Counterexample Generation for Hybrid Automata * **

Johanna Nellen¹, Erika Ábrahám¹, Xin Chen¹, and Pieter Collins²

¹ RWTH Aachen University, Germany
 ² Maastricht University, The Netherlands

Abstract. The last decade brought us a whole range of over-approximative algorithms for the reachability analysis of *hybrid automata*, a widely used modeling language for systems with combined discrete-continuous behavior. Besides theoretical results, there are also some tools available for proving *safety* in the continuous time domain. However, if a given set of critical states is found to be reachable, these tools do not provide *counterexamples* for models beyond timed automata.

This paper investigates the question whether and how available tools can be used to generate counterexamples, even if this functionality is not directly supported. Using the tools SpaceEx and FLOW^{*}, we discuss possibilities to solve our task with and without modifying the tools' source code, report on the effort and the efficiency of implementation, and propose a simulation-based approach for the validation of the resulting (possibly spurious) counterexamples.

1 Introduction

Hybrid systems are systems that exhibit both continuous and discrete behavior. Typical examples are physical systems regulated by discrete controllers, e. g., automotive control systems or controlled chemical plants. Hybrid systems are often modeled as hybrid automata [1], for which the reachability problem is undecidable. Despite undecidability and driven by the fact that most hybrid systems in industrial context are safety-critical, a lot of effort was put into the development of reachability analysis techniques for hybrid automata. State-of-the-art tools like SpaceEx [2] and FLOW* [3] try to compute an over-approximation of the reachable state space and can therefore be used to prove safety, i. e., that a given set of unsafe states cannot be reached from a set of initial states in a given model. However, if the over-approximation of the reachable states contains unsafe states then no conclusive answer can be given.

Counterexamples in form of system runs leading to unsafe states would be extremely valuable, even if they are *spurious*, i.e., if they were considered in the analysis but are not possible in the given model. For safe models they could

^{*} The original publication is available at http://www.springerlink.com.

^{**} This work is supported by the DFG research training group AlgoSyn and the DFG research project HyPro.

help to reduce the approximation error in the analysis efficiently, whereas for unsafe models they could provide important information about the source of the critical system behavior. Counterexamples would enable the application of counterexample-guided abstraction refinement (CEGAR) techniques and could also play an important role in controller synthesis.

Unfortunately, none of the available tools for hybrid automata reachability analysis with continuous time domain computes counterexamples. It is surprising since *internally* they possess sufficient information to generate at least a coarse over-approximation of a counterexample in form of a sequence of jumps (i. e., changes in the discrete part of the system state), augmented with time intervals over-approximating the time durations between the jumps. In this paper we

- 1. examine whether it is possible to either use *augmented* system models or to extract information from the *output* of the SpaceEx tool such that we can *synthesize* over-approximations of counterexamples;
- 2. study how the *efficiency* can be improved by extending the functionality of the FLOW* tool *internally*, i.e., by making modifications to the source code;
- 3. develop a simulation-based approach to *validate* the counterexample overapproximations, i.e., to determine unsafe paths in the over-approximation.

We have chosen SpaceEx and FLOW^{*} for our experiments because on the one hand SpaceEx is one of the most popular hybrid automata reachability analysis tools and on the other hand some of the authors belong to the implementation team of FLOW^{*}, i.e., the modification of the source code of FLOW^{*} could be done safely. Unfortunately, counterexample generation without tool extension is unsatisfactory: we need either expensive additional analysis runs for enlarged systems or parsing hidden information from debug output. The results demonstrate the need to extend the functionality of available analysis tools to generate counterexamples internally. However, even if that task is done, the results strongly over-approximate counterexamples, whose existence can be indicated but not proven. Thus we need novel methods to refine and validate the results, posing highly challenging problems in both theory and practice.

Related work In this paper we focus on reachability analysis techniques for continuous-time hybrid automata that apply a fixed-point-based forward-reachability iteration [1]. Such algorithms need two main ingredients: (a) A technique to represent state sets and to compute certain operations on them like union, intersection, Minkowski sum, etc. All the available tools work with overapproximative representations and computations. Popular approaches use either geometric objects like hyperrectangles [4], polyhedra [5–9], zonotopes [10, 11], orthogonal polyhedra [12] or ellipsoids [13], or symbolic representations like support functions [14, 15] or Taylor models [16, 17]. The choice of the representation is crucial, as it strongly influences the approximation error and the efficiency of the computations. (b) A method to compute one-step-successors of state sets both for continuous flow and discrete jumps. A flowpipe is an over-approximation of the states that are reachable from a given initial set of states by letting time progress within a certain maximal time horizon. To compute a flowpipe, the maximal time horizon is often divided into smaller intervals and the flowpipe is represented as a (finite) union of state sets (flowpipe segments), each covering one of the smaller intervals [5].

The analysis tools HyTech [6], PHAVer [7] and the Multi-Parametric Toolbox [8] use convex polyhedra for the over-approximative representation of state sets, SpaceEx [2] additionally allows the usage of support functions. In [18], the state sets are over-approximated by level sets. The tool d/dt [19] uses grid paving as over-approximations. MATISSE [20] over-approximates state sets by zonotopes. The MATLAB Ellipsoidal Toolbox [21] supports the over-approximative representation of sets by ellipsoids, FLOW* by Taylor models. In ARIADNE [22], the state sets may be over-approximated by Taylor models or grid pavings. In contrast to the other tools, FLOW*, HyTech, PHAVer, ARIADNE and d/dt also support the analysis of non-linear hybrid automata (with non-linear differential equations).

None of these tools supports the generation of counterexamples. There are some works [23, 24] related to counterexample generation for hybrid systems, but they are mostly devoted to CEGAR approaches for restricted classes of hybrid automata like, e.g., (initialized) rectangular automata.

Outline After some preliminaries in Section 2, we describe in the Sections 3 and 4 how we can compute over-approximations of counterexamples for unsafe models, whose validation is discussed in Section 5. Section 6 concludes the paper.

2 Preliminaries

By \mathbb{N} , \mathbb{Z} and \mathbb{R} we denote the set of all natural (with 0), integer and real numbers, respectively, by $\mathbb{R}_{\geq 0}$ the non-negative reals, and use $\mathbb{N}_{>0} = \mathbb{N} \setminus \{0\}$. For some $n \in \mathbb{N}_{>0}$, let $Var = \{x_1, \ldots, x_n\}$ be an ordered set of variables over \mathbb{R} . We use the notation $x = (x_1, \ldots, x_n)$, and denote by Var' and Var the renamed variable sets $\{x'_1, \ldots, x'_n\}$ and $\{\dot{x}_1, \ldots, \dot{x}_n\}$, respectively. Given a real-arithmetic formula ϕ over



Fig. 1. The thermostat example

Var, its satisfaction set is $\llbracket \phi \rrbracket = \{v \in \mathbb{R}^n \mid \phi[v/x] = true\}$; we call ϕ convex if $\llbracket \phi \rrbracket$ is convex. Let $\Phi(Var)$ be the set of all quantifier-free convex real-arithmetic formulas (so-called *predicates*) over *Var*. A predicate is *linear* if it can be expressed in linear real arithmetic.

Definition 1 (Syntax of hybrid automata). A hybrid automaton (HA) is a tuple $\mathcal{H} = (Loc, Var, Edge, Dyn, Inv, Init)$ with the following components:

- Loc is a finite set of locations or modes.
- $Var = \{x_1, ..., x_n\}$ is a finite ordered set of variables over \mathbb{R} . A valuation $v = (v_1, ..., v_n) \in \mathbb{R}^n$ defines for each i = 1, ..., n the value v_i for x_i . A state is a mode-valuation pair $\sigma = (l, v) \in Loc \times \mathbb{R}^n = \Sigma$.



Fig. 2. The navigation benchmark

- $Edge \subseteq Loc \times \Phi(Var \cup Var') \times Loc$ is a finite set of edges. For an edge $e = (l, \phi, l') \in Edge$ we call l (l') the source (target) mode of e and ϕ its transition relation.
- $Dyn: Loc \rightarrow \Phi(Var \cup Var)$ assigns a dynamics to each mode.
- $Inv: Loc \rightarrow \Phi(Var)$ assigns an invariant to each mode.
- Init: Loc $\rightarrow \Phi(Var)$ specifies the initial valuations for each mode.

Since we do not use the parallel composition of hybrid automata in this paper, for simplicity we skipped composition-relevant parts in the above definition.

A toy example of a *thermostat* is depicted graphically in Figure 1. The rectangles represent modes; their names, dynamics and invariants are specified inside the rectangle. Initial valuations are specified on an incoming edge of a given mode without a source mode; a missing incoming edge stays for the initial condition *false*. Figure 2 shows the *navigation benchmark* [25], used later for experiments. It models an object moving in the \mathbb{R}^2 plane. The velocity (v_1, v_2) of the object depends on its position (x_1, x_2) in a grid. For some experiments we add a parameter ε to the navigation benchmark to enlarge the satisfaction sets of guards and invariants by replacing all upper bounds ub (lower bounds lb) by $ub + \varepsilon$ $(lb - \varepsilon)$.

$$\frac{l \in Loc \quad v, v' \in \mathbb{R}^n \quad t \in \mathbb{R}_{\geq 0} \quad f:[0,t] \to \mathbb{R}^n \text{ differentiable } \quad \dot{f} = \frac{df}{dt}}{f(0) = v \quad f(t) = v' \quad \forall t' \in [0,t]. \quad f(t') \in \llbracket Inv(l) \rrbracket \land (f(t'), \dot{f}(t')) \in \llbracket Dyn(l) \rrbracket} \text{ TIME}} \\ \frac{l,l' \in Loc \quad v,v' \in \mathbb{R}^n \quad v \in \llbracket Inv(l) \rrbracket \quad v' \in \llbracket Inv(l') \rrbracket \quad e = (l,\phi,l') \in Edge \quad (v,v') \in \llbracket \phi \rrbracket}{(l,v) \stackrel{e}{\to} (l',v')} \text{ JUMF}}$$



Definition 2 (Semantics of hybrid automata). The operational semantics of a HA $\mathcal{H} = (Loc, Var, Edge, Dyn, Inv, Init)$ with $Var = \{x_1, \ldots, x_n\}$ is given by the rules of Figure 3. The first rule specifies time evolution (time steps), the second one discrete mode changes (jumps).

Let $\to = \bigcup_{t \in \mathbb{R}_{\geq 0}} \xrightarrow{t} \cup \bigcup_{e \in Edge} \xrightarrow{e}$. A path of \mathcal{H} is a (finite or infinite) sequence $(l_0, v_0) \to (l_1, v_1) \to \dots$ For an initial path we additionally require $v_0 \in [Init(l_0)]$. A state $(l, v) \in \Sigma$ is called reachable in \mathcal{H} if there is an initial path $(l_0, v_0) \to (l_1, v_1) \to \dots$ of \mathcal{H} and an index $i \geq 0$ such that $(l_i, v_i) = (l, v)$.

Please note that each reachable state (l, v) of \mathcal{H} can be reached via an initial path of \mathcal{H} of the form $(l_0, v_0) \stackrel{t_0}{\to} (l_0, v'_0) \stackrel{e_0}{\to} \dots (l_{n-1}, v_{n-1}) \stackrel{t_{n-1}}{\to} (l_{n-1}, v'_{n-1}) \stackrel{e_{n-1}}{\to} (l_n, v_n) \stackrel{t_n}{\to} (l_n, v'_n) = (l, v)$ with alternating time steps and jumps for some $n \in \mathbb{N}$. In the following we consider only paths of this form.

A trace e_0, e_1, \ldots describes a sequence of jumps with $e_i \in Edge$ such that the target mode of e_i equals the source mode of e_{i+1} for all $i \in \mathbb{N}$. If we can assume that there is at most one jump between each mode pair, we also identify traces by the sequence l_0, l_1, \ldots of modes visited. Such a trace represents the set of all paths $(l_0, v_0) \stackrel{t''_0}{\to} (l_0, v'_0) \stackrel{e_0}{\to} (l_1, v_1) \stackrel{t''_1}{\to} (l_1, v'_1) \stackrel{e_1}{\to} \ldots$ We say that those paths are contained in the symbolic path.

A timed trace $e_0, [t_0, t'_0], e_1, [t_1, t'_1], \ldots$ annotates a trace e_0, e_1, \ldots with time intervals and represents the set of all paths $(l_0, v_0) \stackrel{t''_0}{\rightarrow} (l_0, v'_0) \stackrel{e_0}{\rightarrow} (l_1, v_1) \stackrel{t''_1}{\rightarrow} (l_1, v'_1) \stackrel{e_1}{\rightarrow} \ldots$ with $t''_i \in [t_i, t'_i]$ for all $i \in \mathbb{N}$. We say that $e_0, [t_0, t'_0], e_1, [t_1, t'_1], \ldots$ is a timed trace of the represented paths, which are *contained* in the timed trace.

Given a HA \mathcal{H} and a set B of unsafe states of \mathcal{H} , the reachability problem poses the question whether the intersection of B with the reachable state set of \mathcal{H} is empty, i.e., whether \mathcal{H} is safe. If \mathcal{H} is unsafe, a counterexample is an initial path of \mathcal{H} leading to an unsafe state from B. For models with weaker expressivity, for example hybrid automata defined by linear predicates and constant derivatives (i.e., dynamics of the form $\bigwedge_{x \in Var} \dot{x} = c_x$ with $c_x \in \mathbb{Z}$ for all $x \in Var$), the bounded reachability problem is decidable and, for unsafe models, counterexamples can be generated (e.g., by bounded model checking using SMT solving with exact arithmetic). However, the computation of counterexamples for general hybrid automata is hard. Theoretically, it could be done by (incomplete) under-approximative reachability computations, but currently there are no techniques available for this task. We propose an approach to generate and refine *presumable counterexamples*, which are timed traces that *might* contain a counterexample; presumable counterexamples that do *not* contain any counterexample are called *spurious*.

3 Generating Traces for Presumable Counterexamples

Existing hybrid automata analysis tools like SpaceEx offer as output options either the computed over-approximation of the reachable state space, its intersection with the unsafe states, or just the answer whether unsafe states are reachable or not (in the over-approximation). However, in contrast to tools for discrete automata, none of the tools for hybrid automata provides counterexamples.

In this section we show how a *trace* explaining the reachability of unsafe states can be computed. We present three different approaches: The first approach augments hybrid automata with auxiliary variables to make observations about the computation history of the analysis. The second approach can be used if the analysis tool outputs sufficient information about the paths that have been processed during the analysis. The third approach suggests to implement some new functionalities efficiently in existing tools. In our experiments we used SpaceEx v0.9.7c, VMware server, and the latest FLOW^{*} version but with the proposed extensions.

3.1 Approach I: Model Augmentation

We extend the model with new variables to make book-keeping about traces that lead to unsafe states in the reachability analysis. First we augment the model and analyze the augmented system to observe the *number of jumps* until an unsafe state is reached. Then we augment and analyze an unrolled model to observe unsafe *traces*.

Determining the counterexample length We augment the model and analyze it to gain information about the length of paths leading to unsafe states. We introduce a counter tr with initial value 0, define $\dot{tr}=0$ in each mode, and let each jump increase the counter value by one.

However, the unboundedness of tr would render the fixed-point analysis to be non-terminating. To bound tr from above, we define a constant max_{tr} and either extend the invariants or the edge guards to forbid higher values.

The value of max_{tr} should be guessed, and in case the analysis of the augmented model reports safety, increased. A possible guess could be the number of iterations during the fixed-point analysis of the original model, which is reported by SpaceEx and specifies how many times the tool computed a (time+jump) successor of a state set. To get a smaller value (and thus shorter counterexamples with less computational effort), the reachability analysis could be stopped when an unsafe state is reached. Unfortunately, SpaceEx does not offer this option.



Fig. 4. The guard (left) and the invariant (right) augmentation of the thermostat model

Definition 3 (Guard and invariant augmentation). Let $\mathcal{H} = (Loc, Var, Edge, Dyn, Inv, Init)$ be a HA and $max_{tr} \in \mathbb{N}$. The guard augmentation of \mathcal{H} is the HA $\mathcal{H}_{quard} = (Loc, Var \cup \{tr\}, Edge', Dyn', Inv, Init')$ with

- $Edge' = \{ (l, (\phi \land tr \le max_{tr} 1 \land tr' = tr + 1), l') \mid (l, \phi, l') \in Edge \};$
- Dyn'(l) = (Dyn(l) \land tr=0) for each $l \in Loc;$
- $Init'(l) = (Init(l) \land tr=0)$ for each $l \in Loc$.

The invariant augmentation of \mathcal{H} is the HA $\mathcal{H}_{inv} = (Loc, Var \cup \{tr\}, Edge'', Dyn', Inv'', Init')$ with Dyn' and Init' as above and

 $- Edge'' = \{(l, (\phi \land tr'=tr+1), l') \mid (l, \phi, l') \in Edge\}; \\ - Inv''(l) = (Inv(l) \land tr \leq max_{tr}) \text{ for each } l \in Loc.$

Figure 4 illustrates the augmentation on the thermostat example. Note that, apart from restricting the number of jumps, the above augmentation does not modify the original system behavior. The size of the state space is increased by the factor $max_{tr}+1$, since the value domain of tr is $[0, max_{tr}] \subseteq \mathbb{N}$ for the constant max_{tr} .

When we analyze the augmented model, SpaceEx returns for each mode in the over-approximated set of reachable unsafe states an over-approximation [l, u]for the values of tr.

Since tr takes integer values only, the lower and upper bounds are not approximated, i.e., during analysis both after l and after u (over-approximative) jump computations unsafe states were reached, but we do not have any information about the values in between. Therefore we fix the number k, describing the length of counterexamples we want to generate, to be either l or u.

We made some experiments for the thermostat example with unsafe states $t \leq$ 19, and for the navigation benchmark with unsafe states $(x_1, x_2) \in [1, 2] \times [0, 1]$. Table 1 compares for different max_{tr} values the number of SpaceEx iterations, the running times, and the resulting tr values for the original models, the guard and the invariant augmentations. For the same number of iterations, the augmented models need in average some more (but still comparable) time for the analysis than the original models; the invariant augmentations seem to be similar in terms of running time.

Table 1. Evaluation of the guard and invariant augmentations with a sampling time of 0.1 and a time horizon of 30

model	augment.	max_{tr}	#iter.	fixed point	running time [secs]	tr
thermostat	none	-	5/11/31	no/no/no	0.11/0.24/0.69	-
example	guard	4/10/30	5/11/31	yes/yes/yes	0.25/0.65/2.15	[1,4]/[1,10]/[1,30]
	invar.	4/10/30	5/11/31	yes/yes/yes	0.30/0.78/2.53	[1,4]/[1,10]/[1,30]
navigation	none	-	29/168/3645	no/no/no	1.54/8.63/1598.63	-
benchmark	guard	4/10/30	29/168/3645	yes/yes/yes	1.92/12.25/2088.88	[4,4]/[4,10]/[4,30]
	invar.	4/10/30	29/168/3160	yes/yes/yes	2.06/13.02/1466.08	[4,4]/[4,10]/[4,30]
navigation2	none	-	32/524	no/no	7.55/254.91	-
benchmark,	guard	4/10	32/524	yes/yes	7.55/284.19	[4, 4]/[4, 10]
$\varepsilon = 0.1$	invar.	4/10	32/524	yes/yes	7.35/293.88	[4, 4]/[4, 10]

Trace encoding In order to observe the traces leading to unsafe states, we need to remember the jumps in the order of their occurrences. We achieve this by unrolling the transition relation of the original model k times, where k is the counterexample length determined in the previous step.

We could define the unrolling by copying each mode k + 1 and each edge k times, and let the *i*th copy of an edge connect the *i*th-copy of the source mode with the (i + 1)st copy of the target mode. To remember the jumps taken, we introduce k auxiliary variables tr_1, \ldots, tr_k and store on the *i*th copy of an edge the edge's identity in tr_i . Such an unrolling would cause a polynomial increase in the size of the model.

However, in such an unrolling there might be different traces leading to the same mode. SpaceEx would *over-approximate* these trace sets by a mode-wise closure, such that we cannot extract them from the result. E. g., for two traces l_1, l_2, l_4, l_5, l_7 and l_1, l_3, l_4, l_6, l_7 , resp., also the trace l_1, l_2, l_4, l_6, l_7 would be included in the over-approximation. Therefore, we copy each mode as many times as the number of different traces of length up to k leading to it. This yields an exponential growth in k for the number of locations and transitions.

We augment the unrolled model to observe the traces to unsafe states. In a naive encoding, we identify edges e_1, \ldots, e_d by numbers $1, \ldots, d$, and introduce new variables tr_i for $i=1,\ldots,k$ to store the edges taken.

We also define an advanced encoding which needs less auxiliary variables. If the domain of a variable includes $[0, |Edge|^n]$ for some $n \in \mathbb{N}$ then we can use it to store a *sequence* of n edges: Each time an edge is taken, we multiply the current value by |Edge| and add the identity of the taken edge. This way we need $\left\lceil \frac{k}{n} \right\rceil$ auxiliary variables to encode a path of length k.

Definition 4 (k-unrolling, trace encoding). Assume a HA $\mathcal{H} = (Loc, Var, Edge, Dyn, Inv, Init)$ with an ordered set $\{e_1, \ldots, e_d\}$ of edges. The k-unrolling of \mathcal{H} is the HA $\mathcal{H}_u = (Loc_u, Var_u, Edge_u, Dyn_u, Inv_u, Init_u)$ with

- $Loc_u = \bigcup_{i=1,\dots,k+1} Loc^i;$
- $Var_u = Var;$
- $Edge_{u} = \{((l_{1}, \dots, l_{i}), \phi, (l_{1}, \dots, l_{i}, l_{i+1})) \mid 1 \le i \le k \land (l_{i}, \phi, l_{i+1}) \in Edge\};$



Fig. 5. Naive trace encoding of the thermostat example with depth 3

- $Dyn_u(l_1, \ldots, l_i) = Dyn(l_i) \text{ for all } (l_1, \ldots, l_i) \in Loc_u;$
- $Inv_u(l_1,...,l_i) = Inv(l_i)$ for all $(l_1,...,l_i) \in Loc_u$;
- $Init_u(l_1, \ldots, l_i) = Init(l_i)$ for i = 1 and false otherwise, for all $(l_1, \ldots, l_i) \in$ Loc_u .

The naive trace encoding of \mathcal{H} with depth k is the HA $\mathcal{H}_1 = (Loc_u, Var_1, Edge_1, edge_1)$ $Dyn_1, Inv_u, Init_1)$ with

- $Var_1 = Var \cup \{tr_1, \dots, tr_k\};$ $Edge_1 = \{((l_1, \dots, l_i), \phi \land tr'_i = j, (l_1, \dots, l_i, l_{i+1})) \mid 1 \le i \le k \land e_j = (l_i, \phi, l_{i+1})$ $\in Edge\}$:
- $Dyn_1(l_1, ..., l_i) = Dyn_u(l_1, ..., l_i) \land \bigwedge_{j=1}^k \dot{t}r_j = 0 \text{ for all } (l_1, ..., l_i) \in Loc_u;$ Init_1(l_1, ..., l_i) = Init_u(l_1, ..., l_i) \land \bigwedge_{j=1}^k tr_j = 0 \text{ for all } (l_1, ..., l_i) \in Loc_u.

Let $n \in \mathbb{N}_{>0}$ such that $[0, d^n]$ is included in the domain of each tr_i and let $z = \left\lceil \frac{k}{n} \right\rceil$. The advanced trace encoding of \mathcal{H} with depth k is the HA \mathcal{H}_2 = $(Loc_u, Var_2, Edge_2, Dyn_2, Inv_u, Init_2)$ with

- $Var_2 = Var \cup \{tr_1, \ldots, tr_z\};$
- $\begin{array}{l} k \wedge e_{j} = (l_{i}, \phi, l_{i+1}) \in Edge; \\ Dyn_{2}(l_{1}, \dots, l_{i}) = Dyn_{u}(l_{1}, \dots, l_{i}) \wedge \bigwedge_{j=1}^{z} \dot{t}r_{j} = 0 \text{ for all } (l_{1}, \dots, l_{i}) \in Loc_{u}; \\ Init_{2}(l_{1}, \dots, l_{i}) = Init_{u}(l_{1}, \dots, l_{i}) \wedge \bigwedge_{j=1}^{z} tr_{j} = 0 \text{ for all } (l_{1}, \dots, l_{i}) \in Loc_{u}. \end{array}$

An example unrolled model for the thermostat with naive trace encoding is shown in Figure 5. Note that depending on the chosen trace encoding, up to kauxiliary variables are added to the system.

Using our implementation for the proposed trace encodings, in Table 2 we compare the model sizes and the analysis running times for the thermostat example and the navigation benchmark. Compared to the original model, the analysis running times for the trace encodings increase only slightly. The last column lists the computed traces, which are (as expected) the same for both encodings.

$\mathbf{3.2}$ Approach II: Parsing the Output of SpaceEx

The approach introduced above does not scale for large systems, since the unrolling blow up the models too strongly. If the verification tool offers enough

Table 2. Evaluation of the naive and advanced trace encodings for the thermostat example and the navigation benchmark (k = 4, time step 0.1, time horizon 30) using $\pi_1 = l_{14}, l_{13}, l_9, l_6, l_2, \pi_2 = l_{14}, l_{10}, l_7, l_6, l_2, \pi_3 = l_{14}, l_{10}, l_7, l_3, l_2$ and $\pi_4 = l_{14}, l_{10}, l_9, l_6, l_2$

model	trace	#locs	#trans	#vars	n	time [secs]	solutions
thermostat	none	2	2	1	-	0.136	-
example	naive	5	4	5	4	0.312	on, off, on, off, on
	adv.	5	4	2	4	0.147	on, off, on, off, on
navigation	none	14	36	5	-	1.372	-
benchmark	naive	81	80	9	3	1.777	$\pi_1; \pi_2; \pi_3; \pi_4$
	adv.	81	80	7	3	1.503	$\pi_1; \pi_2; \pi_3; \pi_4$

information about the analyzed system traces, it is perhaps also possible to extract from the tool's output the same information we gathered by system augmentation and additional analysis runs. We are interested in determining traces that lead to unsafe states during the analysis, since they are candidates for presumable counterexamples. Without loss of generality, we assume that unsafe states are restricted to a single mode.

SpaceEx stores in a FIFO list a sequence of *symbolic states*, each of them consisting of a mode and a state set, whose successors still have to be computed in the forward reachability analysis algorithm. This socalled *waiting list* contains initially each mode with its initial valuation set (if not empty). In each iteration, the next element from the list is taken. Its flowpipe for a user-defined time horizon and all possible (non-empty) jump successors of the flowpipe seg-



Fig. 6. SpaceEx search tree

ments are computed and those that were not yet processed are added to the list. As illustrated in Figure 6, this computation hierarchy corresponds to a tree whose nodes are processed in a breadth-first manner. Each node corresponds to a mode and a set of valuations, which was found to be reachable. The upper indices on the nodes show the number of computed successor sets, whereas gray nodes in the figure represent successors that are contained in another already processed set in the same mode and are therefore not added to the tree.

SpaceEx does not output the structure of this tree. However, using *debug level 2*, we can make use of more verbose console outputs to get additional informations.

- When an iteration starts, SpaceEx outputs a text from which we can extract the iteration number i ("Iteration 5...").
- SpaceEx starts the flowpipe computation and outputs the mode of the current symbolic state ("applying time elapse in location loc()==114").

- The computation of jump successors follows, which is edge-wise. For each edge, whose source is the current mode, its label, source, target is printed ("applying discrete post of transition with label navigation.trans from location loc()==114 to location loc()==113").
- SpaceEx determines, which of the previously computed flowpipe segments intersect with the guard ("found 1 intervals intersecting with guard").
- The jump successors for the intersecting flowpipe segments are computed and, if not yet processed, put to the waiting list. Before switching to the next outgoing edge, some information on the computation time is given ("Discrete post done after 0.098s, cumul 0.098s").
- When all outgoing edges are handled, the iteration is done, and the following output gives us the total number of processed symbolic states and the current size of the waiting list ("1 sym states passed, 2 waiting").
- After termination of the analysis some general analysis results are printed,
 e.g., the number of iterations, whether a fixed point was found or not, the analysis time, and whether unsafe states were reached.

If we would succeed to re-construct the search tree (or at least the involved mode components and their hierarchy) using the above information, we could extract traces that lead to unsafe states in the tree.

The good news is that from the above outputs we can extract quite some information regarding the search tree, such that in some cases we can construct counterexamples. The bad news is that it is not sufficient to reconstruct all details. E. g., since the waiting list size is reported after each iteration, we can determine the number of new waiting list elements added during the last iteration (the new list size minus the old list size minus 1). If this number equals the total number of all intersecting intervals over all analyzed edges then we can determine the mode components of the waiting list elements. However, if some of the successors are already processed and therefore not added to the queue then we cannot know for sure which sets were added. For example, if out of two sets having the same mode component only one was added to the queue, then we cannot know which of them. To avoid wrong guesses, those cases are skipped and not considered further in our implementation.

Without model augmentation, it is not possible to restrict the SpaceEx search to paths of a given length, therefore we cannot directly compare this method to the results of Table 2. We made experiments with the navigation benchmark using the debug output D2 of SpaceEx. For 50 iterations, with a computation time of 32.28 seconds we found 11 traces leading to unsafe states in l_2 . When considering only 25 iterations, the computation time is 12.69 seconds and only 4 traces are found. The increase of running time for using SpaceEx with debug level D2 instead of the default value *medium* was negligible in our experiments.

A single analysis run suffices to extract traces of counterexamples thus this method seems to be superior to the augmentation approaches if the analysis tool communicates enough information about the system traces. However, if not all relevant details are accessible, not all traces can be rebuilt safely.

Table 3. Trace generation using FLOW^{*} (k = 4, time step 0.1, time horizon 30, $\pi_1 = l_{14}, l_{13}, l_9, l_6, l_2, \pi_2 = l_{14}, l_{10}, l_7, l_6, l_2$ and $\pi_3 = l_{14}, l_{10}, l_7, l_3, l_2$)

model	running time [secs]	solutions
thermostat example	0.23	on, off, on, off, on
navigation benchmark	8.37	π_1,π_2,π_3

3.3 Approach III: Extending the Functionality of Flow*

Extracting information from the textual output of a tool is an overhead, since the information was already computed during analysis. Moreover, it might be imprecise if we do not have access to all needed information.

Instead, we could generate counterexample traces on-the-fly by attaching to each symbolic state in the waiting queue the trace that lead to it during the search. The waiting queue initially contains initial symbolic states, to which we attach themselves. If we add a new symbolic state with location l as a successor of another symbolic state, then we attach to the new state the path of the predecessor state extended with the jump whose successor the new state is. The reachability computation will stop when the tree is complete till depth k (the maximal jump depth). Next, FLOW^{*} intersects each tree node with the unsafe set. If a non-empty intersection is detected, the tool dumps the trace attached to the unsafe node.

To implement the above functionality, only minor changes had to be made in FLOW^{*}, but it saves us the time of augmenting the system or parsing tool output. We made experiments in line with Table 2 for the thermostat example and the navigation benchmark. The results are shown in Table 3. Please note that FLOW^{*} does not compute the trace l_{14} , l_{10} , l_9 , l_6 , l_2 , which is spurious. We additionally analyzed the navigation benchmark with k = 8, where FLOW^{*} generated 8 traces to unsafe states in l_2 with initial valuation $x_1 \in [3, 3.5]$, $x_2 \in [3, 4]$, $v_1 \in [-0.1, 0.1]$ and $v_2 \in [-0.8, -0.5]$.

4 Generating a Presumable Counterexample

In this section we show how we can generate presumable counterexamples by extending the previously computed traces to *timed* traces. Given a trace, we compute a reduced model that has the jumps of the trace only. This model is augmented with a clock *timer* and variables $tstamp_i$, i = 1, ..., k, one for each jump in the trace. The clock is initialized to 0 and has derivative 1. Whenever a jump is taken, the clock value is stored in the timestamp of the jump and the clock is reset to 0. Figure 7 illustrates the above transformation.

Definition 5 (Trace model). Given a hybrid automaton $\mathcal{H} = (Loc, Var, Edge, Dyn, Inv, Init)$ and a finite trace e_1, \ldots, e_k of \mathcal{H} with $e_i = (l_i, \phi_i, l_{i+1})$, the trace model of \mathcal{H} for e_1, \ldots, e_k is the HA $\mathcal{H}' = (Loc', Var', Edge', Dyn', Inv', Init')$ with

$$- Loc' = \{(l_1, 0), \dots, (l_k, k), (l_{k+1}, k+1)\};\$$



Fig. 7. Trace model of the thermostat example for k = 3

Table 4. Comparison of the timed traces for the navigation benchmark computed by SpaceEx and FLOW^{*} (k = 4, time step 0.1, time horizon 30, traces from Table 2 and 3)

Initial states: $l_{14}, x_1 \in [3.0, 3.8], x_2 \in [3.0, 4.0], v_1 \in [-0.1, 0.1], v_2 \in [-0.8, -0.5]$							
SpaceEx result in 6.46s:							
$\pi_1: l_{14}, [0.0, 0.6],$	$l_{13}, [0.0, 1.4],$	$l_9, [1.5, 1.8],$	$l_6, [2.3, 2.5], l_2$				
$\pi_2: l_{14}, [0.0, 1.9],$	$l_{10}, [1.5, 1.9],$	$l_7, [0.2, 2.5],$	$l_6, [0.0, 2.4], l_2$				
$\pi_3: l_{14}, [0.0, 1.9],$	$l_{10}, [1.5, 1.9],$	$l_7, [2.3, 2.5],$	$l_3, [0.0, 1.0], l_2$				
$\pi_4: l_{14}, [0.0, 1.9],$	$l_{10}, [0.0, 0.6],$	$l_9, [0.9, 1.4],$	$l_6, [0.0, 0.0], l_2$				
$FLOW^*$ result in $8.37s$:						
$\pi_1: l_{14}, [0.000, 0.566],$	$l_{13}, [0.000, 1.420],$	$l_9, [1.531, 1.880],$	$l_6, [2.421, 2.422], l_2$				
$\pi_2: l_{14}, [0.000, 1.854],$	$l_{10}, [1.534, 1.836],$	l_7 , [0.310, 2.371],	$l_6, [0.000, 2.385], l_2$				
$\pi_3: l_{14}, [0.000, 1.854],$	$l_{10}, [1.534, 1.836],$	l_7 , [2.415, 2.416],	$l_3, [0.000, 0.912], l_2$				

- $Var' = Var \cup \{timer, tstamp_1, \dots, tstamp_k\};$
- $Edge' = \{ ((l_i, i), \phi_i \wedge tstamp'_i = timer \wedge timer' = 0, (l_{i+1}, i+1)) \mid i \in \{1, \dots, k\} \};$
- $Dyn'(l,i) = Dyn(l) \land timer = 1 \land \bigwedge_{i=1,\dots,k} tstamp_i = 0 \text{ for all } (l,i) \in Loc';$
- Inv'(l,i) = Inv(l) for all $(l,i) \in Loc'$;
- $Init'(l, i) = Init(l) \land timer = 0 \text{ for all } (l, i) \in Loc'.$

Another method to get timing information is as follows. Both in SpaceEx and in FLOW^{*}, the time horizon [0, T] of a flowpipe is divided into smaller time intervals $[0, \delta], [\delta, 2\delta], \ldots, [(n-1)\delta, n\delta]$ with $n\delta = T$. The flowpipe is computed as a union of flowpipe segments, one for each smaller interval. Thus the tools have internal information about the timestamps of the symbolic states in the waiting list. We make use of this fact and label the symbolic states in FLOW^{*} with the timed traces which lead to them. This way we get the timing information for free. Please note that this would also be possible for SpaceEx. In FLOW^{*} an additional backward refinement of the time intervals of the timed trace would be also possible, which we cannot describe here due to space limitations.

Table 4 shows some experimental results for the navigation benchmark. We compute timed extensions of the previously computed counterexample traces to build presumable counterexamples. The running times include for FLOW* a complete reachability analysis up to jump depth 4, and for SpaceEx the generation of the traces with Approach II and extracting timing information by building and

analyzing the trace models. Both tools have their advantages: SpaceEx computes the results faster, FLOW^{*} gives sometimes better refinements.

5 Simulation

To gain counterexamples, we identifying some suitable *candidate initial states* (CIS) from the initial state set and *refine* the timed trace separately for each CIS by restricting the timing informations.

Then we apply simulation to each CIS to find concrete counterexamples starting in the given CIS and being contained in the corresponding refined timed trace. Based on the refined timed trace of a CIS, each jump can take place within a bounded but dense time interval. We let the simulation branch on a finite set of jump time points chosen from those intervals. The choice is guided by the invariant and guard satisfaction and uses a heuristics, which iteratively discretizes time intervals with dynamic step sizes to drive the selection towards hitting conditions, e.g., in the presence of strict equations. Furthermore, the heuristics tries to abort simulation paths that do not lead to a counterexample as early as possible.

Finding candidate initial states The task of identifying CISs for simulation is non-trivial, since the timed traces over-approximate counterexamples, such that not all initial states lead to unsafe states within the given time bounds. W. l. o. g., we assume that the initial set is given as a hyperrectangle (otherwise we overapproximate the initial set by a hyperrectangle and use in the following the conjunction of the hyperrectangle with the initial set). We obtain CISs by applying a binary search on the initial set combined with a reachability analysis run to check whether the unsafe states are still reachable. As long as unsafe states are detected to be reachable from a hyperrectangle, the corner points of the hyperrectangle are added to the set of CISs. If in at least one dimension the width of the hyperrectangle is larger than a specified parameter ε , the interval is splitted (in this dimension) in the middle and both halves are analyzed again. The binary search stops if either the specified number of CISs are computed or if all hyperrectangles reach the minimal width in each dimension.

The user can choose between a depth- (DFS) and a breadth-first search (BFS) to generate CISs. DFS computes closely lying points fast, BFS searches for widely spread points at a higher computation time.

For the trace l_{14} , l_{10} , l_7 , l_6 , l_2 of the navigation benchmark, our implementation needs 19ms to create the trace model. For the DFS, SpaceEx has to be run 42 times until 10 CISs are computed from which the unsafe state l_2 is reachable in the SpaceEx over-approximation. The corresponding computation time is 7.14s. The BFS finds the first 10 CISs within 29.90s and with 133 SpaceEx calls.

For each selected CIS we determine a refined timed trace using the same method as before for computing presumable counterexamples, but now restricted to the given CIS as initial state. Simulating the dynamics For linear differential equations the initial value problem is solvable, i.e., we can compute for each state the (unique) state reachable from it in time t. Thus, for linear differential equations we use the matrix exponential (e.g. in the homogeneous case, $\dot{x} = Ax$ is solved by $x(t) = x_0 e^{At}$, where t is the time and x_0 is the initial value of x), whereas for non-linear differential equations numerical methods (e.g. Runge-Kutta) can be used. However, since either exponential function values must be determined or numerical methods are used, the computation is not exact.

Checking invariants Along a simulated timed trace, time can pass in a location only as long as the location's invariant is satisfied. The timed trace provides us for each location l_i a time interval $[t_i, t'_i]$, within which a jump to a successor location should be taken. We have to assure that the invariant is constantly fulfilled from the time where a location was entered till the time point where the jump is taken. Therefore, we compute the time successors for a set of sample time points homogeneously distributed (with a user-defined distance δ) within the time interval $[0, t'_i]$. We check the invariant for those sample time points in an increasing order. If the invariant is violated at a sample time point $t \in [0, t'_i]$, no further time can elapse in the current location. Thus all simulation paths via time points from $[t, t'_i]$ are cut off and the time interval for jumps is restricted to $[t_i, t''_i]$, where t''_i is the time point before t.

Dynamic search for suitable jump time points Non-determinism (at which time point a jump is taken) is handled by branching the simulation for those previously selected sample time points that lie inside $[t_i, t'_i]$. If the edge's guard is fulfilled at a given sample, the jump successor is computed and the corresponding simulation branch is explored in a depth-first search. The first steps of a simulation are shown in Figure 8.

The naive discretization of the dense time intervals has sometimes problems to hit guard conditions. Especially hard for the simulation are guards containing equations. To allow simulation for guards defined by equations, we enlarge the model behavior be replacing the guard equations by inequations, allowing values from a small box around a point instead of hitting the point exactly.

However, even with such an enlarging it can happen that the guard is not fulfilled at any of the selected sample time points, or from the states after the jump no counterexamples can be simulated. In this case we dynamically determine new sample time points for the jump as follows. We use two parameters, an offset and a step size, specifying a set $\{t_i + offset + j \cdot stepsize \in [t_i, t'_i] \mid j \in \mathbb{N}\}$ of sample points. Initially (as described above), the offset is 0 and the step size has the value δ . If the simulation for these sample time points does not succeed, we set the offset to $\delta/2$ and let the step size unchanged. If those points are also not successful, we iteratively half both parameter values. This adaption terminates if either the target location of the timed trace is reached (i. e., a counterexample is found) or the step size reaches some predefined lower bound. The dynamic step-size-adaption is visualized in Figure 9.



Fig. 8. Simulation: 1) Checking the invariant for $[0, t'_0]$; 2) Taking the enabled edges within $[t_0, t''_0]$ to l_1 ; 3) Expanding the next level

If a single counterexample suffices, the simulation can be stopped as soon as the unsafe location was reached. However, by heuristically searching for further counterexamples, it is also possible to provide additional information about a counterexample: Instead of the time points of the jumps along the simulation path, the biggest time intervals can be computed, such that the unsafe state is still reachable.

Table 5 shows the simulation results for some timed traces, each with a single initial state. Note that we find counterexamples (i. e., we reach the maximal jump depth) only in the two middle cases. We additionally run SpaceEx analyses with the given initial point for the first trace and could not reach the bad state with a time step of 0.001, i. e., the first timed trace is spurious. The last trace was not computed by FLOW^{*} and is therefore also spurious.

6 Conclusion and Future Work

In this paper we described an approach to find presumable counterexamples for hybrid automata based on existing reachability tools. Next we plan to improve our method by (1) a backward refinement of the time intervals on timed paths, (2) a rigorous simulation technique for hybrid automata, (3) giving a better



Fig. 9. Adaptive-step-size simulation

Table 5. Simulation results for the navigation benchmark with ε -enlarging $\pi_1 = l_{14}, [0.0, 0.2], l_{13}, [0.0, 0.4], l_9, [1.5, 1.6], l_6, [2.4, 2.5], l_2$

- with initial state (3.0084472656250005, 3.21875, -0.1, -0.8)
- $\begin{aligned} \pi_2 = l_{14}, [1.834, 1.835], l_{10}, [1.779, 1.78], l_7, [1.934, 1.936], l_6, [0.511, 0.514], l_2 \\ \text{with initial state } (3.2, 4.0, 0.1, -0.5) \end{aligned}$
- $\pi_3 = l_{14}, [0.000, 0.001], l_{10}, [1.569, 1.570], l_7, [2.429, 2.431], l_3, [0.514, 0.517], l_2 \\ \text{with initial state } (3.8, 3.0, -0.1, -0.8)$
- $\begin{aligned} \pi_4 &= l_{14}, [0.0, 0.1], l_{10}, [0.0, 0.5], l_9, [1.0, 1.3], l_6, [2.3, 2.5], l_2 \\ & \text{with initial state} \ (3.0125, 3.0, -0.1, -0.8) \end{aligned}$

timed trace	step size	ε ε	reached jump depth	simulated paths	unsafe	time [secs]
π_1	0.0005	0.0005	2	$128 \cdot 10^{8}$	0	20.91
	0.05	0.5	3	128	0	04.26
π_2	0.0005	0.5	4	964	> 50	14.82
π_3	0.0005	0.05	4	96	> 50	14.44
	0.0005	0.0005	4	96	50	10.51
π_4	0.0005	0.0005	2	$480 \cdot 10^{8}$	0	15.46
	0.05	0.05	3	480	0	07.88

heuristics to select the initial points for simulation and (4) use several tools and take the best results to minimize the overestimation in a presumable counterexample. Preliminary results suggest that the function calculus of the tool ARIADNE can be used to validate counterexamples.

References

- Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theor. Comput. Sci. 138 (1995) 3–34
- Frehse, G., Le Guernic, C., Donzé, A., Lebeltel, R.R.O., Ripado, R., Girard, A., Dang, T., Maler, O.: SpaceEx: Scalable verification of hybrid systems. In: Proc. of CAV'11. Volume 6806 of LNCS., Springer (2011) 379–395
- Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. In: Proc. of CAV'13. Volume 8044 of LNCS., Springer (2013) 258–263

- Stursberg, O., Krogh, B.H.: Efficient representation and computation of reachable sets for hybrid systems. In: Proc. of HSCC'03. Volume 2623 of LNCS., Springer (2003) 482–497
- Chutinan, A., Krogh, B.H.: Computing polyhedral approximations to flow pipes for dynamic systems. In: Proc. of CDC'98 (volume 2), IEEE Press (1998) 2089–2094
- Henzinger, T.A., Ho, P., Wong-Toi, H.: HyTech: A model checker for hybrid systems. Software Tools for Technology Transfer 1 (1997) 110–122
- Frehse, G.: PHAVer: Algorithmic verification of hybrid systems past HyTech. In: Proc. of HSCC'05. Volume 3414 of LNCS., Springer (2005) 258–273
- 8. Kvasnica, M., Grieder, P., Baotić, M.: Multi-parametric toolbox (MPT) (2004) http://control.ee.ethz.ch/~mpt/.
- Chen, X., Abrahám, E.: Choice of directions for the approximation of reachable sets for hybrid systems. In: Proc. of EUROCAST'11. Volume 6927 of LNCS., Springer (2011) 535–542
- Kühn, W.: Zonotope dynamics in numerical quality control. In: Mathematical Visualization: Algorithms, Applications and Numerics. Springer (1998) 125–134
- Girard, A.: Reachability of uncertain linear systems using zonotopes. In: Proc. of HSCC'05. Volume 3414 of LNCS., Springer (2005) 291–305
- Bournez, O., Maler, O., Pnueli, A.: Orthogonal polyhedra: Representation and computation. In: Proc. of HSCC'99. Volume 1569 of LNCS., Springer (1999) 46– 60
- Kurzhanski, A.B., Varaiya, P.: On ellipsoidal techniques for reachability analysis. Optimization Methods and Software 17 (2000) 177–237
- 14. Le Guernic, C.: Reachability Analysis of Hybrid Systems with Linear Continuous Dynamics. PhD thesis, Université Joseph Fourier (2009)
- Le Guernic, C., Girard, A.: Reachability analysis of hybrid systems using support functions. In: Proc. of CAV'09. Volume 5643 of LNCS., Springer (2009) 540–554
- Chen, X., Abrahám, E., Sankaranarayanan, S.: Taylor model flowpipe construction for non-linear hybrid systems. In: Proc. of RTSS'12, IEEE Computer Society (2012) 183–192
- Collins, P., Bresolin, D., Geretti, L., Villa, T.: Computing the evolution of hybrid systems using rigorous function calculus. In: Proc. of ADHS'12, IFAC-PapersOnLine (2012)
- Mitchell, I., Tomlin, C.: Level set methods for computation in hybrid systems. In: Proc. of HSCC'00. Volume 1790 of LNCS., Springer (2000) 310–323
- Asarin, E., Dang, T., Maler, O.: The d/dt tool for verification of hybrid systems. In: Proc. of CAV'02. Volume 2404 of LNCS., Springer (2002) 365–370
- Girard, A., Pappas, G.J.: Approximation metrics for discrete and continuous systems. IEEE Transactions on Automatic Control 52 (2007) 782–798
- Kurzhanskiy, A., Varaiya, P.: Ellipsoidal toolbox. Technical report, EECS, UC Berkeley (2006)
- Balluchi, A., Casagrande, A., Collins, P., Ferrari, A., Villa, T., Sangiovanni-Vincentelli, A.L.: Ariadne: A framework for reachability analysis of hybrid automata. In: Proc. of MTNS'06. (2006)
- Prabhakar, P., Duggirala, P.S., Mitra, S., Viswanathan, M.: Hybrid automatabased CEGAR for rectangular hybrid systems. In: Proc. of VMCAI'13. Volume 7737 of LNCS., Springer (2013) 48–67
- Duggirala, P.S., Mitra, S.: Abstraction refinement for stability. In: Proc. of IC-CPS'11, IEEE (2011) 22–31
- Fehnker, A., Ivancic, F.: Benchmarks for hybrid systems verification. In: Proc. of HSCC'04. Volume 2993 of LNCS., Springer (2004) 326–341