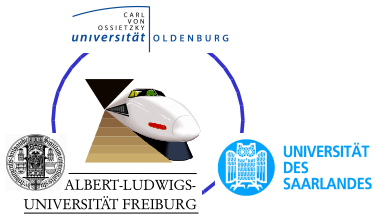


AVACS – Automatic Verification and Analysis of Complex Systems

REPORTS

of SFB/TR 14 AVACS

Editors: Board of SFB/TR 14 AVACS

Verification of Linear Hybrid Systems with Large Discrete State Spaces: Exploring the Design Space for Optimization

by

Ernst Althaus Björn Beber Werner Damm Stefan Disch
Willem Hagemann Astrid Rakow Christoph Scholl
Uwe Waldmann Boris Wirtz

AVACS Technical Report No. 103
April 2016
ISSN: 1860-9821

Publisher: Sonderforschungsbereich/Transregio 14 AVACS
(Automatic Verification and Analysis of Complex Systems)
Editors: Bernd Becker, Werner Damm, Bernd Finkbeiner, Martin Fränzle,
Ernst-Rüdiger Olderog, Andreas Podelski
ATRs (AVACS Technical Reports) are freely downloadable from `www.avacs.org`

Author(s) contact: Werner Damm(`wd@avacs.org`).

Verification of Linear Hybrid Systems with Large Discrete State Spaces: Exploring the Design Space for Optimization

Ernst Althaus[*†] Björn Beber[*†] Werner Damm[‡§] Stefan Disch[¶]
Willem Hagemann[‡] Astrid Rakow[‡] Christoph Scholl[¶] Uwe Waldmann[†]
Boris Wirtz[‡]

April 29, 2016

Abstract

This paper provides a suite of optimization techniques for the verification of safety properties of linear hybrid automata with large discrete state spaces, such as naturally arising when incorporating health state monitoring and degradation levels into the controller design. Such models can – in contrast to purely functional controller models – not analyzed with hybrid verification engines relying on explicit representations of modes, but require fully symbolic representations for both the continuous and discrete part of the state space. The optimization techniques shown yield consistently a speedup of about 20 against previously published results for a similar benchmark suite, and complement these with new results on counterexample guided abstraction refinement. In combination with the methods guaranteeing preciseness of abstractions, this allows to significantly extend the class of models for which safety can be established, covering in particular models with 23 continuous variables and 2^{71} discrete states, 20 continuous variables and 2^{199} discrete states, and 9 continuous variables and 2^{271} discrete states.

Contents

1 Introduction **3**

2 System Model, Basic Data Structures and Model Checking Algorithm **5**
2.1 System Model 5
2.2 Representation of State Sets 8
2.3 Model Checking of Linear Hybrid Automata 9
2.3.1 Step Computation 9
2.3.2 Quantifier Elimination for Linear Real Arithmetic 10
2.3.3 Model Checking Algorithm 11
2.4 Methods for Compact State-Space Representations 13

3 Optimizations of Exact Model Checking **15**
3.1 Optimizing State Sets by Redundancy Removal 15
3.1.1 Motivation 15
3.1.2 Redundancy Detection and Removal for Convex Polyhedra 16
3.1.3 Redundancy Detection for LinAIGs 16
3.1.4 Removal of Redundant Linear Constraints 16

[*]Johannes Gutenberg-Universität Mainz, 55099 Mainz, Germany
[†]Max-Planck-Institut für Informatik, Campus E1.4, 66123 Saarbrücken, Germany
[‡]Carl von Ossietzky Universität Oldenburg, Ammerländer Heerstr. 114–118, 26111 Oldenburg, Germany
[§]OFFIS e.V., Escherweg 2, 26121 Oldenburg, Germany
[¶]Albert-Ludwigs-Universität Freiburg, Georges-Köhler-Allee 51, 79110 Freiburg, Germany

3.2 Optimizing State Space Traversal by Onioning . . . 18
3.2.1 Constraint Minimization . . . 20
3.2.2 Changes in Model Checking Algorithm . . . 21
3.2.3 Per Mode Onioning . . . 21
3.3 Optimized Quantifier Elimination for Linear Real Arithmetic . . . 21
3.3.1 Generalizing Redundancy Removal: Minimizing Polarities . . . 24
3.3.2 Generalizing Constraint Minimization: DC Based Minimization of Polarities . . . 26
3.3.3 Loos–Weispfenning method for several quantifiers of the same type . . . 27
3.3.4 SMT-based Quantifier Elimination . . . 27
3.4 Acceleration at Once . . . 29
3.4.1 Refined System Model . . . 29
3.4.2 Parallel Composition . . . 29
3.4.3 Acceleration at Once . . . 31
3.5 Related Work Exact Model Checking . . . 31

4 Abstractions and Counterexample-Guided Abstraction Refinement 34
4.1 Motivation and Overview . . . 34
4.1.1 A Motivating Example . . . 34
4.2 Model Checking with Abstraction . . . 37
4.3 Abstraction Algorithms for State Set Representations . . . 40
4.3.1 Constraint Minimization Using ϵ-Bloating . . . 40
4.3.2 Proof-based Computation of Simple Interpolants . . . 41
4.3.3 Iterative Computation of Simple Interpolants . . . 46
4.4 Counterexample-Guided Abstraction Refinement . . . 48
4.4.1 Overall Refinement Algorithm . . . 48
4.4.2 Changes in Model Checking Algorithm . . . 49
4.4.3 Learning from Counterexamples . . . 51
4.5 Per Mode Overapproximation . . . 61
4.6 Related Work CEGAR . . . 61

5 Experimental Evaluation 63
5.1 Description of Benchmarks . . . 63
5.2 Evaluation . . . 65
5.2.1 Exact Model Checking . . . 65
5.2.2 CEGAR . . . 77
5.2.3 Scaling to Extremes . . . 86

6 Conclusion 87

1 Introduction

This report describes recent advances achieved in what was originally called *first-order model checking* [1], lifting symbolic model-checking introduced originally by Clarke et al. in the area of hardware verification to the application domain of *models of controllers* of systems.

Controllers are embedded into virtually every system today – in our research we use extensively benchmarks coming from aerospace, automotive, rail, and water-level control in dams. Their common essence is to control a plant, i. e., a physical system, based on observing some part of the plant state through sensors, and to control inputs to the plant by actuators in such a way that overarching safety objectives are guaranteed. To this purpose, control design typically assumes the existence of a mathematical model of the plant, such as given by differential systems of equations. In our work, we use versions of Linear Hybrid Automata [2] as mathematical models of both plants and controllers.

In contrast to most of the research on verifying safety properties of linear hybrid automata, we are interested in models with non-trivial discrete state spaces, which as discussed below naturally arise in later stages of system design. These models explicitly include a set of discrete variables, and allow to take mode switches not only based on observations of the plant, but also based on complex computations in the discrete part, such as coming from dealing with degraded states, redundancy management, monitoring, state information from neighboring controllers, etc. It is worth while to point out the fundamental differences between modes and such discrete variables: modes determine the continuous dynamics of the system, and are – for linear hybrid automata discussed in this paper – thus characterized by differential inclusions defined by linear expressions over derivatives of plant variables. Modes represent knowledge of the state of the plant, such as invariants over plant variables expected to be true in this particular mode. In contrast, discrete variables typically represent knowledge about the health state of the controller itself, as well as information gained about operating modes and health states of other controllers with which it cooperates. Both types of representation of knowledge are fundamental to any realistic system design. We consider thus both modes and discrete variables as first class citizens in our variant of Linear Hybrid Automata, denoted LHA+D, and allow between mode switches an unbounded number of discrete computation steps, updating both continuous and discrete variables, with guards and assignments expressed in linear arithmetic. Examples of such computations are abundant in real system design. Examples include plausibility checks which rely on internal time-discretized approximations of future plant states in order to check for the plausibility of current sensor readings, voting algorithms in redundant systems, resource re-prioritizations due to constraint power budgets, etc.

Overall, the objective of our research is thus to offer a system model which allows succinct representations of such aspects in control design, which invariably are introduced in the transition from pure specification models to design models. This is in contrast with most of the literature on verification of linear hybrid systems, which are not applicable to such models, last not least because they were never designed for coping with the exponential growth in discrete complexity inherent in design models. Such applications are out of reach of existing hybrid verification tools such as HyTech [3], CheckMate [4], d/dt [5], PHAVer [6], SpaceEx [7]: 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 [8] and VIS [9] scale to extremely large discrete systems, but clearly fail to be applicable to systems with continuous dynamics. Only *fully* symbolic verification methods, i. e., those which offer symbolic representations of *both* the continuous *and* the discrete state space, can thus, if at all, cope with safety verification of design models.

This extension from LHA to LHA+D causes a significant increase in complexity. The computational model now evolves along two dimensions of time:

- while being in a mode, the system evolves with physical time, represented by positive real numbers, through flows determined by linear differential inequalities;
- however, between mode switches we can observe arbitrarily complex sequences of discrete computation steps, only advancing the virtual notion of time associated with executing a single such step.

The challenge, then, addressed by our research, is to design a symbolic representation allowing a fully symbolic representation of both the discrete and the continuous state space, and cope with the complexity challenge

both in the continuous and the real dimension. For this, we have proposed LinAIGs [10], i. e., enhanced And-Inverter-Graphs well-known from hardware verification by allowing leaf nodes to represent linear predicates over continuous variables, thus allowing to represent arbitrary disjunctions of convex polyhedra.

In this report we provide a comprehensive overview over techniques developed in the context of the AVACS project to prevent explosion of such symbolic state-space representations when performing backward reachability analysis of LHA+D systems from a given set of unsafe states using LinAIGs. These can be broadly classified in methods preserving preciseness of the representation, and abstraction based approaches.

Fully symbolic model-checking can be seen as performing on-the fly abstraction of the analyzed system in learning, in each iteration step, instances of predicates appearing in the syntactic formulation of the system and its safety properties. During backward analysis along discrete computation steps, such substitution instances are canonically defined from the syntactic description of the system. During backward analysis of flows, we use the Loos–Weispfenning quantifier elimination technique for finding a finite (in fact quadratic) set of test points allowing to reduce existential quantification to a finite disjunction, with substitution instances for the different test points. Since all operations to perform the resulting transformations on LinAIGs, as well as fixed-point detection, can be done using exact arithmetic, such as in exact SMT solvers, this yields a precise abstraction, if the algorithm terminates.

This approach, unfortunately, suffers from two drawbacks:

1. termination is not guaranteed (in fact, because of undecidability of safety properties of LHA, this is not a problem of our approach, but inherent to the safety verification of LHA);

2. the number of substitution instances grows in general exponentially with each backward analysis step.

As to termination, we show that our algorithm yields a semi-decision procedure for the special case of LHA+D, which have the following two characteristics met in any industrial design:

MDT: there is a *minimal dwelling time*, i. e., a constant delta such that whenever a mode is visited, then the LHA stays in this mode for at least delta time units

WCET: there is an upper bound *ub* such that all discrete fixed-point computation stabilize in at most ub discrete steps.

From an industrial perspective, controllers must satisfy both criteria:

- mode switches are typically implemented as task switches, hence a system without MDT would suffer from "flattering", where task switching costs prevent the system to guarantee deadlines of individual tasks

- if such an upper bound would not exist, then the Worst Case Execution Time WCET of the task representing such discrete computations would be unbounded, which is clearly not acceptable in real-time system design.

We develop in Section 3 a range of techniques maintaining preciseness while offering often significant reductions in the symbolic representation of encountered state spaces, as demonstrated by a suite of benchmarks from different applications domains mentioned above, and used throughout the paper. We lift the well-known technique of *onioning* from HW verification to verification of LHA+D. We use SMT solvers to detect redundant linear constraints, and learn interpolants, which allow a succinct representation of the state space when eliminating redundant constraints. We optimize quantifier elimination techniques by polarity detection. We provide two variants of backward reachability analysis of flows, either by factoring for individual modes (targeting single systems, where the number of modes is expected to be small enough), and a technique optimized for parallel composition of independent subsystems (avoiding the state-explosion coming from building the cross-product of their modes).

One of the potential causes for non-termination in safe systems is that our algorithm keeps learning new predicate instances without semantic guidance as to distances to initial states and unsafe states. Such "blind" state-space exploration may thus refine the degree of observation, however, in areas, which are irrelevant for detecting violation of safety properties. Section 4 develops our approach of guided abstractions to overapproximate the state-space, addressing both drawbacks, in learning areas where approximations avoid exploration

of "irrelevant" parts of the state space, and in constructing representations which try to minimize the number of linear constraints needed for representing the approximations. We use incremental SMT solving for testing whether counterexamples found using abstractions are spurious, and develop a range of techniques for learning from spurious counterexamples in subsequent CEGAR loops.

Each of these sections discusses the related state of the art.

These core parts of the paper are complemented by an introductory Section 2 presenting the basic system model and the core model-checking algorithm for LHA+D, and Section 5 giving the experimental results.

Some introductory parts of the paper have been previously published in [2].

2 System Model, Basic Data Structures and Model Checking Algorithm

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 (for simplicity referred to as "linear hybrid automata" (LHA+D) in the sequel). A LHA+D operates on a finite set of continuous and discrete variables and alternates between continuous transitions (flows) and (series of) discrete transitions (jumps). The set of variables is split into four disjoint subsets C, D, I and M. Variables in C are called continuous and range over $\mathbb{R}$. They are used for modeling the dynamics of a plant and represent, e. g., sensor or actuator values or plant states. Variables in $D \cup I \cup M$ are called discrete and range (without loss of generality) over $\mathbb{B} = \{0, 1\}$. They are used to model, e. g., state-machines, counters, or sanity bits. Variables in I are called input variables. Variables in M are called mode variables, they are used to encode a set of boolean vectors $\mathbf{M} \subseteq \{0, 1\}^l$ that correspond to the discrete states of a traditional linear hybrid automaton and determine the possible evolutions of the continuous variables.

Trajectories of the automaton alternate between continuous transitions (flows) and (series of) discrete transitions (jumps). The possible discrete transitions are specified by a finite set DT of guarded assignments of the form

$$\xi \to x_1 := t_1, \ldots, x_n := t_n,$$

where the guard ξ is a boolean expression over the discrete variables and linear constraints over C, the x_i are continuous, discrete, or mode variables, and the t_i are linear expressions over C or boolean expressions as above, respectively. The set DT is split into three disjoint sets DT^d, DT^{d2c}, and DT^{c2d}, where DT^d contains the *purely discrete* transitions, DT^{d2c} 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}, or in other words, that inputs are read only when a trajectory switches from a flow to a jump. 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 (or DT^{d2c} or DT^{c2d}) with $i \neq j$ we have $\xi_i \Rightarrow \neg\xi_j$. 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.

Valuations of the mode variables correspond to discrete states of a traditional linear hybrid automaton: Each *mode*, that is, each element of $\mathbf{M}$ is associated to a linear inequation system $W\vec{x}' \leq w$ that describes the possible derivatives of the evolution of the continuous variables during a continuous transition. The *boundary condition* β_i of a mode $\mathbf{m}_i$ is given by the cofactor of the disjunction of all urgent discrete transition guards from DT^{c2d} w. r. t. $\mathbf{m}_i$, i. e., the partial evaluation of the disjunction w. r. t. $\mathbf{m}_i$. We assume that for each valuation of variables in D, the boundary condition is equivalent to a disjunction of non-strict ($\leq$) linear inequations.

The formula GC, called *global constraint*, is a boolean combinations of variables from $D \cup M$ and linear constraints over C. It is typically used to specify lower and upper bounds for continuous variables in runs to be considered.

A LHA+D can also be accompanied by an *invariant* Inv, which is again a boolean combinations of variables from $D \cup M$ and linear constraints over C. 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.

The following example from [2] illustrates our system model. 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 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 set $D = \{desired\_flappos\}$ and the set $I = \{pilot\_selection\}$. 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

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

  where the variables $deriv\_clock$ and $deriv\_flapangle$ represent the time derivatives of the clock variable $clock$ and the current flap angle $flapangle$, respectively, and $x = c$ is an abbreviation for $x \leq c \wedge x \geq c$.
- 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 pilot's current 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 LHA+D is defined by specifying its trajectories:

Definition 1 *(Semantics of a LHA+D, [2])*

- *A* state *of a 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* $\mathbf{m}^i$*;*
  - $(\mathbf{d}^{i+1}, \mathbf{c}^{i+1}, \mathbf{m}^{i+1}) = (\mathbf{d}^i, \mathbf{c}^i + \int_0^\lambda \mathbf{v}(t)\,\mathrm{d}t, \mathbf{m}^i)$*;*
  - *For every* $0 \leq \lambda' < \lambda$, *the state* $(\mathbf{d}^i, \mathbf{c}^i + \int_0^{\lambda'} \mathbf{v}(t)\,\mathrm{d}t, \mathbf{m}^i)$ *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 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 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.

The LHA+D is safe if there is no trajectory from a given set of initial states (*Init*) to a given set of unsafe states (*Unsafe*) that does not violate the global constraints.

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 [10, 11]. 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. Later on, 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) [12, 13]. 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[1] 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.[2]

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 [13]. Similar techniques have been successfully applied for satisfiability checking of quantified boolean formulas (QSAT), too [14, 15].

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.

[1] 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.

[2] 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.

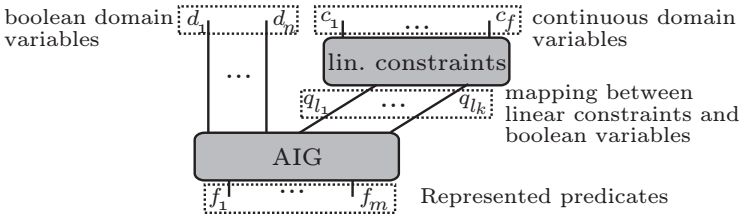


Figure 1: Structure of LinAIGs

2.3 Model Checking of Linear Hybrid Automata

2.3.1 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 [2].[3]

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 [10].

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 $d_j \in D$:

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

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, the constraint variable 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 Sect. 2.1).

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

[3] 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 [16]). By contrast, forward model checking makes use of the discrete image, and computing the latter with a LinAIG representation requires quantifier elimination.

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 $W_i(\mathbf{v}) \Leftrightarrow \bigwedge_j \mathbf{w}_{ij}\mathbf{v} \leq w_{ij}$ 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.[4] 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.[5] We denote this set by $Pre^c(\phi_i, W_i, \beta_i)$. Logically, it can be described by the formula

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

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 $\lambda - \epsilon$, where the formula with ϵ represents the limit for $\epsilon \to +0$. Using the fact that we are only interested in states satisfying GC, the formula can be simplified (modulo GC) to

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

If β_i is a disjunction of linear constraints, one can show that $\neg\beta_i(\mathbf{c}) \wedge \neg\beta_i(\mathbf{c} + (\lambda - \epsilon)\mathbf{v})$ 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 ($\leq$) 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, Henzinger and Ho [17] and replace the product $\lambda\mathbf{v}$ by a new vector $\mathbf{u}$. We obtain:

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

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

It remains to convert this formula over λ, $\mathbf{u} = (u_1, \ldots, u_f)$, 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 described in the next section.

2.3.2 Quantifier Elimination for Linear Real Arithmetic

For the computation of continuous steps based on state set representations given by LinAIGs we need quantifier elimination for linear real arithmetic. For this we use the Loos-Weispfenning test point method [18, 19], which replaces existentially quantified formulas by finite disjunctions using sets of symbolic substitutions.

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

[4] 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.

[5] 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.

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 the usual substitution can easily be extended by these. For instance, if we apply the substitution $[t_j(\vec{y}) - \epsilon/x]$, i. e., x is substituted by $t_j(\vec{y}) - \epsilon$, 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 of ψ for x, depending on whether we consider upper or lower boundaries:

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

Let $TP(\psi(x, \vec{y}), x)$ be the smaller one of the two sets. Then the formula $\exists x.\, \psi(x, \vec{y})$ can be replaced by an equivalent finite disjunction $\bigvee_{\tau \in TP(\psi(x,\vec{y}),x)} \psi(x, \vec{y})[\tau/x]$. 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.

2.3.3 Model Checking Algorithm

Using the pre-image computations described above, we can now define the model checking algorithm, see Alg. 1. 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.[6] Note that discrete transitions form a disjoint and complete case distinction as mentioned in Sect. 2.1 (completeness may be achieved by adding self-loops, if needed). The discrete-to-continuous transitions are disjoint as well, but the case distinction is not complete. Therefore a conjunction with the predicate *d2cTransEnabled* is needed for the evaluation of discrete-to-continuous transitions. *d2cTransEnabled* is true in every state where a discrete-to-continuous transition is enabled. The iteration is performed until a global fixpoint is finally reached or until an initial state is reached.

Model checking and decidability. In Alg. 1 all pre-images computations are performed exactly. Unfortunately, termination of Alg. 1 cannot be guaranteed in general. However, for a large class of industrially relevant models we can show that the algorithm provides a semi-decision procedure: Whenever a model of this class is unsafe the algorithm will detected this in finite time.

This class of models contains all modes for which we can ensure that the successive image computation reaches every instant of time after a finite number of image computations. This properties is satisfied for models (i) having a fixed minimal dwelling time in each mode and (ii) for which the discrete fix point iteration is guaranteed to terminate. Minimal dwelling time is most likely a property of robust models. Moreover, in industrially relevant application we can expect that the worst-case execution time of discrete tasks is known, which implies termination of the discrete fix point computation if modeled correctly.

In [20] it was shown that the model checking problem for a slightly more restricted class, namely the reasonable linear hybrid automata, is indeed decidable. A reasonable linear hybrid automaton is (i) input

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

$\phi_0^{d2c} := \neg safe;\ \phi_0^{d_{fp}} := 0;$
$i := 0;$
repeat
  $i := i + 1;$
  // Discrete fixed point iteration:
  $j := 0;$
  $\phi_0^d := \phi_{i-1}^{d2c} \vee \phi_{i-1}^{d_{fp}};$
  repeat
    $j := j + 1;$
    $\phi_j^d := (Pre^d(\phi_{j-1}^d \wedge GC)) \vee \phi_{i-1}^{d2c};$
  until $GC \wedge \phi_j^d \wedge \neg\phi_{j-1}^d = 0;$
  $\phi_i^{d_{fp}} := \phi_j^d;$
  // Evaluate c2d transitions:
  $\phi_i^{c2d} := \left(\exists d_{n+1}, \ldots d_p(Pre^{c2d}(\phi_i^{d_{fp}} \wedge GC)) \wedge \bigvee_{h=1}^{k}(\beta_h \wedge (\overrightarrow{m} = \mathbf{m}_h))\right) \vee \neg safe;$
  // Evaluate continuous flow:
  $\phi_i^{flow} := \bigvee_{h=1}^{k} Pre^c(\phi_i^{c2d}|_{\overrightarrow{m}=\mathbf{m}_h}, W_h, \beta_h) \wedge (\overrightarrow{m} = \mathbf{m}_h);$
  // Evaluate d2c transitions:
  $\phi_i^{d2c} := (Pre^{d2c}(\phi_i^{flow} \wedge d2cTransEnabled \wedge GC)) \vee \neg safe;$
until $GC \wedge \phi_i^{d2c} \wedge \neg\phi_{i-1}^{d2c} = 0;$
if $GC \wedge (\phi_i^{d_{fp}} \vee \phi_i^{c2d} \vee \phi_i^{flow} \vee \phi_i^{d2c}) \wedge init \neq 0$ **then return** *false*;
return *true*;

Algorithm 1: Backward reachability analysis.

deterministic, i. e., all transitions guards originating from one and the same state are mutually exclusive, and initial states of mode are disjoint, (ii) invariant compatible, i. e., mode invariants are safe and whenever the invariants become false, then there is at least one transition leaving the mode whose guard is enabled, (iii) chatter free, i. e., all transitions enter a mode within its inner envelope, and there is a minimal dwelling time for each mode. In contrast to this result, our algorithm does not guarantee termination on every reasonable hybrid automaton, as the following example, borrowed from [21], shows.

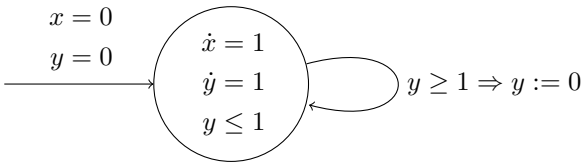


Figure 2: A reasonable linear hybrid automaton

The linear hybrid automaton in Fig. 2 satisfies the criteria for being a reasonable linear hybrid automaton. Since the given automaton consists of exactly one continuous mode we omit the mode variable in the following discussion. Let unsafe be the state $(1.5, 1)$, i. e., the state for which $x = 1.5$ and $y = 1$ holds. The backward reachable states are given by $\{(1.5 - (z + n), 1 - z) \mid z \in [0, 1], n \in \mathbb{N}\}$. Since the initial state $(0, 0)$ is not contained in the set of backward reachable states, the automaton is safe. The hybrid automaton consists of a single c2d-transition and a single continuous flow only. Hence, starting from $\phi_0^{d2c} \equiv \neg safe \equiv \{(1.5, 1)\}$ Alg. 1

yields the sets

$$\phi_1^{d2c} \equiv \{(1.5 - z, 1 - z) \mid z \in [0, \infty]\} \equiv (y = x - 0.5 \wedge x \leq 1.5),$$
$$\phi_2^{d2c} \equiv \{(1.5 - (z + n), 1 - z) \mid z \in [0, \infty], n \in \{0, 1\}\} \equiv (y = x - 0.5 \wedge x \leq 1.5) \vee (y = x + 0.5 \wedge x \leq 0.5),$$
$$\vdots$$
$$\phi_n^{d2c} \equiv \{(1.5 - (z + n), 1 - z) \mid z \in [0, \infty], n \in \{0, n - 1\}\}.$$

Obviously, in every step there are new sets added to the ϕ^{d2c}-predicate and the algorithm will not terminate.

2.4 Methods for Compact State-Space Representations

Here we give a brief review of methods leading to a more compact representation of the LinAIG data structure. Most of them go beyond a separate treatment of the boolean part and the continuous part (linear constraints) [2]. 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. 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.)

The most important methods used in our implementation are:

- Implication-based compaction
- Deciding node equivalences by carefully using decision procedures
- Optimizations using boolean invariants

Implication-based compaction. Dependencies between linear constraints represent information that is easy to detect a priori. We use simple unconditional implications between linear constraints $\alpha_1 c_1 + \ldots + \alpha_n c_n + \alpha_0 \leq 0$ and $\alpha_1 c_1 + \ldots + \alpha_n c_n + \alpha_0' \leq 0$, where $\alpha_0 > \alpha_0'$ as well as implications resulting from global constraints of the form $l_i \leq c_i \leq u_i$ for the continuous variables. If 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). 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.

Deciding node equivalences by using decision procedures. In addition to the eager check for implications between linear constraints above, we use an SMT (SAT modulo theories) solver [22, 23] 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[7] 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). Note that it is also possible to merge nodes which are equivalent "modulo invariants" which have been proved separately (see page 6); details can be found in [2].

It turned out that for our application an *eager* LinAIG compaction is most efficient: Whenever a new node is inserted into the LinAIG, an SMT based node merging is performed, 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. In order to increase the efficiency, we use a layered approach which filters out easy problems, avoiding as many SMT solver calls as possible. The layered approach proceeds as follows:

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.[8] Note that the translation step into a SAT instance includes additional clauses for implications between linear constraints as described before.

   The set of candidate nodes for SAT checks is determined by simulation: 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 this section, 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

[7] In our implementation we use Yices [22] for this task.

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

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 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.

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 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 analysis 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. Forward reachable states $\mathit{fwreach}_{ba}$ of the boolean abstraction of an automaton overapproximate the forward reachable states $\mathit{fwreach}$ of the original automaton. In other words, $\mathit{fwreach}_{ba}$ is a (purely boolean) invariant of the automaton. We compute $\mathit{fwreach}_{ba}$ by a standard symbolic forward model checker for discrete systems [24]. This invariant can then be used to optimize state set representations as described above.

3 Optimizations of Exact Model Checking

3.1 Optimizing State Sets by Redundancy Removal

3.1.1 Motivation

In Sect. 2.2 and Sect. 2.4 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 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 Fig. 3 and 4, which show a typical example from a model checking run representing a small state set based on two real variables: Lines in Figs. 3 and 4 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. 3 contains 24 linear constraints, a closer analysis shows that an optimized representation can be found using only 15 linear constraints as depicted in Fig. 4.

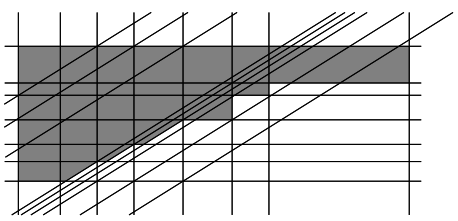


Figure 3: Before redundancy removal

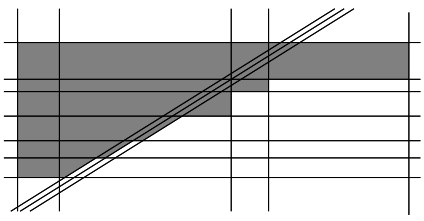


Figure 4: After redundancy removal

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 Sect. 2.3.1). 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.

3.1.2 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 [25, 26], which represent sets of convex polyhedra, i. e., sets of conjunctions $\ell_1 \wedge \ldots \wedge \ell_n$ 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 $\ell_2 \wedge \ldots \wedge \ell_n$ represents the same polyhedron as $\ell_1 \wedge \ldots \wedge \ell_n$, or equivalently, whether $\neg\ell_1 \wedge \ell_2 \wedge \ldots \wedge \ell_n$ represents the empty set. This question can simply be answered by a linear program solver.

3.1.3 Redundancy Detection for LinAIGs

Redundancy of linear constraints is defined as follows [2]:

Definition 2 (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 $\ell_1, \ldots, \ell_r$ $(1 \leq r \leq k)$ 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 $1 \leq i \leq k$ 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 $\ell_1, \ldots, \ell_r$ $(1 \leq r \leq k)$ are* redundant *in the representation of $F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ if and only if the predicate*

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

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 Thm. 1 can be performed by a (conventional) SMT solver (e. g. [22, 23]). A proof for Thm. 1 can be found in [2].

Our overall algorithm for redundancy detection starts by checking redundancy for a single linear constraint and step by step adds further linear constraints to the redundancy check. in that way a maximal set of linear constraints which can be removed from the representation *at the same time* is determined based on iterative applications of Thm. 1. Since the SMT problems occurring in this series of checks are structurally very similar, we make use of an *incremental* SMT solver for the solution, i. e., learned knowledge is transferred from one SMT solver call to the next (by means of learned conflict clauses).

3.1.4 Removal of Redundant Linear Constraints

Suppose that formula (1) of Thm. 1 is unsatisfiable. Now we are looking for an efficient procedure to compute a boolean function G such that $G(d_1, \ldots, d_n, \ell_{r+1}, \ldots, \ell_k)$ and $F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ represent the same predicates. 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 3 *The don't care set dc_{inc} induced by linear constraints $\ell_1, \ldots, \ell_k$ is defined as*

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

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 $\ell_1, \ldots, \ell_r$ 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:[9]

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 $\epsilon_1, \ldots, \epsilon_m$ to constraint variables $q_{\ell_{i_1}}, \ldots, q_{\ell_{i_m}}$ was found to be inconsistent, then the boolean "conflict clause" $(\neg q_{\ell_{i_1}}^{\epsilon_1} \vee \ldots \vee \neg q_{\ell_{i_m}}^{\epsilon_m})$ is added to the set of clauses in the SMT solver to avoid running into the same conflict again.[10] The negation of this conflict clause describes a set of don't cares due to an inconsistency of linear constraints.

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

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

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

If some orbit $\textit{orbit}(v_{d_1}, \ldots, v_{d_n}, v_{\ell_{r+1}}, \ldots, v_{\ell_k})$ 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

[9] Here we refer to the *lazy* approach to SMT solving, see [27], e. g., for an overview.

[10] By definition $q^1_{\ell_{i_j}} := q_{\ell_{i_j}}$ and $q^0_{\ell_{i_j}} := \neg q_{\ell_{i_j}}$.

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 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 $\ell'_1, \ldots, \ell'_k$.[11]

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 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 Def. 2. (This consideration also provides a proof for the "if-part" of Thm. 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. It was shown in [2] that an appropriate representation of such a function G can be computed using Craig interpolation [28, 29, 30].

3.2 Optimizing State Space Traversal by Onioning

According to Sect. 2.3.3 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 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. We start with a simplified version of the fixpoint iteration for explaining the basic ideas. In particular, at first, we assume here a fixpoint iteration just using discrete transitions. The simplified fixpoint iteration is then given by Alg. 2.

```
1 φ_0 := ¬safe; i := 0;
2 if φ_0 ∧ init ≠ 0 then return false;
3 ;
4 repeat
5 |   i := i + 1; φ_i := ∃d_{n+1}, ... d_p Pre^d(φ_{i-1}) ∨ ¬safe;
6 |   if φ_i ∧ init ≠ 0 then return false;
7 until φ_i ∧ ¬φ_{i-1} = 0;
8 return true;
```

Algorithm 2: 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 Alg. 2 we observe that we perform "duplicated work" in each step, since we start the preimage computation in line 5 also from states in ϕ_{i-1} which we were already included in ϕ_{i-2} – and for these states we already performed a preimage computation before.

This observation is not new, it has already been made when symbolic BDD based model checking was introduced for discrete systems in the late eighties [31, 32]. A first idea for solving this problem is just to

[11] 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 $\ell'_1, \ldots, \ell'_k$.

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 Alg. 3.

```
1  φ_0 := ¬safe; ψ_0 := ¬safe; i := 0;
2  if φ_0 ∧ init ≠ 0 then return false;
3  ;
4  repeat
5  |  i := i + 1; ψ_i := ∃d_{n+1}, ... d_p Pre^d(ψ_{i−1});
6  |  ψ_i := ψ_i ∧ ¬φ_{i−1} ;                    // Newly reached states
7  |  if ψ_i ∧ init ≠ 0 then return false;
8  |  φ_i := φ_{i−1} ∨ ψ_i ;                     // All reached states
9  until φ_i ∧ ¬φ_{i−1} = 0;
10 return true;
```

Algorithm 3: 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 less resources while computing preimages. Consider the following example which illustrates this fact:

Example 1 *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 Fig. 5(a). With the assignment $x := x + 1$, $y := y - 1$ the preimage $Pre^d(\phi_0)$ results in

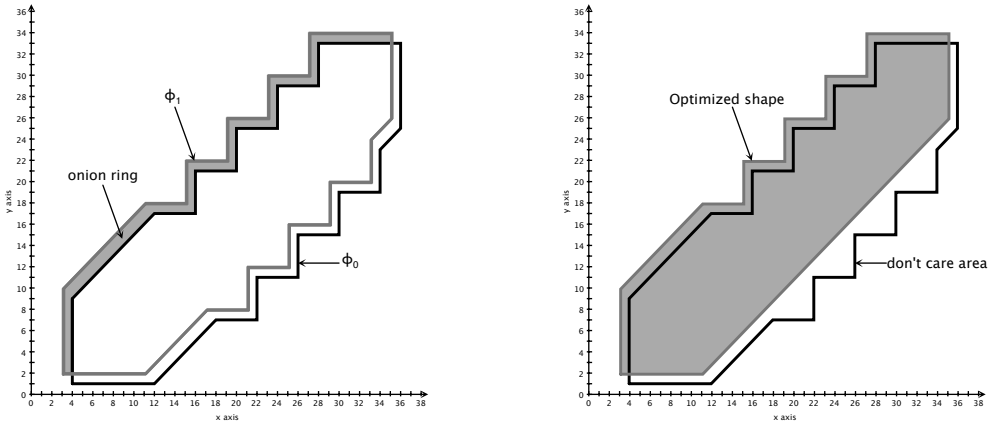
$$\begin{aligned}\phi_1 = &(y > 2) \wedge (y < 34) \wedge (x > 3) \wedge (x < 35) \wedge (y - x < 7)\\ &\wedge [(y < 18) \vee (x > 15)] \wedge [(y < 22) \vee (x > 19)] \wedge [(y < 26) \vee (x > 23)] \wedge [(y < 30) \vee (x > 27)]\\ &\wedge [(x - y < 9) \vee (y > 8) \wedge (x < 21) \vee (y > 12) \wedge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

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 Alg. 3, 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 5 of Alg. 2). 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.[12] *$\phi_1 \wedge \neg\phi_0$ is labeled by "onion ring" in Fig. 5(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 6 of Alg. 3 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 [31, 32] 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 the question what is an appropriate cost measure for optimizing our state set representations which are given by LinAIGs. Here we propose to use the number of linear constraints the representation depends on. As already discussed in Sect. 3.1.1, 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 Alg. 3 where line 6 is replaced by $\psi_i := constraint\_min(\psi_i, \neg\phi_{i-1})$.

[12] 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.



(a) Preimage computation (b) Don't care optimization

Figure 5: Motivating example for constraint minimization

3.2.1 Constraint Minimization

Before we look into the question how to use don't cares in order to minimize the number of linear constraints, we have a look at Ex. 1 again:

Example 2 *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. Fig. 5(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 Sect. 3.1.

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 $\ell_1, \ldots, \ell_r$ $(1 \leq r \leq k)$ of linear constraints which we would like to remove from F using don't care optimization.

Definition 4 *(DC-Removability of linear constraints) The linear constraints $\ell_1, \ldots, \ell_r$ $(1 \leq r \leq k)$ 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.*[13]

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

Theorem 2 (DC-Removability Check) *For all $1 \leq i \leq k$ 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 $\ell_1, \ldots, \ell_r$ $(1 \leq r \leq k)$ 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*

$$\begin{array}{c} F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k) \wedge \neg F(d_1, \ldots, d_n, \ell'_1, \ldots, \ell'_k) \\ \wedge \neg DC(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k) \wedge \neg DC(d_1, \ldots, d_n, \ell'_1, \ldots, \ell'_k) \wedge \bigwedge_{i=r+1}^{k} (\ell_i \equiv \ell'_i) \end{array} \tag{3}$$

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 $c'_1, \ldots, c'_f$.

[13] This means that F and G are the same except for don't cares.

The proof of Thm. 2 can be found in [2].

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 Thm. 2 and makes use of an incremental SMT solver.

Computing an optimized representation. The ideas for a constructive proof for the if-part of Thm. 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 Sect. 3.1.4. (However, in contrast to constraint minimization, redundancy removal does not change represented shapes at all).

We assume that formula (3) from Thm. 2 is unsatisfiable and try to change $F(d_1, \ldots, d_n, q_{\ell_1}, \ldots, q_{\ell_k})$ in a way that the result G will be independent from $q_{\ell_1}, \ldots, q_{\ell_r}$. In addition to the set dc_{inc} of don't cares due to inconsistent assignments to constraint variables (Def. 3) (or the subset $dc'_{inc} \subseteq dc_{inc}$ extracted from an SMT solver checking formula (3)), 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:

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

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

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

$$G(d_1, \ldots, d_n, q_{\ell_{r+1}}, \ldots, q_{\ell_k}) \Longrightarrow (F \vee DC \vee DC'_{inc})(d_1, \ldots, d_n, q_{\ell_1}, \ldots, q_{\ell_k}). \tag{5}$$

With an argument which is analogous to Sect. 3.1.4 the computation of an appropriate function G can be performed by Craig interpolation.

3.2.2 Changes in Model Checking Algorithm

Based on the ideas described above we can define an optimized version of Alg. 1. The algorithm is given by Alg. 4. Since this algorithm is based on optimized "onion rings", we will call it *onion algorithm* in the following. In contrast, the original algorithm from Alg. 1 is called the *non-onion* algorithm.

3.2.3 Per Mode Onioning

There is another way of computing optimized onion rings used in flow steps : Instead of optimizing the complete c2d-onion ring for the previous c2d state set, and then for each mode computing a cofactor of the c2d-onion ring and using it for mode preimage computation, it is possible to compute cofactors of c2d preimage and previous c2d state set for each mode, use them to optimize a mode specific onion ring, and use that for mode preimage computation. This has the advantage that the state sets used for mode preimage computation are optimized specifically for the modes they are used in, which is paid for with a cofactor computation and a constraint minimization for each mode.

The resulting model checking algorithm is called *per mode onion* algorithm, and is given by Alg. 5. It differs to Alg. 4 only in a few lines : The complete c2d-onion ring constraint minimization in Alg. 4, line 14 is no longer necessary, and the mode specific constraint minimization is added (Alg. 5, line 16).

3.3 Optimized Quantifier Elimination for Linear Real Arithmetic

Quantifier elimination for Linear Real Arithmetic turned out to be the crucial and most expensive step in our model checking algorithm. For this reason we look into various approaches for optimizing this step.

$\psi_0^{d2c} := \neg safe;\ \phi_0^{d2c} := \neg safe;\ \phi_0^{c2d} := 0;\ \phi_0^{flow} := 0;\ \phi_0^{d_{fp}} := 0;$
$i = 0;$
repeat
  $i := i + 1;$
  // Discrete fixed point iteration:
  $j := 0;$
  $\psi_0^d := \psi_{i-1}^{d2c}; \quad \phi_0^d = \psi_{i-1}^{d2c} \vee \phi_{i-1}^{d_{fp}};$
  repeat
    $j := j + 1;$
    $\psi_j^d := Pre^d(\psi_{j-1}^d \wedge GC);$
    $\psi_j^d := constraint\_min(\psi_j^d, \phi_{j-1}^d); \quad \phi_j^d := \phi_{j-1}^d \vee \psi_j^d;$
  until $GC \wedge \phi_j^d \wedge \neg\phi_{j-1}^d = 0;$
  $\psi_i^{d_{fp}} := constraint\_min(\phi_j^d, \phi_{i-1}^{d_{fp}}); \quad \phi_i^{d_{fp}} := \phi_j^d;$
  // Evaluate c2d transitions:
  $\psi_i^{c2d} := \exists d_{n+1}, \ldots d_p(Pre^{c2d}(\psi_i^{d_{fp}} \wedge GC)) \wedge \bigvee_{h=1}^k(\beta_h \wedge (\overrightarrow{m} = \mathbf{m}_h));$
  $\psi_i^{c2d} := constraint\_min(\psi_i^{c2d}, \phi_{i-1}^{c2d});$
  if $i = 1$ **then** $\psi_i^{c2d} := \psi_i^{c2d} \vee \neg safe\ ;$
  $\phi_i^{c2d} := \phi_{i-1}^{c2d} \vee \psi_i^{c2d};$
  // Evaluate continuous flow:
  $\psi_i^{flow} := \bigvee_{h=1}^k constraint\_min(Pre^c(\psi_i^{c2d}|_{\overrightarrow{m}=\mathbf{m}_h}, W_h, \beta_h), \phi_{i-1}^{flow}|_{\overrightarrow{m}=\mathbf{m}_h}) \wedge (\overrightarrow{m} = \mathbf{m}_h);$
  $\psi_i^{flow} := constraint\_min(\psi_i^{flow}, \phi_{i-1}^{flow}); \quad \phi_i^{flow} := \phi_{i-1}^{flow} \vee \psi_i^{flow};$
  // Evaluate d2c transitions:
  $\psi_i^{d2c} := Pre^{d2c}(\psi_i^{flow} \wedge d2cTransEnabled \wedge GC);$
  $\psi_i^{d2c} := constraint\_min(\psi_i^{d2c}, \phi_{i-1}^{d2c}); \quad \phi_i^{d2c} := \phi_{i-1}^{d2c} \vee \psi_i^{d2c};$
until $GC \wedge \phi_i^{d2c} \wedge \neg\phi_{i-1}^{d2c} = 0;$
if $GC \wedge (\phi_i^{d_{fp}} \vee \phi_i^{c2d} \vee \phi_i^{flow} \vee \phi_i^{d2c}) \wedge init \neq 0$ **then return** *false*;
return *true*;

Algorithm 4: Backward reachability analysis, onion algorithm.

$\psi_0^{d2c} := \neg safe;\ \phi_0^{d2c} := \neg safe;\ \phi_0^{c2d} := 0;\ \phi_0^{flow} := 0;\ \phi_0^{d_{fp}} := 0;$
$i = 0;$
repeat
  $i := i + 1;$
  // **Discrete fixed point iteration:**
  $j := 0;$
  $\psi_0^d := \psi_{i-1}^{d2c};\quad \phi_0^d = \psi_{i-1}^{d2c} \vee \phi_{i-1}^{d_{fp}};$
  repeat
    $j := j + 1;$
    $\psi_j^d := Pre^d(\psi_{j-1}^d \wedge GC);$
    $\psi_j^d := constraint\_min(\psi_j^d, \phi_{j-1}^d);\quad \phi_j^d := \phi_{j-1}^d \vee \psi_j^d;$
  until $GC \wedge \phi_j^d \wedge \neg\phi_{j-1}^d = 0;$
  $\psi_i^{d_{fp}} := constraint\_min(\phi_j^d, \phi_{i-1}^{d_{fp}});\quad \phi_i^{d_{fp}} := \phi_j^d;$
  // **Evaluate c2d transitions:**
  $\psi_i^{c2d} := \exists d_{n+1}, \ldots d_p(Pre^{c2d}(\psi_i^{d_{fp}} \wedge GC)) \wedge \bigvee_{h=1}^{k}(\beta_h \wedge (\overrightarrow{m} = \mathbf{m}_h));$
  $\psi_i^{c2d} := constraint\_min(\psi_i^{c2d}, \phi_{i-1}^{c2d});$
  if $i = 1$ **then** $\psi_i^{c2d} := \psi_i^{c2d} \vee \neg safe\ ;$
  $\phi_i^{c2d} := \phi_{i-1}^{c2d} \vee \psi_i^{c2d};$
  // **Evaluate continuous flow:**
  $\psi_i^{flow} :=$
  $\bigvee_{h=1}^{k} constraint\_min(Pre^c(constraint\_min(\psi_i^{c2d}|_{\overrightarrow{m}=\mathbf{m}_h}, \phi_{i-1}^{c2d}|_{\overrightarrow{m}=\mathbf{m}_h}), W_h,\ \beta_h), \phi_{i-1}^{flow}|_{\overrightarrow{m}=\mathbf{m}_h}) \wedge$
  $(\overrightarrow{m} = \mathbf{m}_h);$
  $\psi_i^{flow} := constraint\_min(\psi_i^{flow}, \phi_{i-1}^{flow});\quad \phi_i^{flow} := \phi_{i-1}^{flow} \vee \psi_i^{flow};$
  // **Evaluate d2c transitions:**
  $\psi_i^{d2c} := Pre^{d2c}(\psi_i^{flow} \wedge d2cTransEnabled \wedge GC);$
  $\psi_i^{d2c} := constraint\_min(\psi_i^{d2c}, \phi_{i-1}^{d2c});\quad \phi_i^{d2c} := \phi_{i-1}^{d2c} \vee \psi_i^{d2c};$
until $GC \wedge \phi_i^{d2c} \wedge \neg\phi_{i-1}^{d2c} = 0;$
if $GC \wedge (\phi_i^{d_{fp}} \vee \phi_i^{c2d} \vee \phi_i^{flow} \vee \phi_i^{d2c}) \wedge init \neq 0$ **then return** *false*;
return *true*;

Algorithm 5: Backward reachability analysis, per mode onion algorithm.

3.3.1 Generalizing Redundancy Removal: Minimizing Polarities

As already described in Sect. 2.3.2, eliminating existential (or universal) quantifiers for real variables x can be reduced to the application of a certain number of testpoints. The Loos-Weispfenning method used for this assumes that the 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 $\sim_i, \sim'_j \in \{=, \neq, <, \leq, >, \geq\}$. Then two possible sets of test points of ψ for x are computed, the test point set of upper boundaries $TP_1(\psi(x, \vec{y}), x) = \{+\infty\} \cup \{\, t_i(\vec{y}) \mid \sim_i \in \{=, \leq\} \,\} \cup \{\, t_i(\vec{y}) - \epsilon \mid \sim_i \in \{\neq, <\} \,\}$ and the test point set of lower boundaries $TP_2(\psi(x, \vec{y}), x) = \{-\infty\} \cup \{\, t_i(\vec{y}) \mid \sim_i \in \{=, \geq\} \,\} \cup \{\, t_i(\vec{y}) + \epsilon \mid \sim_i \in \{\neq, >\} \,\}$. The set of testpoints $TP(\psi(x, \vec{y}), x)$ is chosen to be the smaller one of the two sets and the existential quantification $\exists x.\, \psi(x, \vec{y})$ is reduced to the equivalent finite disjunction $\bigvee_{\tau \in TP(\psi(x,\vec{y}),x)} \psi(x, \vec{y})[\tau/x]$. It is easy to see that the method does not need to be changed, if the formula contains additional Boolean variables, i. e., if it is a positive Boolean combination of linear constraints and negated or non-negated Boolean variables.

In our model checker predicates are given as a LinAIGs. If negations in the LinAIG are "pushed towards the inputs" by simple transformation rules, then we obtain a representation where the internal nodes represent OR and AND gates and possible negations are restricted to the inputs. Remember that the inputs in turn represent Boolean variables or linear constraints. Of course, negations of linear constraints are again linear constraints, such that this representation can be interpreted as a *positive* Boolean combination of linear constraints and negated or non-negated Boolean variables. The Loos-Weispfenning method can be applied to this transformed representation. It is easy to see that we do not really need to compute the corresponding transformed representation. For the computation of the sets TP_1 and TP_2 it is only needed to compute in which polarity the linear constraints would occur in the transformed representation. Thus, it is enough to traverse the LinAIG and to determine whether a linear constraint ℓ can be reached only via an even number of negations, only via an odd number of negations, or both via an even and an odd number of negations. In the first case, we only need ℓ for computing TP_1 and TP_2, in the second case we only need $\neg\ell$, and in the third case we need both ℓ and $\neg\ell$. In the first two cases, only a single test point results from ℓ (either in TP_1 or TP_2), in the third case two test points result from ℓ, one in TP_1 and one in TP_2.

In Sect. 3.1 we have shown that we can change the representation of a predicate $F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ by removing redundant linear constraints. Removal of redundant linear constraints before quantifier elimination is highly beneficial, since the redundant linear constraints do not contribute to the sets of test points anymore. The observation discussed above suggests a refined method: If a linear constraint ℓ_i occurs in both polarities in a predicate $F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ and ℓ_i is not redundant in F, then we can ask whether there is an equivalent representation for F where ℓ_i occurs only in *one* polarity. If this is the case, one test point can be omitted, either in TP_1 or TP_2. Note that for the computation of test points it is enough to know that such an equivalent representation for F *exists*. It is not needed to compute this equivalent representation before applying quantifier elimination using the Loos-Weispfenning method, since the correctness of the method depends on semantical properties of F, not on syntactic ones.

The question whether there is an equivalent representation for F with ℓ_i occurring only in one polarity can be seen by an analysis of certain monotonicity properties of F.

We start by briefly reviewing known results on monotonicity of Boolean functions.

Definition 5 (Monotonic variables) *Let F be a boolean function over Boolean variables $x_1, \ldots, x_n$. The function $F(x_1, \ldots, x_n)$ is called monotonic in the variables $x_1, \ldots, x_r$ $(1 \leq r \leq n)$ iff the following holds:*
$\forall (v_{x_1}, \ldots, v_{x_n}), (v_{x'_1}, \ldots, v_{x'_r}, v_{x_{r+1}}, \ldots, v_{x_n}) \in \mathbb{B}^n$:
If $(v_{x_1}, \ldots, v_{x_n}) \leq (v_{x'_1}, \ldots, v_{x'_r}, v_{x_{r+1}}, \ldots, v_{x_n})$, then $F(v_{x_1}, \ldots, v_{x_n}) \leq F(v_{x'_1}, \ldots, v_{x'_r}, v_{x_{r+1}}, \ldots, v_{x_n})$.

(The first $\leq$ in the definition means pointwise $\leq$.)

It is well known that monotonic functions can be represented by "monotonic Boolean formulas":

Lemma 1 *Let $F(x_1, \ldots, x_n)$ be a Boolean function which is monotonic in the variables $x_1, \ldots, x_r$. F can be represented by a positive Boolean combination of $x_1, \ldots, x_r$, $x_{r+1}, \neg x_{r+1}, \ldots, x_n, \neg x_n$.*

Proof 1 *Consider an arbitrary DNF for F and assume that the DNF contains a term with a negated variable from $\neg x_1, \ldots, \neg x_r$. Wlog. consider a term*

$$t = \neg x_1 x_2^{\epsilon_2} \ldots x_{r'}^{\epsilon_{r'}} x_{r+1}^{\epsilon_{r_1}} \ldots x_{n'}^{\epsilon_{n'}}$$

($r' \leq r, n' \leq n$).

Here

$$x_i^{\epsilon_i} = \begin{cases} \neg x_i & \text{if } \epsilon_i = 0 \\ x_i & \text{if } \epsilon_i = 1 \end{cases}$$

Then t can be replaced by

$$t' = x_2^{\epsilon_2} \dots x_{r'}^{\epsilon_{r'}} x_{r+1}^{\epsilon_{r_1}} \dots x_{n'}^{\epsilon_{n'}}$$

without changing the represented function. We have to prove that the replacement does not add further satisfying assignments: Consider an arbitrary satisfying assignment $\epsilon = (\epsilon_1, \epsilon_2, \dots, \epsilon_{r'}, \epsilon_{r'+1}, \dots, \epsilon_r, \epsilon_{r+1}, \dots, \epsilon_n)$ for t'. If $\epsilon_1 = 0$, then ϵ is also a satisfying assignment of t and thus of F. If $\epsilon_1 = 1$, then ϵ is a satisfying assignment due to monotonicity: $\epsilon' = (0, \epsilon_2, \dots, \epsilon_{r'}, \epsilon_{r'+1}, \dots, \epsilon_r, \epsilon_{r+1}, \dots, \epsilon_n)$ is a satisfying assignment of t and thus of F. Since $\epsilon' \leq \epsilon$, $F(\epsilon') = 1$, and F monotonic in x_1, we can conclude $F(\epsilon) = 1$.

Based on this argument, all occurrences of negated variable from $\neg x_1, \dots, \neg x_r$ can be removed from the DNF without changing the represented function.

Now we consider Boolean combinations of Boolean variables and linear constraints:

Definition 6 (Monotonicity of linear constraints) *Let F be a Boolean function, let $d_1, \dots, d_n$ be Boolean variables and let $\ell_1, \dots, \ell_k$ be linear constraints over real-valued variables $C = \{c_1, \dots, c_f\}$. $F(d_1, \dots, d_n, \ell_1, \dots, \ell_k)$ is called monotonic in the linear constraints $\ell_1, \dots, \ell_r$ ($1 \leq r \leq k$) iff it holds $\forall (v_{d_1}, \dots, v_{d_n}) \in \mathbb{B}^n$, $\forall \mathbf{v_c}, \mathbf{v'_c} \in \mathbb{R}^f$: If $\forall 1 \leq i \leq r$ $\ell_i(\mathbf{v_c}) \leq \ell_i(\mathbf{v'_c})$, $\forall r+1 \leq j \leq k$ $\ell_j(\mathbf{v_c}) = \ell_j(\mathbf{v'_c})$, then $F(v_{d_1}, \dots, v_{d_n}, \ell_1(\mathbf{v_c}), \dots, \ell_r(\mathbf{v_c}), \ell_{r+1}(\mathbf{v_c}), \dots, \ell_k(\mathbf{v_c})) \leq F(v_{d_1}, \dots, v_{d_n}, \ell_1(\mathbf{v'_c}), \dots, \ell_r(\mathbf{v'_c}), \ell_{r+1}(\mathbf{v'_c}), \dots, \ell_k(\mathbf{v'_c}))$.*

Monotonicity of linear constraints can be easily checked using an SMT solver. For all $1 \leq i \leq k$ let ℓ_i be a linear constraint over real-valued variables $\{c_1, \dots, c_f\}$ and ℓ'_i *exactly the same* linear constraint as ℓ_i, but over a *disjoint copy* $\{c'_1, \dots, c'_f\}$ of the variables. According to Def. 6, F is monotonic in $\ell_1, \dots, \ell_r$, if the following formula is a tautology:

$$\left(\bigwedge_{i=1}^{r} (\neg \ell_i \vee \ell'_i) \wedge \bigwedge_{j=r+1}^{k} (\ell_j \equiv \ell'_j) \right) \Rightarrow$$
$$\left(\neg F(d_1, \dots, d_n, \ell_1, \dots, \ell_r, \ell_{r+1}, \dots, \ell_k) \vee F(d_1, \dots, d_n, \ell'_1, \dots, \ell'_r, \ell'_{r+1}, \dots, \ell'_k) \right)$$

This is equivalent to the unsatisfiability of the following formula:

$$\bigwedge_{i=1}^{r} (\neg \ell_i \vee \ell'_i) \wedge \bigwedge_{j=r+1}^{k} (\ell_j \equiv \ell'_j) \wedge F(d_1, \dots, d_n, \ell_1, \dots, \ell_r, \ell_{r+1}, \dots, \ell_k) \wedge \neg F(d_1, \dots, d_n, \ell'_1, \dots, \ell'_r, \ell'_{r+1}, \dots, \ell'_k)$$

which can be checked using an SMT solver. Thus, we have the following theorem:

Theorem 3 *For all $1 \leq i \leq k$ let ℓ_i be a linear constraint over real-valued variables $\{c_1, \dots, c_f\}$ and ℓ'_i* exactly the same *linear constraint as ℓ_i, but over a* disjoint copy *$\{c'_1, \dots, c'_f\}$ of the variables, $d_1, \dots, d_n$ are Boolean variables. $F(d_1, \dots, d_n, \ell_1, \dots, \ell_k)$ is monotonic in $\ell_1, \dots, \ell_r$ ($1 \leq r \leq k$) iff*

$$\bigwedge_{i=1}^{r} (\neg \ell_i \vee \ell'_i) \wedge \bigwedge_{j=r+1}^{k} (\ell_j \equiv \ell'_j) \wedge F(d_1, \dots, d_n, \ell_1, \dots, \ell_r, \ell_{r+1}, \dots, \ell_k) \wedge \neg F(d_1, \dots, d_n, \ell'_1, \dots, \ell'_r, \ell'_{r+1}, \dots, \ell'_k)$$

is unsatisfiable.

The relation of monotonicity in linear constraints and monotonic Boolean functions is expressed by the following theorem:

Theorem 4 *If $F(d_1, \dots, d_n, \ell_1, \dots, \ell_k)$ is monotonic in $\ell_1, \dots, \ell_r$ ($1 \leq r \leq k$) then there is a Boolean function $G(d_1, \dots, d_n, q_{\ell_1}, \dots, q_{\ell_k})$ with the property that*

1. *G is monotonic in $q_{\ell_1}, \ldots, q_{\ell_r}$ and*

2. *$F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ and $G(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ represent the same predicates.*

Proof 2 *Since $G(d_1, \ldots, d_n, \ell_{r+1}, \ldots, \ell_k)$ and $F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ have to represent the same predicates, the Boolean functions F and G are only allowed to differ on the don't care set dc_{inc} induced by the linear constraints $\ell_1, \ldots, \ell_k$. According to Def. 3, dc_{inc} is defined by*

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

An appropriate function G is computed by making use of the don't care set dc_{inc}. We abbreviate $v := (v_{d_1}, \ldots, v_{d_n}, v_{\ell_1}, \ldots, v_{\ell_k})$, $v' := (v_{d_1}, \ldots, v_{d_n}, v'_{\ell_1}, \ldots, v'_{\ell_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_k})$, and $v'' := (v_{d_1}, \ldots, v_{d_n}, v''_{\ell_1}, \ldots, v''_{\ell_r}, v_{\ell_{r+1}}, \ldots, v_{\ell_k})$. We define G by

$$G(v) = \begin{cases} F(v) & \text{if } v \notin dc_{inc} \\ 0 & \text{if } v \in dc_{inc} \text{ and } \exists v' \notin dc_{inc} \text{ with } v \leq v', F(v') = 0 \\ 1 & \text{otherwise} \end{cases}$$

$F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ and $G(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ represent the same predicates, since the Boolean functions F and G only differ on elements of dc_{inc}, i. e., on elements representing inconsistent assignments of values to linear constraints. It remains to be shown that the Boolean function G is monotonic in $q_{\ell_1}, \ldots, q_{\ell_r}$. Assume that G is not monotonic in $q_{\ell_1}, \ldots, q_{\ell_r}$. Then there exist v and v' such that $v \leq v'$, but $G(v) > G(v')$, i. e., $G(v) = 1$, $G(v') = 0$. We distinguish between different cases for v and v':

1. *$v, v' \notin dc_{inc}$.*
   Then $G(v) = F(v)$ and $G(v') = F(v')$. Moreover, there exist $\mathbf{v_c}, \mathbf{v'_c} \in \mathbb{R}^f$ such that for $1 \leq i \leq r$ $\ell_i(\mathbf{v_c}) = v_{\ell_i}$, $\ell_i(\mathbf{v'_c}) = v'_{\ell_i}$, and $\ell_i(\mathbf{v_c}) \leq \ell_i(\mathbf{v'_c})$, for $r+1 \leq j \leq k$ $\ell_j(\mathbf{v_c}) = \ell_j(\mathbf{v'_c}) = v_{\ell_j}$, but $F(v_{d_1}, \ldots, v_{d_n}, \ell_1(\mathbf{v_c}), \ldots, \ell_r(\mathbf{v_c}), \ell_{r+1}(\mathbf{v_c}), \ldots, \ell_k(\mathbf{v_c})) > F(v_{d_1}, \ldots, v_{d_n}, \ell_1(\mathbf{v'_c}), \ldots, \ell_r(\mathbf{v'_c}), \ell_{r+1}(\mathbf{v'_c}), \ldots, \ell_k(\mathbf{v'_c}))$. This contradicts the monotonicity of $F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ in $\ell_1, \ldots, \ell_r$.

2. *$v \in dc_{inc}$, $v' \notin dc_{inc}$.*
   Then $G(v') = F(v') = 0$. $G(v) = 1$ contradicts the second case of the definition of G.

3. *$v, v' \in dc_{inc}$*
   Since $G(v') = 0$, according to the definition of G there must be a $v'' \notin dc_{inc}$ with $v' \leq v''$, $F(v'') = 0$. $G(v) = 1$ contradicts the second case of the definition of G, since $v \leq v' \leq v''$ and thus $G(v) = 0$.

4. *$v \notin dc_{inc}$, $v' \in dc_{inc}$.*
   Since $G(v') = 0$, there must be a $v'' \notin dc_{inc}$ with $v' \leq v''$, $F(v'') = 0$. Moreover, $G(v) = F(v) = 1$, since $v \notin dc_{inc}$. Thus, there exist $v \notin dc_{inc}$, $v'' \notin dc_{inc}$ with $v \leq v' \leq v''$, $F(v) = 1$, $F(v'') = 0$. As in case 1) this contradicts the This contradicts the monotonicity of $F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ in $\ell_1, \ldots, \ell_r$.

From Thm. 4 and Lemma 1 we can conclude that a predicate $F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ which is monotonic in $\ell_1, \ldots, \ell_r$ $(1 \leq r \leq k)$ has an equivalent representation as a positive Boolean combination of $d_1, \ldots, d_n$, $\ell_1, \ldots, \ell_r, \ell_{r+1}, \neg\ell_{r+1}, \ldots, \ell_n, \neg\ell_n$. For this reason, $\neg\ell_1, \ldots, \neg\ell_r$ do not need to be considered for the computation of the test points.

As already mentioned, for minimizing the number of test points we do not need to compute a Boolean function G according to Thm. 4, but we only need to know whether $F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ is monotonic in $\ell_1, \ldots, \ell_r$. This can be checked using Thm. 3. For constructing sets of monotonic linear constraints step by step, we can make use of incremental SMT solvers as for redundancy detection.

3.3.2 Generalizing Constraint Minimization: DC Based Minimization of Polarities

Just as constraint minimization extends redundancy removal using don't cares, the polarity-based test point minimization from Sect. 3.3.1 can be extended as well. The check for monotonicity in $\ell_1, \ldots, \ell_r$ from Thm. 3

is generalized by using don't cares $DC(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ in a similar manner to the generalization in Thm. 2. In contrast to Sect. 3.3.1, the original predicate $F(d_1, \ldots, d_n, \ell_1, \ldots, \ell_k)$ is semantically *changed* by using don't care conditions. Therefore not only a minimized set of test points has to be computed, but also a corresponding predicate which results from F by exploitation of don't cares. (The Loos-Weispfenning method applied to the original predicate F with the minimized set of test points is not correct in this case.) The computation of the new predicate can be reduced to an appropriate existential quantification or to an application of Craig interpolation according to [30]. Details are omitted here.

3.3.3 Loos–Weispfenning method for several quantifiers of the same type

In Sect. 2.3.2 we have described the Loos-Weispfenning quantifier elimination method for linear real arithmetic. The method eliminates an existentially quantified formula $\exists x.\, \psi(x, \vec{y})$ by a disjunction of instances $\bigvee_{\tau \in TP} \psi(\tau, \vec{y})$, where τ ranges over a finite set TP of test points and the size of TP is proportional to the number of linear constraints in $\psi(x, \vec{y})$. When the formula after the existential quantifier is a disjunction, it is clear that the method profits from moving the existential quantifier inside the disjunction: Performing a quantifier elimination of $(\exists x.\, \psi_1(x, \vec{y})) \vee (\exists x.\, \psi_2(x, \vec{y}))$ results in a significantly smaller formula than performing a quantifier elimination of $\exists x.\, (\psi_1(x, \vec{y}) \vee \psi_2(x, \vec{y}))$. Since the result of a quantifier elimination is again a disjunction, this is even more important when a sequence of several existential quantifiers has to be eliminated, in fact, by moving the outer quantifiers inside whenever the innermost quantifier has been eliminated, one gets a single exponential complexity of the procedure (w. r. t. the number of quantifiers) compared to double exponential for a naïve elimination.

Already in early versions of FOMC, we optionally implemented this special handling of quantified disjunctive formulas. Alas, the option did not pay off in our experiments. As root cause for this behavior we identified our redundancy removal methods. It turned out that redundancy removal often destroys the disjunctive structure of the resulting formulas. Hence, application of redundancy removal inside the removal of several quantifiers is one the one hand desirable to shrink the formula sizes, but on the other hand, it often destroys the disjunctive structure and hence avoids its exploitation. This situation abruptly improved with the availability of polarity minimization. Since polarity minimization reduces the number of test points to the number of test point of an equivalent redundancy-free formula without altering the structure of the given formula, it is often sufficient to perform a final redundancy removal after eliminating the complete sequence of quantifiers.

However, independently of this observation, we implemented a specialized, recursive Loos-Weispfenning quantifier elimination method for several quantifiers that internally preserves and exploits the disjunctive structure of the resulting formulas. Having an enclosed method for elimination of several quantifiers enables us to choose which quantifier to eliminate first. We implemented a simple heuristic which computes the number of test points for each quantifier to be eliminated, and always chooses to eliminate the quantifier with least test points first.

3.3.4 SMT-based Quantifier Elimination

The Loos-Weispfenning test point method uses the equivalence of the existential quantified formula $\exists x.\, \psi(x, \vec{y})$ and the finite disjunction $\bigvee_{\tau \in TP} \psi(\tau, \vec{y})$ of formulas over all test points $\tau \in TP = TP(\psi, x)$. It is an important observation that even for a satisfiable formula $\exists x.\, \psi(x, \vec{y})$ many of the sub-formulas $\psi(\tau, \vec{y})$, $\tau \in TP$, represent the empty set. On the other hand, for any satisfiable formula $\exists x.\, \psi(x, \vec{y})$, there must be at least one test point $\tau \in TP$ such that the corresponding sub-formula $\psi(\tau, \vec{y})$ is not empty. Moreover, $\psi(\tau, \vec{y})$ represents a subset of $\exists x.\, \psi(x, \vec{y})$.

In this section we discuss an SMT-based approach for quantifier elimination. Its main idea is to enumerate all test points τ which lead to non-empty sub-formulas $\psi(\tau, \vec{y})$. It is based on a method which is very similar to a recent technique called *model based projection* [33]. While there are research paper on integrating model based projection into SMT solvers to solve formulas with alternating quantifiers [34, 35], to the best of our knowledge, we are the first which consequently use and evaluate this method for complete existential quantifier elimination on non-convex data structures like LinAIGs.

Using an SMT solver to find non-empty subsets. Let us assume that we already proved the satisfiability of $\exists x.\, \psi(x, \vec{y})$ with an SMT solver. Then we found a satisfying assignment $(u, \vec{v})$ for $(x, \vec{y})$. We show how to identify a test point τ that provides us a non-empty sub-formula $\psi(\tau, \vec{y})$ without any additional SMT solver call.

As a prerequisite we have to recall some properties of test points: For a fixed assignment $\vec{v}'$ of $\vec{y}$ the function $f_{\vec{v}'} : x \mapsto \psi(x, \vec{v}')$ is piecewise constant. A test point symbolically represent an endpoint of an interval over which the function is constant. Since each test point $\tau = \tau(\vec{y})$ is of the form $\pm\infty$, $t(\vec{y})$, or $t(\vec{y}) \pm \epsilon$, it evaluates under the assignment $\vec{v}$ to an extended value $\tau(\vec{v})$ in $\mathbb{R}^*$, where $\mathbb{R}^*$ is $\mathbb{R}$ extended by $\pm\infty$ and addition or subtraction of the infinitesimal ϵ. We expand the usual $<$-ordering to operate on $\mathbb{R}^*$ by setting $-\infty < a < a + \epsilon < b - \epsilon < b < +\infty$ for all $a, b \in \mathbb{R}$ with $a < b$.

Now we show how to identify a test point which leads to a satisfiable sub-formula: For simplicity let us assume that TP contains the lower interval boundaries. Further, let $\tau^* = \tau^*(\vec{y})$ be a test point with $\tau^*(\vec{v}) \leq u$ such that there is no other test point $\tau = \tau(\vec{y})$ with $\tau^*(\vec{v}) < \tau(\vec{v}) \leq u$. Since $f_{\vec{v}}$ is piecewise constant, and since there is no test point which separates c from $\tau^*(\vec{v})$, we have $f_{\vec{v}}(\tau^*(\vec{v})) = f_{\vec{v}}(u) = 1$. Hence τ^* is a test point which leads to the satisfiable sub-formula $\psi(\tau^*, \vec{y})$.

In practice we observe that the SMT solver often returns variable assignments $\vec{v}$ for $\vec{y}$ under which several test points τ_1, τ_2, ..., τ_k coincide with τ^*, i. e., $\tau_1(\vec{v}) = \tau_2(\vec{v}) = \cdots = \tau_k(\vec{v}) = \tau^*(\vec{v})$. In this case all sub-formulas $\phi(\tau_1, \vec{y})$, $\phi(\tau_2, \vec{y})$, $\ldots \phi(\tau_k, \vec{y})$, and $\phi(\tau^*, \vec{y})$ are valid subsets of $\exists x.\, \phi(x, \vec{y})$. Hence, we are free to choose any non-empty disjunctive combination of these sub-formulas as a valid subset of $\exists x.\, \phi(x, \vec{y})$. In the following we discuss to strategies, the *frugal* strategy which choose only one sub-formula nondeterministically, and the *greedy* strategy which takes the disjunction of all these sub-formulas.

The method given above can easily be generalized to formulas also containing Boolean variables. We now discuss the case where we have several nested quantifiers of the same kind: Let $\exists x_m, \ldots x_1.\, \psi(x_1, \ldots, x_m, \vec{y})$ be a satisfiable formula and $(u_1, \ldots, u_m, \vec{v})$ a satisfying assignment of the variables $(x_1, \ldots, x_m, \vec{v})$. We proceed as before: we compute the test points of $\psi(x_1, \ldots, x_m, \vec{y})$ for x_1 and determine a satisfiable sub-formula $\psi(\tau_1^*, x_2, \ldots, x_m, \vec{y})$. Now $\exists x_m, \ldots x_2.\, \psi(\tau_1^*, x_2, \ldots, x_m, \vec{y})$ is a formula that does not contain x_1, represents a non-empty subset of f the original formula, and $(u_2, \ldots, u_m, \vec{v})$ is a satisfying assignment. Hence, we may proceed with partial removal of the next quantifier until no quantifiers are left.

Enumeration of all non-empty test point-generated subsets. So far, we gave a method which allows to generalize the satisfying assignment u and the model $\vec{v}$ of $\exists x.\, \psi(x, \vec{y})$ to a formula describing a subset of models $\psi(\tau^*, \vec{y})$ of $\exists x.\, \psi(x, \vec{y})$ with $\psi(\tau^*, \vec{v}) = 1$. Now we will show how this method can be used to obtain a complete model enumeration of $\exists x.\, \psi(x, \vec{y})$. The idea is quite simple. Initially let $\phi(\vec{y}) = 0$ and $\chi(x, \vec{y}) = \psi(x, \vec{y})$. We use an SMT solver to find a satisfying assignment $(u, \vec{v})$ of $\chi(x, \vec{y})$. We compute the test point set $TP(\psi(x, \vec{y}))$ of $\psi(x, \vec{y})$ for the elimination of x. Then we identify the test points τ^* leading to a satisfiable sub-formula as above, i. e., $\phi(\tau^*, \vec{y})$ is a satisfiable sub-formula of $\exists x.\, \psi(x, \vec{y})$, with $\phi(\tau^*, \vec{v}) = 1$, for each such τ^*. Here we are free to use either the frugal or the greedy strategy. The resulting satisfiable sub-formulas are disjunctively collected in ϕ, i. e., we incrementally assign $\phi(\vec{y}) \vee \psi(\tau^*, \vec{y})$ to $\phi(\vec{y})$. Moreover, we conjunctively add the negations of the resulting sub-formulas to χ, i. e., we incrementally assign $\chi(x, \vec{y}) \wedge \neg\psi(\tau^*, \vec{y})$ to $\chi(x, \vec{y})$. Now we proceed with the satisfiability check of χ as before. The process terminates as soon as χ gets unsatisfiable. Then $\phi(\vec{y})$ is a quantifier-free equivalent of $\exists x.\, \psi(x, \vec{y})$. Alg. 6 shows the pseudo-code of this SMT-based method for quantifier elimination using the frugal strategy.

```
1 φ(y⃗) := false;
2 χ(x, y⃗) := ψ(x, y⃗);
3 while satisfiable(χ(x, y⃗)) do
4 |   (u, v⃗) := findSatisfyingAssignment(χ(x, y⃗));
5 |   TP := computeTestPointSet(ψ(x, y⃗), y);
6 |   τ* := findTestPointForSatisfiableSubFml(TP, u);
7 |   φ(y⃗) := φ(y⃗) ∨ ψ(τ*, y⃗) and χ(x, y⃗) := χ(x, y⃗) ∧ ¬ψ(τ*, y⃗);
8 return φ(y⃗);
```

Algorithm 6: Quantifier elimination by SMT-based model enumeration for single existential quantifier, frugal.

Let us discuss some properties of the SMT-based quantifier elimination.

- $\phi(\vec{y})$ always contain a valid underapproximation of $\exists x.\, \psi(x, \vec{y})$ and all missing states can be found in χ due to the equivalence $\exists x.\, \chi(x, \vec{y}) \equiv \exists x.\, \psi(x, \vec{y}) \wedge \neg\phi(\vec{y})$.
- In each loop, at least one test-point τ^* is identified and the corresponding sub-formula $\psi(\tau^*, \vec{y})$ is guaranteed to contain states which have not been explored in ϕ before.[14]
- Since this algorithm produce a strictly monotonic growing subset of the formulas which would be obtained by the classical Loos-Weispfenning algorithm, the termination of this algorithm is guaranteed.
- Polarity minimization is not needed for this algorithm since only test points are used which lead to a satisfying sub-formula.

Again, this algorithm can easily be extended to work with formulas containing Boolean variables, or having several nested quantifier of the same kind.

3.4 Acceleration at Once

Per mode acceleration (as described in 2.3.3) computes the mode preimage for each global mode separately, and combines them to a global preimage.

If the system is composed of multiple parallel components, each with a number of local modes, then this leads to a large number of mode preimage computations, because each combination of local modes is a valid global mode – in fact the set of global modes is the cross product of the sets of local modes, so the number of global modes grows exponentially with the number of parallel components.

This can be counteracted by doing the image computation without mode splitting : Instead of using a number of (global) mode specific predicates (containing state set cofactors and (global) mode specific derivative and boundary information) and eliminating we build one shared predicate, in which (local) mode specific derivative and boundary information is guarded by the corresponding mode encodings, and eliminate real variables on this single (although bigger) predicate.

3.4.1 Refined System Model

In this section we use a slightly refined system model. It is equivalent to the LHA+D from Sect. 2.1, except that now we explicitly define the linear inequation system describing the possible continuous variable derivatives of modes. We use a finite set GE of guarded evolutions ge_j of the form $\xi_j \to W_j$, where the guard ξ_j is a boolean expression over the mode variables, and W_j is a linear equation system of the form $\vec{x}' \leq w$. The linear inequation system associated with mode $\mathbf{m_i} \in \mathbf{M}$ is the conjunction of the W_j of all $ge_j \in GE$ for which ξ_j holds in $\mathbf{m_i}$.

3.4.2 Parallel Composition

If a system is composed of several parallel subsystems ($S = S_1 || \ldots || S_k$), then the subsystems interact over inputs only, so the output of one subsystem can be the input of another. If such a mapping is given, then the overall system model can be derived out of the component system models.

Since in the system model during a jump all enabled discrete-to-continuous and discrete guarded assignments are considered, it is necessary to ensure that the guarded assignments of a component are only enabled if that component is jumping. This can be achieved by extending each component with a specific jump variable, which is enabled during the jump of that component only.

The extended component S_i^{jump} is the same as S_i, except

- the set of discrete variables is extended with a fresh jump variable : $D_{S_i^{jump}} = D_{S_i} \cup \{jump_i\}$

[14] Note that this clearly holds for the frugal strategy. For non-frugal strategies we can only ensure that the disjunction $\bigvee_{\tau^*} \psi(\tau^*, \vec{y})$ over all suitable test points contains states which have not been explored in ϕ so far.

- each continuous-to-discrete guarded assignment is extended, such that its assignment enables the jump variable :
  $DT^{c2d}_{S_i^{jump}} = \{(\xi \wedge jump_i \rightarrow x_1 := t_1, \ldots, x_n := t_n, jump_i := 1) | (\xi \rightarrow x_1 := t_1, \ldots, x_n := t_n) \in DT^{c2d}_{S_i}\}$
- each discrete guarded assignment is extended, such that its guard depends on the enabled jump variable :
  $DT^{d}_{S_i^{jump}} = \{(\xi \wedge jump_i \rightarrow x_1 := t_1, \ldots, x_n := t_n) | (\xi \rightarrow x_1 := t_1, \ldots, x_n := t_n) \in DT^{d}_{S_i}\}$
- each discrete-to-continuous guarded assignment is extended, such that its guard depends on the enabled jump variable, and its assignment disables the jump variable :
  $DT^{d2c}_{S_i^{jump}} = \{(\xi \wedge jump_i \rightarrow x_1 := t_1, \ldots, x_n := t_n, jump_i := 0) | (\xi \rightarrow x_1 := t_1, \ldots, x_n := t_n) \in DT^{d2c}_{S_i}\}$
- each guarded evolution is extended, such that its guards depends on the disabled jump variable :
  $GE_{S_i^{jump}} = \{(\xi \wedge \neg jump_i \rightarrow W) | (\xi \rightarrow W) \in GE_{S_i}\}$

With the extended component system models and the input mapping $IMap$, the overall system model can be computed out of the component system models as follows :

1. The overall system contains all continuous variables of the components :
   $C_S = \bigcup_{i \in \{1 \ldots k\}} C_{S_i^{jump}}$
2. The overall system input variables are those component input variables, which are not outputs of another component :
   $I_S = (\bigcup_{i \in \{1 \ldots k\}} I_{S_i^{jump}}) / dom(IMap)$
3. The other overall system discrete variables are the discrete variables of the components :
   $D_S = \bigcup_{i \in \{1 \ldots k\}} D_{S_i^{jump}}$
   $M_S = \bigcup_{i \in \{1 \ldots k\}} M_{S_i^{jump}}$
4. The overall system modes are all possible combinations of the component modes :
   $\mathbf{M}_S = \{\mathbf{m}_1. \ \ldots \ .\mathbf{m}_k | \bigwedge_{i \in \{1 \ldots k\}} (\mathbf{m}_i \in \mathbf{M}_{S_i^{jump}})\}$
5. The overall continuous-to-discrete guarded assignments are the continuous-to-discrete guarded assignments of the components, with the component inputs linked to other component variables renamed :
   $DT^{c2d}_S = (\bigcup_{i \in \{1 \ldots k\}} DT^{c2d}_{S_i^{jump}}[i / IMap(i)]_{i \in dom(IMap)}$
6. The other overall system guarded assignments and are those of the component systems :
   $DT^{d}_S = \bigcup_{i \in \{1 \ldots k\}} DT^{d}_{S_i^{jump}}$
   $DT^{d2c}_S = \bigcup_{i \in \{1 \ldots k\}} DT^{d2c}_{S_i^{jump}}$
7. The overall system guarded evolutions are those of the component systems :
   $GE_S = \bigcup_{i \in \{1 \ldots k\}} GE_{S_i^{jump}}$
8. The overalls system initial states is the combination of initial states of the component systems :
   $Init_S = \bigwedge_{i \in \{1 \ldots k\}} Init_{S_i^{jump}}$
9. The overalls system global constraints are the combination of global constraints of the component systems :
   $GC_S = \bigwedge_{i \in \{1 \ldots k\}} GC_{S_i^{jump}}$
10. The overalls system invariants are the combination of invariants of the component systems :
    $Inv_S = \bigwedge_{i \in \{1 \ldots k\}} Inv_{S_i^{jump}}$

3.4.3 Acceleration at Once

Instead of using the per mode preimage $Pre^c_{mode}(\phi_i, W_i, \beta_i)$ for continuous preimage computation

$$\psi_i^{flow} := \bigvee_{h=1}^{k} constraint\_min(Pre^c(\psi_i^{c2d}|_{\overrightarrow{m}=\mathbf{m}_h}, W_h, \beta_h), \phi_{i-1}^{flow}|_{\overrightarrow{m}=\mathbf{m}_h}) \wedge (\overrightarrow{m} = \mathbf{m}_h)$$

with

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

(with $W_i'(\mathbf{u}, \lambda) = \bigwedge_j \mathbf{w}_{ij}\mathbf{u} \leq w_{ij}\lambda$) in the model checking algorithm with explicit iteration over all modes, we use the combined continuous preimage computation

$$\psi_i^{flow} := constraint\_min(Pre^c_{aao}(\psi^{c2d}, W_{aao}, \beta_{aao}), \phi_{i-1}^{flow})$$

whereby

- W_{aao} is a predicate linking mode variables to the linear derivative inequation systems for the modes. For this a conjunction based on the guarded evolutions of the component is used:
  $W_{aao} = \bigwedge_{(\xi_i \to W_i) \in GE_S} \xi_i \implies W_i$
- β_{aao} is a predicate linking mode variables to the mode boundaries. For this a disjunction based on the continuous-to-discrete urgent transition guards of the component is used:
  $\beta_{aao} = \bigwedge_{(\xi_i \to \ldots) \in urgent(DT_S^{c2d})} \xi_i$
  whereby $urgent(DT_S^{c2d})$ is the set of urgent guarded assignments in DT_S^{c2d}.
- Pre^c_{aao} is exactly as Pre^c, just that instead of mode specific W and β the global W_{aao} and β_{aao} are used :
  $Pre^c_{aao}(\phi_i, W_{aao}, \beta_{aao}) = \phi_i(\mathbf{c}) \vee \exists\lambda.\, \lambda > 0 \wedge \exists\mathbf{u}.\, W'_{aao}(\mathbf{u}, \lambda) \wedge \phi_i(\mathbf{c}+\mathbf{u}) \wedge GC(\mathbf{c}+\mathbf{u}) \wedge \beta_{aao}(\mathbf{c}) \wedge \beta'_{aao}(\mathbf{c}+\mathbf{u})$
  The mode variables in W_{aao} and β_{aao} ensure that boundaries and derivatives are linked to their corresponding modes.

The combined preimage computation puts more workload on the quantifier elimination, since the complete onion ring is used (compared to a cofactor in per mode acceleration), and since W_{aao} and β_{aao} are bigger, but the variables are quantified only once per acceleration and not once per mode.

3.5 Related Work Exact Model Checking

On the decidability of safety properties. In the last years, a considerable amount of work has been dedicated to identifying classes of hybrid automata for which checking safety is decidable. Reachability and safety in LHA are in general undecidable, while invariant checking and bounded reachability are decidable. One of the first decidability results for reachability was for initialized rectangular hybrid automata [21], a restricted class of rectangular hybrid automata (i. e., hybrid automata for which at each control location the flow is described by differential inclusions of the form $\dot{x} \in [a, b]$) which require resets for continuous variables upon mode transitions, unless the newly entered and left mode share the same dynamics. These results have been extended in [36] which identifies classes of LHA for which reachability is decidable and in [37] which uses translations of verification problems into satisfiability problems for theories of real numbers (possibly with exponentiation). Decidability of *finite-precision* hybrid automata was studied in [38]. In [39, 40, 41], o-minimal hybrid systems are studied; it is shown that for such hybrid systems reachability can be reduced to reachability in finite models due to existence of finite bisimulations – this is used to give decidability results. The restrictions imposed when defining o-minimal hybrid systems – in particular the requirement that only constant resets are allowed – are quite severe.

In contrast, [20] imposes restrictions on linear hybrid automata (mainly invariant compatibility and chatter freedom) which are much more likely to be met in application. It improves on previous work by Wang [42] in that their approach gives an efficient decision procedure, while his back-reachability based symbolic execution approach is not guaranteed to terminate.

Model checking with exact image computation. Some classes of hybrid automata, like linear hybrid automata, allow an exact image computation, i. e., given a symbolic representation ϕ of a set of states, its post-images under continuous flows and discrete jumps can be computed exactly. Moreover, if the underlying theory admits quantifier elimination, then also the pre-image computation can be computed exactly using the identity $Pre(\phi) = \{\vec{x} \mid \exists \vec{y}.\ \phi(\vec{y}) \wedge \vec{x} = Post(\vec{y})\}$. Typical algorithms for this class of hybrid automata perform a reachability analysis starting from the initial states (or unsafe states in case of backward computation) and successively apply the alternating sequence of continuous and discrete post-image computations (or pre-image computations, respectively) until either an intersection with the unsafe states (or initial states, respectively) is detected or a fixpoint is reached. Due to the undecidablity of checking safety of LHA, these algorithms do not necessarily terminate. But under certain conditions they turn out to be at least semi-decision procedures: If we can ensure that the successive image computation reaches every instant of time after a finite number of image computations, then we will eventually detect whether the model is unsafe. For LHAs without purely discrete steps it suffices to postulate a fixed minimal dwelling time such that it is guaranteed that the state set exploration advances a fixed amount of time units with each continuous image computation. In [20] it has been shown that it is decidable for LHA whether a minimal dwelling time exists or not. For LHA+D we additionally have to stipulate that the discrete fixpoint loop terminates. This is trivially the case for models which do not have any purely discrete steps[15]. In industrially relevant application we can expect that the worst-case execution time of discrete tasks is known, which implies termination of the discrete fixpoint computation if modeled correctly. In [21] it was shown that – with the help of a preceding model transformation – this semi-decision procedure can be extended to a decision procedure for certain sub-classes of rectangular hybrid automata.

The pioneer in this area probably is HyTech [17]. HyTech uses disjunctions of convex polyhedra for state set representation. While in principle all basic image computations could be done exactly, HyTech internally uses limited-precision arithmetic. In order to overcome this and some other limitations PHAVer [43] was developed. PHAVer uses exact arithmetic and the Parma Polyhedral Library for image computation and other manipulation of convex polytopes. Meanwhile, the reachable set computation of PHAVer has been integrated into SpaceEx, a tool for reachability analysis mainly relying on numerical integration. Another exact approach is RED [42]. Internally RED uses a BDD-like representation of state set and Fourier-Motzkin quantifier elimination for image computations. While FOMC uses and directly operates on a symbolic representation which allows arbitrary combination of Boolean variables and linear constraints, the tools mentioned above rely on disjunctions of convex polyhedra and reduce state set manipulation to manipulations of convex polyhedra. For models with a non-trivial discrete complexity, the prime limitation of this tools is state-space explosion due to the number of discrete states.

The extent of model checking with exact image computation. Let us shortly discuss the extent of model checking with exact image computation. So far, we restricted ourselves to linear hybrid automata and the underlying mathematical theory of linear real arithmetic, where image computations can be reduced to linear real quantifier elimination. As shown by Tarski, the theory of reals with addition *and multiplication* also admits quantifier elimination. Hence, exact model checking is also possible for a larger class of hybrid automata, like polynomial hybrid automata [44]. To the best of our knowledge, in practice there has only been little success in extending exact model checking beyond linear hybrid automata. However, in 2005 the topic was picked up by the biological community. In the second paper [45] of a series of papers under the title "Algorithmic Algebraic Model Checking" [46, 45, 47, 48] the model-checker Tolque for semi-algebraic hybrid automata was presented. In their conclusion the authors wrote:

> The real limitation of this quantifier-elimination-based model-checking comes from the computational complexity of Collins' cylindrical algebraic decomposition (CAD) algorithm, with its double-exponential dependence on the number of variables []. In our experience, Qepcad failed to support fully symbolic analysis of the two-cell Delta-Notch system.

Beyond exact image computation. Abandoning exact image computation not only offers the possibility to use fast but imprecise numerical methods, but also opens the perspective to a rich variety of industrially

[15] Note that c2d and d2c transitions are not affected by this restriction.

relevant hybrid automata like hybrid automata with continuous dynamics described by linear ordinary differential equations (ODEs) or non-linear ODEs, possibly subject to bounded disturbances.

Flowpipe based approaches. A basic technique to solve the reachability problem for these classes of hybrid systems is the flowpipe computation. Its basic idea is quite simple and is for forward and backward computations almost the same, it differs only by the sign of the time parameter and the role of exit and entry conditions: In the first step an initial geometric or symbolic representation of a set containing all reachable states within a small time interval of length δ is computed. Proposed representations are convex polytopes, zonotopes, ellipsoids, support functions, or Taylor models ([49, 50]). The initial set is then incrementally shifted forward in time (or backward, respectively) by δ time units until it completely leaves the mode boundaries. Incremental flowpipe computation yields a safe overapproximation of the reachable states and its precision and efficiency can be adjusted by varying δ. However, the details of the flowpipe computation are challenging and highly depend on the choice of the state set representation, especially if we respect the influences of mode invariants and bounded disturbances of the ODE.

As before, a minimal dwelling time ensures time progress such that for any instant of time an overapproximation of the reachable states is computed. These algorithms do not provide a semi-decision procedure, since they could detect non-existent intersections with *Unsafe*, due to overapproximations.

Tools following the flowpipe approach are, among others, SpaceEx and Flow*. SpaceEx [7] directly supports high-dimensional linear hybrid systems with continuous dynamics given by linear ODE with bounded disturbances. It includes the old analysis engine from PHAVer, and offers different support function-based reachability analysis engines. SpaceEx is very efficient, but due to inherent restrictions of support functions it suffers from the weak handling of guard intersections and invariants, e. g., the influence of invariants are only taken into account for the current flow segment, but are not carried over to the next flow segment. Moreover, due to the usage of floating-point arithmetic it does not formally guarantee soundness. Flow* [51] is a model checker for hybrid systems whose transition guards and invariants are given by polynomial inequalities. The continuous dynamic can be defined by non-linear ODEs. Internally, it utilizes Taylor model integration for flowpipe construction. Given a proper parameter setting, Flow* shows a good scalability on non-linear case studies. Since the tool focuses on non-linear systems, its weakness are handling convex guards and invariants [52].

We also experimented with flowpipe computation and prototypically implemented an incremental flowpipe computation for linear ODEs using a combination of polytopes and zonotopes [53]. In backward model checking the pre-images of *Unsafe* are computed. Our experiences showed that these sets are often large and far from the origin. Especially for zonotopes, which represent highly symmetric shapes, this led to huge overapproximation during the computation of the first flowpipe segment. As a possible solution we developed an alternative representation for convex polyhedral sets, the symbolic orthogonal projections (sops). Many geometric operations including convex hulls, Minkowski sums, intersections, and affine transformations can be performed exactly and efficiently on sops. We implemented a self-contained tool for reachability analysis using symbolical orthogonal projections, called SoapBox [54]. Integration of the sop-based flowpipe computations is regarded as future work.

Satisfiability based approaches. The bounded reachability problem, i. e., the question whether certain states can be reached from the initial states, can be formulated as a satisfiability problem. However, for certain classes of hybrid automata, e. g., reasonable linear hybrid automata, it is possible to reduce the unbounded reachability problem to satisfiability problems very similar to bounded model checking [20]. Internally, such tools use combinations of SMT-solvers and ODE-solvers and often utilize fast but incomplete interval constraint propagation. Representatives for this kind of model checkers are HSolver, iSAT-ODE, and dReach. The latter two tools are restricted to bounded model checking only.

In contrast to flowpipe-based approaches, these tools do not ensure an incremental progress in time. The run-time behavior of these tools is often hard to predict. dReach [55] is a bounded model checker for hybrid systems with dynamics given by non-linear ODEs. It unrolls the reachability problem for a hybrid automaton to an SMT formula ϕ of bounded length, and uses the δ-complete decision procedure of the SMT-solver dReal to decide whether its numerical δ-perturbation ϕ^δ is satisfiable or not. iSAT-ODE [56] performs bounded model checking of hybrid systems having dynamics described by non-linear ODEs. It combines

the SMT-solver iSAT with the ODE-solver VNODE-LP. HSolver [57] is an extension of RSolver towards unbounded model checking of hybrid systems with non-linear ODEs. Internally, it uses interval constraint propagation based on floating-point arithmetic with sound rounding. Moreover, HSolver is able to compute abstractions of the input system.

Lack of explicit support of discrete states. Let us note that most mentioned tools have no special support for large discrete state sets. The popular system model for hybrid automata as given by Henzinger [25] only uses real variables plus a mode / location variable which can take discrete values in a finite set and encodes the mode of the associated states. Hence, according to his definition a hybrid automaton mainly consists of continuous systems which are connected by single discrete transitions. Consequently, tools like HyTech and PHAVer and also flowpipe computation-based tools like SpaceEx and Flow* use a disjunction of symbolic states, where each symbolic state represents a set together with a mode information.

On the other hand, satisfiability-based tools use logical state set representations, and, hence, could directly support Boolean variables. However, while iSAT-ODE and dReach support Boolean variables in their input language, HSolver lacks support of Boolean variables.

4 Abstractions and Counterexample-Guided Abstraction Refinement

4.1 Motivation and Overview

In spite of all our optimization techniques presented in Sect. 3, for large real-world examples our state set representations can become large and complicated, since they have to contain a large number of linear constraints in order to describe the border of the (backwards) reachable state space precisely. Therefore we introduce here overapproximations whenever the verification approach suffers from complicated state set representations. These overapproximations approximate the state set representations using a "smoother shape" with less linear constraints. If overapproximations lead to spurious counterexamples, the approximation is refined using the well-known CEGAR paradigm (counterexample-guided abstraction refinement). In that way the approach automatically adapts to the difficulty of the verification problem. It works with higher precision in those parts of the state space which need to be inspected more closely for a successful proof, but uses rough overapproximations in other parts.

4.1.1 A Motivating Example

Here we present a small example which demonstrates problems of our model checking algorithm and shows how these problems can be resolved using overapproximations and refinements with a CEGAR approach.

Example 3 *Here we consider a small toy example with two continuous variables x and y and 4 modes with two mode variables q_1, q_0. Each mode is connected to exactly one quadrant in the coordinate system.*

- *Mode $(q_1, q_0) = (0, 0)$ defines the dynamics in the first quadrant of the coordinate system with $x \geq 0$, $y \geq 0$ by the derivatives $\dot{x} = -1$, $\dot{y} = 1$.*
- *Mode $(q_1, q_0) = (0, 1)$ defines the dynamics in the second quadrant of the coordinate system with $x \leq 0$, $y \geq 0$ by the derivatives $\dot{x} = -1$, $\dot{y} = -1$,*
- *Mode $(q_1, q_0) = (1, 0)$ defines the dynamics in the third quadrant of the coordinate system with $x \leq 0$, $y \leq 0$ by the derivatives $\dot{x} = 1$, $\dot{y} = -1$.*
- *Mode $(q_1, q_0) = (1, 1)$ defines the dynamics in the forth quadrant of the coordinate system with $x \geq 0$, $y \leq 0$ by the derivatives $\dot{x} = 1$, $\dot{y} = 1$.*

The corresponding specification of the mode dynamics is given by

$$\begin{array}{lcl} W_{(0,0)} & = & (\dot{x} = -1 \wedge \dot{y} = 1) \\ W_{(0,1)} & = & (\dot{x} = -1 \wedge \dot{y} = -1) \\ W_{(1,0)} & = & (\dot{x} = 1 \quad \wedge \dot{y} = -1) \\ W_{(1,1)} & = & (\dot{x} = 1 \quad \wedge \dot{y} = 1) \end{array}$$

Global constraints GC define the connection between modes and quadrants as mentioned above, i. e., the predicate GC is given by

$$\begin{array}{lcl} ((q_1, q_0) = (0,0) & \Longrightarrow & x \geq 0 \wedge y \geq 0) \wedge \\ ((q_1, q_0) = (0,1) & \Longrightarrow & x \leq 0 \wedge y \geq 0) \wedge \\ ((q_1, q_0) = (1,0) & \Longrightarrow & x \leq 0 \wedge y \leq 0) \wedge \\ ((q_1, q_0) = (1,1) & \Longrightarrow & x \geq 0 \wedge y \leq 0). \end{array}$$

States not satisfying the GC constraint are disregarded when looking for a trace from the initial states to the unsafe states.

Whenever the x-axis (y-axis) is reached by a continuous flow, a continuous-to-discrete transition is performed which multiplies the x-value (y-value) by a constant 1.1. The following discrete-to-continuous transition simply changes the mode variables to the mode corresponding to the neighboring quadrant. The continuous-to-discrete transitions D^{c2d} are all urgent and are altogether defined by

$$\begin{array}{lcl} (q_1, q_0) = (0,0) \wedge x \leq 0 & \Longrightarrow & y := 1.1y;\ x := x; \\ (q_1, q_0) = (0,1) \wedge y \leq 0 & \Longrightarrow & x := 1.1x;\ y := y; \\ (q_1, q_0) = (1,0) \wedge x \geq 0 & \Longrightarrow & y := 1.1y;\ x := x; \\ (q_1, q_0) = (1,1) \wedge y \geq 0 & \Longrightarrow & x := 1.1x;\ y := y; \end{array}$$

The discrete-to-continuous transitions D^{d2c} select the mode of the following flow and are defined by

$$\begin{array}{lcl} x > 0 \wedge y \geq 0 & \Longrightarrow & (q_1, q_0) := (0,0); \\ x \leq 0 \wedge y > 0 & \Longrightarrow & (q_1, q_0) := (0,1); \\ x < 0 \wedge y \leq 0 & \Longrightarrow & (q_1, q_0) := (1,0); \\ x \geq 0 \wedge y < 0 & \Longrightarrow & (q_1, q_0) := (1,1); \end{array}$$

In this simple model there are no discrete transitions between the continuous-to-discrete and the discrete-to-continuous transitions.

The initial states are specified by $(4.6 \leq x \leq 5.1) \wedge (3.1 \leq y \leq 3.6)$.

The unsafe states are specified by $(q_1, q_0) = (0,0) \wedge (4.5 \leq x \leq 5.5) \wedge 4.5 \leq y \leq 5.5$.

An exact backward analysis starting from the unsafe states leads to a state space computation sketched in Figs. 6 and 7. In order to simplify the exhibition we restrict ourselves to the basic variant of the model checking algorithm here and do not consider the onioning technique from Sect. 3.2. The x-axis and y-axis are shown in bold face lines, linear constraints limiting the state set reached so far by light gray lines. The unsafe states are represented by a red rectangle, the initial states by a blue rectangle. The backward reachability analysis starts with the red rectangle. After the fifth flow, the states marked in yellow in Fig. 6 have been reached. Fig. 7 shows the corresponding state set after 24 flow steps. It can be seen that the backwards evolution of the reachable state set comes closer and closer to the point $(x, y) = (0, 0)$ (in form of a spiral) without ever reaching it. It can also be seen that the backward reachability never ends with a fixed point and that it never stops by reaching some initial state. Moreover, the state set which has to be represented becomes more and more complicated, the number of linear constraints which are definitely needed for representing the limits of the reachable state set increases without any upper bound.

On the other hand we observe that intuitively the later preimage computation steps do not contribute much novel information. The backward evolution does not really "come closer" to the initial states (represented by the blue rectangle). Thus, it is intuitively clear by inspecting Figs. 6 and 7 that it does not help to consider the following (even infinite!) series of preimage computation with maximal precision.

For this reason we decided to consider *overapproximations* of state sets. Whenever we can show using overapproximations that it is not possible to reach the initial states starting from the unsafe states, it is clear

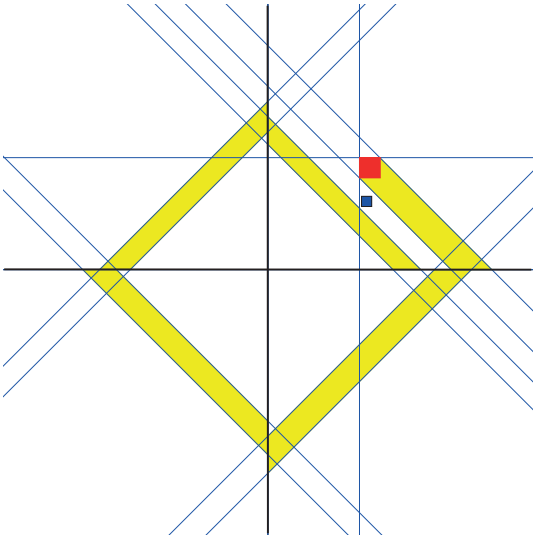


Figure 6: States reached after 5 flow steps.

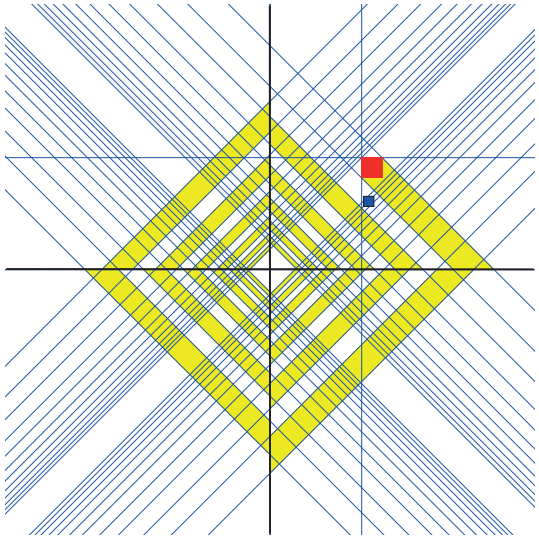


Figure 7: States reached after 24 flow steps.

that the initial states can neither be reached by a precise evolution without overapproximation. If we reach the initial states by our overapproximating backward reachability analysis, then this can have two reasons: Either the precise backward analysis can reach the initial states as well or the overapproximation has been the reason for reaching the initial states. In the later case we will need refinements of the overapproximation in order to decide the reachability problem. The refinements are performs following the well-known CEGAR (Counterexample-Guided Abstraction Refinement) paradigm.

In the following we give a brief introduction to our proposed method by means of our running example.

Our overapproximating backward analysis overapproximates state sets heuristically from time to time triggered by exceeding certain bounds on the total number of linear constraints or certain growth factors w. r. t. the number of linear constraints. Assume that a overapproximation is triggered after the 5th flow computation, see Fig. 6. Our overapproximation approach allows approximations in the environment immediately around the represented state set. The "immediate environment" is computed by the so-called ϵ-bloating operation, for details see Sect. 4.2. The limits for the overapproximation operation for the state set in Fig. 6 are given by the state set shown in Fig. 8. We are not allowed to add states which are "too far away" from the represented state set or which correspond immediately to initial states (this explains the small yellow rectangle in Fig. 8).

Fig. 9 shows the overapproximated state set. In general, our overapproximations are computed using methods described in Sect. 4.3. The goal is to minimize the number of linear constraints in the description of the overapproximated state sets without violating the restrictions for the overapproximation (in our example illustrated in Fig. 8). In Fig. 9 the white area around the blue initial states exactly corresponds to states not included into the overapproximation due to the restriction that initial states are not allowed in the overapproximation.

We replace the exact state set from Fig. 6 by the overapproximated one from Fig. 9 and continue our backward model checking procedure. It is rather easy to see that the next (backward) flow will remove the white area around the initial states in Fig. 9, now leading to the result that the overapproximating reachability analysis detects a path from the initial states to the unsafe states. Such a path is called a *counterexample*.

In general it is not known whether this counterexample is a real one which would also be present in an exact model checking procedure or a spurious counterexample which only results from overapproximations. The next step is to check whether the counterexample is a real one. For this we use a method similar to *Bounded Model Checking* (BMC), i. e., we check whether there is a real counterexample with a length given by the detected path. If our (incremental) BMC procedure fails in its search for a real counterexample, it returns a state e which is reachable from the initial states, but from which the unsafe states can not be reached within the given number of steps. Such a state e must have been added by the overapproximation operation. The state e has to be excluded from the overapproximation in a refinement step. More details on

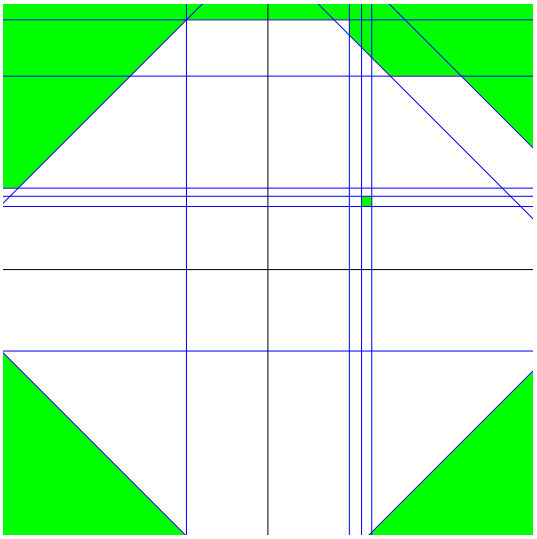


Figure 8: States set restricting the overapproximation.

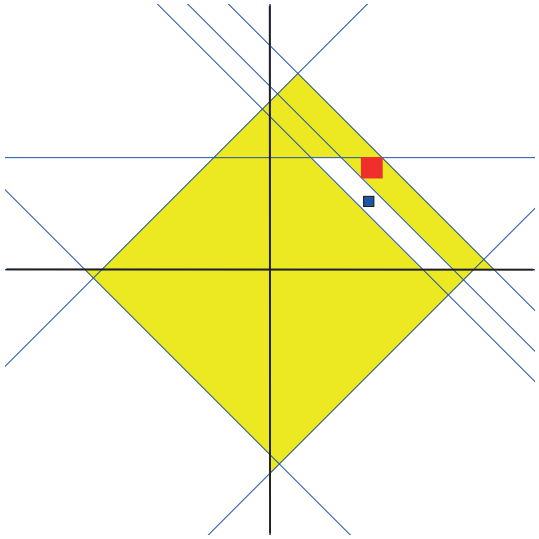


Figure 9: Overapproximated state set.

how to obtain such a state e can be found later on in Sect. 4.4.

In our small example we find such a state e as depicted in Fig. 10. From the initial states there is a flow leading to the state represented by the green circle, but this trace can not be continued to the unsafe states. Therefore e is excluded from the set of states which are allowed to be added by overapproximation in Figs. 8 and 9. Thus, we will backtrack to the situation where we overapproximated the state set from Fig. 6. Now the states restricting the overapproximation are given by Fig. 11 (state e is added). The computed overapproximation is given by Fig. 12. Restarting the backward model checking procedure with the state set from Fig. 12 shows that no additional states are visited and a fixed point has been reached – without arriving at the initial states.

Our small running example reveals several interesting facts:

- Indeed, it was not necessary to look at all details in the exact model checking procedure. Model checking with overapproximations has been able to provide a safety proof.
- A refinement of overapproximations was needed to complete the proof. The refinement was computed using the CEGAR paradigm based on a spurious counterexample.
- Our overapproximation procedure has been able to generalize the information learnt from the spurious counterexample. In particular, the overapproximation procedure did not only exclude the state e from the approximated state set, but also an environment around e (see the difference between Fig. 9 and Fig. 12). This is a crucial property of our methods for overapproximations which will be presented in Sect. 4.3.

We will start by describing model checking with overapproximations in Sect. 4.2. In Sect. 4.3 we will present different approaches for overapproximations minimizing the number of linear constraints in state set representations. Finally, in Sect. 4.4 we will have a detailed view into the CEGAR procedure based on these principles.

4.2 Model Checking with Abstraction

First of all, we present a sketch of our model checking algorithm using abstraction in Alg. 7. Here we confine ourselves to an extension of the *onion algorithm* in Alg. 4. Later on, we will use the model checking algorithm with abstraction in the context of Counterexample-Guided Abstraction Refinement (CEGAR).

The name conventions in Alg. 7 are as follows:

- State sets collecting *all* states visited so far via a corresponding transition type y are denoted by ϕ_x^y.

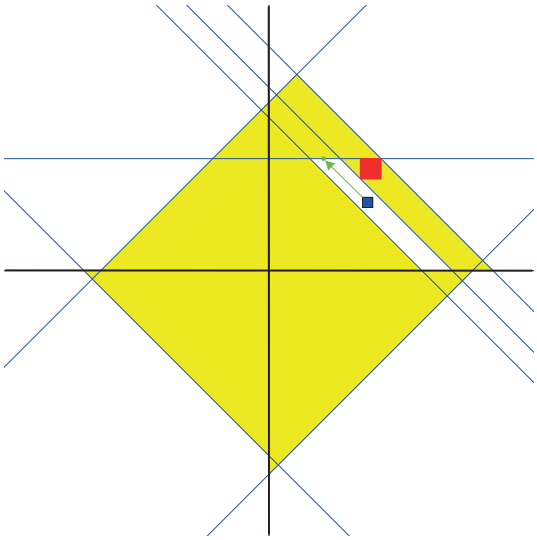


Figure 10: State e which has to be excluded from over-approximation.

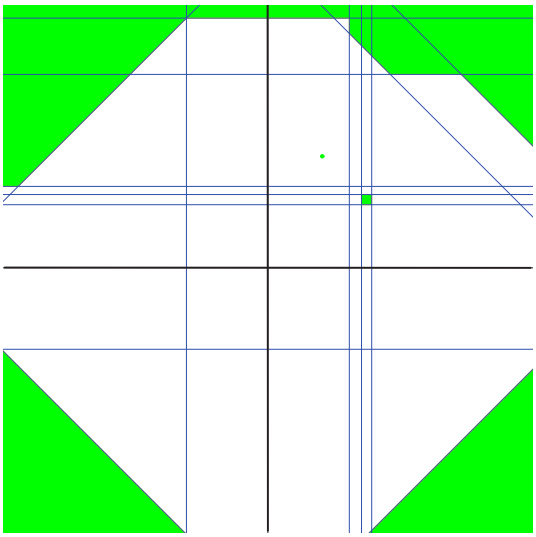


Figure 11: States set restricting the overapproximation (including state e).

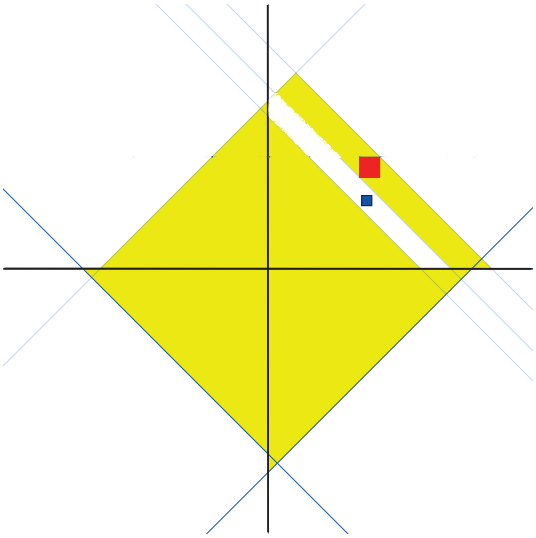


Figure 12: New overapproximated state set and final fixed point of the model checking procedure.

begin

$\psi_0^{d2c} := \neg safe$; $\phi_0^{d2c} := \neg safe$; $\phi_0^{d_{fp}} := 0$; $\phi_0^{c2d} := 0$; $\phi_0^{flow} := 0$;

$i := 0$;

while *true* **do**

$i := i + 1$;

// **Discrete fixed point iteration:**

$j := 0$;

$\psi_0^d := \psi_{i-1}^{d2c}$; $\phi_0^d := \psi_{i-1}^{d2c} \vee \phi_{i-1}^{d_{fp}}$; $\omega_0^d := \phi_{i-1}^{d_{fp}}$;

repeat

$j := j + 1$;

$\psi_{j-1}^{d,ov} := overapprox(\psi_{j-1}^d \wedge \neg\omega_{j-1}^d, eps\_bloat(\psi_{j-1}^d \wedge \neg\omega_{j-1}^d, \epsilon) \vee \phi_{j-1}^d)$;

$\psi_j^d := Pre^d(\psi_{j-1}^{d,ov} \wedge GC)$;

$\omega_j^d := \phi_{j-1}^d \vee \psi_{j-1}^{d,ov}$; $\psi_j^d := constraint\_min(\psi_j^d, \omega_j^d)$; $\phi_j^d := \phi_{j-1}^d \vee \psi_{j-1}^{d,ov} \vee \psi_j^d$;

if $GC \wedge \psi_j^d \wedge init \neq 0$ **then return** *false*;

until $GC \wedge \psi_j^d \wedge \neg\omega_j^d = 0$;

$\psi_i^{d_{fp}} := constraint\_min(\phi_j^d, \phi_{i-1}^{d_{fp}})$;

$\phi_i^{d_{fp}} := \phi_j^d$;

$\psi_i^{d_{fp}} := overapprox(\psi_i^{d_{fp}} \wedge \neg\phi_{i-1}^{d_{fp}}, eps\_bloat(\psi_i^{d_{fp}} \wedge \neg\phi_{i-1}^{d_{fp}}, \epsilon) \vee \phi_i^{d_{fp}})$;

$\phi_i^{d_{fp}} := \phi_{i-1}^{d_{fp}} \vee \psi_i^{d_{fp}}$;

// **Evaluate c2d transitions:**

$\psi_i^{c2d} := \exists d_{n+1}, \ldots d_p(Pre^{c2d}(\psi_i^{d_{fp}} \wedge GC)) \wedge \bigvee_{h=1}^k (\beta_h \wedge (\overrightarrow{m} = \mathbf{m}_h))$;

if $i = 1$ **then** $\psi_i^{c2d} := \psi_i^{c2d} \vee \neg safe$;

if $GC \wedge \psi_i^{c2d} \wedge init \neq 0$ **then return** *false*;

if $GC \wedge \psi_i^{c2d} \wedge \neg\phi_{i-1}^{c2d} = 0$ **then return** *true*;

$\psi_i^{c2d} := constraint\_min(\psi_i^{c2d}, \phi_{i-1}^{c2d})$;

$\phi_i^{c2d} := \phi_{i-1}^{c2d} \vee \psi_i^{c2d}$;

$\psi_i^{c2d} := overapprox(\psi_i^{c2d} \wedge \neg\phi_{i-1}^{c2d}, eps\_bloat(\psi_i^{c2d} \wedge \neg\phi_{i-1}^{c2d}, \epsilon) \vee \phi_i^{c2d})$;

$\phi_i^{c2d} := \phi_{i-1}^{c2d} \vee \psi_i^{c2d}$;

// **Evaluate continuous flow:**

$\psi_i^{flow} := \bigvee_{h=1}^k constraint\_min(Pre^c(\psi_i^{c2d}|_{\overrightarrow{m}=\mathbf{m}_h}, W_h, \beta_h), \phi_{i-1}^{flow}|_{\overrightarrow{m}=\mathbf{m_h}}) \wedge (\overrightarrow{m} = \mathbf{m}_h)$;

if $GC \wedge \psi_i^{flow} \wedge init \neq 0$ **then return** *false*;

if $GC \wedge \psi_i^{flow} \wedge \neg\phi_{i-1}^{flow} = 0$ **then return** *true*;

$\psi_i^{flow} := constraint\_min(\psi_i^{flow}, \phi_{i-1}^{flow})$;

$\phi_i^{flow} := \phi_{i-1}^{flow} \vee \psi_i^{flow}$;

$\psi_i^{flow} := overapprox(\psi_i^{flow} \wedge \neg\phi_{i-1}^{flow}, eps\_bloat(\psi_i^{flow} \wedge \neg\phi_{i-1}^{flow}, \epsilon) \vee \phi_i^{flow})$;

$\phi_i^{flow} := \phi_{i-1}^{flow} \vee \psi_i^{flow}$;

// **Evaluate d2c transitions:**

$\psi_i^{d2c} := Pre^{d2c}(\psi_i^{flow} \wedge d2cTransEnabled \wedge GC)$;

if $GC \wedge \psi_i^{d2c} \wedge init \neq 0$ **then return** *false*;

if $GC \wedge \psi_i^{d2c} \wedge \neg\phi_i^{d_{fp}} = 0$ **then return** *true*;

// See optimization above, $\bigvee_{j=0}^{i-1} \psi_j^{d2c} \subseteq \phi_i^{d_{fp}}$

$\psi_i^{d2c} := constraint\_min(\psi_i^{d2c}, \phi_{i-1}^{d2c})$;

$\phi_i^{d2c} := \phi_{i-1}^{d2c} \vee \psi_i^{d2c}$;

Algorithm 7: Backward reachability analysis, onion algorithm using abstraction.

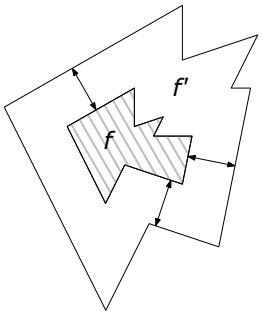


Figure 13: ϵ-bloated state set

- By ψ_x^y we denote the states which can be reached in the *current* step x via transition type y. The "onion ring" of newly visited states is given by $\psi_x^y \wedge \neg\phi_{x-1}^y$. An optimized representation of ψ_x^y using don't cares from ϕ_{x-1}^y is computed by $constraint\_min(\psi_x^y, \phi_{x-1}^y)$.
- $eps\_bloat(f, \epsilon)$ computes a so-called ϵ-bloating of the predicate f: Given a state set f, we produce a bloated version $f' = eps\_bloat(f, \epsilon)$ by pushing all inequations "outwards" by a positive distance. Fig. 13 sketches a 2-dimensional state set f with its bloating f'. More precisely, we assume that each real variable c_i is connected with a lower bound l_i and an upper bound u_i as part of the global constraints. For the direction c_i, the distance for pushing an inequation outwards is given by $\epsilon \cdot (u_i - l_i)$ with $0 \leq \epsilon \leq 1$. By ϵ-bloating we produce room for simplifying predicates with restricted overapproximation. Increasing ϵ leads to more flexibility for improvement, but also to a potentially larger overapproximation.
- The function $overapprox(f, g)$ overapproximates f without computing a representation which is larger than g, i. e., it computes a representation i with $f \leq i \leq g$ (assuming $f \leq g$). In our case, the optimization goal of $overapprox(f, g)$ is to compute an overapproximation for f with a minimized number of linear constraints. The lower bound f and the upper bound g for the overapproximation are computed as follows: The "onion ring" of newly visited states given by $\psi_x^y \wedge \neg\phi_{x-1}^y$ forms the lower bound f. For the upper bound g, the onion ring $\psi_x^y \wedge \neg\phi_{x-1}^y$ is enlarged by ϵ-bloating and the states already reached before are added, resulting in $eps\_bloat(\psi_x^y \wedge \neg\phi_{x-1}^y, \epsilon) \vee \phi_x^y$.

In fact, not every call of *overapprox* in the pseudocode of Alg. 7 really results in an overapproximation of the considered state set. Rather, the places with the *overapprox*-operations mark the places where overapproximation is *possibly* applied, depending on heuristics observing the absolute number of the linear constraints in the predicates as well as the growth of the number of linear constraints compared to the corresponding predicates from the previous iteration. In that way, *overapprox* is heuristically used to keep the number of linear constraints in the predicates within a manageable size.

4.3 Abstraction Algorithms for State Set Representations

Here we consider different variants for the realization of the overapproximation $overapprox(f, g)$. As already mentioned, the goal is to compute a representation for i with the property $f \leq i \leq g$ and i has a minimized number of linear constraints.

4.3.1 Constraint Minimization Using ϵ-Bloating

A first and simple possibility for providing a realization for $overapprox(f, g)$ is using constraint minimization as presented in Sect. 3.2.1: We choose $overapprox(f, g) = constraint\_min(f, g \wedge \neg f)$, i. e., the states in $g \wedge \neg f$ may be added to f or not, forming a don't care set. The operation $constraint\_min(f, g \wedge \neg f)$ minimizes the number of linear constraints in the result by detecting redundant linear constraints "modulo don't cares". This means that the constraints occurring in the result are restricted to the constraints occurring in f or g. Removing this restriction may lead to representations with less linear constraints. Two new methods for computing such representations are presented in Sect. 4.3.2 and 4.3.3.

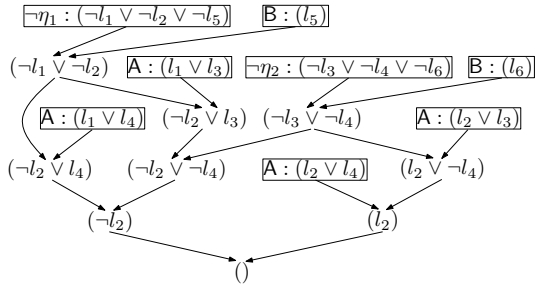


Figure 14: A proof

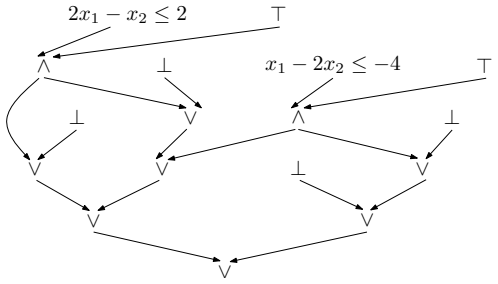


Figure 15: An interpolant

4.3.2 Proof-based Computation of Simple Interpolants

Here we look into a method which uses don't cares to minimize the number of linear constraints in a representation. In contrast to the previous section new linear constraints not occurring in the original representation will be used. The new representation will be obtained by computing so-called "simple interpolants". Such simple interpolants approximate given state set representations by a "smoother shape" with less linear constraints. Given $f \leq g$, a simple interpolant representing $overapprox(f, g)$ is computed based on a proof of unsatisfiability for $f \wedge \neg g$. Proofs of unsatisfiability are computed by an SMT solver with linear arithmetic as the theory.

In the following we write $\phi \models_{\mathcal{T}} \psi$, if a formula ϕ logically implies a formula ψ in all models of the theory $\mathcal{T}$ (i. e., in all models which interpret the function symbols and predicate symbols of linear arithmetic by their usual meaning). Similarly, we call a formula $\mathcal{T}$-satisfiable if it is satisfiable in $\mathcal{T}$.

An SMT solver works as already briefly described in Sect. 3.1.4: It introduces constraint variables q_{ℓ_i} for linear constraints ℓ_i and 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 $\epsilon_{i_1}, \ldots, \epsilon_{i_m}$ to constraint variables $q_{\ell_{i_1}}, \ldots, q_{\ell_{i_m}}$ was found to be inconsistent, the linear program solver derives a cause for the infeasibility of the assignment, say $\eta = m_{j_1}^{\epsilon_{j_1}} \wedge \ldots \wedge m_{j_k}^{\epsilon_{j_k}}$, $(k \leq m)$, where $\{m_{j_1}^{\epsilon_{j_1}}, \ldots, m_{j_k}^{\epsilon_{j_k}}\} \subseteq \{l_{i_1}^{\epsilon_{i_1}}, \ldots, l_{i_m}^{\epsilon_{i_m}}\}$. We call the cause η a *$\mathcal{T}$-conflict*. The SMT solver then adds the negation of the cause, $\neg\eta = \{\neg m_{j_1}^{\epsilon_{j_1}}, \ldots, \neg m_{j_k}^{\epsilon_{j_k}}\}$, which we call *$\mathcal{T}$-lemma*, to its set of clauses and starts backtracking. The added $\mathcal{T}$-lemma prevents the DPLL-procedure from selecting the same invalid assignment again. Usually, the $\mathcal{T}$-conflicts η used in modern SMT solvers are reduced to minimal size (i. e. η becomes satisfiable, if one of its literals is removed) in order to prune the search space as much as possible. Such $\mathcal{T}$-conflicts η are often called *minimal infeasible subsets*.

SMT solvers can be extended in a straightforward way to produce proofs for the unsatisfiability of formulas [58]:

Definition 7 ($\mathcal{T}$-Proof) *Let $S = \{c_1, \ldots, c_n\}$ be a set of non-tautologic clauses and C a clause. A DAG P is a* resolution proof *for the deduction of $\bigwedge c_i \models_{\mathcal{T}} C$, if*
(1) each leaf $n \in P$ is associated with a clause n_{cl}; n_{cl} is either a clause of S or a $\mathcal{T}$-lemma ($n_{cl} = \neg\eta$ for some $\mathcal{T}$-conflict η);
(2) each non-leaf $n \in P$ has exactly two parents n^L and n^R, and is associated with the clause n_{cl} which is derived from n_{cl}^L and n_{cl}^R by resolution, i. e. the parents' clauses share a common variable (the pivot*) n_p such that $n_p \in n_{cl}^L$ and $\neg n_p \in n_{cl}^R$, and $n_{cl} = n_{cl}^L \setminus \{n_p\} \cup n_{cl}^R \setminus \{\neg n_p\}$; n_{cl} (the* resolvent*) must not be a tautology;*
(3) there is exactly one root node $r \in P$; r is associated with clause C; $r_{cl} = C$.

Here we consider the theory of linear arithmetic over reals $\mathcal{LA}(\mathbb{R})$, i. e., we consider $\mathcal{LA}(\mathbb{R})$-proofs.

Intuitively, a resolution proof provides a means to derive a clause C from the set of clauses S and some additional facts of the theory $\mathcal{T}$. If C is the empty clause, P is proving the $\mathcal{T}$-unsatisfiability of S.

Example 4 *Fig. 14 shows a resolution proof for the unsatisfiability of* $S = (l_1 \vee l_3) \wedge (l_1 \vee l_4) \wedge (l_2 \vee l_3) \wedge (l_2 \vee l_4) \wedge l_5 \wedge l_6$ *with* $l_1 = (-x_2 \leq 0)$, $l_2 = (x_1 \leq 1)$, $l_3 = (-x_2 \leq -5)$, $l_4 = (x_1 \leq 6)$, $l_5 = (-2x_1 + x_2 \leq -6)$, $l_6 = (-x_1 + 2x_2 \leq 0)$. *To prove the unsatisfiability, the solver added two* $\mathcal{T}$*-lemmata* $\neg\eta_1 = (\neg l_1 \vee \neg l_2 \vee \neg l_5)$ *and* $\neg\eta_2 = (\neg l_3 \vee \neg l_4 \vee \neg l_6)$.

Here we use a notion of Craig interpolation [59] which is generalized to SMT formulas.

Definition 8 (Craig Interpolant [59]) *Let* A *and* B *be two formulas, such that* $\mathsf{A} \wedge \mathsf{B} \models_{\mathcal{T}} \bot$. *A* Craig interpolant I *is a formula such that (1)* $\mathsf{A} \models_{\mathcal{T}} I$, *(2)* $\mathsf{B} \wedge I \models_{\mathcal{T}} \bot$, *(3) the uninterpreted symbols in* I *occur both in* A *and* B, *the free variables in* I *occur freely both in* A *and* B.

Given a $\mathcal{T}$-unsatisfiable set of clauses $S = \{c_1, \ldots, c_n\}$, a disjoint partition (A, B) of S, and a proof P for the $\mathcal{T}$-unsatisfiability of S, an interpolant for (A, B) can be constructed by the following procedure [60]:[16]

1. For every leaf $n \in P$ associated with a clause $n_{cl} \in S$, set $n_I = n_{cl} \downarrow \mathsf{B}$ if $n_{cl} \in \mathsf{A}$, and set $n_I = \top$ if $n_{cl} \in \mathsf{B}$.
2. For every leaf $n \in P$ associated with a $\mathcal{T}$-lemma $\neg\eta$ ($n_{cl} = \neg\eta$), set $n_I = \mathcal{T}\text{-INTERPOLANT}(\eta \setminus \mathsf{B}, \eta \downarrow \mathsf{B})$.
3. For every non-leaf node $n \in P$, set $n_I = n_I^L \vee n_I^R$ if $n_p \notin \mathsf{B}$, and set $n_I = n_I^L \wedge n_I^R$ if $n_p \in \mathsf{B}$.
4. Let $r \in P$ be the root node of P associated with the empty clause $r_{cl} = \emptyset$. r_I is an interpolant of A and B.

The interpolation procedure differs from pure Boolean interpolation [61] only in the handling of $\mathcal{T}$-lemmata. $\mathcal{T}$-INTERPOLANT$(\cdot, \cdot)$ produces an interpolant for an unsatisfiable pair of conjunctions of $\mathcal{T}$-literals.

In our case conjunctions of m $\mathcal{T}$-literals (i. e., linear inequations) are written as $Ax \leq a$ with real variables $(x_1, \ldots, x_n)^T = x$, $A \in \mathbb{R}^{m \times n}$, $a \in \mathbb{R}^m$.[17] Every row vector in the $m \times n$-matrix A describes the coefficients of the corresponding linear inequation.

There exist several methods to construct an $\mathcal{LA}(\mathbb{R})$-interpolant from conflicts in an $\mathcal{LA}(\mathbb{R})$-proof as described in [60, 62, 58]. Here we review the approach from [62], since our method is based on this approach.

We assume an $\mathcal{LA}(\mathbb{R})$-conflict η which is produced during the proof of unsatisfiability of two formulas A and B. From η we may extract a conjunction $\eta \setminus \mathsf{B}$ of linear inequations only occurring in formula A and a conjunction $\eta \downarrow \mathsf{B}$ of linear inequations occurring in formula B. $\eta \setminus \mathsf{B}$ and $\eta \downarrow \mathsf{B}$ are represented by the inequation systems $Ax \leq a$ and $Bx \leq b$, respectively ($A \in \mathbb{R}^{m_\mathsf{A} \times n}$, $a \in \mathbb{R}^{m_\mathsf{A}}$, $B \in \mathbb{R}^{m_\mathsf{B} \times n}$, $b \in \mathbb{R}^{m_\mathsf{B}}$). Since η is an $\mathcal{LA}(\mathbb{R})$-conflict, the conjunction of $Ax \leq a$ and $Bx \leq b$ has no solution. Then, according to Farkas' lemma, there exists a linear inequation $i^T x \leq \delta$ ($i \in \mathbb{R}^n$, $\delta \in \mathbb{R}$) which is an $\mathcal{LA}(\mathbb{R})$-interpolant for $Ax \leq a$ and $Bx \leq b$. $i^T x \leq \delta$ can be computed by linear programming from the following (in)equations with additional variables $\lambda \in \mathbb{R}^{m_\mathsf{A}}$, $\mu \in \mathbb{R}^{m_\mathsf{B}}$:

(1) $\lambda^T A + \mu^T B = \mathbf{0}^T$, (2) $\lambda^T a + \mu^T b \leq -1$, (3) $\lambda^T A = i^T$, (4) $\lambda^T a = \delta$, (5) $\lambda \geq \mathbf{0}$, $\mu \geq \mathbf{0}$.

The coefficients λ and μ define a positive linear combination of the inequations in $Ax \leq a$ and $Bx \leq b$ leading to a contradiction $0 \leq \lambda^T a + \mu^T b$ with $\lambda^T a + \mu^T b \leq -1$ (see (1) and (2)). The interpolant $i^T x \leq \delta$ just "sums up" the "$Ax \leq a$"-part of the linear combination leading to the contradiction (see (3) and (4)), thus $i^T x \leq \delta$ is implied by $Ax \leq a$. $i^T x \leq \delta$ is clearly inconsistent with $Bx \leq b$, since it derives the same contradiction as before. Altogether $i^T x \leq \delta$ is an interpolant of $Ax \leq a$ and $Bx \leq b$.

Example 5 (cont.) *Fig. 15 shows a Craig interpolant resulting from the proof in Fig. 14, when partitioning* S *into* (A, B) *with* $\mathsf{A} = (l_1 \vee l_3) \wedge (l_1 \vee l_4) \wedge (l_2 \vee l_3) \wedge (l_2 \vee l_4)$ *and* $\mathsf{B} = l_5 \wedge l_6$. *The* $\mathcal{LA}(\mathbb{R})$*-interpolant for the* $\mathcal{LA}(\mathbb{R})$*-conflict* η_1 *is a positive linear combination of* η_1*'s* A*-literals (i. e.,* l_1 *and* l_2*), which is conflicting with a positive linear-combination of the remaining literals (i. e.,* l_5*), e. g.* $1 \cdot (-x_2 \leq 0) + 2 \cdot (x_1 \leq 1) \equiv (2x_1 - x_2 \leq 2)$ *and* $1 \cdot (-2x_1 + x_2 \leq -6)$ *lead to the conflict* $0 \leq -4$. *Similarly, the interpolant* $x_1 - 2x_2 \leq -4$ *is derived from the* $\mathcal{LA}(\mathbb{R})$*-conflict* η_2. *Propagating constants, the final interpolant of* A *and* B *becomes* $(2x_1 - x_2 \leq$

[16] Let C be a clause and ϕ be a formula. With $C \setminus \phi$, we here denote the clause that is created from C by removing all atoms occurring in ϕ; $C \downarrow \phi$ denotes the clause that is created from C by removing all atoms that are not occurring in ϕ.

[17] For simplicity we confine ourselves to non-strict inequations. A generalization to mixed strict and non-strict inequations is straightforward.

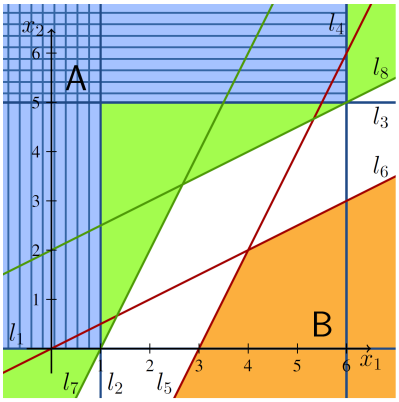


Figure 16: Two $\mathcal{LA}(\mathbb{R})$-interpolants l_7 and l_8 for the interpolation between A and B.

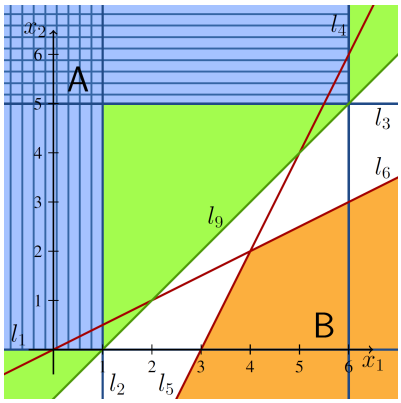


Figure 17: A single $\mathcal{LA}(\mathbb{R})$-interpolant l_9 replacing l_7 and l_8.

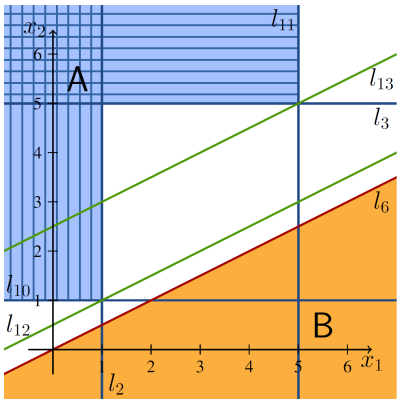


Figure 18: Relaxing constraints to enable l_{12} as shared interpolant.

$2) \vee (x_1 - 2x_2 \leq -4)$. *Fig. 16 gives a geometric illustration of the example.* A *is depicted in blue,* B *in orange, the interpolant is represented by the green or blue areas.* η_1 *says that* $l_1 \wedge l_2$ *(the blue area with vertical lines) does not intersect with* l_5 *(leading to interpolant* $l_7 = (2x_1 - x_2 \leq 2)$*) and* η_2 *says that* $l_3 \wedge l_4$ *(the blue area with horizontal lines) does not intersect with* l_6 *(leading to interpolant* $l_8 = (x_1 - 2x_2 \leq -4)$*).*

Basic idea for computing simple interpolants. Now we present a method computing *simpler* interpolants than the standard methods mentioned above. The basic idea is as follows: It is based upon the observation that in previous interpolation schemes the inconsistency proofs and thus the interpolants derived from different $\mathcal{T}$-conflicts are *uncorrelated*. In most cases different $\mathcal{T}$-conflicts lead to different $\mathcal{LA}(\mathbb{R})$-interpolants contributing to the final interpolant, thus, complicated proofs with many $\mathcal{T}$-conflicts tend to lead to complicated Craig interpolants depending on many linear constraints.

Example 6 (cont.) *In Ex. 5 we have two different* $\mathcal{T}$*-conflicts leading to two different interpolants (see green lines in Fig. 16). However, it is easy to see from Fig. 17 that there is a single inequation* $l_9 = (x_1 - x_2 \leq 1)$ *which can be used as an interpolant for* A *and* B *(*A *implies* l_9 *and* l_9 *does not intersect with* B*).*

Our idea is to *share* $\mathcal{LA}(\mathbb{R})$-interpolants between different $\mathcal{T}$-conflicts. In order to come up with an interpolation scheme using as many shared interpolants as possible, we first introduce a check whether a fixed set of $\mathcal{T}$-conflicts can be proved by a shared proof, leading to a single shared $\mathcal{LA}(\mathbb{R})$-interpolant for that set of $\mathcal{T}$-conflicts.

We assume a fixed set $\{\eta_1, \ldots, \eta_r\}$ of $\mathcal{T}$-conflicts. Each $\mathcal{T}$-conflict η_j defines two systems of inequations: $A_j x \leq a_j$ for the A-part and $B_j x \leq b_j$ for the B-part. Extending [62] we ask whether there is a single inequation $i^T x \leq \delta$ and coefficients λ_j, μ_j with

(1_j) $\lambda_j^T A_j + \mu_j^T B_j = \mathbf{0}^T$, (2_j) $\lambda_j^T a_j + \mu_j^T b_j \leq -1$, (3_j) $\lambda_j^T A_j = i^T$, (4_j) $\lambda_j^T a_j = \delta$, (5_j) $\lambda_j \geq \mathbf{0}$, $\mu_j \geq \mathbf{0}$

for all $j \in \{1, \ldots, r\}$. Note that the coefficients λ_j and μ_j for the different $\mathcal{T}$-conflicts may be different, but the interpolant $i^T x \leq \delta$ is required to be identical for all $\mathcal{T}$-conflicts. Again, the problem formulation consisting of all constraints (1_j)–(5_j) can be solved by linear programming in polynomial time.

Unfortunately, first results showed that the potential to find shared interpolant was not as high as expected using this basic idea. By a further analysis of the problem we observed that more degrees of freedom are needed to enable a larger number of shared interpolants.

Relaxing constraints. Consider Fig. 18 for motivating our first measure to increase the degrees of freedom for interpolant generation. Fig. 18 shows a slightly modified example compared to Figs. 16 and 17 with $\mathsf{A} = (l_{10} \wedge l_2) \vee (l_3 \wedge l_{11})$ and $\mathsf{B} = l_6$. Again we have two $\mathcal{T}$-conflicts: η_3 which says that $l_{10} \wedge l_2 \wedge l_6$ is infeasible and η_4 which says that $l_3 \wedge l_{11} \wedge l_6$ is infeasible. We can show that the interpolation generation according to

[62] only computes interpolants which *touch* the A-part of the $\mathcal{T}$-conflict (as long as the corresponding theory conflict is minimized, and both A-part and B-part are not empty). Thus the only possible interpolants for η_3 and η_4 according to (1)–(5) are l_{12} and l_{13}, respectively. I.e. it is not possible to compute a *shared* interpolant for this example according to equations (1_j)–(5_j). On the other hand it is easy to see that l_{12} may also be used as an interpolant for η_4, if we do not require interpolants to touch the A-part (which is $l_3 \wedge l_{11}$ in the example). We achieve that goal simply by relaxing constraint (4_j) to $(4'_j)$ $\lambda_j a_j \leq \delta$ and by modifying (2_j) to $(2'_j)$ $\delta + \mu_j^T b_j \leq -1$ (all other constraints (i'_j) remain the same as (i_j)).

An inequation $i^T x \leq \delta$ computed according to $(1'_j)$–$(5'_j)$ is still implied by $A_j x \leq a_j$ (since $i^T x \leq \lambda_j a_j$ is implied and $\lambda_j a_j \leq \delta$) and it contradicts $B_j x \leq b_j$, since $0 \leq \lambda_j^T a_j + \mu_j^T b_j \leq \delta + \mu_j^T b_j$ is conflicting with $\delta + \mu_j^T b_j \leq -1$.

Extending $\mathcal{T}$-conflicts. There is a second restriction to the degrees of freedom for shared interpolants which follows from the computation of *minimized* $\mathcal{T}$-conflicts in SMT solvers. (Note that minimized $\mathcal{T}$-conflicts are used with success in modern SMT solvers in order to prune the search space as much as possible. Unfortunately, minimization of $\mathcal{T}$-conflicts impedes the search for shared interpolants.) We can prove the following lemma:

Lemma 2 *If a $\mathcal{LA}(\mathbb{R})$-conflict η is minimized, and both $\eta \setminus \mathsf{B}$ and $\eta \downarrow \mathsf{B}$ are not empty, then the direction of vector i of an $\mathcal{LA}(\mathbb{R})$-interpolant $i^T x \leq \delta$ for $\eta \setminus \mathsf{B}$ and $\eta \downarrow \mathsf{B}$ is fixed.*

Example 7 (cont.) *Again consider Fig. 16. Since the $\mathcal{LA}(\mathbb{R})$-conflict $\eta_1 = l_1 \wedge l_2 \wedge l_5$ is minimized, the direction vector of the interpolant l_7 is fixed. The same holds for $\mathcal{LA}(\mathbb{R})$-conflict $\eta_2 = l_3 \wedge l_4 \wedge l_6$ and the direction vector of l_8. Thus, there is no shared interpolant for η_1 and η_2.*

Fortunately, $\mathcal{T}$-conflicts which are extended by additional inequations remain $\mathcal{T}$-conflicts. (If the conjunction of some inequations is infeasible, then any extension of the conjunction is infeasible as well.) Therefore we may extend η_1 to $\eta'_1 = l_1 \wedge l_2 \wedge l_5 \wedge l_6$ and η_2 to $\eta'_2 = l_3 \wedge l_4 \wedge l_5 \wedge l_6$. It is easy to see that the linear inequation $l_9 = (x_1 - x_2 \leq 1)$ from Fig. 17 is a solution of $(1'_j)$–$(5'_j)$ applied to η'_1 and η'_2 (with coefficients $\lambda_{1,1} = \lambda_{1,2} = 1$, $\mu_{1,1} = \mu_{1,2} = \frac{1}{3}$, $\lambda_{2,1} = \lambda_{2,2} = 1$, $\mu_{2,1} = \mu_{2,2} = \frac{1}{3}$). This means that we really obtain the shared interpolant l_9 from Fig. 17 by $(1'_j)$–$(5'_j)$, if we extend the $\mathcal{T}$-conflicts appropriately.

We learn from Ex. 7 that an appropriate extension of $\mathcal{T}$-conflicts increases the degrees of freedom in the computation of interpolants, leading to new shared interpolants. Clearly, in the general case an extension of $\mathcal{T}$-conflicts η_j (and thus of $\mathcal{T}$-lemmata $\neg\eta_j$) may destroy proofs of $\mathcal{T}$-unsatisfiability. In the following we derive conditions when interpolants derived from proofs with extended $\mathcal{T}$-lemmata are still correct. We consider two approaches: Adding implied literals and lemma localization.

Implied literals.

Definition 9 *Let* A *and* B *be two formulas, and let l be a literal. l is an* implied literal for A *(*implied literal for B*), if* $\mathsf{A} \models_{\mathcal{T}} l$ *and l does not occur in* B *(if* $\mathsf{B} \models_{\mathcal{T}} l$*).*

Lemma 3 *Let P be a proof of $\mathcal{T}$-unsatisfiability of* $\mathsf{A} \wedge \mathsf{B}$*, let $\neg\eta$ be a $\mathcal{T}$-lemma in P not containing literal $\neg l$, and let l be implied for* A *(for* B*). Then Craig interpolation according to [60] applied to P with $\neg\eta$ replaced by $\neg\eta \vee \neg l$ computes a Craig interpolant for* A *and* B*.*

The proof can be found in [63].

We can conclude from Lemma 3 that we are free to arbitrarily add negations of implied literals for A or B to $\mathcal{T}$-lemmata without losing the property that the resulting formula according to [60] is an interpolant of A and B.

Example 8 (cont.) *Again consider Fig. 16. l_1 and l_4 are clearly implied literals for* A*, l_5 and l_6 are implied literals for* B*. Therefore we can extend $\mathcal{T}$-conflict η_1 to $\eta''_1 = l_1 \wedge l_2 \wedge l_4 \wedge l_5 \wedge l_6$ and η_2 to $\eta''_2 = l_1 \wedge l_3 \wedge l_4 \wedge l_5 \wedge l_6$. $(1'_j)$–$(5'_j)$ applied to η''_1 and η''_2 and interpolation according to [60] leads to l_9 as an interpolant of* A *and* B *(similarly to Ex. 7, see Fig. 17).*

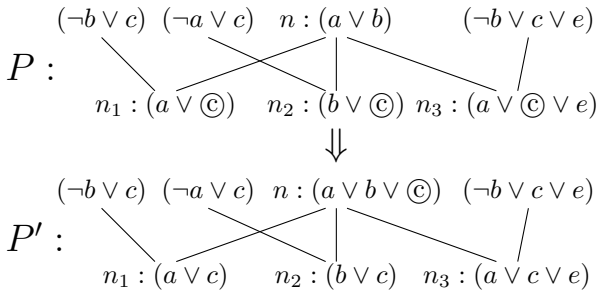


Figure 19: Pushing-up literals

Lemma Localization. A second method for extending $\mathcal{T}$-conflicts is given by so-called Lemma Localization [64]. The additional $\mathcal{T}$-literals for extending $\mathcal{T}$-conflicts are derived by the *pushup operation* which detects redundant $\mathcal{T}$-literals in the proof structure. A $\mathcal{T}$-literal l is called redundant in a proof node, if the clauses of all successor nodes contain the literal l and the current node does not use it as the pivot variable for the resolution. Such a redundant $\mathcal{T}$-literal can then be added to the current node without losing correctness of the proof.

Fig. 19 illustrates the idea: The upper part of Fig. 19 shows a detail of a bigger resolution proof P: the node n has three children n_1, n_2, and n_3, whose clauses are derived from their parents by resolution. The clauses of all three children contain the literal c. The clause of n, however, does not contain the literal c. "Pushing up" c from n_1, n_2, and n_3 to n (i. e. adding c to n's clause) creates a modified graph P' (shown in the lower part of Fig. 19), in which the resolutions at n's children n_1, n_2, and n_3 are still valid. If, on the other hand, one child's clause did not contain the literal c (e. g. n_3's clause), and we pushed up c to n's clause, the resolution at this child would not be valid anymore (the resolvent would then contain c but the original clauses did not). Furthermore, one must not push a literal into a node's clause if the literal matches the node's pivot, since the resolvent of two clauses can not contain the pivot variable. Therefore, in general, one can "push up" literals to a node's clause that are (1) in the intersection of all of its children's clauses, and (2) do not match the node's pivot.

A redundant literal l may eventually be pushed into a leaf of the proof which represents a $\mathcal{T}$-lemma $\neg\eta$. The corresponding $\mathcal{T}$-*conflict* η is extended by $\neg l$. In that way the pushup operation is used to increase the degrees of freedom for computing shared $\mathcal{T}$-interpolants.

Overall algorithm for computing shared interpolants. First of all, as a preprocessing step, we use the pushup-algorithm in order to replace potentially complex $\mathcal{T}$-interpolants by constants: If $(\eta \wedge \neg l) \setminus \mathsf{B}$ is still a $\mathcal{T}$-conflict, then $\bot$ is a valid $\mathcal{T}$-interpolant, and if $(\eta \wedge \neg l) \downarrow \mathsf{B}$ is still a $\mathcal{T}$-conflict, then $\top$ is a valid $\mathcal{T}$-interpolant. Of course, extending theory conflicts by additional literals increases the chance of obtaining constant $\mathcal{T}$-interpolants. If we are able to detect constant $\mathcal{T}$-interpolants, then the sizes of overall Craig-interpolants may decrease significantly due to the propagation of constants.

After detecting constant $\mathcal{T}$-interpolants, we continue by computing *shared* $\mathcal{T}$-interpolants. We use implied literals and the pushup operation to increase the degrees of freedom for computing shared $\mathcal{T}$-interpolants.

Our overall algorithm starts with a $\mathcal{T}$-unsatisfiable set of clauses S and a disjoint partition (A, B) of S, and computes a proof P for the $\mathcal{T}$-unsatisfiability of S. P contains r $\mathcal{T}$-lemmata $\neg\eta_1, \ldots, \neg\eta_r$. The system of (in)equations $(1'_j)$–$(5'_j)$ from above with $j \in \{j_1, \ldots, j_k\}$ provides us with a check whether there is a shared interpolant $i^T x \leq \delta$ for the subset $\{\eta_{j_1}, \ldots, \eta_{j_k}\}$ of $\mathcal{T}$-conflicts. This check is called `SharedInterpol`$(\{\eta_{j_1}, \ldots, \eta_{j_k}\})$. Our goal is to find an interpolant for A and B with a minimal number of different $\mathcal{T}$-interpolants. At first, we use `SharedInterpol` to precompute an (undirected) compatibility graph $G_{cg} = (V_{cg}, E_{cg})$ with $V_{cg} = \{\eta_1, \ldots, \eta_r\}$ and $\{\eta_i, \eta_j\} \in E_{cg}$ iff there is a shared interpolant of η_i and η_j.

Then we use a simple iterative greedy algorithm based on `SharedInterpol` for minimizing the number of different $\mathcal{T}$-interpolants used in the Craig interpolant. For this, we iteratively compute sets SI_i of $\mathcal{T}$-conflicts which have a shared interpolant. We start with $SI_1 = \{\eta_s\}$ for some $\mathcal{T}$-conflict η_s. To extend a set SI_i we select a new $\mathcal{T}$-conflict $\eta_c \notin \cup_{j=1}^{i} SI_j$ with $\{\eta_c, \eta_j\} \in E_{cg}$ for all $\eta_j \in SI_i$. Then we check whether

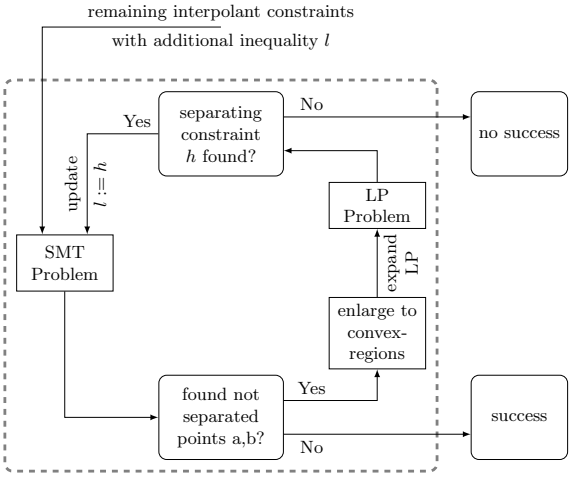


Figure 20: Sketch of the Iterative Computation

`SharedInterpol`$(SI_i \cup \{\eta_c\})$ returns true or false. If the result is true, we set $SI_i := SI_i \cup \{\eta_c\}$, otherwise we select a new $\mathcal{T}$-conflict as above. If there is no appropriate new $\mathcal{T}$-conflict, then we start a new set SI_{i+1}. The algorithm stops when all $\mathcal{T}$-conflicts are inserted into a set SI_j. Of course, the quality of the result depends on the selection of $\mathcal{T}$-conflicts to start new sets SI_i and on the decision which candidate $\mathcal{T}$-conflicts to select if there are several candidates. So the cardinality of sets SI_i and their total number (i. e., the number of computed $\mathcal{LA}(\mathbb{R})$-interpolants) is not necessarily minimal. For improving on the order dependency of the iterative greedy algorithm, in [63] a method based on linear programming is proposed that computes maximum subsets of $\mathcal{T}$-conflicts having a shared interpolant.

4.3.3 Iterative Computation of Simple Interpolants

The iterative computation of simple interpolants is based on the following extension of Proposition 3 in [65].

Proposition 1 *The set of (linear) constraints $L = \{L_1, \ldots, L_h\}$ separates the state sets f and g through some Boolean formula if and only if for every pair of points $p_f \in f$ and $p_g \in g$ with the same assignment of boolean variables $((p_f)_{\mathbb{B}} = (p_g)_{\mathbb{B}})$ there exists a j $(1 \leq j \leq h)$ such that $L_j(p) \neq L_j(q)$.*

Starting with an interpolant i computed by Constraint Minimization (Sect. 3.2.1) we create L, the set of linear constraints used in this interpolant. Since i is an interpolant, L satisfies Prop. 1.

The algorithm now iteratively improves the set L by replacing two linear constraints by one, while preserving the invariant that L satisfy Prop. 1. The linear constraints in L are called *interpolant constraints*. Hence, our algorithm is a local search heuristic.

Notice that when removing two interpolant constraints, there will be pairs $p_f \in f, p_g \in g$ of points with $((p_f)_{\mathbb{B}} = (p_g)_{\mathbb{B}})$ that can not be distinguished by the remaining interpolant constraints L, i. e., $l(p_f) = l(p_g)$ for all $l \in L$. We will test whether all those pairs of points can be distinguished by a single new linear constraint.

Basically, we iteratively collect such pairs of points and construct a linear constraint l^* separating all pairs of points already found, until either all pairs of points can be distinguished with the additional help of l^* or no such linear constraint can be found. Fig. 20 gives a sketch of the algorithm.

In order to guarantee termination, one cannot just collect pairs of points, as there can be an infinite number. Therefore, we construct convex regions C_f and C_g around the points p_f and p_g described only by constraints known to the system, i. e., for C_f we only use linear constraints used in the description of f and additionally all remaining interpolant constraints. The convex sets are contained in the respective sets, i. e., $C_f \subseteq f$ and $C_g \subseteq g$. Since there are only a finite number of linear constraints in the description of the state sets and interpolant constraints, there are only a finite number of possible convex sets describable by these constraints. This lead to a termination of this part of the algorithm in a finite number of steps.

Hence, we need three sub-algorithms: one for finding pairs of points that can not be distinguished, one for constructing the convex regions around them, and one for constructing a linear constraint separating a given set of pairs of convex regions.

Finding Pairs of Indistinguishable Points with SMT. Notice that we are in the situation that we have to test whether our tentative new linear constraints l^* together with the remaining linear constraints L of the interpolant satisfies to construct an interpolant between state sets f and g. The linear constraints are not sufficient if and only if we can find points $p_f \in f, p_g \in g$ with $((p_f)_{\mathbb{B}} = (p_g)_{\mathbb{B}})$ that are not distinguishable, i. e., for all $l \in L \cup \{l^*\}$ $l(p_f) = l(p_g)$ holds.

To solve this by an SMT solver we use the following formula:

$$(p_f \in f) \wedge (p_g \in g) \wedge ((p_f)_{\mathbb{B}} = (p_g)_{\mathbb{B}}) \wedge \left(\bigwedge_{l \in L \cup \{l^*\}} l(p_f) = l(p_g) \right) \tag{6}$$

Either we found a valid solution and therefore two points p_f, p_g that are not separable by any linear constraint in $L \cup \{l^*\}$ or we found that $L \cup \{l^*\}$ is a valid set of interpolant constraints. The problem has only minor changes in every iteration, because we only substitute the old condition $l^*(p) = l^*(q)$ with an updated linear constraint l^*. This gives us the opportunity to use the advantage of incremental SMT.

Enlarge Pairs of Indistinguishable Points to Convex Regions. If we have found a pair of points p_f, p_g as described in the previous section, we want to find a set of points $C_f \subseteq f$ around p_f that is preferably large, such that no point in C_f can be distinguished from p_g and vice versa.

We achieve this by computing convex regions around p_F, such that all points within C_f are equal with respect to all linear constraints in $L \setminus \{l^*\}$ and all linear constraints used in the description of f.

For computing C_f test for every linear constraint l that is used in the description of f, if a satisfies this linear constraint, i. e., we compute $l(p_f)$. We therefore collect linear constraints in a set C. We add l to C when $l(p_f)$ is *true*, or $\neg\, l$ if $l(p_f)$ is *false*. After evaluating that for every linear constraint in f, we compute the same for every interpolant constraint. We do not use the current l^* in the description of the convex region, since we change this constraint in the next step, where we search for a new candidate. C_f is then computed as a conjunction of all constraints in C.

Finding a Linear Constraint Separating Pairs of Convex Regions with LP. We try to find a solution to this problem by constructing a linear program whose solution represents a separating linear constraint. The LP does not solve the problem in general, as it will fix all convex regions of f on one side and the convex regions of g on the other side, which is not necessarily required. Furthermore, it assumes that all inequalities in f and g are non-strict, i. e., convex regions are enlarged by their boundary. Both deficiencies are handled heuristically later. We use an LP-solver that handles rational arithmetic as errors in the coefficients prevent the algorithm from termination since we could be forced to separate the same pair of convex region multiple times.

The construction of the LP is similar to the one that computes the linear constraints in resolution proofs, hence based on Farkas' Lemma. We expanded the approach to separate multiple pairs of convex regions.

Therefore, we define the variables $d \in \mathbb{Q}^m$ and $d_0 \in \mathbb{Q}$ that describe the new linear constraint l^* in the form $d^T x \leq d_0$. Pairs of convex sets (F^j, G^j) for $j \in \{1, \ldots, k\}$, which are present in the k-th iteration of the LP-problem for finding a new constraint for one test. The constraint $d^T x \leq d_0$ is implied by F^j, if there is a non-negative linear combination of the inequalities of F^j leading to $d^T x \leq d_0$. Similarly, the constraint $a^T x > b$ is implied by G^j, if there is a positive ϵ such that there is a non-positive linear combination of the inequalities of G^j leading to $d^T x \geq d_0 + \epsilon$. All constraints can easily be formulated as linear constraints.

If the LP is solvable and $\epsilon > 0$ the computed linear constraint l^* separates each pair of convex regions. Additionally the margin can be shifted to one region by adding specific scalar factors to the ϵ in the constraints.

For more details, we refer to [66].

4.4 Counterexample-Guided Abstraction Refinement

Abstracting state sets comes with a price – it is possible that model checking reaches an initial state due to abstraction only. This can happen directly, if a state added by overapproximation is also an initial state, or indirectly, if the (direct or indirect) preimage of a state added by overapproximation intersects the initial state set. In any case, model checking would consider the model unsafe, even if the model is safe.

Since the model checking result should not change due to abstraction, these cases need to be identified and excluded. This is a rather typical situation when using abstractions, and often the problem is covered with *counterexample-guided abstraction refinement* (CEGAR): The *spurious counterexamples* are used to refine the abstraction, and the whole process is restarted.

In our case we use state set overapproximation as abstraction technique, and we want to avoid adding states, whose (direct or indirect) preimage closure intersects the init initial state set – or, in other words, we want to avoid adding states, which are reachable from the initial state set.

So the basic idea is to use reach set approximations for refinement – whenever a state (or state set) is identified as reachable, it is added to the corresponding reach set approximation. If overapproximation computation is modified, so that the overapproximated state sets do not extend into the reach set approximation, then spurious counterexamples can be safely excluded.

4.4.1 Overall Refinement Algorithm

For implementing CEGAR using reach set approximations, we need several ingredients: First spurious counterexamples need to be identified and used for refinement of the reach sets, then model checking needs to be restartable and must use the reach sets, and finally a CEGAR loop must coordinate both.

The overall CEGAR algorithm is distributed into two main subalgorithms: A special counterexample generation algorithm generates counterexamples, identifies spurious counterexamples, performs reach set approximation refinement and computes refined state set overapproximations. The model checking algorithm is modified, so that it uses the reach set approximations, and can be restarted at any depth.

For performing their tasks successfully, information needs to be transferred between the subalgorithms: For counterexample generation, spurious counterexample identification, and refinement computation, the counterexample generation algorithm needs the computed state sets (saved in a state set vector $\vec{\phi}$), overapproximations (saved in a state set vector $\vec{\phi^{ov}}$), and overapproximation lower bounds (saved in a state set vector $\vec{\phi^{ov_{lb}}}$) computed by the model checking algorithm. For convenience, the state set vectors are stored in the result generated by the model checking algorithm.

In reverse, the model checking algorithm needs the refined reach sets, the refined overapproximations and the restart depth provided by the counterexample generation algorithm.

Model checking and counterexample generation are controlled by the CEGAR loop in Alg. 8.

```
1  cegar(init, safe, GC) :
2  begin
3      restartData.restartDepth := 0;
4      repeat
5          mcResult := modelCheck(init, safe, GC, restartData) ;
6          if ¬mcResult.safe then
7              cegResult := generateCounterexample(mcResult.φ⃗, mcResult.φ^ov⃗);
8              if ¬cegResult.ceFound then
9                  restartData := cegResult.restartData;
10     until mcResult.safe ∨ cegResult.ceFound;
```

Algorithm 8: Cegar loop

4.4.2 Changes in Model Checking Algorithm

Using Reach Set Approximations. We use the reach set approximations $reachA^{c2d}$, $reachA^{d}$, $reachA^{d2c}$, and $reachA^{flow}$, each of which contains states which are known to be reachable with transitions of the corresponding type. All approximations are initialized with *init*.

The reach set approximations are used to limit the overapproximations used in the next preimage computation, i. e. the overapproximation to be used in a flow preimage computation is intersected with $\neg reachA^{flow}$.

So the line of Alg. 7

29 $\psi_i^{flow} := overapprox(\psi_i^{flow} \wedge \neg\phi_{i-1}^{flow}, eps\_bloat(\psi_i^{flow} \wedge \neg\phi_{i-1}^{flow}, \epsilon) \vee \phi_i^{flow})$;

is changed to

29 $\psi_i^{flow} := overapprox(\psi_i^{flow} \wedge \neg\phi_{i-1}^{flow}, (eps\_bloat(\psi_i^{flow} \wedge \neg\phi_{i-1}^{flow}, \epsilon) \wedge \neg reachA^{d2c}) \vee \phi_i^{flow})$;

The use of $reachA^{c2d}$, $reachA^{d}$ and $reachA^{d2c}$ follows the same pattern.

State Set Parameters and Results. The state sets in the state set vectors $\vec{\phi}$, $\vec{\phi^{ov}}$ and $\vec{\phi^{ov_{lb}}}$ are labeled, so that the names used in the algorithm are preserved.

In the following algorithm descriptions, the state set vectors to return are not explicitly extended. Instead we just enumerate the variables, which are added to the state sets immediately after they are computed:

During the first initialization, ϕ_0^{d2c} is added to $\vec{\phi}$. During execution of mainloop i with discrete fixpoint iteration taking j steps, the following state sets are added: $\phi_{i,1}^{d}, \ldots \phi_{i,j-1}^{d}, \phi_i^{d_{fp}}, \phi_i^{c2d}, \phi_i^{flow}, \phi_i^{d2c}$.

For $\vec{\phi^{ov}}$, the following state sets are added in mainloop i after they are computed: $\phi_{i,0}^{d,ov}$, ..., $\phi_{i,j-1}^{d,ov}$, $\phi_i^{d_{fp},ov}$, $\phi_i^{c2d,ov}$, $\phi_{i,ov}^{flow}$.

For $\vec{\phi^{ov_{lb}}}$, the following state sets are added in mainloop i after they are computed: $\phi_{i,0}^{d,ov_{lb}}$, ..., $\phi_{i,j-1}^{d,ov_{lb}}$, $\phi_i^{d_{fp},ov_{lb}}$, $\phi_i^{c2d,ov_{lb}}$, $\phi_{i,ov_{lb}}^{flow}$.

Restarting Model Checking. Model check restart is indicated by a restart depth larger than 0. If the restart depth is 0, then normal model checking is executed. If model checking is restarted, then the algorithm is resumed exactly at the position, where overapproximation introduced a state, whose (direct or indirect) preimage intersects the initial state set, so that the minimum number of necessary preimage computations are repeated. Instead of the original overapproximation, the refined overapproximated provided by counterexample generation is used as base for preimage computation.

Now we will describe the algorithm in detail. Since the overall algorithm is lengthy, the description is divided into segments.

Initialization. Initialization is done differently in the context of restarting than during normal model checking. If restarting, then the needed model checking data is restored. If not restarting, then model checking data is initialized as in Alg. 7. The variable *restarting* signals that restarting is going on. Once normal model checking has been resumed, the variable is reset, and normal model checking continues. The initialization is described in Alg. 9.

Mainloop. The model checking mainloop as such is separated in four sections: discrete fixpoint iteration, continuous-to-discrete transition evaluation, flow evaluation, and discrete-to-continuous transition evaluation (as in Alg. 7). When restarting, some sections may be skipped until the restarting point is reached. The mainloop is described in Alg. 10.

Discrete Fixed Point Iteration. The discrete fixed point iteration section is skipped, if restarting and the restarting point is not the discrete section.

If not restarting, then the section is equivalent to the corresponding section in Alg. 7, with the small additions that during overapproximation the reach set approximation $reachA^{d}$ is used and the lower bound $\psi_{j-1}^{d,ov_{lb}}$ is explicitly named, since it is needed for refinement.

If restarting, then after restoring the needed environment data, the refined overapproximation data is used as base for preimage computation. After that, restarting is completed and normal model checking resumes.

```
1  modelCheck(init, safe, GC, restartData) :
2  begin
       // Initialization:
3      if restartData.restartDepth > 0 then
4          ψ^d2c := restartData.ψ^d2c; φ^d2c := restartData.φ^d2c; φ^d_fp := restartData.φ^d_fp;
           φ^c2d := restartData.φ^c2d; φ^flow := restartData.φ^flow;
5          i := restartData.restartDepth − 1; restarting := true;
6      else
7          ψ_0^d2c := ¬safe; φ_0^d2c := ¬safe; φ_0^d_fp := 0; φ_0^c2d := 0; φ_0^flow := 0;
8          i := 0; restarting := false;
       ⋮
```

Algorithm 9: Backward onion algorithm using abstraction with restart, initialization.

```
1  modelCheck(init, safe, GC, restartData) :
2  begin
       // Initialization:
       ⋮
       // Mainloop:
9      while true do
10         i := i + 1;
           // Discrete fixed point iteration:
           ⋮
           // Evaluate c2d transitions:
           ⋮
           // Evaluate continuous flow:
           ⋮
           // Evaluate d2c transitions:
           ⋮
```

Algorithm 10: Backward onion algorithm using abstraction with restart, mainloop.

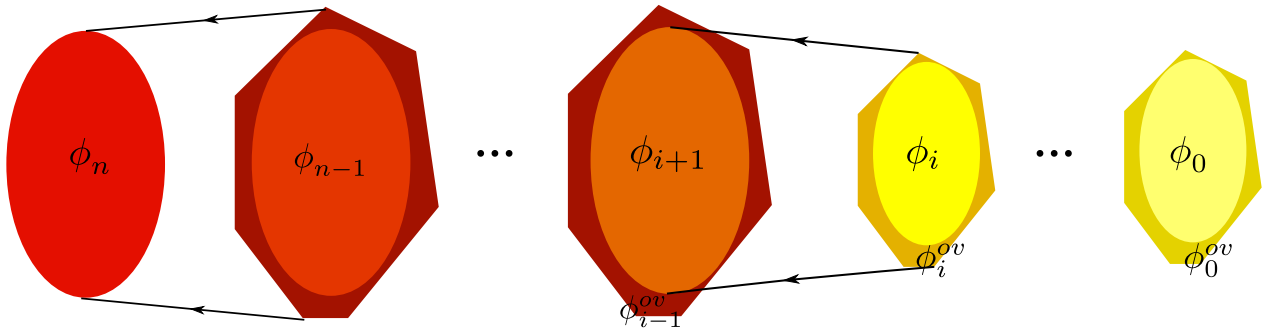


Figure 21: State set sequence

The discrete fixpoint iteration section is described in Alg. 11.

Continuous-to-discrete Transitions. Again the continuous-to-discrete transition section is skipped, if restarting and the restarting point is not the continuous-to-discrete section. Again, if not restarting, then the section is equivalent to the corresponding section in Alg. 7 (except that the overapproximation of the previous state set has been shifted to this section, so that restarting can be performed). Also in this section there are some small additions: During overapproximation the reach set approximation $reachA^{c2d}$ is used, and overapproximation $\psi_{j-1}^{d_{fp},ov}$ and lower bound $\psi_{j-1}^{d_{fp},ov_{lb}}$ are explicitly named, since they are needed for refinement. The continuous-to-discrete transition section is described in Alg. 12.

Continuous Flow and Discrete-to-Continuous Transitions. The changes for continuous flow section (Alg. 13) and discrete-to-continuous transition section (Alg. 14) match the pattern of those of the continuous-to-discrete transition section.

4.4.3 Learning from Counterexamples

Counterexample generation is only triggered, if the model checker generated a state set sequence, whose final state set intersects the set *init* of initial states. If this happens, counterexample generation has two main goals: If the state set sequence does yield a path leading from *init* to *unsafe*, this path should be computed and returned. If there is no such path, and the intersection was only detected because state set overapproximation added one (or several) state(s), whose (direct or indirect) preimage(s) intersect *init*, then these state(s) should be identified and added to the corresponding reach set approximation.

Since the hybrid state space is infinite, it is well possible that an infinite number of states are subject of refinement. Because an infinite number of refinement steps is clearly not acceptable, it is important to learn as much as possible from spurious counterexamples, and in as few steps as possible.

In this section, we describe a method using incremental SMT for trying to compute a path from *init* to *unsafe*. If such a path exists, it is found and returned. If there is no such path, the approach gives enough information for successfully triggering refinement.

As a first step, we describe the basic idea using a uniform transition relation, and then extend the approach to the more complex LHA+D system model.

Computing Counterexamples with Incremental SMT (Uniform Transition Relation). We use a simple system with hybrid variable set Var and the uniform transition relation $Trans$, which is applicable for every step of the system. We use $s_1 \to s_2$ as abbreviation for $(s_1, s_2) \in Trans$. We assume $\vec{\phi} = \phi_0, \ldots, \phi_n$ and $\vec{\phi^{ov}} = \phi_0^{ov}, \ldots, \phi_{n-1}^{ov}$ are vectors of hybrid state sets computed by a model checker, with $\phi_0 = unsafe$ and $\phi_n \wedge init \neq \emptyset$. Additionally $\forall_{i \in 0 \ldots n-1} \phi_i \subset \phi_i^{ov}$, so each ϕ_i^{ov} is either equal to the corresponding ϕ_i or an overapproximation. Furthermore $\forall_{i \in 1 \ldots n} \phi_i = \phi_{i-1} \cup Pre^{Trans}(\phi_{i-1}^{ov})$, so each ϕ_i is the ϕ_{i-1} extended with (possible) overapproximation's preimage Pre^{Trans} according to $Trans$. This situation is depicted in Fig. 21.

Obviously there is a real counterexample, if there is a sequence of states connected by the transition relation, starting from *init* and ending in *unsafe*: $\forall_{i \in n \ldots 0} : \exists s_i \in \phi_i : s_n \in init \wedge s_n \to s_{n-1} \to \cdots \to s_1 \to s_0$.

```
1  modelCheck(init, safe, GC, restartData) :
2  begin
       // Initialization:
       ⋮
       // Mainloop:
9      while true do
10       i := i + 1;
         // Discrete fixed point iteration:
11       if ¬restarting ∨ (restarting ∧ restartData.restartSection = d) then
12         if ¬restarting then
13           j := 0;
14           ψ^d_{i,0} := ψ^{d2c}_{i-1}; φ^d_{i,0} := ψ^{d2c}_{i-1} ∨ φ^{d_fp}_{i-1}; ω^d_{i,0} := φ^{d_fp}_{i-1};
15         else
16           j := restartData.j − 1;
17           if restartData.j = 1 then
18             ψ^d_{i,0} := restartData.ψ^{d2c}_{i-1};
19             φ^d_{i,0} := restartData.ψ^{d2c}_{i-1} ∨ restartData.φ^{d_fp}_{i-1};
20             ω^d_{i,0} := restartData.φ^{d_fp}_{i-1};
21           else
22             ψ⃗^d_i := restartData.ψ⃗^d_i; φ⃗^d_i := restartData.φ⃗^d_i;
23             ω⃗^d_i := restartData.ω⃗^d_i;
24         repeat
25           j := j + 1;
26           if ¬restarting then
27             ψ^{d,ov_lb}_{i,j-1} := ψ^d_{i,j-1} ∧ ¬ω^d_{i,j-1};
28             ψ^{d,ov}_{i,j-1} := overapprox(ψ^{d,ov_lb}_{i,j-1}, (eps_bloat(ψ^{d,ov_lb}_{i,j-1}, ϵ) ∧ ¬reachA^d) ∨ φ^d_{i,j-1});
29           else
30             ψ^{d,ov}_{i,j-1} := restartData.ψ^{d,ov}_{i,j-1};
31             restarting := false;
32           ψ^d_{i,j} := Pre^d(ψ^{d,ov}_{i,j-1} ∧ GC);
33           ω^d_{i,j} := ω^d_{i,j-1} ∨ ψ^{d,ov}_{i,j-1}; ψ^d_{i,j} := constraint_min(ψ^d_{i,j}, ω^d_{i,j}); φ^d_{i,j} := φ^d_{i,j-1} ∨ ψ^{d,ov}_{i,j-1} ∨ ψ^d_j;
34           if GC ∧ ψ^d_{i,j} ∧ init ≠ 0 then return false;
35         until GC ∧ ψ^d_j ∧ ¬ω^d_j = 0;
36       φ^{d_fp}_i := φ^d_{i,j};
37       ψ^{d_fp}_i := constraint_min(φ^{d_fp}_i, φ^{d_fp}_{i-1});
         ⋮
```

Algorithm 11: Backward onion algorithm using abstraction with restart, discrete fixed point iteration section.

```
1  modelCheck(init, safe, GC, restartData) :
2  begin
       // Initialization:
       ⋮
       // Mainloop:
9      while true do
10       i := i + 1;
         // Discrete fixed point iteration:
         ⋮
         // Evaluate c2d transitions:
38       if ¬restarting ∨ (restarting ∧ restartData.restartSection = c2d) then
39         if ¬restarting then
40           ψ_i^{d_fp,ov_lb} := ψ_i^{d_fp} ∧ ¬φ_{i-1}^{d_fp};
41           ψ_i^{d_fp,ov} := overapprox(ψ_i^{d_fp,ov_lb}, (eps_bloat(ψ_i^{d_fp,ov_lb}, ϵ) ∧ ¬reach^{c2d}) ∨ φ_i^{d_fp});
42         else
43           ψ_i^{d_fp,ov} := restartData.ψ_i^{d_fp,ov};
44           restarting := false;
45         φ_i^{d_fp,ov} := φ_{i-1}^{d_fp} ∨ ψ_i^{d_fp,ov};
46         ψ_i^{c2d} := ∃d_{n+1}, ... d_p(Pre^{c2d}(ψ_i^{d_fp,ov} ∧ GC)) ∧ ⋁_{h=1}^{k}(β_h ∧ (m⃗ = m_h));
47         if i = 1 then ψ_i^{c2d} := ψ_i^{c2d} ∨ ¬safe;
48         ;
49         if GC ∧ ψ_i^{c2d} ∧ init ≠ 0 then return false;
50         ;
51         if GC ∧ ψ_i^{c2d} ∧ ¬φ_{i-1}^{c2d} = 0 then return true;
52         ;
53         ψ_i^{c2d} := constraint_min(ψ_i^{c2d}, φ_{i-1}^{c2d});
54         φ_i^{c2d} := φ_{i-1}^{c2d} ∨ ψ_i^{c2d};
         ⋮
```

Algorithm 12: Backward onion algorithm using abstraction with restart, continuous-to-discrete transition section.

```
1  modelCheck(init, safe, GC, restartData) :
2  begin
       // Initialization:
       ⋮
       // Mainloop:
3      while true do
4          i := i + 1;
           // Discrete fixed point iteration:
           ⋮
           // Evaluate c2d transitions:
           ⋮
           // Evaluate continuous flow:
52         if ¬restarting ∨ (restarting ∧ restartData.restartSection = flow) then
53             if ¬restarting then
54                 ψ_i^{c2d,ov_lb} := ψ_i^{c2d} ∧ ¬φ_{i-1}^{c2d};
55                 ψ_i^{c2d,ov} := overapprox(ψ_i^{c2d,ov_lb}, (eps_bloat(ψ_i^{c2d,ov_lb}, ε) ∧ ¬reachA^{flow}) ∨ φ_i^{c2d});
56             else
57                 ψ_i^{c2d,ov} := restartData.ψ_i^{c2d,ov};
58                 restarting := false;
59             φ_i^{c2d,ov} := φ_{i-1}^{c2d} ∨ ψ_i^{c2d,ov};
60             ψ_i^{flow} := ⋁_{h=1}^{k} constraint_min(Pre^c(ψ_i^{c2d,ov}|_{m⃗=m_h}, W_h, β_h), φ_{i-1}^{flow}|_{m⃗=m_h}) ∧ (m⃗ = m_h);
61             if GC ∧ ψ_i^{flow} ∧ init ≠ 0 then return false;
62             ;
63             if GC ∧ ψ_i^{flow} ∧ ¬φ_{i-1}^{flow} = 0 then return true;
64             ;
65             ψ_i^{flow} := constraint_min(ψ_i^{flow}, φ_{i-1}^{flow});
66             φ_i^{flow} := φ_{i-1}^{flow} ∨ ψ_i^{flow};
           ⋮
```

Algorithm 13: Backward onion algorithm using abstraction with restart, continuous flow section.

```
1  modelCheck(init, safe, GC, restartData) :
2  begin
       // Initialization:
       ⋮
       // Mainloop:
3      while true do
4          i := i + 1;
           // Discrete fixed point iteration:
           ⋮
           // Evaluate c2d transitions:
           ⋮
           // Evaluate continuous flow:
           ⋮
           // Evaluate d2c transitions:
65         if ¬restarting ∨ (restarting ∧ restartData.restartSection = d2c) then
66             if ¬restarting then
67                 ψ_i^{flow,ov_lb} := ψ_i^{flow} ∧ ¬φ_{i-1}^{flow};
68                 ψ_i^{flow,ov} := overapprox(ψ_i^{flow,ov_lb}, (eps_bloat(ψ_i^{flow,ov_lb}, ϵ) ∧ ¬reachA^{d2c}) ∨ φ_i^{flow});
69             else
70                 ψ_i^{flow,ov} := restartData.ψ_i^{flow,ov};
71                 restarting := false;
72             φ_i^{flow,ov} := φ_{i-1}^{flow} ∨ ψ_i^{flow,ov};
73             ψ_i^{d2c} := Pre^{d2c}(ψ_i^{flow,ov} ∧ GC);
74             if GC ∧ ψ_i^{d2c} ∧ init ≠ 0 then return false;
75             ;
76             if GC ∧ ψ_i^{d2c} ∧ ¬φ_i^{d_fp} = 0 then return true;
77             ;
               // See optimization above, ⋁_{j=0}^{i-1} ψ_j^{d2c} ⊆ φ_i^{d_fp}
78             ψ_i^{d2c} := constraint_min(ψ_i^{d2c}, φ_{i-1}^{d2c});
79             φ_i^{d2c} := φ_{i-1}^{d2c} ∨ ψ_i^{d2c};
```

Algorithm 14: Backward onion algorithm using abstraction with restart, discrete-to-continuous transition section.

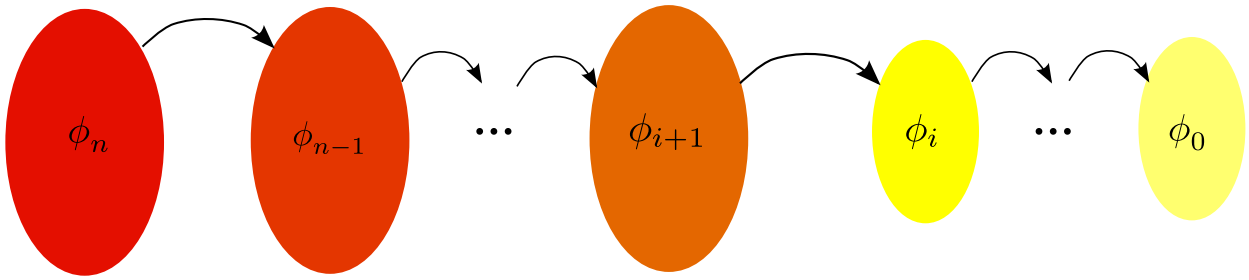


Figure 22: Real counterexample

If there is a gap between two state sets ($\exists i \in n-1 \ldots 0 : \forall s_i \in \phi_j : \forall j \in n \ldots i : s_j \in \phi_j : s_n \in init \wedge s_n \rightarrow \cdots \rightarrow s_{i+1} \implies \neg(s_{i+1} \rightarrow s_i)$), then overapproximation must be involved, because $\phi_{i-1} \subset \phi_{i-1}^{ov}$ and $\phi_i = \phi_{i-1} \cup Pre^{Trans}(\phi_{i-1}^{ov})$.

This can be coined into an algorithm with the help of incremental SMT: We build and solve a sequence of SMT problems (similar to bounded model checking (BMC)). Similar to BMC, the system variables exist in a separate instance for each unroll depth, connected by the transition relation. Using the abbreviations $\phi\langle n\rangle = \phi[v_n/v]_{v \in Var}$ and $Trans\langle n, m\rangle = Trans[v_n/v, v_n/v']_{v \in Var}$, we begin with

$init\langle n\rangle \wedge \phi_n\langle n\rangle$
$init\langle n\rangle \wedge \phi_n\langle n\rangle \wedge Trans\langle n, n-1\rangle \wedge \phi_{n-1}\langle n-1\rangle$
$init\langle n\rangle \wedge \phi_n\langle n\rangle \wedge Trans\langle n, n-1\rangle \wedge \phi_{n-1}\langle n-1\rangle \wedge Trans\langle n-1, n-2\rangle \wedge \phi_{n-2}\langle n-2\rangle$
$\vdots$

If the SMT problem for depth i is solvable, then we generate (and solve) the problem for depth $i-1$, until (hopefully)

$init\langle n\rangle \wedge \phi_n\langle n\rangle \wedge Trans\langle n, n-1\rangle \wedge \phi_{n-1}\langle n-1\rangle \wedge \ldots \wedge Trans\langle 1, 0\rangle \wedge \phi_0\langle 0\rangle$

can be solved and a real counterexample has been found. However if a SMT problem for depth i is not solvable, then we can switch from problem

$init\langle n\rangle \wedge \phi_n\langle n\rangle \wedge Trans\langle n, n-1\rangle \wedge \phi_{n-1}\langle n-1\rangle \wedge \ldots \wedge Trans\langle n-1+1, n-i\rangle \wedge \phi_{n-i}\langle n-i\rangle$

to the problem

$init\langle n\rangle \wedge \phi_n\langle n\rangle \wedge Trans\langle n, n-1\rangle \wedge \phi_{n-1}\langle n-1\rangle \wedge \ldots \wedge Trans\langle n-1+1, n-i\rangle \wedge \phi_{n-i}^{ov}\langle n-i\rangle$

by replacing the last SMT frame denoting $\phi_{n-i}\langle n-i\rangle$ by $\phi_{n-i}^{ov}\langle n-i\rangle$. Each solution of this problem describes a spurious counterexample and can be used for refinement.

Computing Counterexamples with Incremental SMT (LHA+D). In this paragraph, we try to transfer the idea sketched above for a uniform transition relation to a LHA+D. Instead of a simple system with variable set Var and uniform transition relation $Trans$, we use a LHA+D with $Var = C \cup D \cup M \cup I$ and the transition relations $Trans^{c2d}$, $Trans^{d}$, $Trans^{d2c}$ and $Trans^{flow}$, corresponding to the step types of the LHA+D. We use abbreviations $s_1 \xrightarrow{t} s_2$ for $(s_1, s_2) \in Trans^t$ for $t \in \{c2d, d, d2c, flow\}$.

Again we assume $\vec{\phi} = \phi_0, \ldots, \phi_n$ and $\vec{\phi^{ov}} = \phi_0^{ov}, \ldots, \phi_{n-1}^{ov}$ are vectors of hybrid state sets, now computed by the model checking algorithm described in 4.4.2. For any ϕ in $\vec{\phi}$, we write ϕ^t if $type(\phi) = t$. For any ϕ^{ov} in $\vec{\phi^{ov}}$, we write $\phi^{t,ov}$ if $type(\phi^{ov}) = t$.

For a real counterexample, again a state sequence leading from $init$ to $unsafe$ needs to be established. Again we start with a SMT problem describing the states intersecting $init$, then add transition relations and target state sets as long as the problem keeps solvable or $unsafe$ is reached, or use a modified problem involving an overapproximated state set for identification of spurious counterexamples and refinement, if the SMT problem gets unsolvable. However many details are different to the case of a uniform transition relation.

Initial SMT Problem. The model checking algorithm terminates, as soon as an intersection with $init$ is detected, thus $init \cup \vec{\phi}[n]^t \neq \emptyset$ is guaranteed. So the first SMT problem is

$init\langle n\rangle \wedge \vec{\phi}[n]^{t_n}\langle n\rangle$
where $t_n \in \{c2d, d, d2c, flow\}$.

Extending the SMT Problem. Extension of the incremental SMT problem depends on the type of the last state set added to the problem. The extension of a SMT problem of depth i

$\cdots \wedge \vec{\phi}[i]^{t_i}\langle i\rangle$

is simple for $t_i = flow$ and $t_i = d2c$: If the problem is solvable, then it is extended with $Trans^{t_i}$ and the next state set $\vec{\phi}[i-1]^{t_{i-1}}$:

$\cdots \wedge \vec{\phi}[i]^{t_i}\langle i\rangle \wedge Trans^{t_{i-1}} < i, i-1 > \wedge \vec{\phi}[i-1]^{t_{i-1}}\langle i-1\rangle$

If the problem is not solvable, we replace $\vec{\phi}[i]^{t_i}$ with its overapproximation $\vec{\phi}[i]^{t_i,ov}$ and start identifying spurious counterexamples:

$\cdots \wedge \vec{\phi}[i]^{t_i,ov}\langle i\rangle$

For $t_i = c2d$ and $t_i = d$ the discrete fixpoint iteration in the model checking algorithm complicates extension. The iteration computes overapproximations and preimages and disjoins them, until a fixpoint is reached, whereby the number of needed iterations corresponds to the maximum number of discrete steps the system may take, although also less steps (including no discrete steps at all) are possible. The fixpoint is then used as base for $c2d$-overapproximation.

This poses several problems, which need to be dealt with separately: First each possible number of discrete steps needs to be considered for finding a real counterexample. Second each possible number of discrete steps needs to be considered for finding spurious counterexamples, if necessary. Finally the special properties of the discrete fixpoint needs to be regarded for finding spurious counterexamples.

So for a real counterexample bridging state sets from a discrete fixpoint iteration with j steps, we must consider the path segments

$$\begin{array}{l}
\ldots s_i^{c2d} \overset{c2d}{\rightarrow} s_{i-1}^{d2c} \ldots \\
\ldots s_i^{c2d} \overset{c2d}{\rightarrow} s_{i,1}^{d} \overset{d}{\rightarrow} s_{i-1}^{d2c} \ldots \\
\vdots \\
\ldots s_i^{c2d} \overset{c2d}{\rightarrow} s_{i,j-1}^{d} \overset{d}{\rightarrow} \ldots \overset{d}{\rightarrow} s_{i,1}^{d} \overset{d}{\rightarrow} s_{i-1}^{d2c} \ldots
\end{array}$$

For embedding into an incremental SMT problem, it is much more convenient to have only path segments of the same length, so we fill up the missing steps with τ-transitions, which do not alter the state.

$$\begin{array}{lll}
\ldots\ s_i^{c2d} \overset{c2d}{\rightarrow} s_{i,j-1}^{d} \overset{\tau}{\rightarrow} \quad \ldots \quad \overset{\tau}{\rightarrow} s_{i,1}^{d} \overset{\tau}{\rightarrow} s_{i-1}^{d2c} \ldots & // & s_{i,j-1}^{d} = \ldots = s_{i-1}^{d2c} \\
\ldots\ s_i^{c2d} \overset{c2d}{\rightarrow} s_{i,j-1}^{d} \overset{d}{\rightarrow} s_{i,j-2}^{d} \overset{\tau}{\rightarrow} \ldots \overset{\tau}{\rightarrow} s_{i,1}^{d} \overset{\tau}{\rightarrow} s_{i-1}^{d2c} \ldots & // & s_{i,j-2}^{d} = \ldots = s_{i-1}^{d2c} \\
\vdots & & \\
\ldots\ s_i^{c2d} \overset{c2d}{\rightarrow} s_{i,j-1}^{d} \overset{d}{\rightarrow} \quad \ldots \quad \overset{d}{\rightarrow} s_{i,1}^{d} \overset{d}{\rightarrow} s_{i-1}^{d2c} \ldots & &
\end{array}$$

The τ-transition is introduced by extending the transition relation $Trans^d$ to $Trans^{d_\tau}$ by adding identity pairs for all states in which no guarded assignment in DT^d is enabled.

Using this extension, we can deal with SMT problems with a discrete state set $\vec{\phi}[i]^d\langle i\rangle$ on-top: If the problem is solvable, it is extended with $Trans^{d_\tau}$ and the next state set $\vec{\phi}[i-1]^{t_{i-1}}$. If the problem is not solvable, the state set on-top is replaced with $\vec{\phi}[i]^{d,ov}\langle i\rangle$ and the result problem is used for spurious counterexample detection.

For continuous-to-discrete steps, one final problem needs to be solved: The base of continuous-to-discrete overapproximation is the discrete fixpoint, which is a disjunction of (possibly overapproximated) discrete state sets. So during modelchecking continuous-to-discrete preimages are computed for states added during overapproximation of the fixpoint and during overapproximation of discrete state sets, which need to be treated separately for spurious counterexample detection. This is done with a little trick: First we check whether a state in the discrete fixpoint is reachable with a continuous-to-discrete step. If this is not the case, then the fixpoint overapproximation is the cause of the spurious counterexample. If a state in the discrete fixpoint is reachable, then we switch to searching a path through the discrete state sets (and treating discrete state set overapproximations, if necessary). This is achieved with a sequence of incremental SMT problems. Starting from the SMT problem of depth i

$\cdots \wedge \vec{\phi}[i]^{c2d}\langle i\rangle$

first an extension with $Trans^{c2d}$ and $\vec{\phi}[i-1]^{d_{fix}}$ is made. If the resulting SMT problem

$\cdots \wedge \vec{\phi}[i]^{c2d}\langle i\rangle \wedge Trans^{c2d} < i, i-2 > \wedge \vec{\phi}[i-1]^{d_{fix}}\langle i-2\rangle$

is not solvable, then the state set on-top is replaced with the corresponding overapproximation $\vec{\phi}[i-1]^{d_{fix},ov}$, and spurious counterexample detection and refinement is performed. If the problem is solvable, then the state set on-top is replaced with the last discrete state set of the fixpoint iteration $\vec{\phi}[i-2]^{d}$. If now the resulting SMT problem

$\cdots \wedge \vec{\phi}[i]^{c2d}\langle i\rangle \wedge Trans^{c2d} < i, i-2 > \wedge \vec{\phi}[i-2]^{d}\langle i-2\rangle$

is not solvable, then the state set on-top is replaced with the corresponding overapproximation $\vec{\phi}[i-2]^{d,ov}$, and spurious counterexample detection and refinement is performed. If the problem is solvable, then SMT problem extension for discrete steps is started.

End of Counterexample Generation. Counterexample generation ends, if either $unsafe$ is reached, or if no further step towards $unsafe$ can be made. Unlike in the case of a uniform transition relation above, $unsafe$ can be reached in any step of the last mainloop.

Counterexample Generation Algorithms. All components are put together in Alg. 15.

Excluding CEX points. Refinement is triggered with a SMT problem, whose solutions describe spurious counterexamples. The spurious counterexamples are used to fill reach set approximations for the corresponding step types.

There is one basic problem: The overapproximation, which caused the spurious counterexample, may have added an infinite number of states. Each solution itself provides only a finite number of reachable points, so if only these points are excluded, possibly an infinite number of solutions must be created. Luckily the reach set approximations are used during interpolation to limit overapproximation, and with each additional point in the reach set approximation each performed interpolation possibly removes an infinite number of reachable states, thus spurious counterexamples.

Thus we integrate interpolation and reach set approximation tightly: Immediately after adding a counterexample point to the reach set approximation, a new state set overapproximation limited by the extended reach set approximation and previous overapproximation is computed. If the new reach set approximation still yields spurious counterexamples, a new counterexample point is computed and added to the reach set approximation. This process is repeated in a loop, until no further spurious counterexamples can be generated.

Depending on how well the generated counterexample points stake out the reachable state space, more or less loop iterations are needed. In practice the default counterexample points work reasonably well, however in rare cases additional measures need to be taken to ensure a good distribution, including shifting counterexample points as close to the state set as possible, or generating counterexample points only with a minimum distance to each other. For all models used in the evaluation, this was not necessary, and the algorithm Alg. 16 was executed without modifications.

Generalization of CEX points using Quantifier Elimination. Generally speaking, a counterexample is a finite sequence of states $(s_n, s_{n-1}, \ldots, s_0)$, where s_n is an initial state, together with a sequence of transition types $(t_n, t_{n-1}, \ldots, t_1)$, $t_i \in \{c2d, d, d2c, flow\}$ for $i = n, \ldots, 1$, such that we obtain the connected sequence:

$$s_n \xrightarrow{t_n} s_{n-1} \xrightarrow{t_{n-1}} s_{n-2} \xrightarrow{t_{n-2}} \ldots \xrightarrow{t_1} s_0.$$

All states s_i, $i = n, \ldots, 0$, of the counterexample are reachable from the initial states.

In this section we discuss the generalization of *spurious* counterexamples. Let us assume that we detected a spurious counterexample $(s_n, \ldots, s_k)$ and the corresponding transitions $(t_n, \ldots, t_{k+1})$ during our CEGAR approach, i. e., the overapproximation ϕ_k^{ov} of ϕ_k caused the spuriousness of the counterexample. Note that the sequences (s_i) and (t_i) can be extracted from Alg. 15. In order to avoid the spurious counterexample, the causing overapproximation has to be revised – a procedure which we call refinement. The simplest method is to add s_k to the exclude set and to redo the overapproximation. The newly computed overapproximation does not contain s_k and $(s_n, \ldots, s_k)$ is no longer a spurious counterexample. Since this approach does not guarantee

```
1  generateCounterexample(init, unsafe, GC, φ⃗, φ^{ov⃗_lb}, φ^{ov⃗}) :
2  begin
3      SMTproblem.push ((φ[n] ∧ Init ∧ GC) < n >);
4      SMTmodel := SMTproblem.solve ();
5      if inFirstMainloop(φ[n]) then
6          SMTproblem.push (unsafe < n >);
7          SMTmodel := SMTproblem.solve ();
8          unsafeReached := SMTmodel.solvable;
9          SMTproblem.pop ();
10     i := n; i_next := i − 1;
11     while i > 0 ∧ ¬unsafeReached ∧ SMTmodel.solvable do
12         if type(φ⃗[i]) = c2d ∧ type(φ⃗[i_next]) = dfix then
13             SMTproblem.push (Trans^{type_i} < i, i − 2 >);
14         else if type(φ⃗[i]) ≠ c2d then
15             SMTproblem.push (Trans^{type_i} < i, i − 1 >);
16         SMTproblem.push ((φ⃗[i_next] ∧ GC) < i_next >);
17         SMTmodel := SMTproblem.solve ();
18         if inFirstMainloop(φ⃗[i_next]) then
19             SMTproblem.push (unsafe < n >);
20             SMTmodel = SMTproblem.solve ();
21             unsafeReached = SMTmodel.solvable;
22             SMTproblem.pop ();
23         if type(φ⃗[i]) = c2d ∧ type(φ⃗[i_next]) = dfix then
24             SMTproblem.pop ();
25             i_next = i_next − 1; ;
26         else
27             i := i_next; i_next := i − 1;
28     if SMTmodel.solvable then
29         result.ceFound := true;
30         computeTrace(SMTmodel);
31     else
32         SMTproblem.pop ();
33         result.ceFound := false;
34         computeRefinement(SMTproblem, φ^{ov⃗}, φ^{ov⃗_lb}, i_next, ReachA^{type(φ⃗[i])});
```

Algorithm 15: Counterexample generation.

```
1  computeRefinement(SMTproblem, φ^{ov⃗}, φ^{ov⃗_lb}, i_next, ReachA) :
2  begin
3      repeat
4          SMTproblem.push ((φ^{ov⃗}[i_next] ∧ GC) < i_next >);
5          SMTmodel := SMTproblem.solve ();
6          if SMTmodel.solvable then
7              cexPoint := computeCexPoint(SMTmodel);
8              ReachA := ReachA ∪ {cexPoint} ;
9              φ^{ov⃗}[i_next] := interpolate(φ^{ov_lb}[i_next] ∧ GC ∧ Inv, (¬φ^{ov⃗}[i_next] ∨ ReachA) ∧ GC ∧ Inv);
10     until ¬SMTmodel.solvable;
```

Algorithm 16: Refinement.

the absence of other spurious counterexamples, we have to check again for a spurious counterexample and exclude it just as before until no further spurious counterexample can be found. While in practice the repeated search does often terminate due to interpolation, we cannot formally guarantee termination of the refinement.

We would like to have methods which allow to exclude many spurious counterexamples at once and introduce only mild additionally computational costs. Ideally, such a method also guarantees termination of the refinement. The following two paragraphs describe two possible methods to generalize single counterexample by usage of quantifier elimination.

Excluding Cones. Let $s_n \xrightarrow{t_n} s_{n-1} \xrightarrow{t_{n-1}} s_{n-2} \xrightarrow{t_{n-2}} \dots \xrightarrow{t_{k+1}} s_k$ be a spurious counterexample. Thus, s_{k+1} is a reachable state and its post-image under $\mathit{Trans}^{t_{k+1}}$, i. e., the set $\{\vec{x} \mid \mathit{Trans}^{t_{k+1}}(s_{k+1}, \vec{x})\}$, is also reachable and its quantifier free equivalent can be computed using quantifier elimination. Obviously, the post-image is a generalization of the exclude point s_k and can be added to the exclude set. A variant of this method can be used in the special case where $t_{k+1} = d2c$. In this case, the sequence $s_n \xrightarrow{t_n} s_{n-1} \xrightarrow{t_{n-1}} s_{n-2} \xrightarrow{t_{n-2}} \dots \xrightarrow{t_{k+1}} s_k$ can be extended by any state x which is reachable from s_k by a continuous flow. Hence, instead of adding s_k to the exclude set we may add the continuous post-image of s_k to the exclude set. Clearly, the continuous post-image of s_k also contains s_k.

Generalization of CEX points using Partial Quantifier Elimination. Let $s_n \xrightarrow{t_n} s_{n-1} \xrightarrow{t_{n-1}} s_{n-2} \xrightarrow{t_{n-2}} \dots \xrightarrow{t_{k+1}} s_k$ be a spurious counterexample. This sequence is a valid trajectory of the hybrid automaton describing a possible evolution of the initial state s_n along the transitions given by the sequence $(t_n, \dots, t_{k+1})$. On the other hand, the following formula characterizes all reachable states from some initial state for the same sequence of transitions:

$$\exists x_n, x_{n-1}, \dots x_{k+1}.\, \mathit{init}(x_n) \wedge \mathit{Trans}^{t_n}(x_n, x_{n-1}) \wedge \dots \wedge \mathit{Trans}^{t_{k+1}}(x_{k+1}, x_k)$$

An easy transformation of the formula provides a formula which collects all reachable states along the given sequence of transitions:

$$\begin{aligned} \exists x_n, x_{n_1}, \dots x_k.\, \mathit{init}(x_n) \wedge \mathit{Trans}^{t_n}(x_n, x_{n-1}) \wedge \dots \wedge \mathit{Trans}^{t_{k+1}}(x_{k+1}, x_k) \\ \wedge (x = x_n \vee x = x_{n-1} \vee \dots \vee x = x_k). \end{aligned} \tag{7}$$

In principle, we could use quantifier elimination to eliminate the variables $x_n, \dots, x_k$ and obtain a quantifier free formula $\mathit{Reach}(x)$ representing all reachable states along the given sequence of transitions. Adding Reach to the exclude set would prevent the overapproximation to contain any spurious counterexample for the given sequence of transitions. Hence, using expensive quantifier elimination would guarantee termination of the refinement.

Instead of using the expensive but complete quantifier elimination, we propose to use a partial quantifier elimination, like it is presented in Sect. 3.3.4. The method takes a formula and a satisfying assignment of the variables to compute an underapproximation of the quantifier-free equivalent. In the case of a spurious counterexample $s_n \xrightarrow{t_n} s_{n-1} \xrightarrow{t_{n-1}} s_{n-2} \xrightarrow{t_{n-2}} \dots \xrightarrow{t_{k+1}} s_k$ the t_i, $i = n, \dots, k+1$, are used to determine the formula (7) and the s_i, $i = n \dots, k$, provide a satisfying assignment for (7). Hence, we can determine those test points of the Loos-Weispfenning method, which lead to a satisfying sub-formula $\mathit{PartialReach}(x)$ of $\mathit{Reach}(x)$. Now, we add $\mathit{PartialReach}(x)$ to the exclude set, refine and check for further spurious counterexamples. Let us discuss the termination of this refinement procedure: First, let us remark that there are only finitely many different sequences of transitions $(t_n, \dots, t_{k+1})$. Hence, it suffices to show that the proposed method excludes all spurious counterexamples for a fixed sequence of transition within a finite number of refinement loops. For a given sequence of transitions, we are done as soon as the exclude set contains all states of $\mathit{Reach}(x)$, where $\mathit{Reach}(x)$ is exactly the set of all reachable states for the given sequence of transitions. By construction, the end point of a spurious counterexample $s_n \xrightarrow{t_n} s_{n-1} \xrightarrow{t_{n-1}} s_{n-2} \xrightarrow{t_{n-2}} \dots \xrightarrow{t_{k+1}} s_k$ cannot be a point of the exclude set. That is, each newly found spurious counterexample either has a different transition sequence, or it differs in at least one state $s_n, \dots, s_k$ from the exclude set. Hence, the corresponding satisfiable sub-formula $\mathit{PartialReach}(x)$ has at least one point which is not yet contained in the exclude set. On the other hand, the method in Sect. 3.3.4 only generates finitely many different sub-formulas of $\mathit{Reach}(x)$. Hence, the refinement procedure is guaranteed to terminate.

4.5 Per Mode Overapproximation

If per mode acceleration is used, then it is also possible to perform *per mode overapproximation*. Unlike in the approach described in Alg. 13 (which we call *overapproximation at once*), where we use the complete c2d onion ring, the complete flow reach set and the complete c2d state set as base for overapproximation, and then for each mode compute a cofactor of the overapproximation and use it for mode preimage computation, we can compute cofactors of the c2d onion ring, the flow reach set and c2d state set, and then use these cofactors as base for computing mode specific c2d onion ring cofactor overapproximations, which again are used for mode preimage computation, in hope that the gain of the mode specific overapproximation exceeds the cost of computing the additional cofactors.

Also for per mode overapproximation the integration into the overall cegar algorithm needs to be considered. Luckily counterexample generation and refinement computation do not need to be changed, as long as the mode specific overapproximations are combined to an overall c2d overapproximation, which can be used in the counterexample generation algorithm. For restarting, however, again mode specific cofactors of the refined c2d overapproximation are computed, and used for mode preimage computation.

The approach is described in Alg. 17, which contains the updated continuous flow section of the onion algorithm using abstraction with restart.

4.6 Related Work CEGAR

The general undecidability of model checking of hybrid systems and the often observed state set explosion in typical reach set computations makes the usage of abstraction techniques attractive. Clearly, the choice of an appropriate abstraction is again a difficult problem. Ideally, an abstraction is fine enough to preserve the safeness and it is coarse enough to allow efficient computation. An appealing idea is to start with a coarse abstraction and successively refine the abstraction guided by counterexamples.

Many approaches, like [67, 68, 69], use abstractions which transform the hybrid systems inherently having an infinite state space into finite state models: The papers [68, 69] use discrete transitions systems as abstractions. In [68] the validation of counterexamples and the refinements are computed simultaneously along a concretization of the abstract path of the counterexample. In contrast, [69] extracts conflicting guard-/jump-sequences from spurious counterexamples. These conflicting sequences are then ruled out in the abstract model. [67] utilizes a predicate abstraction of the concrete hybrid system and searches the abstract state space for counterexamples for the abstract safety proof. If no such intersection exists, the concrete model is safe and we are done. Otherwise, it has to be checked whether the abstract counterexample describes concrete counterexamples. To this end, the abstract counterexample has to be concretized using the concretization of the involved abstract states and transitions, and the concrete post-image computation. Either such a concretization is successful and a concrete counterexample has been found, or the concretization fails. In the later case, the abstraction has to be refined in order to exclude the spurious abstract counterexample from further search. The concretization of an abstract counterexample is an expensive post-image computation of bounded length and ideally uses an exact post-image computation to avoid unnecessary refinements. Interestingly, during their refinement procedure they use a subdivision approach to find a separating predicate for two disjoint sets of polyhedra, a problem which is highly related to our interpolant computation.

[70] uses hyper-rectangles as abstract states. A refinement step corresponds to splitting the boxes and re-computation of the possible transitions. In order to avoid the possible exponential blowup due to the splitting, the authors propose to use constraint propagation. The approach is implemented in HSolver and deals with non-linear hybrid systems.

[71] propose a CEGAR approach for rectangular hybrid systems. They use abstractions which map an hybrid system to a hybrid system. The *strong reset* abstraction maps a rectangular hybrid system to an abstract initialized rectangular hybrid system by altering the assignments of certain transitions into a non-deterministic choice within an admissible interval. For the later class of hybrid systems it is well-known that there exists a procedure to decide safety [21]. The corresponding refinement introduces additional transitions by splitting the interval of non-deterministic choice. This abstraction yields a semi-decision procedure for rectangular hybrid systems. For the class of initialized rectangular systems the authors propose to use control abstraction merging locations and transitions, and a flow abstraction based on dropping and scaling of variables. The authors show that their CEGAR approach using the later two abstractions yields a decision

```
1  modelCheck(init, safe, GC, restartData) :
2  begin
       // Initialization:
       ⋮
       // Mainloop:
3      while true do
4          i := i + 1;
           // Discrete fixed point iteration:
           ⋮
           // Evaluate c2d transitions:
           ⋮
           // Evaluate continuous flow:
52         if ¬restarting ∨ (restarting ∧ restartData.restartSection = flow) then
53             if ¬restarting then
54                 ψ_i^{c2d,ov_lb} := ψ_i^{c2d} ∧ ¬φ_{i-1}^{c2d};
55             else
56                 ψ_i^{c2d,ov} := restartData.ψ_i^{c2d,ov};
57             φ_i^{c2d,ov} := φ_{i-1}^{c2d};
58             foreach h ∈ {1 ... k} do
59                 if ¬restarting then
60                     φ_{i,m_h}^{c2d} := φ_i^{c2d}|_{m⃗=m_h};
61                     ψ_{i,m_h}^{c2d,ov_lb} := ψ_i^{c2d,ov_lb}|_{m⃗=m_h};
62                     reachA_{m_h}^{flow} := reachA^{flow}|_{m⃗=m_h};
63                     ψ_{i,m_h}^{c2d,ov} := overapprox(ψ_{i,m_h}^{c2d,ov_lb}, (eps_bloat(ψ_{i,m_h}^{c2d,ov_lb}, ϵ) ∧ ¬reachA_{m_h}^{flow}) ∨ φ_{i,m_h}^{c2d});
64                     ψ_i^{c2d,ov} := ψ_i^{c2d,ov} ∨ (ψ_{i,m_h}^{c2d,ov} ∧ (m⃗ = m_h));
65                     φ_i^{c2d,ov} := φ_i^{c2d,ov} ∨ (ψ_{i,m_h}^{c2d,ov} ∧ (m⃗ = m_h));
66                 else
67                     ψ_{i,m_h}^{c2d,ov} := restartData.ψ_i^{c2d,ov}|_{m⃗=m_h}
68                 ψ_{i,m_h}^{flow} := constraint_min(Pre^c(ψ_{i,m_h}^{c2d,ov}, W_h, β_h), φ_{i-1}^{flow}|_{m⃗=m_h});
69                 ψ_i^{flow} := ψ_i^{flow} ∨ (ψ_{i,m_h}^{flow}(m⃗ = m_h);
70             if restarting then
71                 restarting := false;
72             if GC ∧ ψ_i^{flow} ∧ init ≠ 0 then return false;
73             if GC ∧ ψ_i^{flow} ∧ ¬φ_{i-1}^{flow} = 0 then return true;
74             ψ_i^{flow} := constraint_min(ψ_i^{flow}, φ_{i-1}^{flow});
75             φ_i^{flow} := φ_{i-1}^{flow} ∨ ψ_i^{flow};
           ⋮
```

Algorithm 17: Backward onion algorithm using abstraction with restart, continuous flow section with per mode overapproximation.

procedure for initialized hybrid systems.

5 Experimental Evaluation

5.1 Description of Benchmarks

In the following we describe the models used for benchmarking. In general the models describe one or several controllers on a continuous plant. The models are specified in an HLANG [72] dialect for FOMC, which provides integer values for user convenience. The integer variable are translated to a boolean representation internally.

We characterize the complexity of the modeled system in terms of the discrete and continuous state spaces. The discrete state space is determined by the cross-product of discrete variables ranges. The continuous state space is characterized by the number of real variables in the model.

In order to examine how the FOMC copes with increasingly more complex behavior, we also consider scaled up version of the same system. The prefix of a model name refers to the model (family) and is followed by a reference to the scaled instance. The suffix *inv* encodes that lemmata are part of the model description, that are tagged as invariants, whereas the suffix *gc* encodes that lemmata are part of the model description, that are tagged as global constraints. In both cases, the FOMC is able to establish the annotated lemmata on the given model.

Flap/Slat Controller. The Flap/Slat Controller benchmark (*flapCtrl*,*flapSlatCtrl_ 2of2Lp2F*) is derived from a case study for Airbus [73]. During take-off (and landing) flaps and slats of the aircraft are extended to generate more lift at low velocity and have to be retracted in time, as they are not robust enough for high velocities. A controller corrects the pilot's commands, if they endanger the flaps or slats. The safety property to be established for our model is that the velocity of the aircraft never exceeds the allowed velocities for the current flap/slat positions, respectively.

Discrete variables encode the number of lever positions, stages of the controller state machine. A mode variable per flap/slat encodes whether the flap/slat is decreasing, increasing or at stand-still. Continuous variables are the plane velocity and the angles of flaps/slats. To generate increasingly challenging model instances, we scaled the model in the number of lever positions and in the number of flaps/slats.

Scaling only the number of lever positions (*flapCtrl*), we considered models with a discrete state space up to $2048 = 2^{11}$ states and two continuous variables, *flapCtrl_ 10of10Lp* .

The model family *flapSlatCtrl_ 2of2Lp2FXXS* has been derived to generate models with increasingly more parallel components. Therefore we scale up the number `XX` of slats on an aircraft with already two flaps, while the lever positions are limited to two at all considered instances. The smallest model *flapSlatCtrl_ 2of2Lp2F0S* has a discrete state space of 2^{11} states and three continuous variables, whereas the most complex instance considered, *flapSlatCtrl_ 2of2Lp2F20S*, has a discrete state space of 2^{71} states and 23 continuous variables.

Approach Velocity Controller. These benchmarks model controllers regulating the speed of a car via its acceleration. We consider two distinct controller implementations. *acc_ pi* applies a PI control loop mechanism. Its objective is to make the car drive a given speed. We consider a linearized version of this model. The resulting model has a discrete state space of 16 states and two continuous variables.

The second controller switches between constant accelerations to implement a strategy for a prioritized list of objectives. Its top priority is to guarantee collision freedom, its second priority objective is to follow the preceding car not closer than within a certain distance. Also the controller tries to make the car drive at a certain goal speed, if this does not endanger its more important objectives. We examine this controller in two variants for collision freedom. *(i)* The goal speed is arbitrary but constant (*avc_ arb*), *(ii)* the goal speed changes (*avc_ goal*). The continuous evolution of the distance between two cars is overapproximated via linearization. We implement the linearization via modes, that correspond to velocity ranges and for which they define appropriate bounded derivatives for the distance evolution. The linearization granularity is encoded into the model name via the suffix *_ hX* where `X` is the number of linearization modes. Both, *avc_ arb* and *avc_ goal*, models have 8 continuous variables and discrete state space ranging from 512 to 2^{11} states, depending on the considered linearization granularity.

Bouncing Ball. *bb_ h22* is a simple model of a bouncing ball. A continuous variable x represents the current height of the ball over the floor, a second continuous variable v represents the current velocity of

the ball. The dynamics of the ball is given by $\dot{x} = v$, $\dot{v} = -1 \pm 0.05$. The ball bounces as soon as it reaches the floor. At a bounce the speed of the ball is discretely reduced. It is verified that the ball, being dropped from an initial height, never exceeds a certain height. This continuous dynamics is overapproximated into 22 modes of bounded derivatives. The model has a discrete state space of 32 states and three continuous variables.

Intersection Controller. The intersection controller monitors the traffic flow at a two road intersection. The controller has to guarantee that a car approaching the intersection on a minor road is granted access to the intersection only if its save passage can be guaranteed without disrupting the traffic flow of the major road. In order to determine whether a car on a minor road may cross the intersection, the controller conservatively overapproximates the future traffic evolution. Based on its traffic prediction, the controller computes a waiting time for the car on the minor road, after which the intersection can be crossed safely.

For these models we examine two kinds of safety properties. The first safety property describes collision freedom at the intersection and holds on the considered models, whereas the second safety property does not hold (model names with *dist*) and therewith illustrate that a car on a minor road actually may enter the intersection.

We consider several distinct but closely related controllers. The *interC_ s+g* controller orders a car on a minor road not to reduce its speed, if this enables the car to cross the intersection right away. In contrast, *interC_ s* controllers only decide on an appropriate waiting time for a gap between two modeled cars.

The model has continuous variables for wait timers, distances and velocities. Discrete variables encode the state of the controller and encode the protocol for a car on a minor road approaching and crossing the intersection. Further we consider overapproximated linearized dynamics for the distance evolution implemented via modes of bounded derivatives. The continuous evolution of waiting time and velocity is specified via constant derivatives.

We consider two different encodings of *interC_ s+g* controllers. The first variant has a discrete state space of $4096 = 2^{12}$ states and 5 continuous variables, whereas the second variant has $16384 = 2^{14}$ discrete states and 5 continuous variables. We also consider two variants of *interC_ s* controller models. The first variant has a discrete state space of $1024 = 2^{10}$ states and 5 continuous variables, whereas the second variant has 128 discrete states and 5 continuous variables.

ETCS Train Crossing Controller. We derived a couple of benchmark models (names with *etcs*) from the Etcs collision avoidance protocol. The protocol consists of two parts, the speed supervision of the train and a cooperation protocol between the train and a radio block center that grants movement authorities to the train. The train has to stop before reaching the end of its current movement authority. The protocol also maintains some error control and signals a failure to the on-board speed supervision. Details on the reference model in Matlab/Simulink and the derived models can be found in [74].

In the plain version the model has 16 modes, two continuous variables, the speed and train position. The discrete variables describe the different phases of the protocol. An input variable models whether the gate is closed, in which case the train can safely continue. We consider two different encodings, where the discrete state space has 2^{10} or 2^{14} states, respectively.

We also consider model variants (*etcs_ error*) that capture additionally failures at the train, radio block center or signaling. The plain *etcs_ error* models have a discrete state space of 2^{24} (including inputs) and an additional continuous variable with constant derivative to model a time-out. The variant *interrupt* has a discrete state space of 2^{25} states and two continuous variables.

Dam. Dam models describe a setting where water impounds and the dam uses it to produce energy with the help of n turbines. The water level of the reservoir is determined by an inflow of water towards the dam and an outflow of water through n turbines. A turbine has to undergo maintenance, depending on the number of switching operations of the turbine. The controller switches the turbines based on the current water level; its strategy is "disturbed" by races due to turbines which are currently in maintenance mode. The controller has to guarantee the safety property that the water level always stays within given bounds.

Again we derive from the basic model several variants of different complexity. Whether the turbine has to undergo maintenance depends on the number of times the turbine has been switched on and off. A turbine spends a certain time in maintenance. Hence these models have continuous variables to monitor the time spend at maintenance and additional discrete variables to count their switchings. The model family *dam_ 3turb_ mYY−XX.hlang*, is derived by varying the maintenance time and switching thresholds. The most

Option	Redundancy Removal	Onioning	Constraint Minimization	Polarity Minimization	Quantifier Elimination
1	off	off	off	off	naïve
2	on	off	off	off	naïve
3	on	on	off	off	naïve
4	on	on	on	off	naïve
5	on	perMode	on	off	naïve
6	on	on	on	on	naïve
7	on	on	on	on	distributive
8	on	on	on	off	SMT-based

Table 1: Settings for exact model checking using acceleration per mode

complex model of these has 4608 discrete states and 4 continuous variables.

We extended this model, so that for each turbine there are two kind of maintenance modes, short and long. Whether the turbine has to undergo maintenance depends on the number of switchings and on its total run time. Also *dam_ random_ XXturb_ m13t3.hlang* uses inputs to implement random selection in a turbine switching strategy for turbines with a maintenance duration of 13 and a switching threshold of three. The smallest instance of this family within this benchmark set is *dam_ random_ 4turb_ m13t3.hlang* with a discrete state space of 2^{24} and 5 continuous variables. The biggest instance for which the FOMC terminated within two hours is *dam_ random_ 7turb_ m13t3.hlang* with a discrete state space of 2^{42} and 8 continuous variables. The models *dam_ random_ XXturb_ m13.9t7.hlang* are closely related. They have an extended maintenance duration and an increased thresholds for switchings before the turbine has to undergo maintenance. *dam_ random_ 2turb_ m13.9t7_ .hlang* has a discrete state space of 2^{17} states and three continuous variables, whereas *dam_ random_ 4turb_ m13.9t7_ random.hlang* has a discrete state space of 2^{28} and 5 continuous variables.

5.2 Evaluation

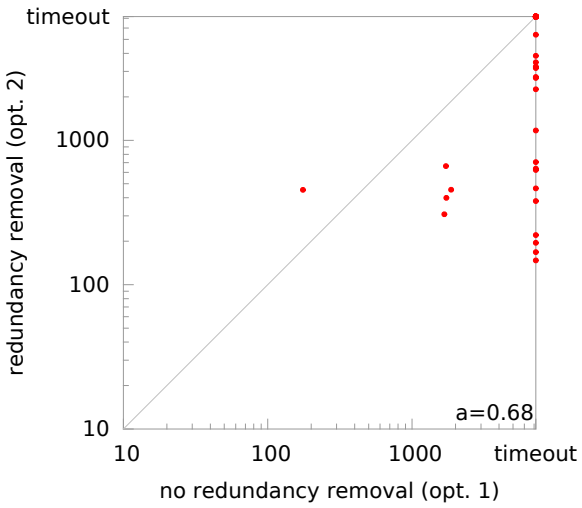
5.2.1 Exact Model Checking

In this section we present experimental results for various optimizations of exact model checking. All evaluated techniques have been described in Chapter 3. The interface of FOMC provides the possibility to activate or deactivate these techniques independently, resulting in an enormous number of possible combinations. However, some combinations are extremely useful while other are not. For instance, enabling onioning without constraint minimization is usually a bad idea, since it would prevent onioning from using don't cares. Hence, we are not comparing all possible combination against each other. Instead, we defined several *evaluation flows* and *evaluation groups*. So the aim of this chapter is twofold: We would like to show the benefits of several optimizations by experimental results in a meaningful order while, on the other hand, we try to follow the structure of Chapter 3 as close as possible, and we would like to isolate proper settings for efficient model checking with FOMC.

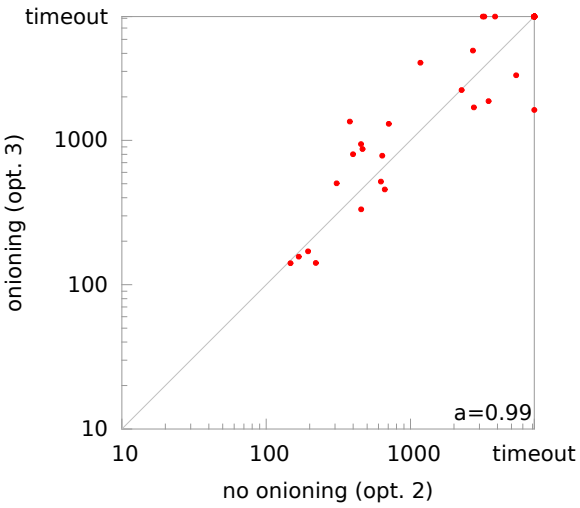
Settings for exact acceleration per mode. Table 1 provides a quick overview on the different settings for our evaluation of exact model checking with mode-wise computation of the continuous flow. The settings vary from no optimization to onioning and a different choices of quantifier elimination methods.

The benefits of onioning are evaluated in the evaluation flow 1–2–3–4, corresponding to Sect. 3.1 to Sect. 3.2.2. In the evaluation group 4–5 we compare the global onioning technique with per mode onioning, see Sect. 3.2.3. The optimizations for quantifier elimination as presented in Sect. 3.3 are compared in the evaluation group 4–6–7–8.

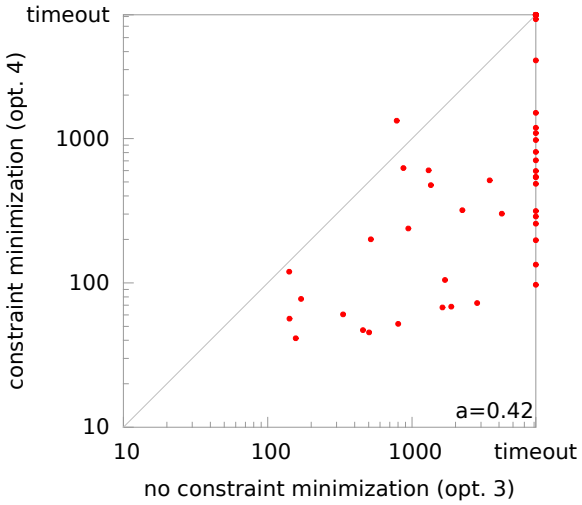
How to interpret the figures? In each figure we compare two options against each other. Each coordinate of a red point represents the run time of the respective options. The maximal possible value of each coordinate is agrees with the timeout-value. The gray diagonal indicates the location of all points for which the run time



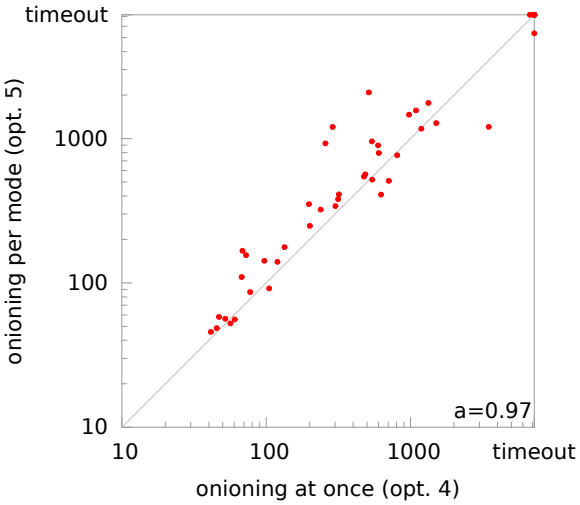
(a) Redundancy removal



(b) Onioning



(c) Constraint minimization



(d) Per mode onioning

Figure 23: From no optimizations to onioning

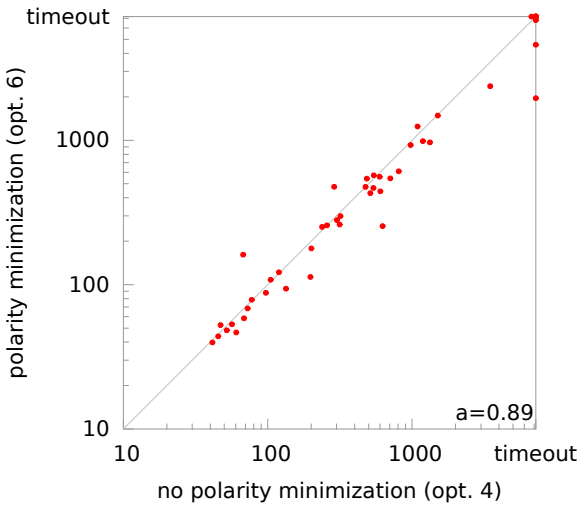
Model	Opt 1	Opt 2	Opt 3	Opt 4
acc_pi_h14	t.o.	147.18	140.71	**119.51**
avc_arb_h5	t.o.	t.o.	t.o.	**1090.21**
avc_goal_h11	t.o.	t.o.	t.o.	**977.05**
avc_goal_h5	t.o.	3165.03	t.o.	**133.68**
avc_goal_h7	t.o.	3855.24	t.o.	**314.86**
avc_goal_h9	t.o.	t.o.	t.o.	**596.10**
bb_h22	t.o.	**636.52**	782.50	1329.60
dam_3turb_m20-20	1671.13	307.63	504.20	**45.43**
dam_3turb_m20-40	t.o.	3468.90	1868.91	**68.46**
dam_3turb_m20-60	t.o.	t.o.	t.o.	**706.94**
dam_3turb_m40-20	1715.59	662.50	456.43	**47.11**
dam_3turb_m40-40	t.o.	5395.03	2823.66	**72.50**
dam_3turb_m40-60	t.o.	t.o.	t.o.	**543.08**
dam_3turb_m60-20	1862.96	454.62	332.34	**60.48**
dam_3turb_m60-40	t.o.	t.o.	1623.18	**67.61**
dam_3turb_m60-60	t.o.	t.o.	t.o.	**807.25**
dam_3turb_m80-20	1726.03	399.78	800.33	**51.98**
dam_3turb_m80-40	t.o.	2746.36	1690.46	**104.86**
dam_3turb_m80-60	t.o.	t.o.	t.o.	**485.44**
dam_random_2turb_m13.9t7	**175.21**	453.43	942.53	237.98
dam_random_4turb_m13.9t7	t.o.	t.o.	t.o.	**288.21**
dam_random_4turb_m13t3	t.o.	t.o.	t.o.	**197.50**
dam_random_7turb_m13t3	t.o.	t.o.	t.o.	**6713.08**
etcs	t.o.	220.73	141.43	**56.57**
etcs_error-detect	t.o.	623.14	517.70	**200.52**
etcs_error_interrupt	t.o.	2702.70	4185.44	**301.74**
etcs_error_inv	t.o.	194.80	170.14	**77.51**
etcs_inv	t.o.	168.11	156.30	**41.30**
etcs_v2_inv	t.o.	2257.34	2231.15	**319.02**
flapCtrl_10of10Lp	t.o.	t.o.	t.o.	**1508.14**
flapCtrl_6of10Lp	t.o.	**379.20**	1348.98	475.57
flapCtrl_7of10Lp	t.o.	707.07	1300.55	**601.95**
flapCtrl_8of10Lp	t.o.	1169.14	3451.12	**513.90**
flapCtrl_9of10Lp	t.o.	t.o.	t.o.	**1188.40**
interC_s_gc	t.o.	t.o.	t.o.	**257.40**
interC_s_v2	t.o.	t.o.	t.o.	**3476.31**
interC_s_v2_dist	**t.o.**	**t.o.**	**t.o.**	**t.o.**
interC_s_v2_gc	t.o.	t.o.	t.o.	**97.09**
interC_s+g_dist	t.o.	**464.67**	869.77	624.62
interC_s+g_v2_dist	t.o.	3242.16	t.o.	**539.50**

(a) Redundancy removal, onioning, and constraint minimization

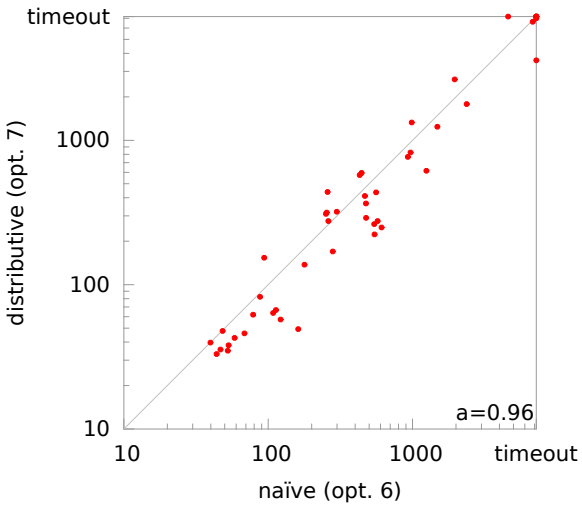
Opt 4	Opt 5
119.51	139.92
1090.21	1565.18
977.05	1462.36
133.68	176.95
314.86	380.68
596.10	895.78
1329.60	1765.71
45.43	48.61
68.46	166.84
706.94	**509.59**
47.11	58.06
72.50	155.50
543.08	**519.26**
60.48	**55.85**
67.61	109.80
807.25	**765.48**
51.98	56.44
104.86	**91.62**
485.44	565.63
237.98	321.88
288.21	1201.50
197.50	350.52
6713.08	t.o.
56.57	**52.37**
200.52	248.68
301.74	340.36
77.51	86.30
41.30	45.74
319.02	410.70
1508.14	**1279.09**
475.57	545.36
601.95	793.59
513.90	2087.34
1188.40	**1169.48**
257.40	925.20
3476.31	**1204.26**
t.o.	**5359.23**
97.09	142.28
624.62	**409.18**
539.50	955.88

(b) Onioning at once and per mode onioning

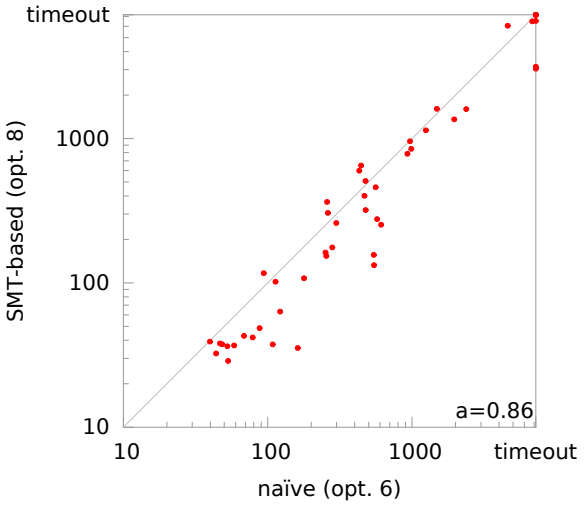
Table 2: The timeout (t.o.) was set to 7200 sec. Best results are shown in bold numbers. Rows containing timeouts only are not shown.



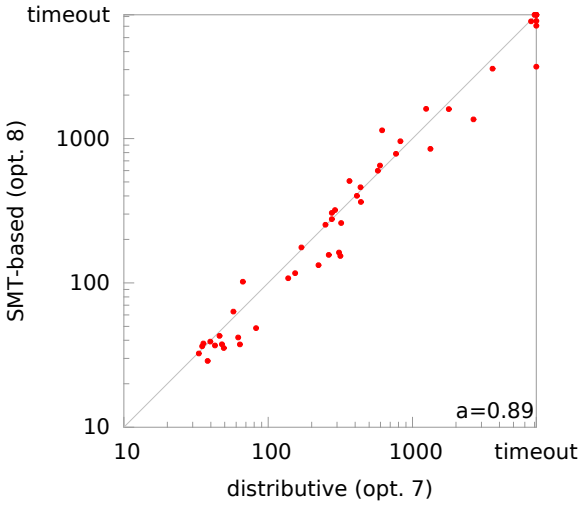
(a) Polarity minimization



(b) Distributive quantifier elimination



(c) SMT-based quantifier elimination



(d) SMT-based against distributive quantifier elimination

Figure 24: Polarity minimization and different quantifier elimination methods

Model	Opt 4	Opt 6	Opt 7	Opt 8
acc_pi_h14	119.51	121.78	**57.32**	63.27
avc_arb_h5	1090.21	1247.90	**615.62**	1139.56
avc_arb_h7	t.o.	1958.46	2644.47	**1358.64**
avc_arb_h9	t.o.	t.o.	3580.82	**3048.52**
avc_goal_h11	977.05	927.22	**767.62**	784.66
avc_goal_h21	t.o.	6803.18	6633.37	**6485.79**
avc_goal_h5	133.68	**93.89**	153.66	116.67
avc_goal_h7	314.86	**261.18**	275.77	305.29
avc_goal_h9	596.10	559.50	**435.84**	459.85
bb_h22	1329.60	968.17	**823.36**	956.98
dam_3turb_m20-20	45.43	43.90	33.04	**32.47**
dam_3turb_m20-40	68.46	58.46	42.79	**36.88**
dam_3turb_m20-60	706.94	545.22	222.77	**132.67**
dam_3turb_m40-20	47.11	52.43	**34.89**	36.35
dam_3turb_m40-40	72.50	68.45	46.00	**42.99**
dam_3turb_m40-60	543.08	572.58	**275.95**	276.23
dam_3turb_m60-20	60.48	46.70	**35.51**	38.04
dam_3turb_m60-40	67.61	161.36	49.22	**35.36**
dam_3turb_m60-60	807.25	610.25	**248.96**	253.05
dam_3turb_m80-20	51.98	48.32	47.78	**37.60**
dam_3turb_m80-40	104.86	108.23	63.61	**37.47**
dam_3turb_m80-60	485.44	542.12	262.73	**156.30**
dam_random_2turb_m13.9t7	237.98	251.43	309.30	**162.31**
dam_random_4turb_m13.9t7	**288.21**	476.69	290.26	319.27
dam_random_4turb_m13t3	197.50	113.16	**66.70**	101.89
dam_random_7turb_m13t3	**6713.08**	t.o.	t.o.	t.o.
etcs	56.57	53.18	38.07	**28.78**
etcs_error-detect	200.52	178.39	137.57	**107.72**
etcs_error_interrupt	301.74	279.75	**169.66**	176.24
etcs_error_inv	77.51	78.62	61.95	**41.91**
etcs_inv	41.30	39.78	39.71	**39.22**
etcs_v2_inv	319.02	298.48	320.20	**259.89**
flapCtrl_10of10Lp	1508.14	1484.68	**1241.63**	1604.85
flapCtrl_6of10Lp	475.57	475.32	**365.16**	507.90
flapCtrl_7of10Lp	601.95	**443.03**	594.38	650.02
flapCtrl_8of10Lp	513.90	**430.66**	574.92	598.67
flapCtrl_9of10Lp	1188.40	985.82	1327.86	**847.84**
interC_s_gc	**257.40**	257.63	438.60	363.52
interC_s_v2	3476.31	2369.68	1784.18	**1598.80**
interC_s_v2_dist	t.o.	**4594.53**	t.o.	6049.18
interC_s_v2_gc	97.09	87.74	82.45	**48.58**
interC_s+g_dist	624.62	254.52	316.26	**153.76**
interC_s+g_gc	t.o.	t.o.	**7008.12**	t.o.
interC_s+g_v2_dist	539.50	467.20	411.59	**401.52**
interC_s+g_v2_gc	t.o.	t.o.	t.o.	**6529.16**
interC_s+g_v2_inv	t.o.	t.o.	t.o.	**3143.96**

Table 3: Results for the comparison of different quantifier elimination methods. The timeout (t.o.) was set to 7200 sec. Best results are shown in bold numbers. Rows containing timeouts only are not shown.

of both runs agrees. For example, in Fig. 23(a) we compare the run times of option 1 (redundancy removal deactivated) on the x-axis against the run times of option 2 (redundancy removal activated) on the y-axis. In the upper right corner there is a red dot on the gray diagonal, indicating that there is at least one model in our benchmark set for which both options led to a timeout. Most of the red points lie below the diagonal, meaning that the x-value of their coordinates is larger than the y-value. Hence, for corresponding benchmarks their run time for option 1 (redundancy removal deactivated) is larger than the run time for option 2 (redundancy removal activated).

Most run times are relatively small, hence, we decided to scale both axes logarithmically for better clarity. Detailed values of the benchmark results are provided in an additional table. For example, let us look at Fig. 23(a) again. There is a single red point above the gray diagonal having coordinates of roughly $(170, 450)$. Indeed, the corresponding table Table 2(b) shows in the row labeled as "dam_ random_ 2turb_ m13.9t7" the values 175.21 and 453.43 for the associated columns to the options 1 and 2, respectively.

Each figure is decorated with the parameter a providing the slope of the regression line through the origin of the points. It was computed using the method of least squares, i. e., $a = \frac{\sum_{i=1}^{k} x_i y_i}{\sum_{i=1}^{k} x_i^2}$ where x_i and y_i are the coordinates of the point of the ith benchmark. Please note that the value alone has only limited explanatory power since it is greatly affected by outliers.

Redundancy removal, onioning, and constraint minimization. In order to show the benefits of the onioning technique as presented in Sect. 3.1 to Sect. 3.2.2 we show experimental results comparing the following optimization:

- *Redundancy removal* is a key technology to reduce the size of the state set representation. Moreover, it builds the basis for more elaborated techniques like *constraint minimization* and *polarity minimization*. Fig. 23(a) compares a non-optimized FOMC against FOMC with enabled redundancy removal. Redundancy removal significantly reduces the number of timeouts and is enabled in all following experiments.

- *Onioning* allows the partial pre-image computation of newly added states while possibly avoiding the pre-image computation of states whose pre-images have already been computed.

- *Constraint minimization* extends redundancy removal by don't cares. Constraint minimization uses the same principle of redundancy detection. The main difference to redundancy removal lies in the generation of the requested redundancy-free formula. The resulting formula has to be equivalent to the given formula outside of the don't care set only. Within the don't cares, constraint minimization is free to choose any representation. Constraint minimization exploits this additional freedom to compute a formula with a minimized number of linear constraints. In the context of onioning, we easily obtain don't cares, consisting of those states for which the pre-image has already been computed. Fig. 23(b) shows that onioning without constraint minimization is in general a bad idea. In contrast, Fig. 23(c) shows that the combination of onioning and constraint minimization is a powerful technique in exact symbolic model checking. In all following experiments onioning will used in the combination with constraint minimization only.

Per mode onioning. We compare onioning at once against per mode onioning. Per mode onioning is described in Sect. 3.2.3. Fig. 23(d) and Table 2(b) show that onioning at once emerges to be the better choice. Here we have an interesting case where the parameter a is greatly influenced by the outliers. While the parameter indicates, that per mode onioning would be the better choice in general, the figure and the table clearly show that onioning at once is the better choice for the majority of our benchmarks in our repository. For all further experiments we use onioning at once only.

Lesson learned from evaluation of onioning. Experience shows that the onioning technique is the best choice for the majority of our models and is a good starting point if less is known about the model. However, later on we will see that for certain model classes and upcoming settings, non-onioning might still be a good choice. At this point we note that onioning should not be used without constraint minimization and that per mode onioning usually does not pay off.

Polarity minimization and quantifier elimination. We compare different techniques for quantifier elimination as presented in Sect. 3.3.

- *Polarity Minimization* is a specialized version of redundancy detection. It does not only detect non-redundant constraints, but also the polarities for which the given constraints are non-redundant. Identifying non-redundant constraints and their polarity noticeable reduces the number of test points in the Loos-Weispfenning method and leads to smaller resulting formulas. This effect is confirmed in Fig. 24(a).
- The *distributive quantifier elimination* is described in Sect. 3.3.3. This method is specialized for elimination of several quantifiers of the same kind and exploits the structure of the intermediate resulting formulas by distributing the remaining quantifiers over this structure. This method heavily depends on the redundancy *detection* of the polarity minimization. In fact, beside of an initial and final redundancy removal, there is no intermediate redundancy *removal*. The benefits of the distributive quantifier elimination compared to naïve quantifier elimination are shown in Fig. 24(b). Note that distributive quantifier elimination has its full strength if there are more than one quantifier to eliminate. For example, the model class of Flap/Slat Controllers has only constant derivatives, leading to an elimination of a single quantifier only.
- The *SMT-based quantifier elimination* is described in Sect. 3.3.4. In contrast to former methods it does not profit from polarity minimization since only test point leading to satisfiable sub-formulas are used. As the distributive quantifier elimination intermediate redundancy removals are avoided. Redundancy removal is applied to the initial and finally resulting formula only. Fig. 24(c) documents the superior performance of the SMT-based approach compared to the naïve approach. Finally, Fig. 24(d) indicates that the SMT-based approach is generally of better performance than the distributive quantifier elimination.

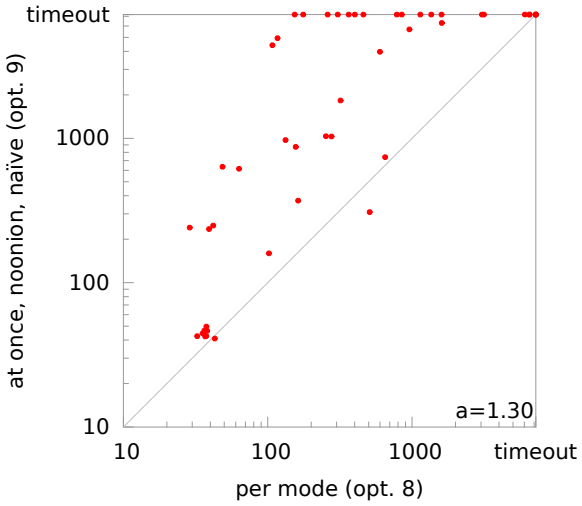
Lesson learned from evaluation of polarity minimization and quantifier elimination. The benchmark results provide clear evidence that the SMT-based quantifier elimination is the best choice for the majority of models. Moreover, with exception of SMT-based quantifier elimination, polarity minimization efficiently reduces the number of test points and should be activated. Prior experience have shown that the presented trend also carries over to non-onioning settings. Altogether, our experience and the presented results so far meet our expectations. However, all results so far were obtained using acceleration per mode. For acceleration at once we will make different observations which are partly more difficult to explain.

Acceleration at once. So far, all presented results were obtained by per mode acceleration. That is, the continuous pre-image is computed piecewise over the mode cofactors of the current state set. In this paragraph we evaluated an alternative technique called *acceleration at once*, see Sect. 3.4. Acceleration at once computes the continuous pre-image computation in a single step. On one hand, this leads to a more complex formula expressing the compound of the mode specific continuous flows, but, on the other hand, has the advantage that we do not need to compute the Cartesian product of several mode variables as it is the case in models with several parallel components.

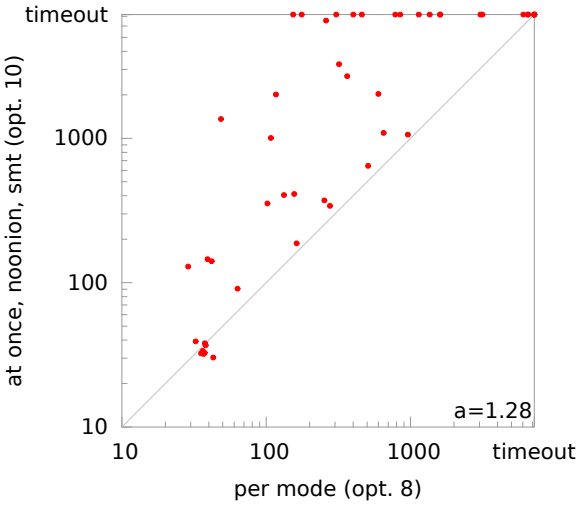
First, we present an evaluation of acceleration at once on the same benchmark set as it was used for acceleration per mode. Afterwards, we present very promising results on a specialized benchmark set.

Option	Acceleration	Onioning	Quantifier Elimination Method
(from Table 1) 8	perMode	on	SMT-based
9	atOnce	off	naïve
10	atOnce	off	SMT-based
11	atOnce	on	naïve
12	atOnce	on	SMT-based

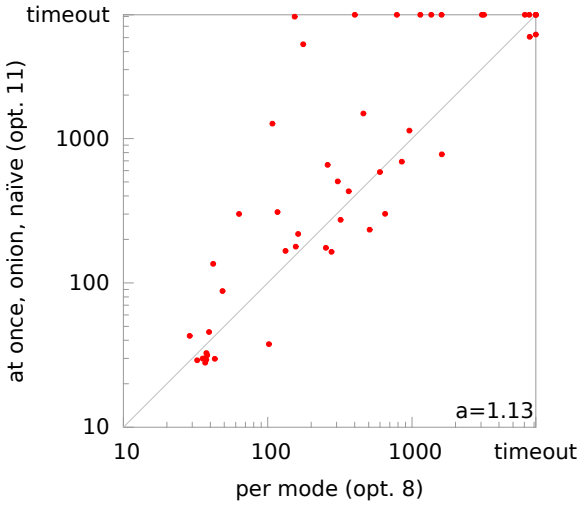
Table 4: Settings for exact model checking using acceleration at once



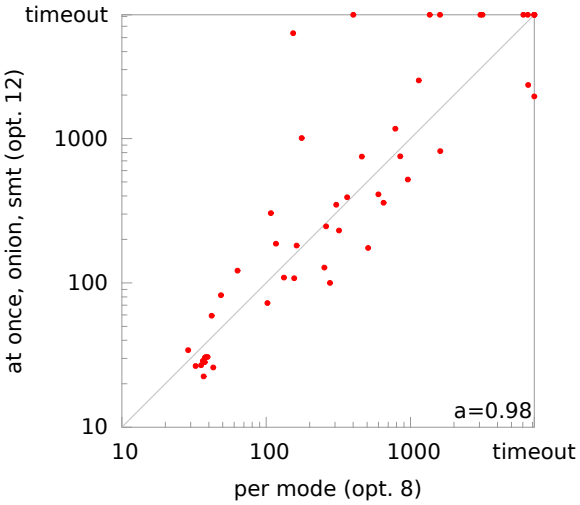
(a) Acceleration at once, no onioning, naïve



(b) Acceleration at once, no onioning, SMT-based



(c) Acceleration at once, onioning, naïve



(d) Acceleration at once, onioning, SMT-based

Figure 25: Acceleration at once compared with acceleration per mode

Model	Opt 8	Opt 9	Opt 10	Opt 11	Opt 12
acc_pi_h14	**63.27**	614.72	90.80	300.42	121.67
avc_arb_h5	**1139.56**	t.o.	t.o.	t.o.	2527.15
avc_arb_h7	**1358.64**	t.o.	t.o.	t.o.	t.o.
avc_arb_h9	**3048.52**	t.o.	t.o.	t.o.	t.o.
avc_goal_h11	**784.66**	t.o.	t.o.	t.o.	1169.64
avc_goal_h21	**6485.79**	t.o.	t.o.	t.o.	t.o.
avc_goal_h5	**116.67**	4937.50	2014.74	309.59	186.96
avc_goal_h7	**305.29**	t.o.	t.o.	504.73	347.58
avc_goal_h9	**459.85**	t.o.	t.o.	1489.50	750.01
bb_h22	956.98	5687.29	1060.41	1136.69	**520.25**
dam_3turb_m20-20	32.47	42.58	39.20	29.08	**26.56**
dam_3turb_m20-40	36.88	42.39	32.07	27.99	**22.51**
dam_3turb_m20-60	132.67	971.70	404.52	166.44	**108.89**
dam_3turb_m40-20	36.35	46.54	33.71	29.94	**28.81**
dam_3turb_m40-40	42.99	41.02	30.33	29.79	**25.95**
dam_3turb_m40-60	276.23	1029.95	340.46	164.08	**100.07**
dam_3turb_m60-20	38.04	46.42	36.87	31.56	**30.61**
dam_3turb_m60-40	35.36	44.56	32.39	29.90	**26.88**
dam_3turb_m60-60	253.05	1033.48	371.42	175.04	**127.60**
dam_3turb_m80-20	37.60	49.68	38.02	32.62	**30.23**
dam_3turb_m80-40	37.47	42.68	32.64	29.33	**28.23**
dam_3turb_m80-60	156.30	871.33	411.93	178.32	**107.62**
dam_random_2turb_m13.9t7	**162.31**	370.38	187.15	218.40	181.59
dam_random_4turb_m13.9t7	319.27	1827.96	3261.76	273.68	**230.83**
dam_random_4turb_m13t3	101.89	159.53	353.82	**37.64**	72.54
etcs	**28.78**	240.90	129.39	42.92	34.22
etcs_error-detect	**107.72**	4409.09	1006.21	1268.80	304.28
etcs_error_interrupt	**176.24**	t.o.	t.o.	4499.53	1007.86
etcs_error_inv	**41.91**	248.70	140.78	135.41	59.28
etcs_inv	39.22	235.18	145.24	45.68	**30.66**
etcs_v2_inv	259.89	t.o.	6545.56	657.19	**246.68**
flapCtrl_10of10Lp	1604.85	6291.64	t.o.	**777.02**	818.08
flapCtrl_6of10Lp	507.90	307.84	644.63	233.90	**174.49**
flapCtrl_7of10Lp	650.02	740.84	1090.03	**300.72**	358.91
flapCtrl_8of10Lp	598.67	3977.58	2030.74	585.83	**410.79**
flapCtrl_9of10Lp	847.84	t.o.	t.o.	**690.97**	752.90
interC_s_gc	**363.52**	t.o.	2691.60	431.07	391.79
interC_s_v2	**1598.80**	t.o.	t.o.	t.o.	t.o.
interC_s_v2_dist	**6049.18**	t.o.	t.o.	t.o.	t.o.
interC_s_v2_gc	**48.58**	635.40	1359.62	87.80	82.20
interC_s+g_dist	**153.76**	t.o.	t.o.	6989.27	5372.76
interC_s+g_gc	t.o.	t.o.	t.o.	5249.61	**1957.61**
interC_s+g_v2_dist	**401.52**	t.o.	t.o.	t.o.	t.o.
interC_s+g_v2_gc	6529.16	t.o.	t.o.	5068.16	**2352.97**
interC_s+g_v2_inv	**3143.96**	t.o.	t.o.	t.o.	t.o.

Table 5: Results for acceleration at once. The timeout (t.o.) was set to 7200 sec. Best results are shown in bold numbers. Rows containing timeouts only are not shown.

Option	AAO	Onioning	Polarity Minimization	Quantifier Elimination Method
1	on	off	on	naïve
2	on	on	on	naïve
3	on	off	off	SMT-based
4	on	on	off	SMT-based
5	off	off	on	naïve
6	off	on	on	naïve

Table 6: Settings for exact model checking using acceleration per mode, extra benchmark set

Model	Opt 1	Opt 2	Opt 3	Opt 4	Opt 5	Opt 6
flapSlatCtrl_2of2Lp2F0S	3.04	3.24	3.06	**2.32**	5.95	4.78
flapSlatCtrl_2of2Lp2F1S	2711.59	4161.67	3679.10	2292.60	**827.44**	1637.03
flapSlatCtrl_2of2Lp2F2S	**4471.50**	76558.80	11264.62	17797.80	6661.73	8534.59
flapSlatCtrl_2of2Lp2F3S	13034.39	41637.19	13075.40	**12689.87**	33685.32	28398.80
flapSlatCtrl_2of2Lp2F4S	**5538.01**	t.o.	20894.88	22533.31	t.o.	t.o.
flapSlatCtrl_2of2Lp2F5S	**10039.74**	68331.08	31822.52	40078.90	t.o.	t.o.
flapSlatCtrl_2of2Lp2F6S	**10475.10**	72094.26	39121.84	31960.95	t.o.	t.o.
flapSlatCtrl_2of2Lp2F7S	**11788.56**	34393.26	24468.82	96978.52	t.o.	t.o.
flapSlatCtrl_2of2Lp2F8S	**42442.06**	t.o.	51147.38	t.o.	t.o.	t.o.
flapSlatCtrl_2of2Lp2F9S	**13404.15**	t.o.	37998.77	t.o.	t.o.	t.o.
flapSlatCtrl_2of2Lp2F10S	**12613.76**	42371.93	39147.02	t.o.	t.o.	t.o.
flapSlatCtrl_2of2Lp2F11S	78640.62	t.o.	**71780.02**	t.o.	t.o.	t.o.
flapSlatCtrl_2of2Lp2F12S	92810.50	t.o.	**63650.54**	t.o.	t.o.	t.o.
flapSlatCtrl_2of2Lp2F13S	**24042.32**	t.o.	39675.72	t.o.	t.o.	t.o.
flapSlatCtrl_2of2Lp2F14S	**14236.59**	t.o.	107183.42	t.o.	t.o.	t.o.
flapSlatCtrl_2of2Lp2F15S	115035.47	t.o.	**63841.51**	t.o.	t.o.	t.o.
flapSlatCtrl_2of2Lp2F16S	**45215.36**	t.o.	66656.62	t.o.	t.o.	t.o.
flapSlatCtrl_2of2Lp2F17S	**16418.40**	t.o.	116709.49	t.o.	t.o.	t.o.
flapSlatCtrl_2of2Lp2F18S	t.o.	t.o.	**117839.88**	t.o.	t.o.	t.o.
flapSlatCtrl_2of2Lp2F19S	**36114.34**	t.o.	84296.29	t.o.	t.o.	t.o.
flapSlatCtrl_2of2Lp2F20S	t.o.	t.o.	**98048.64**	t.o.	t.o.	t.o.

Table 7: Results for acceleration at once, extra benchmark set. The timeout (t.o.) was set to 129600 sec. Best results are shown in bold numbers.

Settings for exact acceleration at once. Table 4 provides a quick overview on the different settings which we have used to evaluate acceleration at once.

Acceleration at once on standard benchmark set. Fig. 25(b) and Fig. 25(a) show that acceleration at once without onioning is not competitive with the best setting for acceleration per mode (option 8). The situation improves substantially as soon as we enable onioning for acceleration at once. Many data points are evenly scattered in a broad stripe around the diagonal. Especially in combination with the SMT-based quantifier elimination, acceleration at once seems to be an interesting alternative to acceleration per mode. Nevertheless, acceleration at once produces more timeouts and a closer look onto Table 5 reveals that presumably there are certain model classes for which acceleration at once is extremely well-suited, while for other model classes it seems to be a bad choice. Identifying such model classes is considered as future work.

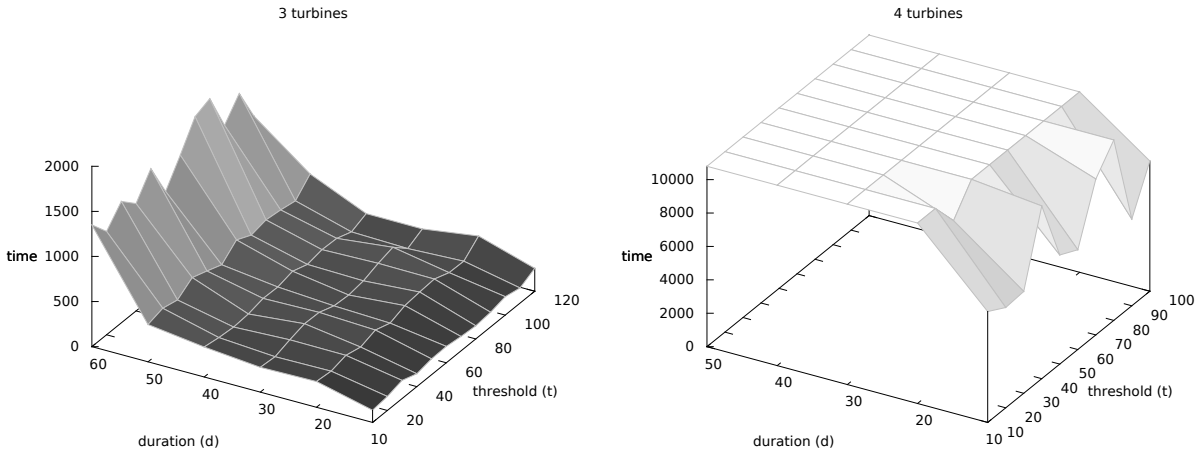
A benchmark set for acceleration at once. Acceleration at once was designed to cope with models having many parallel components. Hence, we evaluate acceleration at once on a specialized benchmark set containing models with many components. Table 6 shows the settings which we used for our second evaluation of acceleration at once.

The run times for this benchmark set are presented in Table 7. Interestingly, we observe that using SMT-based quantifier elimination does not always yield the best results. For some unknown reasons, the naïve implementation of the Loos–Weispfenning quantifier elimination method often leads to better run times. A deeper analysis of this behavior is planned for the future.

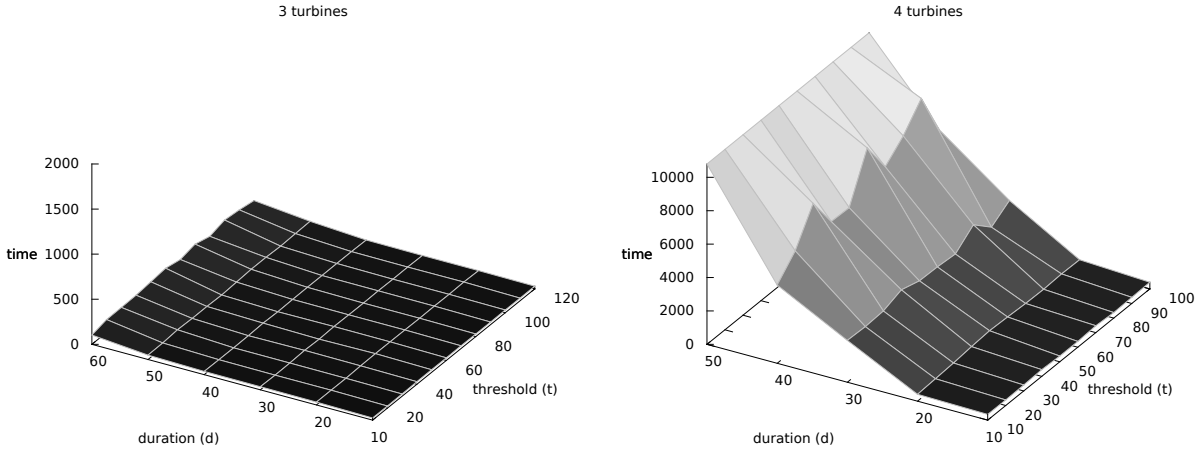
Lessons learned form evaluation of acceleration at once. In contrast to acceleration per mode, acceleration at once is a novel approach whose run time behavior is not that well understood. The results for acceleration per mode came along with theoretical insights. For acceleration at once, theory and evaluation are only loosely coupled, a factor which certainly has to be investigated in more detail. Moreover, we have to isolate well-suited model classes for deceleration-at-once.

However, already in the current state we partly observe competitive, yet superior run time results for acceleration at once.

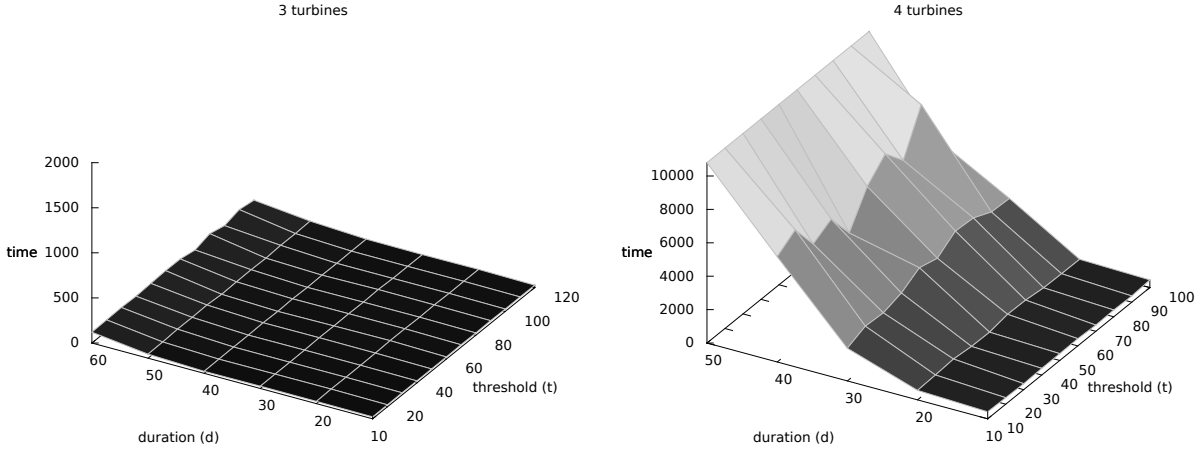
Putting everything together. In order to illustrate our progress in exact model checking, we extended and re-run the dam case study in [2] with an old FOMC version from August 23rd, 2010 using the onioning



(a) Old version of FOMC

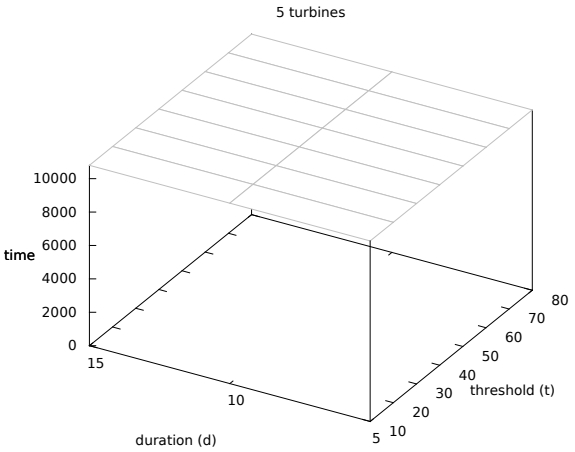


(b) Current FOMC with Optimal Settings for Acceleration Per Mode

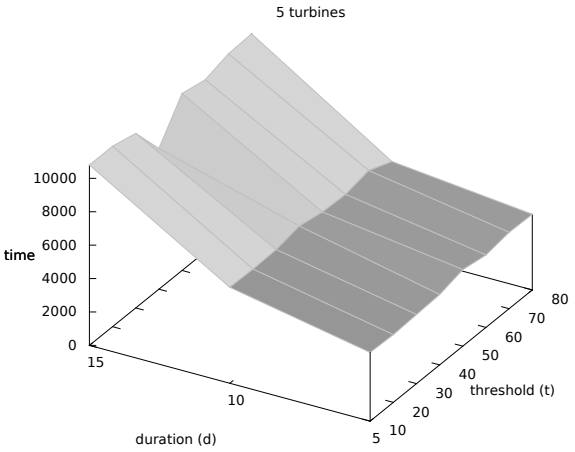


(c) Current FOMC with acceleration at once

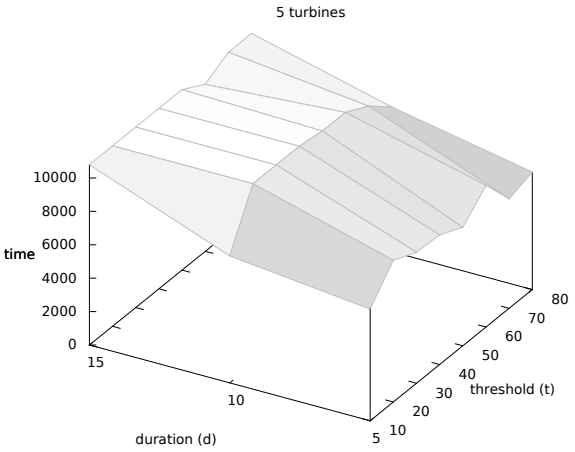
Figure 26: Comparison for 3 and 4 turbines



(a) Old version of FOMC



(b) Current FOMC with optimal settings for acceleration per mode



(c) Current FOMC with acceleration at once

Figure 27: Comparison for 5 turbines

approach against the current FOMC with optimal settings for acceleration per mode and optimal settings for acceleration at once. Compared to the case study in [2] we increased the number of turbines to 4 and 5. The timeout was set to 10800 sec.

For acceleration per mode we have chosen the settings of option 8 in Table 1. Based on the results shown in Table 5 we decided to use acceleration at once in combination with SMT-based quantifier elimination and onioning.

The run times for the old FOMC version for 3 turbines are roughly half of the run times presented in [2], which is probably due to the improved hardware. The run times vary from 135.48 sec to 1692.56 sec. The respective run times for the current FOMC vary from 23.55 sec to 189.38 sec for acceleration per mode and from 21.30 sec to 158.29 sec for acceleration at once.

For 4 turbines the old FOMC version is only able to finish a few benchmark run within the given timeout. The current FOMC performs much better: especially for small values of duration, we obtain the model checking results in $\frac{1}{20}$ of the time needed by the old FOMC.

Finally, for 5 turbines we do not obtain any model checking results form the old FOMC.

5.2.2 CEGAR

In this section we present experimental results for various optimizations of CEGAR model checking. All evaluated techniques have been described in Chapter 4. The interface of FOMC provides many different options which influence the CEGAR approach leading to vast variety of possible settings. The goal of this section is to provide and evaluate some promising settings.

Common settings to all experiments. For the exact pre-image computation we used the same settings as Option 8 in Table 1, i. e., we used continuous pre-image computation per mode, together with onioning and SMT-based quantifier elimination.

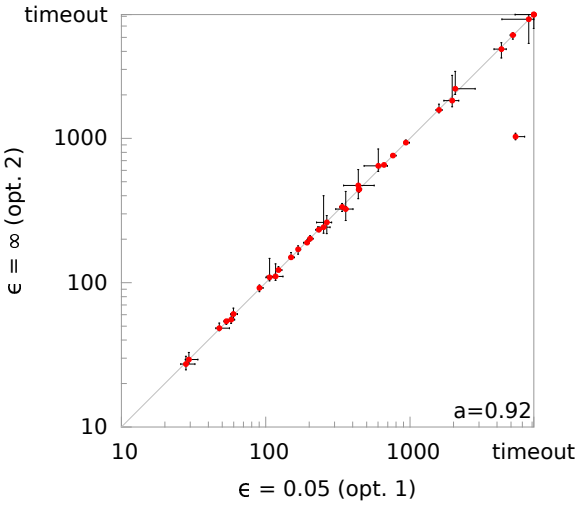
While we implemented the ability to trigger overapproximations of the current state set before any pre-image computation, our evaluation is restricted to overapproximations before the continuous pre-image computation. This restriction is motivated by our experiences and offers us the possibility to compare two different overapproximation methods, overapproximation at once and overapproximation per mode. The latter is currently only supported for the continuous pre-image computation.

Throughout all experiments we disabled the bloating of the initial set. Furthermore, the first overapproximation is only triggered if the number of linear constraints is at least 60 ($minThresholdLC$), all further overapproximations are triggered if the number of linear constraints reached at least the 1.5-fold of the number of linear constraints since the last overapproximation ($relThresholdLC$).

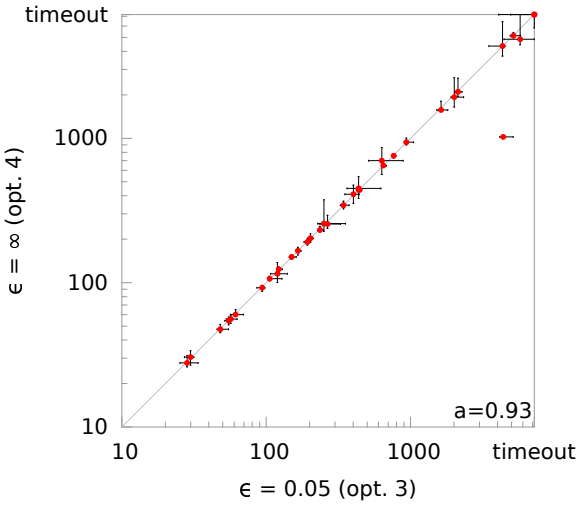
Please note that the presented benchmark set does not contain the *dam_ 3turb_ m#-#* models. Due to our overapproximation strategy CEGAR was not triggered on this model class.

Option	CEGAR	Overapproximation	Epsilon	Interpolation Method
1	on	atOnce	0.05	CM
2	on	atOnce	∞	CM
3	on	atOnce	0.05	FINT
4	on	atOnce	∞	FINT
5	on	atOnce	0.05	MINT
6	on	atOnce	∞	MINT
7	on	perMode	0.05	CM
8	on	perMode	∞	CM
9	on	perMode	0.05	FINT
10	on	perMode	∞	FINT
11	on	perMode	0.05	MINT
12	on	perMode	∞	MINT
13	off	n.a.	n.a.	n.a.

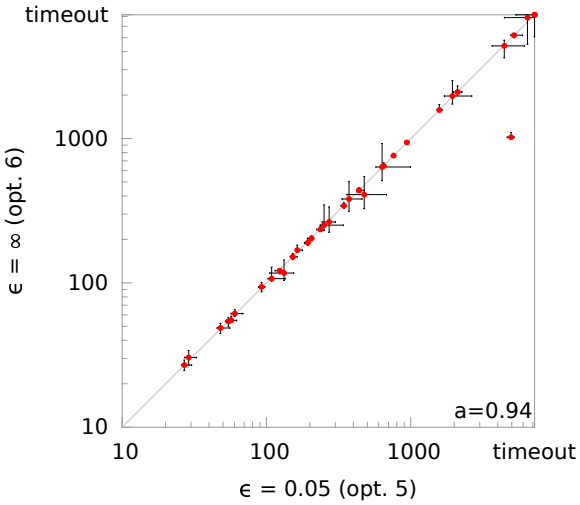
Table 8: Settings for CEGAR model checking



(a) Constraint minimization

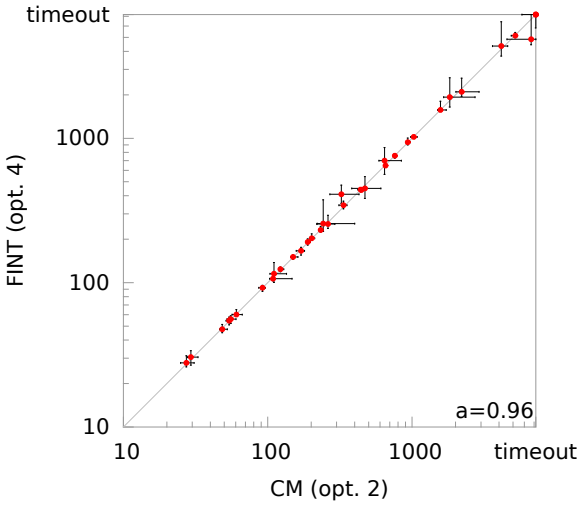


(b) FINT

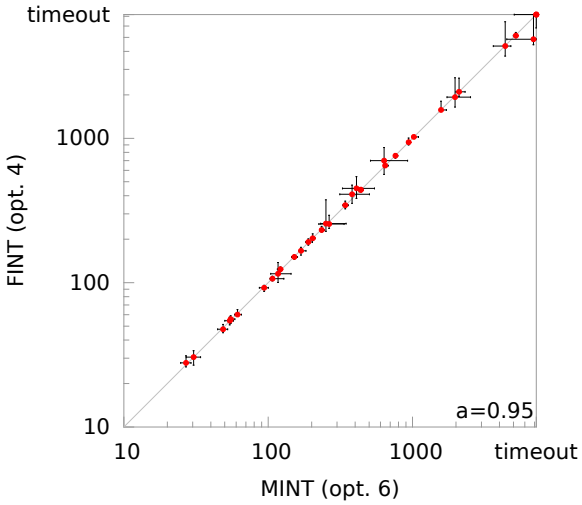


(c) MINT

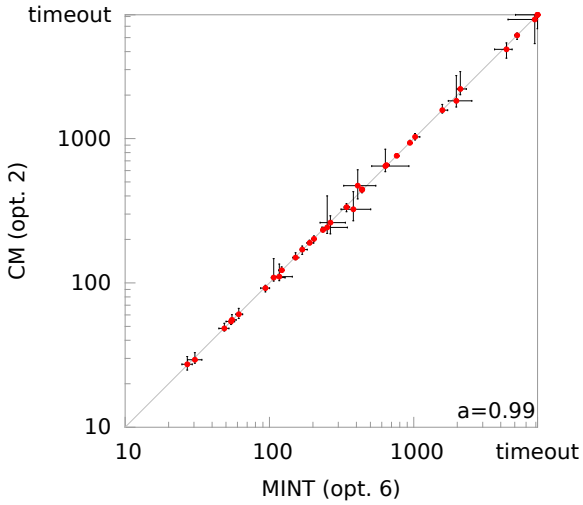
Figure 28: Evaluation of ϵ-bloating for overapproximation at once



(a) FINT vs. constraint minimization



(b) FINT vs. MINT



(c) Constraint minimization vs. MINT

Figure 29: Evaluation of interpolation methods for overapproximation at once

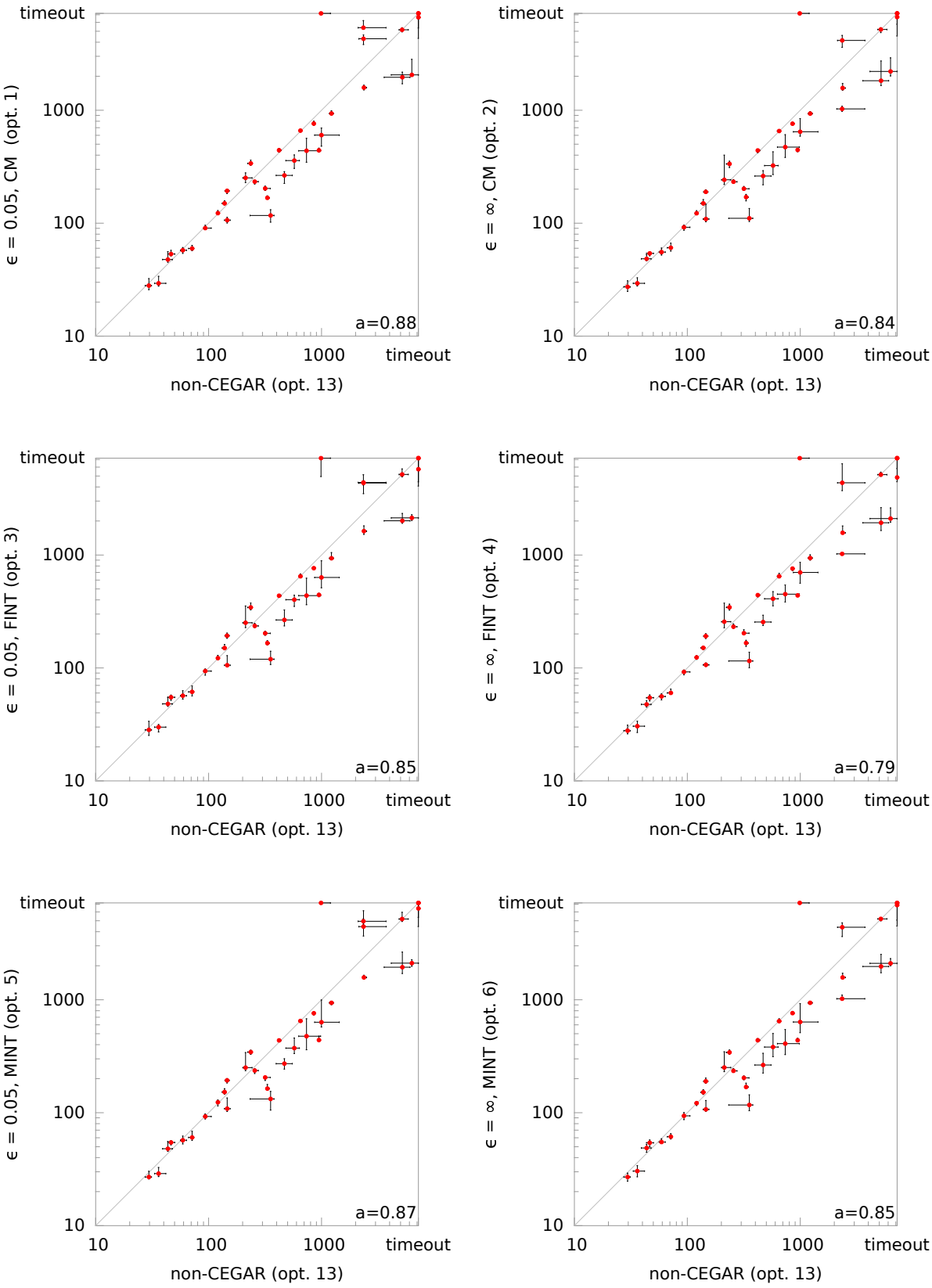


Figure 30: Overapproximation at once vs. non-CEGAR

Overview							
Model	Opt 1	Opt 2	Opt 3	Opt 4	Opt 5	Opt 6	Opt 13
acc_pi_h14	57.58	55.49	56.77	55.92	56.99	**55.00**	59.06
avc_arb_h5	441.28	445.54	443.52	438.52	439.98	**437.48**	947.84
avc_arb_h7	934.86	**934.24**	934.69	940.06	939.65	938.85	1222.71
avc_arb_h9	1586.23	**1571.15**	1625.59	1573.28	1577.24	1576.55	2371.14
avc_goal_h11	659.46	654.37	646.90	647.50	649.47	**646.01**	650.58
avc_goal_h21	5158.72	5191.65	5165.23	**5147.82**	5184.33	5202.76	5168.05
avc_goal_h5	122.71	122.24	122.00	124.01	123.77	121.61	**120.93**
avc_goal_h7	233.08	232.78	235.58	**231.72**	236.10	234.68	256.12
avc_goal_h9	442.31	440.44	435.87	441.23	436.57	437.36	**420.77**
bb_h22	760.82	759.95	764.00	**758.48**	760.19	760.76	854.18
dam_random_2turb_m13.9t7	193.42	189.36	192.94	191.46	193.47	189.66	**145.69**
dam_random_4turb_m13t3	**59.80**	60.62	61.41	60.07	60.37	61.31	71.40
dam_random_4turb_m13.9t7	338.68	335.08	343.77	344.46	344.50	341.66	**236.41**
etcs	28.03	27.28	28.31	27.81	**26.96**	**26.96**	29.58
etcs_error-detect	**90.39**	91.98	93.70	92.17	92.56	93.70	93.49
etcs_error_interrupt	106.29	108.88	**105.58**	106.46	108.68	106.93	145.90
etcs_error_inv	47.68	48.42	48.12	47.58	48.00	48.73	**43.64**
etcs_inv	29.32	29.38	29.92	30.48	**28.89**	30.44	36.10
etcs_v2_inv	251.62	242.24	251.43	256.86	250.60	251.01	**212.29**
flapCtrl_10of10Lp	**601.97**	644.74	632.58	700.17	632.53	634.78	997.04
flapCtrl_6of10Lp	117.01	**110.40**	119.56	115.24	132.29	117.07	353.48
flapCtrl_7of10Lp	264.92	261.68	265.78	**255.18**	271.70	264.26	468.12
flapCtrl_8of10Lp	358.55	**323.63**	401.78	409.53	373.54	381.07	574.16
flapCtrl_9of10Lp	438.05	471.29	437.11	450.06	475.84	**408.57**	735.98
interC_s_dist	**2055.94**	2205.13	2130.89	2102.05	2110.35	2098.94	6301.36
interC_s_gc	167.54	170.10	165.76	166.32	**163.58**	168.88	331.16
interC_s_v2	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	**987.83**
interC_s_v2_dist	5374.79	1027.38	4385.74	1022.47	4947.38	**1020.01**	2346.41
interC_s_v2_gc	53.32	54.01	55.02	54.49	54.50	54.13	**46.48**
interC_s+g_dist	149.87	149.88	150.04	150.69	152.48	151.56	**138.34**
interC_s+g_gc	6647.00	6677.58	5742.02	**4853.36**	6433.74	6886.74	t.o.
interC_s+g_v2_dist	202.90	**201.79**	202.82	203.04	205.20	203.44	316.63
interC_s+g_v2_gc	1958.71	**1824.06**	2008.94	1928.63	1942.95	1970.37	5183.29
interC_s+g_v2_inv	4294.36	4147.03	4336.47	4361.76	4437.16	4381.88	**2351.19**

Table 9: Evaluation of CEGAR with overapproximation at once. The timeout (t.o.) was set to 7200 sec. Best results are shown in bold numbers. Rows containing timeouts only are not shown.

Settings for CEGAR. Table 8 provides an overview on the different settings which we have used for our experiments. Option 13 corresponds to Option 8 from Table 1. We evaluated two different overapproximation methods, overapproximation at once (*atOnce*) and per mode overapproximation (*perMode*). An ϵ-bloating factor of 0.05 pushes the linear constraints outwards by 5% of the interval length of the allowed values of a variable. The factor ∞ allows to use the complete continuous state space for interpolation. For the interpolation methods we evaluated three different methods: constraint minimization (*CM*, see Sect. 3.2.1), *FINT* (see Sect. 4.3.2), and *MINT* (see Sect. 4.3.3).

How to interpret the figures? We repeated our evaluation several times. The presented results are obtained from a total number of 11 runs. The timeout was set to 7200 sec. In order to handle timeouts correctly, we decided to use the median instead of the arithmetic mean as statistic average of all runs. In each figure we compare two options against each other. Each coordinate of a red point represents the medians of the run times. The error lines span from the minimal to the maximal value occurred in our measurement. Most run times are relatively small, hence, we decided to scale both axes logarithmically for better clarity. The number indicated by a gives the slope of the regressions line of the median points through the origin. It was computed using the method of least squares, i. e., $a = \frac{\sum_{i=1}^{k} x_i y_i}{\sum_{i=1}^{k} x_i^2}$ where x_i and y_i are the coordinates of the median point of the ith benchmark.

Overapproximation at once. Let us first discuss the influence of the different methods for ϵ-bloating. The figures in Fig. 28 indicate that there is slight advantage of using an ϵ value of ∞ in overapproximation at once irrespective the choice of interpolation method. However, this result is mainly caused by a single survey point, namely *interC_ s_ v2_ dist* (see Table 9) and it is not clear whether this observation can be generalized or traced back to some unknown model characteristic.

After identifying $\epsilon = \infty$ as the better choice for overapproximation at once setting, we compare the different interpolations methods. We observe the following order of interpolation methods: Fint seems to produce the best results, followed from constraint minimization, and finally Mint. Hence, we finally identified option 4 as the best choice for overapproximation at once. A comparison of option 1–6 against the run times of exact model checking (option 13) is shown in Fig. 30. It shows that our assessment of the results was right: Indeed option 4 turns out to be the best setting. Moreover and possible more important: Regardless which of the CEGAR option 1-6 we choose, in average we obtain better results than with the non-CEGAR approach.

Per mode overapproximation. In contrast to overapproximation at once, the actual choice of the option, i. e. the choice of the ϵ-bloating method or the choice of the interpolation method, has only little impact on the resulting run times, as Fig. 31 shows. Indeed, the comparison with the run times of the exact model checking in Fig. 32 shows a very coherent picture. So we cannot identify a clear candidate for optimal settings in the per mode overapproximation scenario. However, irrespective of the actual choice, all proposed settings turned out to be faster than the non-CEGAR approach.

Comparing overapproximation at once with per mode overapproximation. While we could identify a best candidate for overapproximation at once, namely option 4, we were not able to identify such an candidate for per mode overapproximation. A comparison of Fig. 30 and Fig. 32 indicates that overapproximation at once is the better choice for our benchmark set.

Lesson learned from evaluation of CEGAR. For the majority of our benchmarks the CEGAR approach using any setting leads to better run times than the non-CEGAR approach. However, for our benchmark set we were able to identify option 4 as the best choice, see comparison of option 4 and option 13 in Fig. 30. The question whether this setting should be used as default setting for CEGAR model checking with FOMC is considered as future work.

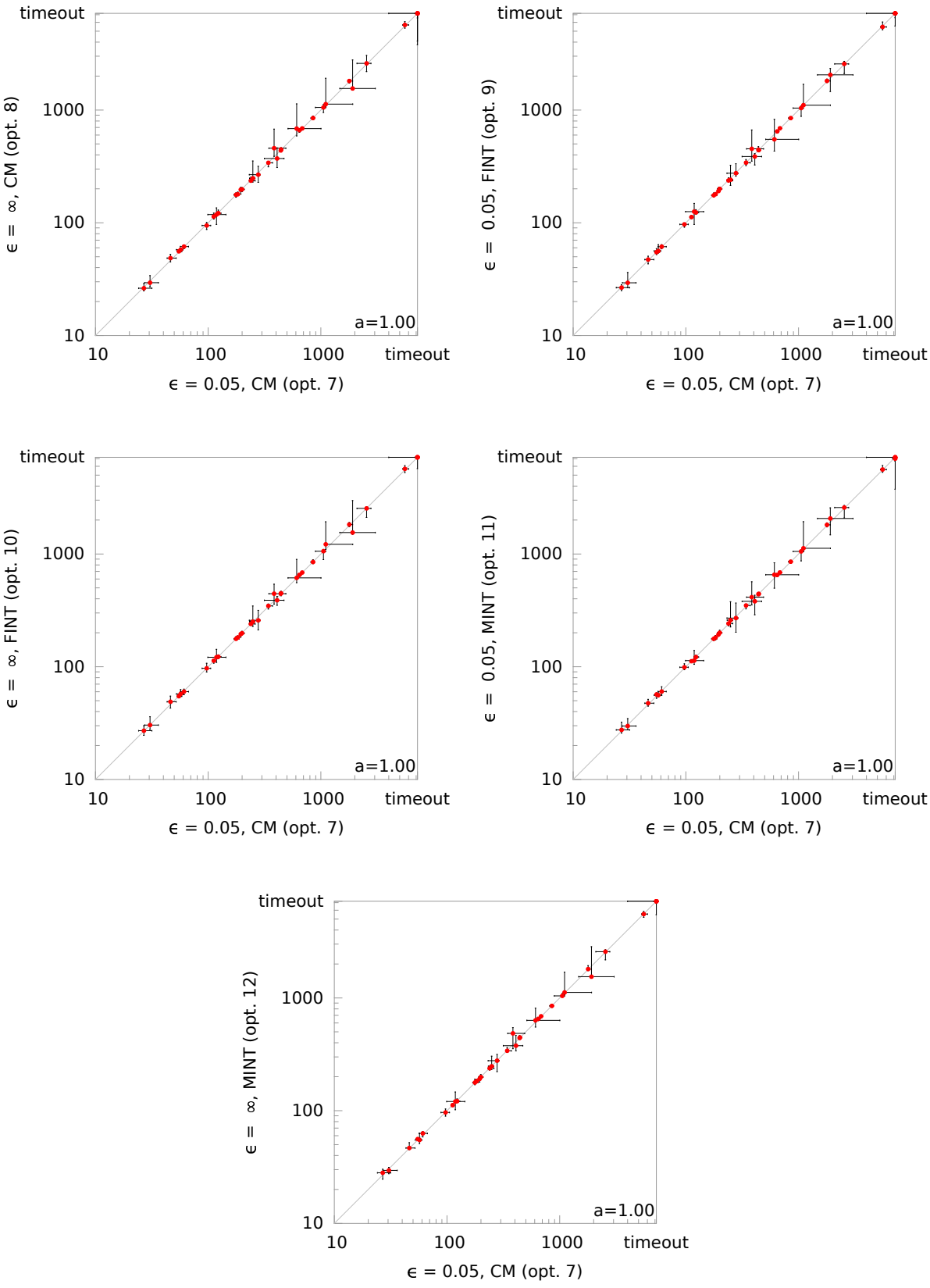


Figure 31: Different settings for per mode overapproximation

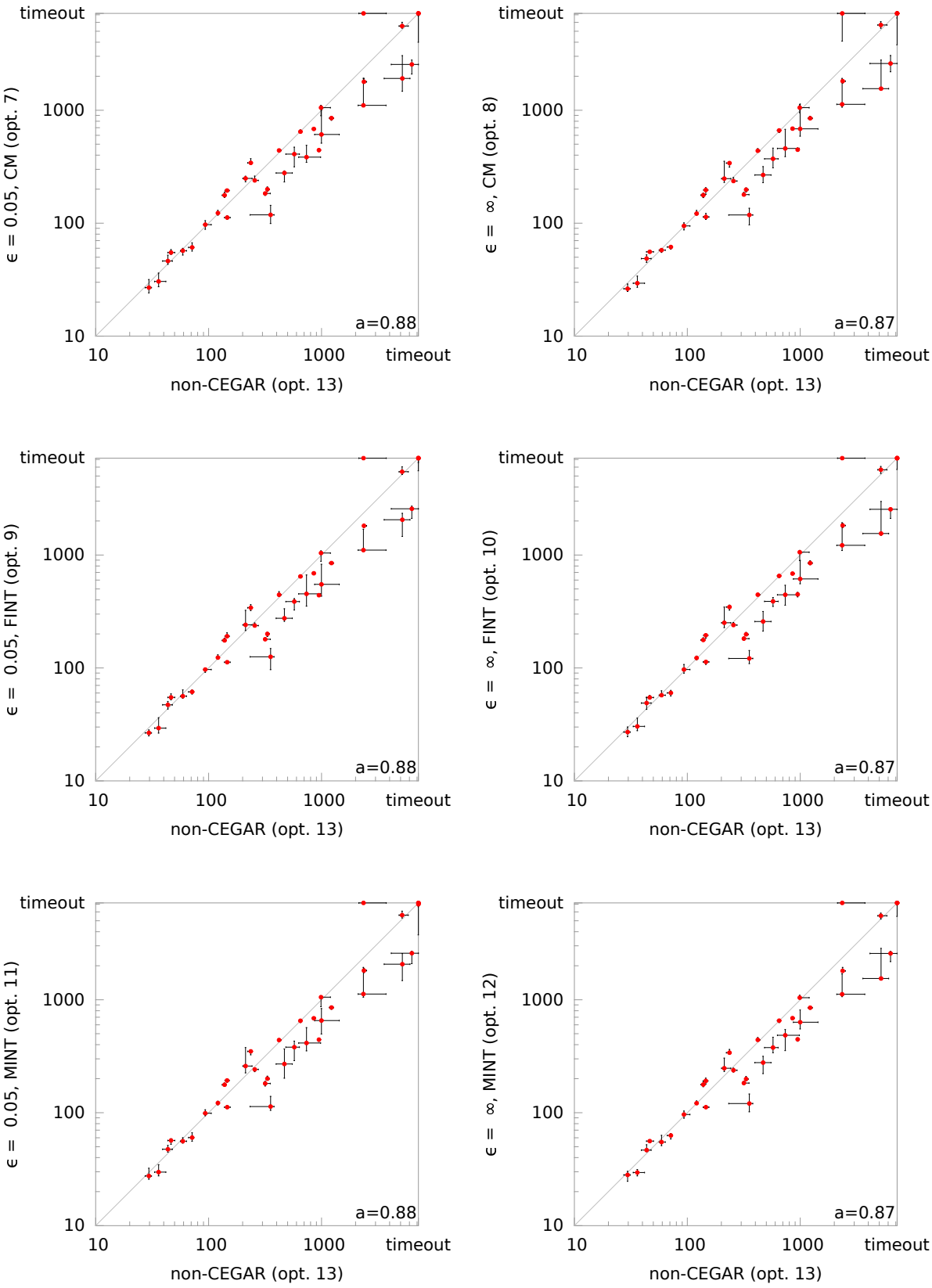


Figure 32: Per mode overapproximation vs. non-CEGAR

Overview							
Model	Opt 7	Opt 8	Opt 9	Opt 10	Opt 11	Opt 12	Opt 13
acc_pi_h14	57.03	57.77	56.28	57.31	55.90	**54.96**	59.06
avc_arb_h5	443.48	446.80	**440.08**	447.85	442.98	446.09	947.84
avc_arb_h7	852.77	848.82	**848.09**	848.96	852.99	851.09	1222.71
avc_arb_h9	**1787.98**	1810.31	1812.84	1823.64	1811.86	1802.26	2371.14
avc_goal_h11	646.70	662.69	**646.15**	652.46	652.18	653.57	650.58
avc_goal_h21	5563.81	5684.93	5461.38	5680.90	5612.97	5545.98	**5168.05**
avc_goal_h5	122.90	121.48	123.46	122.56	121.96	121.40	**120.93**
avc_goal_h7	239.42	**236.23**	237.00	239.47	241.62	237.76	256.12
avc_goal_h9	440.62	**437.64**	444.17	445.04	439.56	441.06	**420.77**
bb_h22	**683.37**	686.82	689.04	684.76	686.50	687.29	854.18
dam_random_2turb_m13.9t7	194.47	196.73	190.86	194.48	192.81	191.24	**145.69**
dam_random_4turb_m13t3	61.04	61.60	61.42	**60.15**	60.26	62.90	71.40
dam_random_4turb_m13.9t7	342.12	340.80	341.24	346.50	349.05	340.05	**236.41**
etcs	26.92	**26.27**	26.62	27.05	27.52	28.11	29.58
etcs_error-detect	97.04	**94.61**	96.89	96.85	98.85	96.33	**93.49**
etcs_error_interrupt	112.11	113.63	112.36	113.01	111.81	**111.79**	145.90
etcs_error_inv	46.25	48.56	47.14	48.93	47.48	46.72	**43.64**
etcs_inv	30.52	29.41	**29.32**	30.34	29.78	29.56	36.10
etcs_v2_inv	248.98	248.26	240.70	251.17	258.76	247.46	**212.29**
flapCtrl_10of10Lp	610.11	683.82	**549.54**	614.24	654.18	632.97	997.04
flapCtrl_6of10Lp	118.56	118.17	125.62	121.44	**113.40**	120.68	353.48
flapCtrl_7of10Lp	278.30	267.46	275.40	**257.90**	270.14	277.84	468.12
flapCtrl_8of10Lp	408.61	**371.63**	386.80	388.94	380.77	376.94	574.16
flapCtrl_9of10Lp	**384.20**	458.44	452.87	444.07	413.96	484.47	735.98
interC_s_dist	2544.30	2594.94	2564.53	**2537.70**	2580.53	2571.24	6301.36
interC_s_gc	199.64	**198.05**	199.96	198.48	200.22	198.77	331.16
interC_s_v2	1054.03	1053.03	1041.01	1057.91	1054.47	1043.30	**987.83**
interC_s_v2_dist	**1104.30**	1127.52	1104.51	1219.36	1124.04	1118.76	2346.41
interC_s_v2_gc	55.00	55.80	54.88	54.72	56.74	55.99	**46.48**
interC_s+g_dist	176.98	177.18	175.20	176.97	176.74	177.48	**138.34**
interC_s+g_gc	t.o.	t.o.	t.o.	t.o.	**7010.97**	t.o.	t.o.
interC_s+g_v2_dist	183.21	**179.04**	179.52	181.90	181.52	182.88	316.63
interC_s+g_v2_gc	1914.24	1552.26	2049.88	1547.96	2064.74	**1542.68**	5183.29
interC_s+g_v2_inv	t.o.	**t.o.**	t.o.	t.o.	t.o.	t.o.	**2351.19**

Table 10: Evaluation of CEGAR with per mode overapproximation. The timeout (t.o.) was set to 7200 sec. Best results are shown in bold numbers. Rows containing timeouts only are not shown.

5.2.3 Scaling to Extremes

In this report we concentrate on systematically presenting and relating the new developments within FOMC. Therefore we presented above experiments whose aim was to compare the different FOMC options. At these experiments the FOMC was limited to run on the benchmark models for certain maximal time period and the models chosen for benchmark had a complexity that is about manageable within the specified time period.

To give a glimpse into what the most complex models can be checked via FOMC, we chose another benchmark set and started the FOMC without a run time limit. Furthermore, we only started a small number of processes on each model with a parameter combination that seemed auspicious based on previous results. We selected only models that should be proven as *safe*, i.e., satisfying the target safety property. Hence the model checking process terminates only after determining the fix point for iterative preimage computation starting from the state set representing the negated target property.

In order to determine the most complex models, for which the FOMC still can establish satisfaction of the safety property, we relied on the model families that are scalable in the number of components, that are, the *dam* and the *flapSlat* controller (cf. Sect. 5.1). At the *dam*, we can scale up the number of turbines and the controller has to switch on and off the turbines, so that the water level stays within bounds. At the *flapSlat* controller models, we can scale up the number of slats at an aircraft with already two flaps. The controller has to ensure that the angles of flaps and slats match the current plane velocity. In the following we summarize the best of results, that we observed.

- 23 **continuous variables and 2^{71} discrete states:** The largest number of continuous variables in a model for which the FOMC successfully established safety is 23. The *flapSlatCtrl_ 2of2Lp2F20S* model has a continuous variable for the plane velocity and one continuous variable for the angle of each of the two flaps and 20 slats. It has a discrete state space of 2^{71}.

  The evolutions of the plane velocity, flaps and slats are specified by constant derivatives. This model is also extreme in having the largest number of global modes. Each flap/slat has three modes, so for 22 flaps and slats, this amounts to 3^{22} global modes. The model is enriched by invariants, that is with lemmata annotations, expressing that all flaps and, respectively, all slats are equal. In separate runs it was proven that the annotated lemmata hold.

  The successful run was started as part of benchmarking acceleration at once (cf. p. 73). Here, actually, a time out was set to 36 hours. For this model only acceleration at once (cf. Sect. 3.4) was able to successfully establish safety.

- 20 **continuous variables and 2^{199} discrete states:** The model having simultaneously a large continuous and discrete state space is *dam_ 6turb_ m10_ use.hlang*. The model specifies a dam with 6 turbines. Each turbine induces three continuous variables for its run time, minor and major maintenance time. The model encoding is optimized to have a small number of modes, by specifying the continuous evolution for the water level referring only to the number of currently running turbines.

  The FOMC process established safety after more than three days of run time.

- 9 **continuous variable and 2^{271} discrete states** The model *dam_ random_ 8turb_ m13t3.hlang* is the model with the largest discrete state space. It also encodes a dam, in a variant where each turbine introduces a continuous variable for its maintenance time and the number of modes is optimized to be small – just like for *dam_ 6turb_ m10_ use.hlang*. Also each turbine comes with a counter for the number on/off switchings since its last maintenance. In this model variant the threshold for going into maintenance has been set to $2^{29} + 1$. So the resulting model of concurrent turbines has an immense discrete state space.

  The FOMC process established safety after more than six days of run time.

We remain in debt of a more systematic exploration of the limits of manageable model complexity. We will carefully analyze our first results above. In particular, we plan to do further experiments for different FOMC options, and reexamine the models for different properties to get an impression of the influence of the target property and to simultaneously check the sanity of the model.

Nevertheless, these results are impressive and give a bright outlook on the capabilities of the FOMC.

6 Conclusion

We have systematically explored the design space of optimization methods maintaining preciseness of abstractions, and shown that techniques such as onioning can be successfully lifted from the domain of hardware verification to the class of controller models with large discrete state spaces, when combined with techniques such as redundancy removal and constraint minimization. We have identified classes of loosely coupled controller models which can be scaled in the number of controller, for which a particular method for symbolic flow evaluation coined acceleration at once significantly outperformed the standard mode-wise evaluation of flows because it avoids the construction of cross products of modes. We have seen, that for certain classes of applications the complexity of the symbolic representation as LinAIGs can be drastically reduced with a tailored combination of overapproximation and redundancy removal techniques, and that combining this with counter-example guided abstraction refinement allowed to significantly extend the class of models, for which we can establish safety properties. We have provided a comprehensive assessment of the relative merits of these techniques with a rich set of benchmarks representing typical control applications from automation, avionics, automotive and rail.

We feel, though, that a still deeper assessment is needed to understand the limits and potentials of our approach. What characteristics of models would allow scaling both the number of discrete and continuous variables? For which classes of models can we infer automatically invariants so as to reduce the complexity of the verification problem? Can we give design guidelines for decomposing large designs into loosely coupled systems? How can we apply hierarchical reasoning by black-boxing subcomponents and learning invariants about these sufficient to establish global safety problems? How can we go beyond linear hybrid automata, by integrating symbolic representation methods used for reachability analysis of linear differential systems of equations, such as the recently proposed symbolic orthogonal projections [75]? We hope that the level of precision provided in this paper, the insights gained, and the intuitions passed on, will help to continue this line of research on first-order model checking inaugurated more than 20 years ago [76].

References

[1] J. Bohn, W. Damm, O. Grumberg, H. Hungar, K. Laster, First-order-CTL model checking, in: FSTTCS, Vol. 1530 of LNCS, Springer, 1998, pp. 283–294.

[2] W. Damm, H. Dierks, S. Disch, W. Hagemann, F. Pigorsch, C. Scholl, U. Waldmann, B. Wirtz, Exact and fully symbolic verification of linear hybrid automata with large discrete state spaces, Sci. Comput. Program. 77 (10-11) (2012) 1122–1150.

[3] T. A. Henzinger, P.-H. Ho, H. Wong-toi, HyTech: A model checker for hybrid systems, Software Tools for Technology Transfer 1 (1997) 460–463.

[4] 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.

[5] E. Asarin, T. Dang, O. Maler, The d/dt tool for verification of the hybrid systems, in: 14th Conference on Computer Aided Verification, Vol. 2404 of LNCS, Springer, 2002, pp. 365–370.

[6] G. Frehse, PHAVer: Algorithmic verification of hybrid systems past HyTech, International Journal on Software Tools for Technology Transfer (STTT) 10 (3).

[7] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, O. Maler, SpaceEx: Scalable verification of hybrid systems, in: G. Gopalakrishnan, S. Qadeer (Eds.), Computer Aided Verification, Vol. 6806 of Lecture Notes in Computer Science, Springer, Heidelberg, 2011, pp. 379–395.

[8] K. L. McMillan, Symbolic Model Checking, Kluwer Academic Publishers, 1993.

[9] T. V. Group, VIS: A system for verification and synthesis, in: CAV, 1996, pp. 428–432.

[10] 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, Vol. 4218 of LNCS, 2006, pp. 276–291.

[11] 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, Vol. 4762 of LNCS, Springer, 2007, pp. 425–440.

[12] 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).

[13] 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.

[14] 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.

[15] F. Pigorsch, C. Scholl, An AIG-based QBF-solver using SAT for preprocessing, in: Design Automation Conference, 2010, pp. 170–175.

[16] C. A. R. Hoare, An axiomatic basis for computer programming, Communication of the ACM 12 (1969) 576–583.

[17] R. Alur, T. A. Henzinger, P.-H. Ho, Automatic symbolic verification of embedded systems, IEEE Transactions on Software Engineering 22 (3) (1996) 181–201.

[18] R. Loos, V. Weispfenning, Applying linear quantifier elimination, The Computer Journal 36 (5) (1993) 450–462.

[19] A. Dolzmann, Algorithmic strategies for applicable real quantifier elimination, Ph.D. thesis, Universität Passau (2000).

[20] W. Damm, C. Ihlemann, V. Sofronie-Stokkermans, PTIME parametric verification of safety properties for reasonable linear hybrid automata, Mathematics in Computer Science, Special Issue on Numerical Software Verification 5 (4) (2011) 469–497.

[21] T. A. Henzinger, P. W. Kopke, A. Puri, P. Varaiya, What's decidable about hybrid automata?, J. Comput. Syst. Sci. 57 (1) (1998) 94–124.

[22] B. Dutertre, L. de Moura, A fast linear-arithmetic solver for DPLL(T), in: CAV, Vol. 4144 of LNCS, Springer, 2006, pp. 81–94.

[23] R. Bruttomesso, A. Cimatti, A. Franzen, A. Griggio, R. Sebastiani, The MathSAT 4 SMT solver, in: CAV, LNCS, Springer, 2008, pp. 299–303.

[24] The VIS Group, VIS: A system for verification and synthesis, in: 8th Conference on Computer Aided Verification, Vol. 1102 of LNCS, Springer, 1996, pp. 428–432.

[25] T. A. Henzinger, The theory of hybrid automata, in: 11th IEEE Symposium on Logic in Computer Science, IEEE Press, 1996, pp. 278–292.

[26] G. Frehse, Compositional verification of hybrid systems using simulation relations, Ph.D. thesis, Radboud Universiteit Nijmegen (2005).

[27] R. Sebastiani, Lazy satisability modulo theories, JSAT 3 (2007) 141–224.

[28] 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.

[29] P. Pudlák, Lower bounds for resolution and cutting plane proofs and monotone computations, Journal on Symbolic Logic 62 (3) (1997) 981–998.

[30] K. L. McMillan, Interpolation and SAT-based model checking, in: CAV, Vol. 2725 of LNCS, Springer, 2003, pp. 1–13.

[31] O. Coudert, C. Berthet, J. C. Madre, Verification of synchronous sequential machines based on symbolic execution, in: Automatic Verification Methods for Finite State Systems, Vol. 407 of LNCS, Springer Verlag, 1989, pp. 365–373.

[32] 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.

[33] A. Komuravelli, A. Gurfinkel, S. Chaki, SMT-based model checking for recursive programs, in: 26th Conference on Computer Aided Verification, 2014, pp. 17–34.

[34] A. Reynolds, T. King, V. Kuncak, An instantiation-based approach for solving quantified linear arithmetic, arXiv preprint arXiv:1510.02642.

[35] B. Dutertre, Solving exists/forall problems with yices, in: Workshop on Satisfiability Modulo Theories, 2015.

[36] J. S. Miller, Decidability and complexity results for timed automata and semi-linear hybrid automata, in: N. A. Lynch, B. H. Krogh (Eds.), HSCC, Vol. 1790 of LNCS, Springer, 2000, pp. 296–309.

[37] G. Lafferriere, G. J. Pappas, S. Yovine, A new class of decidable hybrid systems, in: F. W. Vaandrager, J. H. van Schuppen (Eds.), HSCC, Vol. 1569 of LNCS, Springer, 1999, pp. 137–151.

[38] M. Agrawal, P. S. Thiagarajan, The discrete time behavior of lazy linear hybrid automata, in: M. Morari, L. Thiele (Eds.), HSCC, Vol. 3414 of LNCS, Springer, 2005, pp. 55–69.

[39] G. Lafferriere, G. Pappas, S. Sastry, O-minimal hybrid systems, Mathematics of Control, Signals, and Systems 13 (1) (2000) 1–21.

[40] T. Brihaye, C. Michaux, C. Rivière, C. Troestler, On o-minimal hybrid systems, in: R. Alur, G. J. Pappas (Eds.), HSCC, Vol. 2993 of LNCS, Springer, 2004, pp. 219–233.

[41] T. Brihaye, C. Michaux, On the expressiveness and decidability of o-minimal hybrid systems, Journal of Complexity 21 (4) (2005) 447–478.

[42] F. Wang, Symbolic parametric safety analysis of linear hybrid systems with BDD-like data-structures, IEEE Trans. on Software Engineering 31 (1) (2005) 38–52.

[43] G. Frehse, PHAVer: Algorithmic verification of hybrid systems past HyTech, in: HSCC, Vol. 3414 of LNCS, Springer, 2005, pp. 258–273.

[44] M. Fränzle, What will be eventually true of polynomial hybrid automata?, in: N. Kobayashi, B. C. Pierce (Eds.), Theoretical Aspects of Computer Software, Springer, 2001, pp. 340–359.

[45] V. Mysore, C. Piazza, B. Mishra, Algorithmic algebraic model checking II: Decidability of semi-algebraic model checking and its applications to systems biology, in: Automated Technology for Verification and Analysis, Springer, 2005, pp. 217–233.

[46] C. Piazza, M. Antoniotti, V. Mysore, A. Policriti, F. Winkler, B. Mishra, Algorithmic algebraic model checking I: Challenges from systems biology, in: K. Etessami, S. K. Rajamani (Eds.), Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 5–19.

[47] V. Mysore, B. Mishra, Algorithmic algebraic model checking III: Approximate methods, Electronic Notes in Theoretical Computer Science 149 (1) (2006) 61–77, proceedings of the 7th International Workshop on Verification of Infinite-State Systems (INFINITY 2005) Verification of Infinite-State Systems 2005.

[48] V. Mysore, B. Mishra, Algorithmic algebraic model checking IV: Characterization of metabolic networks, in: H. Anai, K. Horimoto, T. Kutsia (Eds.), Algebraic Biology: Second International Conference, AB 2007, Castle of Hagenberg, Austria, July 2-4, 2007. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 170–184.

[49] X. Chen, E. Ábrahám, S. Sankaranarayanan, Taylor model flowpipe construction for non-linear hybrid systems, in: Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd, IEEE, 2012, pp. 183–192.

[50] X. Chen, Reachability analysis of non-linear hybrid systems using Taylor models, Ph.D. thesis, Ph. D. thesis, RWTH Aachen University, Germany (2015).

[51] X. Chen, E. Ábrahám, S. Sankaranarayanan, Flow*: An analyzer for non-linear hybrid systems, in: Computer Aided Verification, Springer, 2013, pp. 258–263.

[52] S. Schupp, E. Ábrahám, X. Chen, I. B. Makhlouf, G. Frehse, S. Sankaranarayanan, S. Kowalewski, Current challenges in the verification of hybrid systems, in: Cyber Physical Systems. Design, Modeling, and Evaluation, Springer, 2015, pp. 8–24.

[53] W. Damm, S. Disch, W. Hagemann, C. Scholl, U. Waldmann, B. Wirtz, Integrating incremental flow pipes into a symbolic model checker for hybrid systems, Reports of SFB/TR 14 AVACS 76, SFB/TR 14 AVACS, iSSN: 1860-9821, http://www.avacs.org (July 2011).

[54] W. Hagemann, Reachability analysis of hybrid systems using symbolic orthogonal projections, in: A. Biere, R. Bloem (Eds.), Computer Aided Verification, Vol. 8559 of Lecture Notes in Computer Science, Springer International Publishing Switzerland, 2014, pp. 406–422.

[55] S. Kong, S. Gao, W. Chen, E. Clakre, dreach: δ-reachability analysis for hybrid systems, in: TACAS, Springer, 2015, pp. 200–205.

[56] A. Eggers, Direct handling of ordinary differential equations in constraint-solving-based analysis of hybrid systems, Ph.D. thesis, Universität Oldenburg (2014).

[57] S. Ratschan, Z. She, Safety verification of hybrid systems by constraint propagation-based abstraction refinement, ACM Transactions on Embedded Computing Systems (TECS) 6 (1) (2007) 8.

[58] A. Cimatti, A. Griggio, R. Sebastiani, Efficient Generation of Craig Interpolants in Satisfiability Modulo Theories, ACM Trans. Comput. Logic 12 (1) (2010) 7:1–7:54.

[59] W. Craig, Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory, Journal of Symbolic Logic 22 (3) (1957) pp. 269–285.

[60] K. L. McMillan, An Interpolating Theorem Prover, Theoretical Computer Science 345 (1) (2005) 101–121.

[61] K. L. McMillan, Interpolation and SAT-Based Model Checking, in: Proc. of CAV, Vol. 2742, Springer Berlin / Heidelberg, 2003, pp. 1–13.

[62] A. Rybalchenko, V. Sofronie-Stokkermans, Constraint Solving for Interpolation, in: Proc. of VMCAI, 2007, pp. 346–362.

[63] C. Scholl, F. Pigorsch, S. Disch, E. Althaus, Simple interpolants for linear arithmetic, in: Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, 2014, pp. 1–6.

[64] F. Pigorsch, C. Scholl, Lemma localization: A practical method for downsizing SMT-interpolants, in: Proc. of DATE, 2013, pp. 1405–1410.

[65] N. Megiddo, On the complexity of polyhedral separability, Discrete & Computational Geometry 3 (4) (1988) 325–337.

[66] E. Althaus, B. Beber, J. Kupilas, C. Scholl, Improving interpolants for linear arithmetic, in: Automated Technology for Verification and Analysis, Springer, 2015, pp. 48–63.

[67] R. Alur, T. Dang, F. Ivančić, Counter-example guided predicate abstraction of hybrid systems, in: Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2003, pp. 208–223.

[68] E. Clarke, A. Fehnker, Z. Han, B. Krogh, J. Ouaknine, O. Stursberg, M. Theobald, Abstraction and counterexample-guided refinement in model checking of hybrid systems, International Journal of Foundations of Computer Science 14 (04) (2003) 583–604.

[69] M. Segelken, Abstraction and counterexample-guided construction of ω-automata for model checking of step-discrete linear hybrid models, in: Computer Aided Verification, Springer, 2007, pp. 433–448.

[70] S. Ratschan, Z. She, Safety verification of the hybrid systems by constraint propagation based abstraction refinement, in: 8th Workshop on Hybrid Systems: Computation and Control, Vol. 3414 of LNCS, Springer, 2005, pp. 573–589.

[71] P. Prabhakar, P. S. Duggirala, S. Mitra, M. Viswanathan, Hybrid automata-based cegar for rectangular hybrid systems, in: Verification, Model Checking, and Abstract Interpretation, Springer, 2013, pp. 48–67.

[72] M. Fränzle, H. Hungar, C. Schmitt, B. Wirtz, HLang: Compositional representation of hybrid systems via predicates, Reports of SFB/TR 14 AVACS 20, SFB/TR 14 AVACS, ISSN: 1860-9821, http://www.avacs.org (July 2007).

[73] H3 FOMC Team, The flap controller description, `http://www.avacs.org/fileadmin/Benchmarks/Open/FlapSlatSystem.pdf`.

[74] W. Damm, A. Mikschl, J. Oehlerking, E.-R. Olderog, J. Pang, A. Platzer, M. Segelken, B. Wirtz, Automating verification of cooperation, control, and design in traffic applications., in: C. Jones, Z. Liu, J. Woodcock (Eds.), Formal Methods and Hybrid Real-Time Systems, Vol. 4700 of LNCS, Springer, 2007, pp. 115–169.

[75] W. Hagemann, Efficient geometric operations on convex polyhedra, with an application to reachability analysis of hybrid systems, Mathematics in Computer Science 9 (3) (2015) 283–325.

[76] H. Hungar, O. Grumberg, W. Damm, What if model checking must be truly symbolic, in: 8th Conference on Correct Hardware Design and Verification Methods, Vol. 987 of LNCS, Springer, 1995, pp. 1–20.