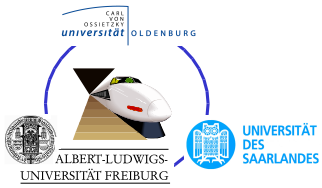

AVACS – Automatic Verification and Analysis of Complex Systems

REPORTS

of SFB/TR 14 AVACS

Editors: Board of SFB/TR 14 AVACS

Integrating Incremental Flow Pipes into a Symbolic Model Checker for Hybrid Systems

by

Werner Damm Stefan Disch Willem Hagemann
Christoph Scholl Uwe Waldmann Boris Wirtz

AVACS Technical Report No. 76
July 2011
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: Uwe Waldmann (`uw@avacs.org`).

Integrating Incremental Flow Pipes into a Symbolic Model Checker for Hybrid Systems

Werner Damm[*‡] Stefan Disch[§] Willem Hagemann[♯♭]
Christoph Scholl[§] Uwe Waldmann[♯] Boris Wirtz[*]

August 24, 2011

We describe an approach to integrate incremental flow pipe computation into a fully symbolic backward model checker for hybrid systems. Our method combines the advantages of symbolic state set representation, such as the ability to deal with large numbers of boolean variables, with an efficient way to handle continuous flows defined by linear differential equations, possibly including bounded disturbances.

1 Introduction

When we want to verify safety properties for embedded control applications in the transportation domain, we have to deal with two problems: The first one is the presence of a large discrete state space. Even if the number of control laws (continuous modes) is rather small, the number of discrete inputs (such as setting of control switches), counters, sanity checkbits, possibly multiple concurrent state machines, system-degradation modes, and finite switching variables makes it impossible to check the safety of such systems using verification tools that treat all discrete states explicitly. To handle such models in a model checker, it is therefore indispensable to use a fully symbolic representation. Examples include HRDs (Hybrid Restriction Diagrams), as used in the model checker RED [20], and LinAIGs (Linear constraint And-Inverter-Graphs), as used in our own FOMC tool [7, 6]. Thanks to powerful simplification and redundancy elimination techniques, LinAIGs allow the efficient representation of hybrid state sets involving large numbers of discrete variables.

The second problem we have to tackle is the system dynamics. So far, fully symbolic model checking approaches are restricted to dynamic behaviours that can be described by bounded derivatives [3]. For more complex dynamics – linear differential equations and beyond – zonotopes have been shown to be an efficient device for real-time and hybrid reachability testing [1, 2, 11, 10, 15]. Using zonotopes continuous flow pipes can be computed fast and without

[*]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
[♯]Max-Planck-Institut für Informatik, Campus E1.4, 66123 Saarbrücken, Germany
[♭]Universität des Saarlandes, 66123 Saarbrücken, Germany

wrapping effect. However, these approaches are restricted to systems with an explicit representation of a small number of discrete states, and usually to bounded model checking. Moreover, while the continuous flow can be computed rather precisely using zonotopes, unavoidable conversions from general polytopes to zonotopes when switching between discrete steps and continuous flows can induce significant overapproximation errors.

In this paper we present a first approach to integrate a combined polytope/zonotope-based flow pipe computation into a fully symbolic backward model checker for hybrid systems. The paper is structured as follows: we start by describing the class of models we consider. In Sect. 3 we recapitulate the basic properties of LinAIG-based backward model checking. Then we consider flow pipe computations for single polytopes: we show how to compute incremental flow pipes for backward model checking and explain the combination of polytopes and zonotopes that we use. In Sect. 5 we lift this process to full LinAIGs and discuss measures to deal with non-terminating flow pipes. Finally, we demonstrate the viability of our approch using two case studies that we have analyzed with a prototypical integration of our method within FOMC: an approach velocity controller from a driver assistance system and a railroad crossing model involving speed supervision of a train and a cooperation protocol between the train and a Radio Block Center (RBC).

2 Models

A *hybrid system with discrete states* consists of the following components:

- $\mathbf{x}$ is a vector of continuous variables $x_1, \ldots, x_r$ ranging over $\mathbb{R}$. These variables represent sensor values, actuator values, plant states, and other real-valued variables used for the modeling of control-laws and plant dynamics.

- $\mathbf{d}$ and $\mathbf{i}$ are vectors of discrete variables $d_1, \ldots, d_n$ and $i_1, \ldots, i_p$ ranging over $\mathbb{B} = \{0, 1\}$. These variables represent states from state machines, switches, counters, sanity bits of sensor values, etc. Elements of $\mathbf{i}$ are inputs.

- $\mathbf{m}$ is a vector of boolean mode variables that is used to represent a finite set of modes $\mathbf{M} = \{\mathbf{m}_1, \ldots, \mathbf{m}_k\} \subseteq \{0, 1\}^l$ using a suitable encoding. Each mode $\mathbf{m}_i$ is associated with a linear differential equation (LDE)

  $$\dot{\mathbf{x}}(t) = A_i\mathbf{x}(t) + B_i\mathbf{u}(t)$$

  describing the time derivative of the evolution of the continuous variables $\mathbf{x}$ during the mode $\mathbf{m}_i$. Here $\mathbf{u}(t) \in U \subseteq \mathbb{R}^k$ ranges over some bounded set of disturbances U, $A_i \in \mathbb{R}^{r\times r}$, and $B_i \in \mathbb{R}^{r\times k}$.

- GC is a global constraint given by a formula in $\mathcal{P}(\mathbf{d}, \mathbf{x}, \mathbf{m})$, where $\mathcal{P}(\mathbf{d}, \mathbf{x}, \mathbf{m})$ denotes the set of all boolean combinations of variables in $\mathbf{d}$ and $\mathbf{m}$ and linear constraints over $\mathbf{x}$ (i. e., (in)equations of the form $\sum \alpha_i x_i \sim \alpha_0$, where $\sim \in \{=, <, \leq\}$). We assume that GC specifies lower and upper bounds for all continuous variables.

- *Init* is a set of initial states given by a formula in $\mathcal{P}(\mathbf{d}, \mathbf{x}, \mathbf{m})$.

- DT is the set of discrete transitions; each discrete transition is given as a guarded assignment of the form

  $$\xi_i \quad \rightarrow \quad \mathbf{d} := \mathbf{g}_i(\mathbf{d}, \mathbf{i}, \mathbf{x}, \mathbf{m}); \quad \mathbf{x} := L_i\mathbf{x} + \mathbf{b}_i; \quad \mathbf{m} := \mathbf{m}_{j_i}$$

where $\xi_i \in \mathcal{P}(\mathbf{d}, \mathbf{x}, \mathbf{m})$, $\mathbf{g}_i$ is a vector of formulas in $\mathcal{P}(\mathbf{d} \cup \mathbf{i}, \mathbf{x}, \mathbf{m})$, $L_i \in \mathbb{R}^{r \times r}$, $\mathbf{b}_i \in \mathbb{R}^r$ and $\mathbf{m}_{j_i} \in \mathbf{M}$.

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

The guards of the transitions from DT^d (DT^{d2c}, DT^{c2d}) must be mutually exclusive. 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. Transitions from DT^{c2d} are urgent: If the guard ξ_i of a transition from DT^{c2d} holds, the continuous flow must stop.

For each mode $\mathbf{m}_i$ its *boundary condition* β_i is given by the cofactor of the disjunction of all discrete transition guards from DT^{c2d} w. r. t. $\mathbf{m}_i$.[1] For each valuation of variables in $\mathbf{d}$, the boundary condition must be equivalent to a disjunction of non-strict ($\leq$) linear inequations.

We assume that each mode has a minimum dwell time, i. e., it is impossible that a mode is left instantaneously in the moment when it is entered.

A state of a hybrid systems is a valuation of the variables $\mathbf{d}$, $\mathbf{x}$ and $\mathbf{m}$. Compared to standard definitions of hybrid automata [12], the transitions in DT^{c2d} correspond to "jumps" out of continuous flows. Since we allow a (possibly empty) series of purely discrete transitions after each jump, we additionally introduce discrete transitions in DT^d and discrete-to-continuous transitions in DT^{d2c} (which again lead to a continuous flow). Thus, the system evolves by alternating between continuous flows, in which time passes and only continuous variables are changed according to the LDE associated with the currently active mode of the system, and sequences of discrete transitions, which – following the synchrony hypothesis [4] – happen in zero time.

3 LinAIG-based Model Checking

For efficiently implementing formulas from $\mathcal{P}(D, C, M)$ (for sets of states) we use a specific data structure called LinAIGs [8, 7]. In that way both the discrete part and the continuous part of the hybrid state space are represented by one symbolic representation.

LinAIGs are And-Inverter-Graphs (AIGs) [13] enriched by linear constraints. AIGs are basically boolean circuits consisting only of AND gates and inverters. In [18] a variant of AIGs, the so-called Functionally Reduced AND-Inverter Graphs (FRAIGs) [17, 18] were tailored towards the representation and manipulation of sets of states in symbolic model checking, replacing BDDs as a compact representation of large discrete state spaces. To represent formulas from $\mathcal{P}(D, C, M)$ in a LinAIG we introduce a set of new (boolean) *constraint variables* Q. Each occurring linear constraint ℓ_i is then encoded by some $q_{\ell_i} \in Q$ (see Fig. 1), and the entire formula is encoded as an AIG over the extended set of boolean variables.

For keeping the representation as compact as possible we use a multitude of methods. For instance, inserting different nodes representing the same predicate is avoided using an SMT

[1] The cofactor is the partial evaluation of the disjunction w. r. t. $\mathbf{m} = \mathbf{m}_i$. It does not depend on the mode variables anymore.

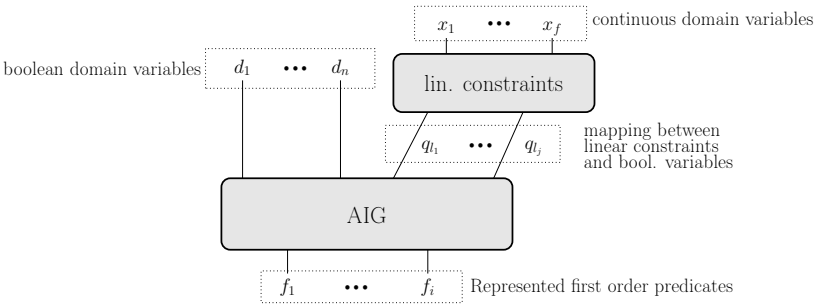

Figure 1: Structure of LinAIGs

(SAT modulo theories) solver which combines DPLL with linear programming as a decision procedure [8, 7]. Another method which is essential for keeping the representation compact is *redundancy removal* [19] which removes so-called redundant linear constraints from LinAIG representations. A linear constraint is called redundant for a (possibly non-convex) polyhedron (i.e., a boolean combination of linear constraints), if the polyhedron can be described without using this linear constraint.

Our goal is to check whether all states reachable from the initial states of the system (given by a LinAIG for $\mathit{Init} \in \mathcal{P}(D, C, M)$) are within a given set of (safe) states *Safe*. To establish this, a backwards fixpoint computation is performed. We start with the set $\neg \mathit{Safe}$ 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. We have chosen the backwards direction, because in this case the pre-image for discrete transitions can be expressed essentially by a substitution [8]. If the continuous behaviour is described by bounds on the derivatives as in [7, 6], then the pre-image $\mathit{flow}(\phi)$ of a state set ϕ for continuous transitions can be computed by quantifier elimination for real-valued variables, e. g., using the Loos-Weispfenning method [16]. The modifications necessary to deal with more complex dynamic behaviour will be the topic of the rest of this paper.

The structure of the model checking loop allows us to use a particular kind of optimization. Suppose that we have to compute the pre-image of a state set ψ_{i+1} at some point of the iteration, and that we have already computed the pre-image of the state set ϕ_i in previous iterations. Then all states in ϕ_i are "don't-cares" for the current step: we may add them to ψ_{i+1} or we may leave them out without changing the result of the fixpoint iteration. In other words, instead of continuing the iteration with ψ_{i+1}, we may as well use $\psi_{i+1} \vee \phi_i$, or $\psi_{i+1} \wedge \neg\phi_i$, or any state set in between. The don't-care optimization procedure introduced in [6] computes such a state set, and it does it in such a way that the result depends on a minimal set of linear constraints.

4 Computing Continuous Pre-Images

4.1 Entry Hyperplanes

If the dynamic behaviour of a hybrid systems is described by constant or bounded derivatives, the flow pipe or continuous pre-image $\mathit{flow}(\phi)$ of a state set $\phi \in \mathcal{P}(D, C, M)$ is again a state set in $\mathcal{P}(D, C, M)$ (and it can be computed in one step by quantifier elimination). This continuous pre-image is the starting point of the pre-image computation for the next discrete super step.

For richer dynamics, such as LDEs and beyond, such a one-step computation is in general impossible. In most cases, one cannot avoid an incremental computation of the flow pipe using a sufficiently small time step.

However, we do not need the entire pre-image to continue the computation: any subset of $\mathit{flow}(\phi)$ that contains all states where the continuous mode can be entered in the forward direction is sufficient. Assume for simplicity that the hybrid system starts in *Init* in some continuous mode and that the continuous mode is not left immediately by a c2d transition[2]. Under these circumstances there are two ways to enter a continuous mode: (a) directly from *Init* or (b) via a discrete super-step from a predecessor continuous mode.

If ϕ itself does not already intersect *Init*, $\mathit{flow}(\phi) \cap \mathit{Init}$ is non-empty if and only if $\mathit{flow}(\phi)$ intersects the boundary of *Init*, so for (a) it is sufficient to compute the intersections of the continuous pre-image $\mathit{flow}(\phi)$ with the hyperplanes delimiting *Init*. This is completely analogous to the forward model-checking case, where we have to check for intersections of the flow-pipe with $\neg \mathit{Safe}$.

For (b), the situation is slightly more complicated. Under the assumption that each mode of the model has a minimal dwell time and that all c2d-transitions are urgent, a continuous mode can be left (in the forward direction) only at one of its exit hyperplanes, i. e., at some hyperplane of the form $\mathbf{a}^\top \mathbf{x} = c$, where $\mathbf{a}^\top \mathbf{x} \leq c$ occurs in the guard of a c2d transition starting from the mode. In contrast to the forward model-checking case, we cannot use these exit hyperplanes directly, but we have to compute the possible images of the exit hyperplanes under discrete transformations. We may assume that every possible sequence of discrete transformations can be described as a set of guarded assignments

$$\xi_i \quad \rightarrow \quad \mathbf{d} := \ldots; \quad \mathbf{x} := L_i \mathbf{x} + \mathbf{b}_i; \quad \mathbf{m} := \ldots$$

where L_i is a linear mapping. The image of an exit hyperplane $\mathbf{a}^\top \mathbf{x} = c$ under the assignment $\mathbf{x} := L_i \mathbf{x} + \mathbf{b}_i$ can be computed as follows:

Case 1: If L_i is the identity mapping, then $\mathbf{a}^\top \mathbf{x} = c$ is mapped to the hyperplane $\mathbf{a}^\top \mathbf{x} = c + \mathbf{a}^\top \mathbf{b}$.

Case 2: If L_i is bijective and $\mathbf{b}_i = \mathbf{0}$, then $\mathbf{a}^\top \mathbf{x} = c$ is mapped to the hyperplane $(\mathbf{a}^\top L_i^{-1})\mathbf{x} = c$.

Case 3: If L_i is a projection $[x_1 := 0, \ldots, x_j := 0]$ and $\mathbf{b}_i = \mathbf{0}$, then $\mathbf{a}^\top \mathbf{x} = \sum a_i x_i = c$ is mapped to an intersection of j or $j{+}1$ hyperplanes, namely $x_1 = 0, \ \ldots, \ x_j = 0$, and, if $a_1 = \cdots = a_j = 0$, additionally $\mathbf{a}^\top \mathbf{x} = c$.

Case 4: Otherwise, the linear transformation L_i can be written as a composition $A_2 B A_1$, where A_1 and A_2 are bijective and B is a projection, and the result is obtained by composing the three special cases above.

The set of potential entry hyperplanes for a mode can then be computed from the exit hyperplanes of its predecessor modes by selecting at least one of the images of each exit hyperplane under each applicable transformation sequence.

However, this is mostly a theoretical result. In practice, those continuous variables that are changed during the continuous modes are usually not changed arbitrarily in the discrete steps in between. Only two cases are frequent: Either the variables occurring in exit hyperplanes are not changed at all in any of the discrete supersteps. This is typical for variables describing physical properies of a plant. Or at least one of the variables is a timer variable that is reset to

[2]Otherwise we have to extend *Init* by the set of all states that are reachable from *Init* using only discrete transitions.

a fixed value. In the first case, we can simply take the exit hyperplane as an entry hyperplane of the following mode; in the second case, we can use $x_j = \mathit{reset\_value}$ for the timer variable x_j.[3]

It should be clear that hybrid model checking using an incremental flow pipe computation can only be efficient if mode changes are mostly deterministic. If transitions out of a continuous mode can happen at any time, either because they are non-urgent, or because they depend on external events (e. g., interrupts), any point of the flow pipe can be the starting point of another flow pipe, and enumerating the resulting tree of successor states becomes intractable very quickly. To analyze such models, bounded derivatives are usually a better choice. In fact, at this point there is little difference between forward and backward model checking.

4.2 Incremental Flow Pipe Computation

We will now turn to the question how to compute the actual flow pipe. Let us first consider flow pipes for individual polytopes; the extension to LinAIGs will be explained later.

If we have an undisturbed linear evolution

$$\dot{\mathbf{x}}(t) = A\mathbf{x}(t)$$

an overapproximation of the set of points from which some convex $P \subseteq \mathbb{R}^r$ is reachable can be computed as follows. Let $\tau > 0$ be the time discretization step. We first compute the matrix exponential $e^{-\tau A}$ of $-\tau A$. Second, we overapproximate the convex hull of P and $e^{-\tau A}P$ and add a bloating ball to take care of the non-linearity of the evolution. The resulting set P_1 is is an overapproximation of the set of all points from which P is reachable during the time interval $[0, \tau]$. The set of points from which P is reachable during the time interval $[i\tau, (i{+}1)\tau]$ can then be computed using the recurrence relation $P_{i+1} = e^{-\tau A}P_i$.[4]

The operations considered so far can be implemented easily using a polytope representation. If P is described by the linear inequations $\bigwedge_{i\in I} \mathbf{n}_i^\top \mathbf{x} \leq u_i$, then $e^{-\tau A}P$ is determined by the linear inequations $\bigwedge_{i\in I} \mathbf{n}_i'^\top \mathbf{x} \leq u_i$, where $\mathbf{n}_i' = (e^{\tau A})^\top \mathbf{n}_i$, and the convex hull can be overapproximated by the weak join.

The situation changes considerably when the linear evolution is disturbed, that is,

$$\dot{\mathbf{x}}(t) = A\mathbf{x}(t) + B\mathbf{u}(t)$$

for $\mathbf{u}(t)$ in some bounded subset $U \subseteq \mathbb{R}^k$. In this case, the recurrence relation takes the form $P_{i+1} = e^{-\tau A}P_i \oplus V$, where V is a set that depends on τ, A, B and U and $\oplus$ is the Minkowski sum (pointwise sum of sets). Computing the Minkowski sum repeatedly is problematic for polytopes, though: the exact computation is rather inefficient; computing an overapproximation in each iteration leads to a severe lack of precision, due to the so-called wrapping effect.

Zonotopes are an alternative representation that has been proposed for this task by Girard [10]. A zonotope is a point-symmetric polytope given by a central point $\mathbf{c} \in \mathbb{R}^r$ and a list of direction vectors (generators) $\mathbf{v}_1, \ldots, \mathbf{v}_k \in \mathbb{R}^r$:

$$Z = \{\, \mathbf{c} + \sum \alpha_j \mathbf{v}_j \mid -1 \leq \alpha_j \leq 1 \,\}.$$

[3] There may be several choices; in this case one should ensure that the variable(s) of the entry hyperplane are actually modified in the LDE of the successor mode.

[4] The computations for backward reachability, as considered here, and forward reachability differ only in the sign of the matrix τA.

The set of zonotopes is closed under linear transformation and under the Minkowski sum. Both operations can be computed very efficiently: The result of applying a linear transformation Φ to a zonotope $Z = (\mathbf{c}, \langle \mathbf{v}_1, \ldots, \mathbf{v}_n \rangle)$ is obtained by applying Φ to each component of Z, that is, $\Phi Z = (\Phi\mathbf{c}, \langle \Phi\mathbf{v}_1, \ldots, \Phi\mathbf{v}_n \rangle)$. To compute the Minkowski sum of $Z = (\mathbf{c}, \langle \mathbf{v}_1, \ldots, \mathbf{v}_n \rangle)$ and $Z' = (\mathbf{c}', \langle \mathbf{v}'_1, \ldots, \mathbf{v}'_n \rangle)$, one adds the central points and concatenates the lists of direction vectors, hence $Z \oplus Z' = (\mathbf{c} + \mathbf{c}', \langle \mathbf{v}_1, \ldots, \mathbf{v}_n, \mathbf{v}'_1, \ldots, \mathbf{v}'_n \rangle)$. Zonotopes also have the advantage that it is very easy to check whether a zonotope intersects a halfspace $\mathbf{n}^\top \mathbf{x} \leq u$ (or analogously, a hyperplane): it suffices to compute $\min\{ \mathbf{n}^\top (\mathbf{c} + \sum \alpha_j \mathbf{v}_j) \mid -1 \leq \alpha_j \leq 1 \}$, which is the same as $\mathbf{n}^\top \mathbf{c} + \sum \gamma_j \mathbf{n}^\top \mathbf{v}_j$, where $\gamma_j = 1$ for $\mathbf{n}^\top \mathbf{v}_j < 0$ and $\gamma_j = -1$ otherwise.

However, zonotopes are a rather restrictive class of polytopes. In hybrid system verification, we have to deal with alternations between continuous and discrete steps, and even if we assume that the result of some continuous step is in fact a zonotope, the following discrete step will in general not preserve this structure, so the input for the next continuous step can be an arbitrary polytope. While it is possible to decompose general polytopes into a union of zonotopes, this would lead to an explosion of the state set representation. The alternative method would be overapproximation of polytopes by zonotopes; the error introduced this way turned out to be much too large for the benchmarks we analyzed.

The solution to this problem is to use a mixed polytope/zonotope representation: To compute the continuous pre-image of a convex polytope P, we first overapproximate it by a zonotope Z. Then we compute in parallel the undisturbed flow pipe for the original polytope, that is, a sequence of polytopes P_i, and the disturbed flow pipe for the overapproximating zonotope, that is, a sequence of zonotopes of the form $Z_i = \Phi^{i-1} Z_1 \oplus \bigoplus_{0 \leq j \leq i-2} \Phi^j V$. The latter has a dual purpose: On the one hand, it serves to accumulate the influence of the disturbances, on the other hand, we use it to get a first inexpensive estimate whether the flow pipe intersects *Init* or an entry or exit hyperplane. More costly methods only have to be applied if we detect such an intersection with some Z_i. In this case, we combine the corresponding polytope P_i of the undisturbed flow pipe and the disturbance part $\bigoplus_{0 \leq j \leq i-2} \Phi^j V$ of the disturbed flow pipe; that is, we compute a polytope P'_i that uses the same normal vectors as P_i and overapproximates $P_i \oplus \bigoplus_{0 \leq j \leq i-2} \Phi^j V$. Then we use linear programming to test whether there is also a non-empty intersection of P'_i and *Init* or the hyperplane and to get a suitable representation of the intersection.[5]

There is one more problem that we have to deal with. It can happen that the computed flow pipe or a part of it leaves the boundary of the current mode. The first case is easy: if the flow pipe leaves the mode boundary completely, we can simply abort the computation. The second case is more difficult, though: if only some part of the flow pipe lies outside of the mode boundary, we have to cut out those trajectories that cross the mode boundary but we have to keep the rest. The solution that we use is essentially the one proposed in [14], namely to compute also the pre-image of the entire mode invariant, which is then intersected with P'_i and *Init* or the hyperplane.

4.3 Termination

Termination is in general a problem for incremental flow pipe computation techniques. In forward as well as in backward model checking, it can happen that the sequence of polytopes in the flow pipe never completely traverses an entry or exit hyperplane, for instance because

[5] Due to the fact that Z_1 is an overapproximation of P_1, it can actually happen that Z_i intersects *Init*, but P'_i does not.

the LDE describes a circular motion ($\dot{x} = y$, $\dot{y} = -x$). It can also happen that some points of the initial polytope move across the entry or exit hyperplane, whereas others remain fixed; as an example consider the LDE $\dot{x} = y$, $\dot{y} = 0$ and any point lying on the axis $y = 0$.

Still computing backward flow pipes poses some additional problems: In forward model checking, a flow pipe computation can always be stopped whenever the sequence of polytopes has traversed an *exit* hyperplane completely. On the other hand, an *entry* hyperplane can be traversed several times in the forward direction, hence traversing an entry hyperplane in backward model checking is not a sufficient criterion for stopping the flow pipe computation. The computation of a backward flow pipe can only be stopped if we have traversed an *exit* hyperplane[6] or a global constraint of the hybrid system.

In the sequel we will first consider terminating flow pipe computations. A solution to the non-termination problem will be presented in Sect. 5.3.

5 Lifting the Flow Pipe Computation to LinAIGs

5.1 Converting a LinAIG to a Polytope-based Representation

So far, we have restricted ourselves to flow pipes of individual polytopes. Our goal is now to extend this approach to arbitrary LinAIG representations of the state space. To this end, we have to extract convex polytopes from the LinAIG representation, perform flow pipe computation and build a new LinAIG representation of the resulting state space using the computed polytopes. However, since LinAIGs allow an arbitrary boolean combination of boolean variables and linear constraints, it is necessary to adapt the structure of the LinAIG first. Since e.g. the flow pipe of a polytope $P_1 \wedge P_2$ is not equal to the conjunction of the flow pipes for P_1 and P_2, we can not allow conjunctions of polytopes in the LinAIG we start from. This leads us to the definition of so-called Z-formulas.

Z-formulas are defined recursively in the following way:

$$ZF := P \mid ZF \wedge BF \mid ZF \vee BF \mid ZF \vee ZF$$

where P are polytopes (conjunction of linear constraints) and BF are arbitrary purely boolean formulas.

A Z-formula does not allow conjunctions of polytopes, nor does it allow negations of polytopes. Thus a Z-formula always represents the continuous part of the state space as a disjunction (union) of polytopes. Note that every formula in disjunctive normal form (DNF) with terms given by conjunctions of boolean variables and polytopes is a special case of a Z-formula.

One option would be to keep the LinAIG during all operations in a Z-formula structure. This would imply that after every operation on the LinAIG (e.g. a simple AND-operation) the resulting LinAIG representation has to be restructured in order to satisfy the definition of Z-formulas, which may be very time-consuming. Another drawback is that the LinAIG loses compaction potential if it is limited to a certain structure. Therefore we decided to convert the LinAIG only when we really need the Z-formula structure.

The overall procedure for conversion is sketched in Alg. 1. The input is a LinAIG l which represents the current state set. The conversion starts by a transformation of the AIG part of

[6] More precisely: an unconditional exit hyperplane. If the guard of a c2d transition is not simply a linear constraint but a more complex formula, the situation is more complicated.

the LinAIG into a (reduced ordered) binary decision diagram (BDD) [5]. In a BDD every path from the root node to the 1-terminal node represents a term of a corresponding DNF. Such a path is called a *1-path*. Thus, a BDD representation of the boolean structure of a LinAIG implicitly represents a valid Z-formula. Since the polytope/zonotope-based flow computation only operates on the polytopes of a Z-formula, it would be sufficient to extract polytopes from 1-paths by intersecting all linear constraints represented by abstraction variables $q_{l_i} \in Q$ occurring along the 1-path.

Input: LinAIG l
Result: BDD b, list of polytopes PL
begin
  BDD b := convertToBDD(l);
  partialReordering($b, (d_1, \ldots, d_n, q_{l_1}, \ldots, q_{l_k})$) ;
  $(n_1, \ldots, n_m)$:= searchConnectionNodes(b);
  PL := [];
  foreach $n_i \in (n_1, \ldots, n_m)$ **do**
    $(P_{i_1}, \ldots, P_{i_j})$:= collectPolytopes(n_i);
    PL.append($P_{i_1}, \ldots, P_{i_j}$) ;
  addVarBoundsToPolytopes(PL);
  removeUnnecessaryLinConst(PL);
  return (b,PL);
end

Alg. 1. Extracting the polytopes of a Z-formula out of a corresponding LinAIG

However, to keep our representations as compact as possible we avoid a complete DNF conversion by enumerating 1-paths and we separate the purely Boolean part of the BDD by reordering the variables. After reordering, every purely boolean variable $d_i \in D$ has a lower level in the variable order than any abstraction variable q_i. Now on every path starting from the root the boolean variables always occur before the abstraction variables. Nodes which are labeled by abstraction variables and have at least one incoming edge from a node labeled by a purely boolean variable are then called *connection nodes*. Fig. 2 shows a sketch of a BDD with a variable order respecting the partition of variables into purely boolean and abstraction variables. Each connection node represents a disjunction of polytopes. Now the set of polytopes needed for flow computation is given by all 1-paths beginning at some connection node. We can easily extract the polytopes by following these 1-paths. When we have collected all polytopes in a list PL we intersect each polytope with the bounds on real variables in order to ensure that it is bounded. Furthermore, we detect and eliminate redundant constraints in the descriptions of the polytopes. The result of Alg. 1 is the upper part of the BDD b with pointers to connection nodes as shown in Fig. 2 and a list of polytopes PL. The (upper part of the) BDD b is needed to be able to transfer the flow computation result back into the LinAIG.

5.2 Converting the Flow Computation back to a LinAIG

In the BDD b every connection node n_i represents a disjunction of the polytopes $P_{i_1} \vee \ldots \vee P_{i_j}$. The flow computation returns for every single initial polytope P a set of resulting polytopes R_P. We now build for every connection node n_i the union u_i of all resulting polytopes in $R_{P_{i_1}} \cup \ldots \cup R_{P_{i_j}}$. Finally, we traverse the BDD b recursively starting from the root node to

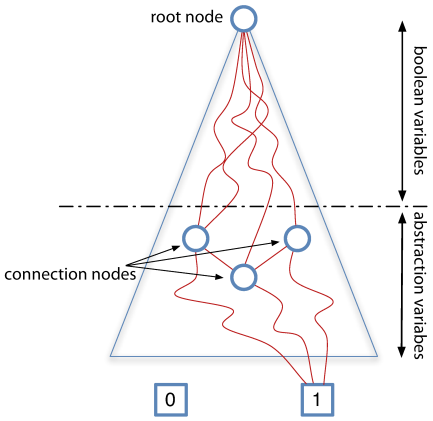

Figure 2: Sketch of a BDD with purely boolean variables above abstraction variables.

convert it back into a LinAIG. When reaching a connection node n_i we replace the original function of the connection node with the computed union of polytopes u_i.

5.3 Termination Revisited

Combining the results of the previous three sections, we obtain a function *flow′* that takes a LinAIG as input, converts it into a Z-formula, computes the flow pipe for each of its polytopes, collects the intersections of each flow pipe with *Init* and possible entry hyperplanes, inserts these intersections into the Z-formula, and returns the corresponding LinAIG.

We can then use $\mathit{flow}'(\phi)$ as a replacement for the entire continuous pre-image $\mathit{flow}(\phi)$ in the backward model-checking loop. However, this works only under the condition that every individual flow pipe computation terminates.

The obvious technique to deal with non-terminating flow pipe computations is dovetailing: We have to interleave the computation of continuous flow pipes and the rest of the model-checking loop. The don't-care optimization technique explained in Sect. 3 makes it rather easy to implement this. As a first step, we extend the function *flow′*: *flow′* gets an an additional argument, a time bound n that determines the number of iteration steps, and it stores its intermediate results in a hash table so that the computation can be resumed whenever a flow pipe has to be computed for an identical polytope with a larger value of n. As a side effect, *flow′* returns a state set that describes those states for which the flow pipe has been computed completely within n time steps. Now let ϕ_i a set of states for which we have to compute the continuous pre-image[7]. Let ω_{i-1} be the set of all states for which the flow pipe has already been computed completely during previous iterations, and let χ_{i-1} be the set of all states for which the flow pipe has been started but not finished during previous iterations. We set $\chi_i := \phi_i \vee \chi_{i-1}$, don't-care optimize χ_i with respect to ω_{i-1}, and take the result as input for *flow′*. The set of states returned by *flow′* for which the flow pipe has been computed completely within n time steps is then added to ω_{i-1}. Finally the time limit n is incremented for the next model checking loop.

Note that, if all flow pipes terminate, then χ_i becomes a subset of ω_i. If furthermore $\phi_{i+1} = \phi_i$, then don't-care optimizing χ_{i+1} w. r. t. ω_i yields *false*, so we get the same behaviour as for the simple implementation without dovetailing.

[7] That is, ϕ_i is the result of the last discrete pre-image computation.

6 Experimental Results

We have integrated the polytope/zonotope-based flow pipe generation into our FOMC model checker. In the following, we describe the experimental results that we obtained on two case studies.

6.1 Approach Velocity Controller (AVC)

Our first case study comes from an advanced driver assistance system. We consider two cars, a leading car c_l and a following car c_f. While c_l drives at nearly constant velocity, the acceleration of c_f is controlled by its *Approach Velocity Controller* (AVC). The goal of the AVC is to approach the leading car until a desired distance *dist* is established and to maintain this distance afterwards.

Dynamics of the AVC The AVC has three modes, the *Normal* mode, where the acceleration $\dot{v}_f$ is computed by a linear differential equation, and two border modes, namely *HighAcceleration* and *LowAcceleration*, which are introduced to avoid extreme accelerations or decelerations.

Let v_l the velocity of the leading car c_l and d the measured distance from c_f to c_l. In the *Normal* mode the AVC computes the acceleration $\dot{v}_f$ by

$$\dot{v}_f = a(v_l - v_f) + b(d - dist),$$

where a and b are parameters with $a > 0$, $b > 0$ and $\frac{a^2}{4} \geq b$.[8]

If the computed acceleration $\dot{v}_f$ exceeds the upper limit a_{max} the controller switch to the *HighAcceleration* mode, where $\dot{v}_f = a_{max}$ is set. The analogous behavior holds if the computed acceleration falls below the minimal allowed acceleration a_{min}. The controller switches back to *Normal* mode, as soon as the computed acceleration is within the allowed values.

The Model For the model we have chosen the parameter $a = 0.29$ and $b = 0.01$. The initial velocity of the following car is between $v_{min} = 0\frac{\text{m}}{\text{s}}$ and $v_{max} = 40\frac{\text{m}}{\text{s}}$ and the initial distance between both cars is between 450m and 500m. The desired distance is $dist = 50\text{m}$. We restricted the considered time to the interval $t \in [0\text{s},\ 500\text{s}]$.

We proved a combined safety property depending on the initial velocity v_l of the leading car

(a) collision freedom: for $v_l \in [v_{min},\ v_{max}]$ the distance d is always greater than 0m,

(b) reaching the desired distance: for $v_l \in [v_{min},\ 0.8v_{max}]$ we can ensure that the desired distance *dist* $\pm$ 5m is reached after 160s and kept afterward. The velocity of the following car stays within the bounds $v_f \in [v_{min} - 1\frac{\text{m}}{\text{s}},\ v_{max} + 5\frac{\text{m}}{\text{s}r}]$.

In addition we allow small disturbances on the acceleration of the leading car, $\dot{v}_l = 0 + u$, with $|u(t)| \leq disturb$.

[8] The right choice for these parameters can be read from the homogeneous solution of this linear differential equation. Only for positive values of a and b a stable behavior can be guaranteed, which is approaching the desired distance *dist*. If the discriminant $\frac{a^2}{4} - b$ is chosen negatively the AVC would oscillate around *dist*.

Results On an Intel Core 2 Duo T7500, 2.2 GHz, 2 GB RAM using one processor we measured a running time of 57sec for the undisturbed model. For the disturbed one we measured a running time of 77sec for the safe version ($disturb = 0.1$) and 37sec for the unsafe version ($disturb = 0.25$).

6.2 Railroad Crossing

Our second sample model is derived from the verification of collision avoidance protocols for train applications [9]. The system model consists of two parts, the speed supervision of the train and a cooperation protocol between the train and a Radio Block Center (RBC).

Speed Controller The speed controller drives the current speed v towards a desired speed v_d. In the *Normal* mode the acceleration is computed by $\dot{v} = 0.1(v_d - v)$. If the computed acceleration exceeds the maximal or minimal acceleration, that is, if $\dot{v} \geq 1.0$ or $\dot{v} \leq -1.5$ the controller switches to the *Acceleration* or *Braking* mode. The *Acceleration* mode sets the acceleration to the fixed value $\dot{v} = 1.0$ and allows a transition back to the *Normal* mode if $0.1(v_d - v) \leq 0.8$ is reached. The *Braking* mode behaves similarly with the fixed deceleration $\dot{v} = -1.5$; it is left if $0.1(v_d - v) \geq -1.2$. In addition the controller provides an *Emergency* mode with maximum deceleration of $\dot{v} = -3$. The *Emergency* mode is entered if the cooperation protocol signals a failure.

Cooperation Protocol The cooperation protocol distinguishes different phases. These phases are modelled position dependent. In the *Far* phase the train receives optionally an *isCrossing* message. In this case it sends a *lockCrossing* request to the railroad crossing and switches to the *Request* phase. The railroad crossing has to acknowledge the request and starts locking the crossing. In the *Negotiation* phase the train gets the signals *isLockedCrossing* and *newEOA*. If the crossing is not locked or no new *End of Authority* (EOA) is provided the train has to stop before reaching the current EOA. The protocol also maintains some error control and signals a failure to the speed supervision.

Safety Property The safety property of the model is

$$\textit{Emergency} \text{ mode activated} \rightarrow p \leq EOA, \tag{1}$$

i.e. if the *Emergency* mode is activated the position of train does not exceed EOA.

On the other hand we wanted to ensure that the train crosses the EOA if no failure occurs, thus we showed that the target

$$\text{not } \textit{Emergency} \text{ mode activated} \rightarrow p \leq EOA, \tag{2}$$

is not always globally true, i.e. the train crosses EOA if *Emergency* is not activated.

Results For target 1 we ran two variants with different disturbances, on an Intel Core 2 Duo T7500, 2.2 GHz, 2 GB RAM using one processor. The safe variant with smaller disturbances took 6 main loop iterations and a running time of 286 sec; the unsafe variant with higher disturbances took 4 main loop iterations and a running time of 320 sec.

For target 2 we measured a running time of 164 sec; it took 3 main loop iterations.

7 Conclusions

We have demonstrated that it is possible to integrate incremental flow pipe computation using a combination of polytopes and zonotopes into a fully symbolic backward model checker for hybrid systems. A prototype implementation has been integrated into our FOMC system and so far the results are promising. We plan to extend the system to more complex dynamics, and we will also investigate the potential of building a forward model checker based on the same techniques – note that the termination problem becomes easier for the forward direction, whereas the computation of discrete steps becomes more difficult, since it requires quantifier elimination.

References

[1] M. Althoff, O. Stursberg, and M. Buss. Reachability analysis of linear systems with uncertain parameters and inputs. In *46th Conference on Decision and Control*, 2007, pp. 726–732.

[2] M. Althoff, O. Stursberg, and M. Buss. Computing reachable sets of hybrid systems using a combination of zonotopes and polytopes. *Nonlinear Analysis: Hybrid Systems*, 2009.

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

[4] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[5] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[6] W. Damm, H. Dierks, S. Disch, W. Hagemann, F. Pigorsch, C. Scholl, U. Waldmann, and B. Wirtz. Exact and fully symbolic verification of linear hybrid automata with large discrete state spaces. *Science of Computer Programming*, to appear, 2011.

[7] W. Damm, S. Disch, H. Hungar, S. Jacobs, J. Pang, F. Pigorsch, C. Scholl, U. Waldmann, and B. Wirtz. Exact state set representations in the verification of linear hybrid systems with large discrete state space. In *ATVA*, 2007, *LNCS 4762*, pp. 425–440. Springer.

[8] W. Damm, S. Disch, H. Hungar, J. Pang, F. Pigorsch, C. Scholl, U. Waldmann, and B. Wirtz. Automatic verification of hybrid systems with large discrete state space. In *ATVA*, 2006, *LNCS 4218*, pp. 276–291.

[9] W. Damm, A. Mikschl, J. Oehlerking, E.-R. Olderog, J. Pang, A. Platzer, M. Segelken, and B. Wirtz. Automating verification of cooperation, control, and design in traffic applications. In *Formal Methods and Hybrid Real-Time Systems*, 2007, *LNCS 4700*, pp. 115–169. Springer.

[10] A. Girard. Reachability of uncertain linear systems using zonotopes. In M. Morari and L. Thiele, eds., *HSCC*, 2005, *LNCS 3414*, pp. 542–556. Springer.

[11] A. Girard and C. Le Guernic. Zonotope/hyperplane intersection for hybrid systems reachability analysis. In *HSCC*, 2008, *LNCS 4981*, pp. 215–228. Springer.

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

[13] A. Kuehlmann, V. Paruthi, F. Krohm, and M.K. Ganai. Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification. *IEEE Trans. on CAD*, 2002.

[14] C. Le Guernic. *Calcul d'Atteignabilité des Systèmes Hybrides à Partie Continue Linéaire.* PhD thesis, Université Grenoble I, 2009.

[15] C. Le Guernic and A. Girard. Reachability analysis of linear systems using support functions. *Nonlinear Analysis: Hybrid Systems*, 4(2):250–262, 2010.

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

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

[18] F. Pigorsch, C. Scholl, and S. Disch. Advanced unbounded model checking by using AIGs, BDD sweeping and quantifier scheduling. In *FMCAD*, 2006, pp. 89–96. IEEE Press.

[19] C. Scholl, S. Disch, F. Pigorsch, and S. Kupferschmid. Computing optimized representations for non-convex polyhedra by detection and removal of redundant linear constraints. In *TACAS*, 2009, *LNCS 5505*, pp. 383–397. Springer.

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