	14	33,193	334
+ bool. abstr.	38.2	34	12,882	315	273.2	14	10,223	156	802.6	14	33,193	334
+ quant-RR	1264.3	34	3,095	116	17.7	14	1,436	53	22.3	14	1,669	87
+ Craig-RR	126.5	34	4,512	97	12.5	14	1,706	53	18.6	14	1,964	87
+ onion & RR	3.5	34	519	118	8.3	14	2,172	54	15.4	14	1,967	88
+ constr. minim.	6.4	34	562	86	4.7	14	471	53	5.4	14	602	87

7.2. Discrete-time model checking

In a first series of experiments we have a closer look at the discrete-time variant of our model checking algorithm (for dt-LHA+Ds as described in Section 2).

7.2.1. Case study — flap controller

Our first case study is derived from a case study for Airbus, a controller for the flaps of an aircraft. The flaps are extended during take-off and landing to generate more lift at low velocity. As they are not robust enough for high velocity, they must be retracted during other flight phases. It is the controller's task to correct the pilot's commands if he endangers the flaps. Additionally, there is also an extensive monitoring of the health of its sub-systems, checking for instance for hardware failures. The health monitoring system interacts with the flap control by enforcing a more conservative behavior of the control when errors are supposed to be in the system.

The benchmark used here is a simplified version of the full system including the flap controller and a health monitoring system. The model has two continuous variables: the velocity, and the flap angle. Discrete states of the controller and of the health monitoring system contribute to the discrete state space.

The safety property to be established for our model is "For the current flap setting, the aircraft's velocity shall not exceed the nominal velocity (w.r.t. the flap position) plus 7 knots". Whether this requirement holds for our model depends on a "race" between flap retraction and speed increase. The controller is correct, if it initiates flap retraction (by correcting the pilot) early enough.

For the experiments we considered different variants of this case study. We have benchmarks with 3 and 4 lever positions for the flaps (denoted by *3LP* and *4LP*, respectively). Additionally there are simplified versions without the health monitoring system and a simple one-hot encoding of discrete states (*3LP-noHM* and *4LP-noHM*). For the models with health monitoring we considered two variants which differ in the dynamics of the flaps: The first variant (*3LP-I*, *4LP-I*) retracts or increases the flaps with a constant rate. The second variant (*3LP-II*, *4LP-II*) uses a slower rate when the flaps are near to the commanded position.

Both for *4LP-I* and *4LP-II* the discrete state space contains 2^{18} discrete states.

The results of the experiments are given in Tables 1 and 2. All of these models are safe, i.e., the step computation has to iterate until a fixpoint is reached. In the first row of both tables the names of the used models are given. For every benchmark we list the run time given in seconds (*Runtime*), the number of steps until fixpoint or timeout (*# Steps*), the maximum of the number of LinAIG nodes during the complete model checking run (*Nodes*), and the total number of linear constraints created

Table 2
Discrete-time flap controller case study with 4 lever positions.

Configuration	4LP-noHM				4LP-I				4LP-II			
	Runtime	# Steps	Nodes	LC	Runtime	# Steps	Nodes	LC	Runtime	# Steps	Nodes	LC
basic version	TO	(27)			TO	(8)			TO	(5)		
+ implications	TO	(48)			TO	(16)			TO	(11)		
+ eager SMT	10,630.7	86	77,604	480	TO	(22)			TO	(12)		
+ bool. abstr.	122.7	33	24,763	389	TO	(24)			TO	(12)		
+ quant-RR	2,568.9	33	91,525	136	TO	(14)			TO	(12)		
+ Craig-RR	1,500.5	33	20,315	153	2212.3	26	22,349	111	601.7	26	11,896	199
+ onion & RR	5.8	33	1,213	171	9143.3	26	139,055	117	TO	(25)		
+ constr. minim.	16.5	33	1,267	127	16.2	26	1,369	111	19.1	26	1,792	199

during the run (*LC*). When a timeout after 3 hours occurred we denote this with "TO" and omit the numbers for nodes and linear constraints. The step number is then the step in which the timeout occurred.

In the first column different configurations of our model checker with different optimizations are given. First, we used a basic version without strengthening Boolean reasoning by implications between linear constraints (see Section 4.1). An SMT solver was only used during explicit checks like the fixpoint detection. Then we added the usage of implications to SAT checks for equivalence of nodes (Section 4.1.1). The next configuration uses the SMT solver in an eager manner (see Section 4.1.2), i.e., the SMT solver is used to prevent the insertion of LinAIG nodes which are equivalent to already existing nodes (with interpretation of constraint variables by linear constraints). Then we add the usage of the Boolean invariants as described in Section 4.2. In the next configuration we add redundancy elimination (Section 5). Here, we compare two variants: The first one uses quantification to remove redundant constraints (*quant-RR*) and the second uses Craig interpolation (see Section 5.4). Finally, we have two configurations which compute state sets only based on "onion rings" as described in Section 6. The first one uses only redundancy elimination based on Craig interpolation and the second one implements the complete constraint minimization approach (Section 6.2) where redundancy elimination is enhanced by DC-removability.

Our first and simplest configuration ("basic version") is not able to solve any of the benchmarks within the time limit.

When we add the usage of implications to strengthen the SAT solver, FOMC is able to solve two of the simpler benchmark variants (3LP and 3LP-HM-I) within the time limit. However, compared to the following configurations with additional optimizations, the number of created nodes and linear constraints is much higher, because this approach is not able to detect all equivalences between the nodes of the LinAIG data structure.

The picture changes drastically, if we use an SMT solver during node insertions ("eager SMT"). For the two benchmarks solved by the previous configuration, the maximal number of LinAIG nodes and linear constraints is reduced by about one half. Also the run times are reduced to a large extent (e.g., by a factor of 16 for 3LP-noHM). With the "eager SMT" configuration we can solve two more benchmarks (3LP-II and 4LP) within the time limit. The results show that it is worthwhile to invest additional effort by an SMT solver to arrive at more compact representations which can be handled more easily in subsequent model checking steps.

The next optimization we added is the computation of a Boolean abstraction of the forward reachable states. Here one can see that this method is able to reduce the number of necessary steps to compute the fixpoint in two experiments (3LP-noHM and 4LP-noHM). E.g. for the 4LP-noHM model it reduces the number of necessary steps from 86 to 33. In these cases the approximate forward reachability information is effectively used to prune the backward search space w.r.t. states which are definitely not reachable from the initial states. In other experiments we observe that this optimization is not helpful, but at least it does not slow down the run time very much (for 4LP-noHM, e.g., computation of Boolean invariants needs 0.8 CPU seconds).

When redundancy elimination is used, we can observe that the number of linear constraints is reduced by about two-thirds. Comparing redundancy elimination based on quantification ("quant-RR") and redundancy elimination based on Craig interpolation ("Craig-RR") we see that the second version is clearly superior to the first (w.r.t. run time and in most cases also w.r.t. node counts), since it removes a complete set of redundant linear constraints in one step. Redundancy elimination based on Craig interpolation is the first configuration which is able to solve all benchmarks. The results demonstrate that removing redundant linear constraints is very important to get compact representations of the sets of reachable states.

In the last two experiments ("onion & RR" and "constr. minim.") we used a different pre-image computation method which computes pre-images only for "onion rings" of newly reached states. As motivated in Section 6.1 by Example 2, the difference in the pre-image computation alone does not necessarily lead to better results. Thus the configuration "onion & RR" is sometimes faster than "Craig-RR" and uses fewer LinAIG nodes (e.g., for 3LP-noHM and 4LP-noHM), and sometimes it is much worse (e.g., for 4LP-I and 4LP-II). However, especially for the more complex models, the full constraint minimization method using DC-removability clearly outperforms pure redundancy elimination "Craig-RR" with speedups up to 136.[21]

[21] At first sight it may be surprising that the numbers of linear constraints for constraint minimization are not always much smaller than those for other methods such as "Craig-RR". However note that (in contrast to Fig. 8 in Section 6.5) the numbers listed here are the maximal numbers of linear constraints

Table 3
Continuous-time flap controller case study with 3 lever positions.

Configuration	3LP-cont-noHM				3LP-cont-I				3LP-cont-II			
	Runtime	# Lps.	Nodes	LC	Runtime	# Lps.	Nodes	LC	Runtime	# Lps.	Nodes	LC
basic version	TO	(3)			TO	(2)			TO	(2)		
+ implications	TO	(3)			TO	(3)			TO	(2)		
+ eager SMT	7.4	3	779	250	TO	(4)			TO	(3)		
+ bool. abstr.	8.5	3	782	258	TO	(4)			TO	(3)		
+ quant-RR	8.8	3	717	99	408.8	4	9,953	140	TO	(3)		
+ Craig-RR	9.3	3	787	109	418.0	4	10,422	120	492.7	4	9,932	226
+ onion & RR	8.7	3	883	96	290.6	4	11,495	140	1943.1	4	15,914	705
+ constr. minim.	8.5	3	581	85	142.9	4	5,306	106	157.4	4	4,618	218
PHAVer (fwd)	1.2				TO				TO			
PHAVer (bwd)	2.5				TO				TO			
RED (fwd)	TO				TO				TO			
RED (bwd)	0.3				TO				TO			

Altogether, in almost all cases our last version with all the optimizations presented in this paper provides the fastest method with the most compact state set representations. Apparently, it is the most stable method amongst all the other configurations.

7.3. *Continuous-time model checking*

In a second set of experiments we evaluated our approach for two different case studies with continuous evolutions interleaved with sequences of discrete transitions (ct-LHA+Ds as given in Section 2). For these benchmarks we also have built models for PHAVer [7] and RED [22], two state-of-the-art model checkers for hybrid systems with continuous evolutions.[22] In contrast to FOMC, PHAVer uses unions of convex polyhedra to represent the continuous part of the state space and the discrete part is represented by (explicit) locations. By default, PHAVer computes the reachable state space in a forward manner. RED uses so-called HRDs which are BDD-like data-structures and allow symbolic representations of the state space. By default, the traversal of the state space in RED works in a backward manner.

7.3.1. *Case study — flap controller*

The models used here are similar to the models from Section 7.2.1. Minor differences consist in a slightly more complex behavior allowed for the pilot and a slightly different health-monitoring which raises only an alarm if an error is visible for a certain time. The main difference however consists in the movement of the flaps and in the change of velocity: In contrast to the time-discrete modeling used in Section 7.2.1, the system dynamics is described by continuous evolutions here. We denote the continuous-time variants of the benchmarks with *cont* in the benchmark names. Both for the 4LP-cont-I and the 4LP-cont-II benchmark the discrete state space contains 2^{37} discrete states. Again, all the models are safe.

Results for the flap controller case study. We used the same configurations of our model checker with different optimizations as in the discrete-time experiments. The results of this case study are given in Tables 3 and 4. The tables are labeled like in Section 7.2.1 with the only difference that the columns *"# Steps"* are replaced by columns *"# Lps"*. *# Lps* gives the number of loops needed until a fixpoint is detected. As shown in Section 3.2 a single loop consists of a sequence of discrete transitions until a fixpoint is reached, a continuous-to-discrete transition, a continuous flow, and finally a discrete-to-continuous transition. In case of a timeout, *# Lps* gives (in parentheses) the loop in which the timeout occurred.

The last four rows show the run times for PHAVer and RED, where *(fwd)* denotes model checking in the forward direction and *(bwd)* denotes the backward direction, respectively.

The overall picture is similar to the discrete-time experiments. Neither the basic version nor the version with usage of implications were able to solve any of the benchmarks. In contrast to the results from Section 7.2.1, optimization with Boolean abstraction does not reduce the number of needed loops and it does not help for any variant of the models. Using redundancy elimination, the number of used linear constraints is significantly reduced and again, redundancy elimination using Craig interpolation (Craig-RR) is superior to the version using quantification (quant-RR). Comparing rows "onion & RR" and "constr. minim." one can see that the pre-image image computation based on "onion rings" is only helpful in combination with constraint minimization. Altogether, the approach using constraint minimization is clearly the best configuration both in terms of run time and in terms of the representation size.

used during the complete model checking run. For the complexity of the step computation, the numbers of linear constraints in the *state sets used for the current step computation* are much more important and these numbers are indeed much smaller for constraint minimization — leading to much smaller run times.

[22] We used PHAVer 0.38 and RED 8 for our experiments.

Table 4
Continuous-time flap controller case study with 4 lever positions.

Configuration	4LP-cont-noHM				4LP-cont-I				4LP-cont-II			
	Runtime	# Lps.	Nodes	LC	Runtime	# Lps.	Nodes	LC	Runtime	# Lps.	Nodes	LC
basic version	TO	(2)			TO	(2)			TO	(2)		
+ implications	TO	(3)			TO	(2)			TO	(2)		
+ eager SMT	19.8	4	2,985	651	TO	(3)			TO	(2)		
+ bool. abstr.	23.3	4	2,976	652	TO	(3)			TO	(2)		
+ quant-RR	18.5	4	1,524	177	TO	(2)			TO	(2)		
+ Craig-RR	17.0	4	1,475	182	5,821.6	5	31,866	461	TO	(4)		
+ onion & RR	20.9	4	1,741	180	7,668.5	5	39,388	485	TO	(3)		
+ constr. minim.	15.9	4	1,291	163	3,392.0	5	23,750	434	9,165.7	5	46,943	871
PHAVer (fwd)	3.3				TO				TO			
PHAVer (bwd)	6.7				TO				TO			
RED (fwd)	TO				TO				TO			
RED (bwd)	0.3				TO				TO			

Comparison with PHAVer. Compared with our best configuration using constraint minimization, PHAVer is much faster on the simplified benchmarks with a reduced discrete complexity (3LP-cont-noHM, 4LP-cont-noHM). However, PHAVer is not able to solve any of the benchmarks with full complexity (3LP-cont-I, 3LP-cont-II, 4LP-cont-I, and 4LP-cont-II). In contrast, FOMC is able to solve these benchmarks within moderate run times which indicates that our approach can handle discrete complexity much better.

Comparison with RED. The model checker RED shows a similar picture as PHAVer. The simplified models without health-monitoring (3LP-cont-noHM, 4LP-cont-noHM) are solved faster than FOMC and PHAVer when the default backward model checking algorithm is used. On the other hand RED was not able to solve any model using health-monitoring. We also observed that RED used up to 3.6 GB of memory before timeout whereas FOMC never used more than 600 MB. These results show that our sophisticated methods for keeping the state set representations as compact as possible really pay off for the more complex benchmarks.

7.3.2. Case study — dam

In our second case study we consider a dam that impounds water and uses this to produce energy with the help of turbines which the water has to pass. The safety property of this system is that the water level shall always stay within given bounds.

The water level is influenced by an inflow of water towards the dam (with a rate between *min_in* and *max_in*) and by the outflow through 3 turbines which can be controlled separately. The turbines can be switched into three different operating modes with different outflows: In mode *low* a turbine consumes a reduced amount of water per time unit. This is the preferable mode, because it is the most efficient one in terms of energy produced per water unit. In mode *high* a turbine consumes the maximum amount of water it can consume. The mode *maintenance* represents a turbine that is stopped for maintenance reasons. Here, no water passes the turbine.

Each turbine uses a separate continuous timer variable with values between 0 and the constant *duration* for measuring the time until maintenance is finished. Maintenance of a turbine is triggered by a maintenance counter with $\lceil \log(\mathit{threshold}) \rceil$ discrete state bits per turbine. The turbines switch into the maintenance mode after *threshold* changes from the *low* to the *high* mode.

The controller of the system observes the current water level and changes the modes of the turbines. It determines an expected range for the water level depending on the number of turbines in mode *low* and mode *high* and uses a fixed strategy switching turbines into mode *high*, if the water level is higher than expected, and switching turbines into mode *low*, if the water level is lower than expected. The strategy of the controller is "disturbed" by turbines switching into *maintenance* mode.

Given concrete instances of all parameters the verification question is whether the controller is able to keep the water level always within the given bounds.

For our experiments we varied the the parameter *threshold* (t for short) from 10 to 80 and the parameter *duration* (d for short) from 10 to 60. For instance, the model with $t = 10$ and $d = 30$ has 2^{18} discrete states whereas the model with $t = 80$ and $d = 30$ has 2^{27} discrete states. Our results showed that all considered models were safe.

Again we used the same configurations of our model checker with different optimizations as in the previous experiments. To give an overview of all run times we present 3-dimensional charts, where the z-axis represents the run time and the other two axes represent the two parameters d and t, as labeled in the chart.

Results for the dam case study. Altogether, our results for this case study are in line with the previous results:

Both for the basic version of FOMC and the version with usage of implications between linear constraints none of the benchmarks finished within the CPU limit, so we omit figures for these configurations.

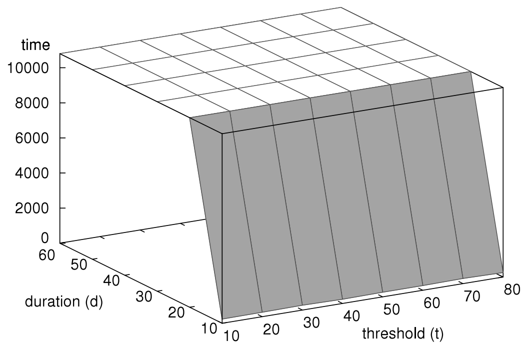


Fig. 9. Dam case study: using eager SMT solving.

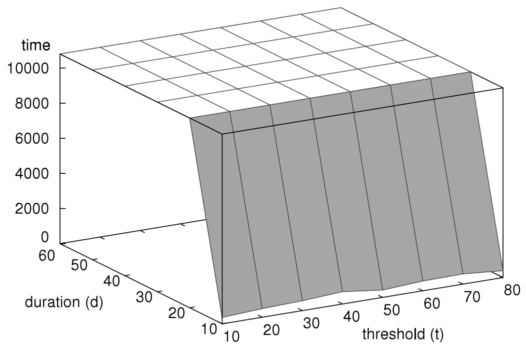


Fig. 10. Dam case study: using boolean invariants.

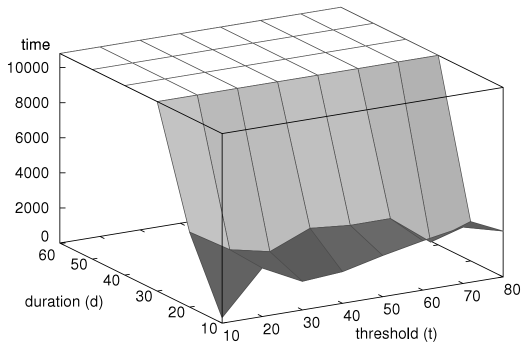


Fig. 11. Dam case study: using redundancy elimination by quantification.

When eager SMT checking is enabled, all benchmarks with $d = 10$ can be solved (see Fig. 9), but increasing the duration parameter leads to timeouts. The variant with additional Boolean invariants shows the same behavior (see Fig. 10).

The run times of our model checker improve when we activate redundancy elimination. Again we observe that the version using Craig interpolation is superior to the version using existential quantification (compare Figs. 11 and 12). The configuration using Craig interpolation is now able to solve all benchmarks with a duration up to $d = 40$. Another interesting observation is that our model checker does not seem to be very sensitive to the value of t, which is influencing the size of discrete state space.

Now we consider the two methods with pre-image computation based on the "onion rings" of newly reached states. As already observed for previous benchmarks, the changed pre-image computation combined only with redundancy removal (without full constraint minimization) does not help much (see Fig. 13). However, as we can see in Fig. 14, run times improve to a great extent for full constraint minimization. The configuration with constraint minimization is the only one which is able to solve all considered benchmarks. As observed before for redundancy elimination, the run time of FOMC is hardly influenced by an increase of the discrete complexity (i.e., by increasing values of t). The maximum run time (among all experiments) of almost 70 CPU minutes occurred for $d = 60$ and $t = 20$. Here the maximum number of active LinAIG nodes was 19,137 and a total number of 1551 linear constraints was generated during the complete model checking run.

Comparison with PHAVer. The results of PHAVer applied to the dam case study are shown in Fig. 15.

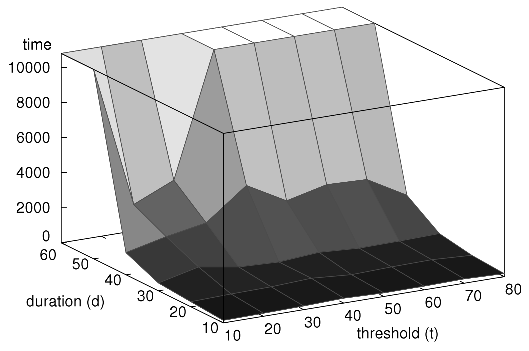


Fig. 12. Dam case study: using redundancy elimination by Craig interpolation.

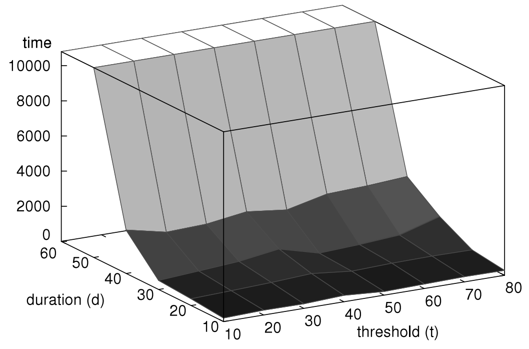


Fig. 13. Dam case study: using onion-slices and redundancy elimination.

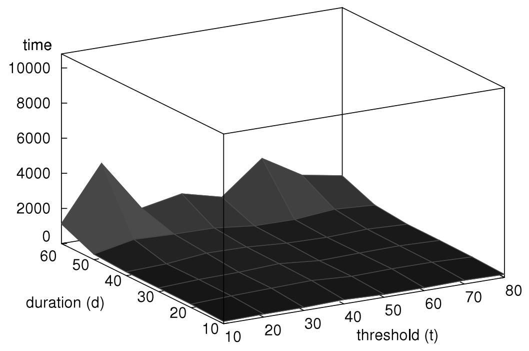


Fig. 14. Dam case study: using constraint minimization.

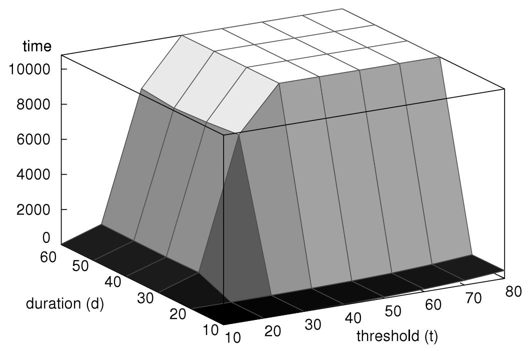


Fig. 15. Run time of PHAVer using encoding I.

When the discrete complexity is low ($t = 10$), PHAVer is faster than FOMC. In this class of model configurations PHAVer is able to solve all benchmarks within 26 s, whereas FOMC (with all optimizations) needs up to 19 min. Here one can see the strength of the PHAVer approach which is able to handle continuous complexity combined with a small discrete complexity

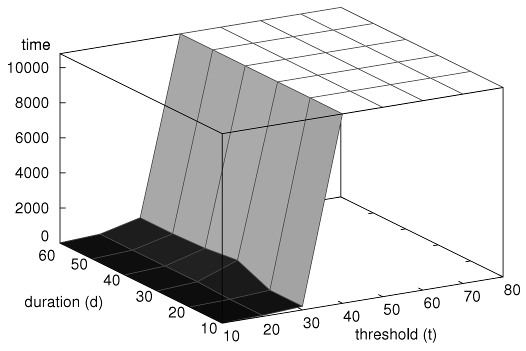


Fig. 16. Run time of PHAVer using encoding II.

well. However, the run times for PHAVer quickly increase when there is a non-trivial discrete complexity combined with a reasonable continuous complexity. Here PHAVer is running into timeouts soon. In contrast, our model checker FOMC is able to solve all instances within 70 CPU minutes (compare Figs. 14 and 15). Concerning the memory consumption we observed that PHAVer is using up to 1.7 GB shortly before a timeout occurs, whereas FOMC uses only up to 228 MB.

In order to obtain a fair comparison for PHAVer we also varied the modeling style: In a first variant (which was presented so far) we modeled the maintenance counters by single variables which remain constant during continuous evolutions (encoding I). In a second variant, the different values for the discrete maintenance counters were represented by different locations (encoding II). Comparing Figs. 15 and 16 we observe that for the first variant strictly more benchmarks can be solved. The second variant suffers from a huge memory consumption due to the large number of explicit locations; we always run out of memory when the value of the threshold t exceeds 30.[23] Of course, the first style of modeling, which avoids large numbers of explicit locations for discrete complexity, is only possible for special cases.

Altogether, these differences of PHAVer and FOMC show that the strengths of the approaches are somehow complementary. PHAVer can handle hybrid state spaces efficiently as long as the discrete part is small, whereas FOMC is tailored towards models with large discrete state spaces due to its fully symbolic approach.

In addition, we used PHAVer with backward model checking for the dam case study. We obtained the result that the backward model checking variant of PHAVer exceeds the time limit for all instances of our parameterized benchmark, even for the simplest ones. So we omitted the figure for this experiment. However, this clearly demonstrates that the results discussed above are not based on the fact that FOMC performs backward model checking and PHAVer performs forward model checking.

Comparison with RED. Finally we applied RED to this class of benchmarks. Again, we used the same benchmark parameters and timeout. Unfortunately, RED was not able to solve any of the benchmarks before the timeout (neither with the backward nor with the forward model checking variant), hence we omitted the diagrams for these results.

8. Conclusion and future work

We presented a method for model checking safety properties of hybrid systems with large discrete state spaces. To compress state set representations our method relies on a complex interaction of a series of different methods which altogether contribute to the overall performance: Simple sub-problems are filtered out and are solved using simulation, purely Boolean reasoning etc., whereas more demanding sub-problems are solved profiting from the strengths of modern SMT solvers. Using novel optimization techniques called redundancy elimination and constraint minimization we are able to handle non-trivial benchmarks which we could not cope with before. Our experiments also demonstrate the advantages of our method compared to hybrid model checkers like PHAVer as soon as the discrete part of the state space is non-trivial. A comparison to RED, which also provides symbolic representations of state sets, again demonstrates the success of our sophisticated methods for compressing state set representations.

Whereas we have still relied on exact state space representations, we plan to extend our optimization methods by incorporating over-approximations. Moreover, we are currently working on supporting richer dynamics.

Acknowledgements

We would like to thank Farn Wang from National Taiwan University for his support regarding our experiments with the RED tool and appreciate his prompt answers to all of our questions.

[23] We set the run time to the timeout value when the memory limit of 4 GB is exceeded.

References

[1] W. Damm, S. Disch, H. Hungar, J. Pang, F. Pigorsch, C. Scholl, U. Waldmann, B. Wirtz, Automatic verification of hybrid systems with large discrete state space, in: ATVA, in: LNCS, vol. 4218, 2006, pp. 276–291.
[2] W. Damm, S. Disch, H. Hungar, S. Jacobs, J. Pang, F. Pigorsch, C. Scholl, U. Waldmann, B. Wirtz, Exact state set representations in the verification of linear hybrid systems with large discrete state space, in: ATVA, in: LNCS, vol. 4762, Springer, 2007, pp. 425–440.
[3] C. Scholl, S. Disch, F. Pigorsch, S. Kupferschmid, Computing optimized representations for non-convex polyhedra by detection and removal of redundant linear constraints, in: TACAS, in: LNCS, vol. 5505, Springer, 2009, pp. 383–397.
[4] M. Segelken, Abstraction and counterexample-guided construction of ω-automata for model checking of step-discrete linear hybrid models, in: CAV, in: LNCS, vol. 4590, Springer, 2007, pp. 433–448.
[5] B.I. Silva, K. Richeson, B.H. Krogh, A. Chutinan, Modeling and verification of hybrid dynamical system using CheckMate, in: 4th Conference on Automation of Mixed Processes, 2000.
[6] T.A. Henzinger, P.-H. Ho, H. Wong-Toi, HyTech: a model checker for hybrid systems, Software Tools for Technology Transfer 1 (1–2) (1997) 110–122.
[7] G. Frehse, PHAVer: algorithmic verification of hybrid systems past HyTech, International Journal on Software Tools for Technology Transfer (STTT) 10 (3) (2008) 263–279.
[8] E. Asarin, T. Dang, O. Maler, The d/dt tool for verification of the hybrid systems, in: 14th Conference on Computer Aided Verification, in: LNCS, vol. 2404, Springer, 2002, pp. 365–370.
[9] K.L. McMillan, Symbolic Model Checking, Kluwer Academic Publishers, 1993.
[10] The VIS Group, VIS: a system for verification and synthesis, in: 8th Conference on Computer Aided Verification, in: LNCS, vol. 1102, Springer, 1996, pp. 428–432.
[11] B. Dutertre, L. de Moura, A fast linear-arithmetic solver for DPLL(T), in: CAV, in: LNCS, vol. 4144, Springer, 2006, pp. 81–94.
[12] R. Bruttomesso, A. Cimatti, A. Franzen, A. Griggio, R. Sebastiani, The MathSAT 4 SMT solver, in: CAV, in: LNCS, Springer, 2008, pp. 299–303.
[13] R. Loos, V. Weispfenning, Applying linear quantifier elimination, The Computer Journal 36 (5) (1993) 450–462.
[14] W. Craig, Three uses of the Herbrand–Gentzen theorem in relating model theory and proof theory, Journal on Symbolic Logic 22 (3) (1957) 269–285.
[15] P. Pudlák, Lower bounds for resolution and cutting plane proofs and monotone computations, Journal on Symbolic Logic 62 (3) (1997) 981–998.
[16] K.L. McMillan, Interpolation and SAT-based model checking, in: CAV, in: LNCS, vol. 2725, Springer, 2003, pp. 1–13.
[17] C.-C. Lee, J.-H.R. Jiang, C.-Y. Huang, A. Mishchenko, Scalable exploration of functional dependency by interpolation and incremental SAT solving, in: ICCAD, 2007, pp. 227–233.
[18] O. Coudert, C. Berthet, J.C. Madre, Verification of synchronous sequential machines based on symbolic execution, in: Automatic Verification Methods for Finite State Systems, in: LNCS, vol. 407, Springer Verlag, 1989, pp. 365–373.
[19] O. Coudert, J.C. Madre, A unified framework for the formal verification of sequential circuits, in: Int'l Conf. on CAD, 1990, pp. 126–129.
[20] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, J. Hwang, Symbolic model checking: 10^{20} states and beyond, in: 5th Annual IEEE Symposium on Logic in Computer Science, IEEE Press, 1990, pp. 428–439.
[21] D. Beyer, A. Noack, Can decision diagrams overcome state space explosion in real-time verification? in: Proceedings of the 23rd IFIP International Conference on Formal Techniques for Networked and Distributed Systems, in: Lecture Notes in Computer Science, vol. 2767, Springer, 2003, pp. 193–208.
[22] F. Wang, Symbolic parametric safety analysis of linear hybrid systems with BDD-like data-structures, IEEE Transactions on Software Engineering 31 (1) (2005) 38–52.
[23] G. Frehse, PHAVer: algorithmic verification of hybrid systems past HyTech, in: HSCC, in: LNCS, vol. 3414, Springer, 2005, pp. 258–273.
[24] G. Frehse, Compositional verification of hybrid systems using simulation relations, Ph.D. Thesis, Radboud Universiteit Nijmegen, 2005.
[25] A. Girard, G.J. Pappas, Approximation metrics for discrete and continuous systems, IEEE Transactions on Automatic Control 52 (5) (2007) 782–798.
[26] W. Damm, G. Pinto, S. Ratschan, Guaranteed termination in the verification of LTL properties of non-linear robust discrete time hybrid systems, in: 3th Symposium on Automated Technology for Verification and Analysis, in: LNCS, vol. 3707, 2005, pp. 99–113.
[27] M. Agrawal, P.S. Thiagarajan, Lazy rectangular hybrid automata, in: 7th Workshop on Hybrid Systems: Computation and Control, in: LNCS, vol. 2993, Springer, 2004, pp. 1–15.
[28] M. Agrawal, P.S. Thiagarajan, The discrete time behavior of lazy linear hybrid automata, in: 8th Workshop on Hybrid Systems: Computation and Control, in: LNCS, vol. 3414, Springer, 2005, pp. 55–69.
[29] A. Platzer, E. Clarke, The image computation problem in hybrid systems model checking, in: 10th Workshop on Hybrid Systems: Computation and Control, in: LNCS, vol. 4416, Springer, 2007, pp. 473–486.
[30] S. Jha, B. Brady, S. Seshia, Symbolic reachability analysis of lazy linear hybrid automata, Tech. Rep., EECS Dept., UC Berkeley, 2007.
[31] E. Asarin, T. Dang, A. Girard, Hybridization methods for the analysis of non-linear systems, Acta Informatica 43 (7) (2007) 451–476.
[32] T.A. Henzinger, The theory of hybrid automata, in: 11th IEEE Symposium on Logic in Computer Science, IEEE Press, 1996, pp. 278–292.
[33] A. Benveniste, G. Berry, The synchronous approach to reactive and real-time systems, Proceedings of the IEEE 79 (9) (1991) 1270–1282.
[34] A. Mishchenko, S. Chatterjee, R. Jiang, R.K. Brayton, FRAIGs: a unifying representation for logic synthesis and verification, Tech. Rep., EECS Dept., UC Berkeley, 2005.
[35] F. Pigorsch, C. Scholl, S. Disch, Advanced unbounded model checking by using AIGs, BDD sweeping and quantifier scheduling, in: FMCAD, IEEE Press, 2006, pp. 89–96.
[36] F. Pigorsch, C. Scholl, Exploiting structure in an AIG based QBF solver, in: Conf. on Design, Automation and Test in Europe, 2009, pp. 1596–1601.
[37] F. Pigorsch, C. Scholl, An AIG-based QBF-solver using SAT for preprocessing, in: Design Automation Conference, 2009.
[38] A. Dolzmann, Algorithmic strategies for applicable real quantifier elimination, Ph.D. Thesis, Universität Passau, 2000.
[39] C.A.R. Hoare, An axiomatic basis for computer programming, Communication of the ACM 12 (1969) 576–583.
[40] R. Alur, T.A. Henzinger, P.-H. Ho, Automatic symbolic verification of embedded systems, IEEE Transactions on Software Engineering 22 (3) (1996) 181–201.
[41] F. Pigorsch, C. Scholl, Using implications for optimizing state set representations of linear hybrid systems, in: GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, 2009, pp. 77–86.
[42] A.V. Aho, M.R. Garey, J.D. Ullman, The transitive reduction of a directed graph, SIAM Journal on Computing 1 (2) (1972) 131–137.
[43] R. Sebastiani, Lazy satisability modulo theories, JSAT 3 (2007) 141–224.
[44] G. Tseitin, On the complexity of derivations in propositional calculus, in: Studies in Constructive Mathematics and Mathematical Logics, 1968.
[45] M. Lewis, T. Schubert, B. Becker, Multithreaded SAT solving, in: 12th Asia and South Pacific Design Automation Conference, 2007, pp. 926–931.
[46] B. Haible, R.B. Kreckel, CLN — class library for numbers, http://www.ginac.de/CLN/.
[47] N. Eén, N. Sörensson, An extensible SAT-solver, in: SAT, in: LNCS, vol. 2919, Springer, 2003, pp. 541–638.
[48] F. Somenzi, CUDD: CU decision diagram package, http://vlsi.colorado.edu/~fabio/CUDD/.