Assume-Guarantee Abstraction Refinement Meets Hybrid Systems

Sergiy Bogomolov[1], Goran Frehse[2], Marius Greitschus[1], Radu Grosu[3], Corina Pasareanu[4], Andreas Podelski[1], and Thomas Strump[1]

[1] University of Freiburg, Germany
{bogom,greitsch,podelski,strumpt}@informatik.uni-freiburg.de
[2] Université Joseph Fourier Grenoble 1 – Verimag, France
goran.frehse@imag.fr
[3] Vienna University of Technology, Austria
radu.grosu@tuwien.ac.at
[4] NASA Ames Research Center, USA
Corina.S.Pasareanu@nasa.gov

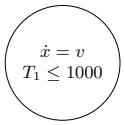
Abstract. Compositional verification techniques in the assume-guarantee style have been successfully applied to transition systems to efficiently reduce the search space by leveraging the compositional nature of the systems under consideration. We adapt these techniques to the domain of hybrid systems with affine dynamics. To build assumptions we introduce an abstraction based on location merging. We integrate the assume-guarantee style analysis with automatic abstraction refinement. We have implemented our approach in the symbolic hybrid model checker SpaceEx. The evaluation shows its practical potential. To the best of our knowledge, this is the first work combining assume-guarantee reasoning with automatic abstraction-refinement in the context of hybrid automata.

1 Introduction

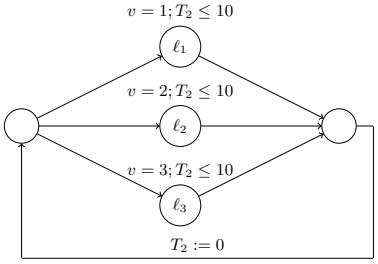
Assume-guarantee (AG) reasoning [14] is a well-known methodology for the verification of large systems. The idea behind is to decompose the verification of a system into the verification of its components, which are smaller and therefore easier to verify. A typical example of such systems would be a system comprised of a controller and a plant. In this work, we mainly concentrate on hybrid systems [1] with *stratified* controllers, i.e., controllers consisting of multiple strata (layers), where each of them is responsible for some particular plant parameter. Assume-guarantee reasoning can be performed using the following rule, ASym, where P is a safety property and $\mathcal{H}_1 \parallel \mathcal{H}_2$ denotes the parallel composition of components $\mathcal{H}_1$ and $\mathcal{H}_2$, where $\mathcal{H}_1$ is a plant and $\mathcal{H}_2$ is a controller.

$$\frac{1 : \mathcal{H}_1 \parallel A \models P \qquad 2 : \mathcal{H}_2 \models A}{\mathcal{H}_1 \parallel \mathcal{H}_2 \models P}$$

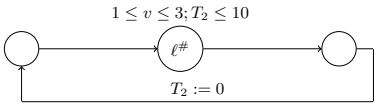
Rule ASym



(a) Plant $\mathcal{H}_1$.



(b) Controller $\mathcal{H}_2$ – unmerged.



(c) Controller $\mathcal{H}_2^{\#}$ – merged.

Fig. 1: A motivating example.

In this rule, A denotes an *assumption* about the controller of $\mathcal{H}_1$. Premise 1 ensures that when $\mathcal{H}_1$ is a part of a system that satisfies A, the system also guarantees P. Premise 2 ensures that any system that contains $\mathcal{H}_2$ satisfies A. Together the two premises imply the conclusion of the rule. The rule ASym is applicable if the assumption A is more abstract than $\mathcal{H}_2$, but still reflects $\mathcal{H}_2$'s behavior. Additionally, an appropriate assumption for the rule needs to be strong enough for $\mathcal{H}_1$ to satisfy P in premise 1.

The most challenging part of applying assume-guarantee reasoning is to come up with appropriate assumptions to use in the application of the assume-guarantee rules. Several learning and abstraction-refinement techniques [5, 13] have been proposed for automating the generation of assumptions for the verification of transition systems.

In this paper, we focus on the automated generation of assumptions in the context of hybrid systems. Similar to the work by Bobaru et al. [5] we use abstraction-refinement techniques to iteratively build the assumptions for the rule ASym. In our case, $\mathcal{H}_2$, i.e., the controller of $\mathcal{H}_1$, is abstracted. The use of over-approximations guarantees that the assumption describes the component correctly and hence premise 2 holds by construction. However, it is possible that premise 1 does not hold, in which case a counterexample is provided. The counterexample is analyzed to see if it is spurious, in which case the abstraction of $\mathcal{H}_2$ is refined to eliminate it. If the counterexample is real, then $\mathcal{H}_1 \parallel \mathcal{H}_2$ violates P.

We present a framework which can efficiently handle the class of affine hybrid systems. Due to the mixed discrete-continuous nature of hybrid systems, we need to pay special attention on the abstraction of continuous dynamics. We illustrate the idea of our compositional analysis on a toy example. Fig. 1 shows a simple hybrid automaton consisting of the plant $\mathcal{H}_1$ in Fig. 1a and controller $\mathcal{H}_2$ in Fig. 1b. We observe that the derivative of variable x in plant $\mathcal{H}_1$ depends on

the value of v governed by the controller $\mathcal{H}_2$. Furthermore, we see that the controller operates in iterations of length 10. The possible controller options are grouped in a stratum. While analyzing this system, a hybrid model checker will consider all the three options on every controller iteration which results in 3^n branches for n iterations. By noting that for some properties only the minimal and maximal values of v are of relevance, we come up with an abstracted version of the automaton $\mathcal{H}_2$ in Fig. 1c. We replace the three alternative options by only one *coarser* option. To ensure that the resulting automaton is indeed an over-approximation of the original system, we use $1 \leq v \leq 3$ as an invariant of the merged location $\ell^{\#}$, i.e., we replace the exact values of v with its bounds. This abstraction will be especially useful to prove, e.g., that within the first 1000 seconds of system operation the state $x = 4000$ will still not be reached. In the abstraction we will reduce an exponential number of branchings to a linear one. Note that this kind of location-merging abstractions is especially useful for the class of stratified controllers. The reason is that the controller structure can be exploited to efficiently generate an initial abstraction by merging locations belonging to the same stratum. Intuitively, this step allows us to adjust the precision level at which the system parameters are taken into account. If the resulting abstraction is too coarse, a finer-grained abstraction is generated in the refinement step.

The lesson we learn from this example is that merging of locations is a promising approach to generate abstractions in scope of the assume-guarantee reasoning paradigm. To ensure the conservativeness of the resulting abstraction, we compute the invariants as a convex hull of the original locations. Note that the computation of minimal and maximal values of v shown above represents a simple case of a general convex hull computation. Given the continuous, affine dynamics of the form $\dot{x}(t) = Ax(t) + u(t)$, the merged locations are computed by first eliminating the (unprimed) state variables and consequently computing the convex hull of the resulting polytopes over the derivatives. As outlined above, sometimes we might end up with *spurious* counterexamples. To overcome this issue we proceed to the phase of spuriousness checking. If the found path is indeed spurious, we refine the system by splitting one or multiple locations and continue with the analysis of this new system. Note that the assume-guarantee reasoning methodology is a variant of the CEGAR approach [6]. The essential difference of AGAR compared to CEGAR is the compositional handling of the system. We develop our approach along these lines by ensuring that the proposed algorithms work in the compositional fashion, e.g., we only abstract a part of the system and the refinement algorithm considers a projection of the found counterexample on the abstracted component. Our implementation in SpaceEx [9] shows the practical potential.

The remainder of the paper is organized as follows. We introduce the necessary preliminary notions in Sec. 2. In Sec. 3, we introduce our compositional framework. This is followed by a discussion about related work in Sec. 4. Afterwards, we present our experimental evaluation in Sec. 5. Finally, we conclude the paper in Sec. 6.

2 Preliminaries

Hybrid automata [11] provide an expressive formalism suitable for modeling complex real-world systems.

Definition 1 (Affine Hybrid Automaton). *An* affine hybrid automaton *is a tuple* $\mathcal{H} = (Loc, Var, Init, Flow, Trans, I)$*, where* Loc *is a finite set of locations,* $Var = \{x_1, \ldots, x_n\}$ *is a set of real-valued variables,* $Init(\ell) \subseteq \mathbb{R}^n$ *is the convex set of initial values for* $x_1, \ldots, x_n$ *for all locations* $\ell \in Loc$*. For each* $\ell \in Loc$*,* $Flow(\ell)$ *is a relation over the variables in* Var *and their derivatives*

$$\dot{x}(t) = Ax(t) + u(t), u(t) \in \mathcal{U},$$

where $x(t) \in \mathbb{R}^n$*,* A *is a real-valued* $n \times n$ *matrix and* $\mathcal{U} \subseteq \mathbb{R}^n$ *is a closed and bounded convex set.* $Trans$ *is a set of discrete transitions* (ℓ, g, ξ, ℓ')*, where* ℓ *and* ℓ' *are the source and the target locations,* g *is the guard (given as a linear constraint), and* ξ *is the update (given by an affine mapping).* $I(\ell) \subseteq \mathbb{R}^n$ *is the convex invariant for all locations* $\ell \in Loc$*.*

The semantics of hybrid automata is defined as follows. A *state* of $\mathcal{H}$ is a tuple $(\ell, \mathbf{x})$ consisting of a location $\ell \in Loc$ and a point $\mathbf{x} \in \mathbb{R}^n$. More formally, $\mathbf{x}$ is a valuation of the continuous variables in Var. Let $T = [0, \Delta]$ be a time interval for some $\Delta \geq 0$. A *trajectory* of $\mathcal{H}$ from state $s = (\ell, \mathbf{x})$ to state $s' = (\ell', \mathbf{x}')$ is defined by a tuple $\rho = (L, \mathbf{X})$, where $L : T \to Loc$ and $\mathbf{X} : T \to \mathbb{R}^n$ are functions that define for each time point in T the location and values of the continuous variables, respectively. The trajectory ρ starts in $(\ell, \mathbf{x})$, ends in $(\ell', \mathbf{x}')$, and obeys the following constraints:

- The sequence of time points in ρ, where the location is changed (according to L) increases strictly monotonically, starts with time point 0, and ends with time point Δ.
- There are no location changes which are not defined by L (i. e., locations are not changed during the continuous evolution).
- For all $t \in T$, the continuous variable evolution is consistent with the differential equation and invariant of $L(t)$.

We define $traj(\mathcal{H})$ as a set of all trajectories ρ for $\Delta \geq 0$. The *length* of the trajectory $|\rho|$ is equal to the number of different locations on it. The initial set of states $S_{init}(\mathcal{H})$ of $\mathcal{H}$ is defined as $\bigcup_\ell (\ell, Init(\ell))$. We say that s' is *reachable* from s if a trajectory from s to s' exists. The reachable state space $\mathcal{R}(\mathcal{H})$ of $\mathcal{H}$ is defined as the set of states such that a state s is reachable from $S_{init}(\mathcal{H})$. In this paper, we also refer to *symbolic states*. A symbolic state $s = (\ell, R)$ is defined as a tuple, where $\ell \in Loc$, and R is a convex set consisting of points $\mathbf{x} \in \mathbb{R}^n$. The continuous part R of a symbolic state is also called *region*. The symbolic state space of $\mathcal{H}$ is called the *region space*. The *convex hull* of two regions R_1 and R_2 is denoted by $\mathcal{CH}(R_1 \cup R_2)$. The *path* in the region space is a sequence of symbolic states $\pi = s_0, \ldots, s_{n-1}$. The *length* of the path $|\pi| = n$ is equal to

the number of symbolic states on it. We assume without loss of generality that there is a single bad location ℓ_{bad} with unrestricted invariant and flow. Our goal is to find a trajectory from $S_{init}(\mathcal{H})$ to the bad location. A trajectory that starts in a state s and leads to a bad location is called an *error trajectory* $\rho_e(s)$.

Composition of hybrid automata. A *product automaton* $\mathcal{N} = \mathcal{H}_1 || \ldots || \mathcal{H}_m$ denotes a set of interacting hybrid automata. The semantics of $\mathcal{N}$ is defined based on the semantics of a single hybrid automaton, with the following extensions. Every automaton in $\mathcal{N}$ is associated with a finite set of *synchronization labels*, including a special label τ in all label sets. The discrete component of a *state* s of $\mathcal{N}$ is defined as a *vector* of locations that denotes the current locations of every component in $\mathcal{N}$. Similarly, in addition to single automata, a *trajectory* of $\mathcal{N}$ maps time points to vectors of locations of each automaton. For a time point t, changes in the location vectors in a trajectory can either be caused by a single transition labeled with τ of one automaton in $\mathcal{N}$ ("interleaving transition"), or there are several automata in $\mathcal{N}$ that simultaneously fire transitions with equal synchronization labels other than τ ("synchronized transition"). We refer to the work by Donzé et al. [7] for more details.

3 Compositional Framework for Hybrid Systems

In this section, we introduce the main ingredients of our compositional framework: the abstraction of a hybrid system, an algorithm for spuriousness check, and a refinement algorithm.

3.1 Abstraction Algorithm

We construct our abstraction by partially merging system locations. To formally define the abstraction, we introduce a location abstraction function α and a location concretization function α^{-1} as follows.

Definition 2 (Location abstraction function). *Location abstraction function $\alpha : Loc \to Loc^{\#}$ provides a mapping from every concrete location in Loc to its abstract counterpart. Furthermore, we require $|Loc^{\#}| \leq |Loc|$, i.e., the abstract system should have at most the same number of locations as the original one.*

Definition 3 (Location concretization function). *Location concretization function $\alpha^{-1} : Loc^{\#} \to 2^{Loc}$ provides a mapping from every abstract location in $Loc^{\#}$ to the set of concrete locations which were merged into it.*

If $\ell \in \alpha^{-1}(\ell^{\#})$, then ℓ is a *corresponding* location to the abstract location $\ell^{\#}$. Furthermore, we abuse the notation and apply a concretization function not only to abstract locations, but also to abstract symbolic states and abstract symbolic paths. We define an abstract hybrid automaton $\mathcal{H}^{\#}$ induced by the location abstraction function α and concrete hybrid automaton $\mathcal{H}$ as follows:

Definition 4 (Location-merging abstraction).
Let $\mathcal{H} = (Loc, Var, Init, Flow, Trans, I)$ *be a hybrid automaton and* $\alpha : Loc \to Loc'$ *be a location abstraction function. The abstract automaton* $\mathcal{H}^\# = (Loc^\#, Var^\#, Init^\#, Flow^\#, Trans^\#, I^\#)$ *induced by the location-merging abstraction with respect to the location function* α *is defined as follows:*

- $Loc^\# = Loc'$*, i.e., the location abstraction function provides which locations of* $\mathcal{H}$ *are to be merged. We assume that* α *keeps the bad location* ℓ_{bad} *as a singleton.*
- $Var^\# = Var$*, i.e., the abstraction preserves the continuous variables of the original system.*
- $\forall \ell^\# \in Loc^\# : Init^\#(\ell^\#) = \mathcal{CH}(\bigcup_{\ell \in \alpha^{-1}(\ell^\#)} Init(\ell))$*, i.e., the regions describing the initial values in concrete locations are first merged into one (possibly non-convex) set and afterwards are over-approximated by a convex hull.*
  Note that if an abstract location is a singleton, the application of the convex hull operator will result in the original set as we consider only hybrid automata with $Init(\ell)$ *being a convex set (see Def. 1).*
- $\forall \ell^\# \in Loc^\# :$

$$Flow^\#(\ell^\#)(x, \dot{x}) = \begin{cases} \mathcal{CH}(\bigcup_{\ell \in \alpha^{-1}(\ell^\#)} F_\ell), & |\alpha^{-1}(\ell^\#)| > 1 \\ Flow(\alpha^{-1}(\ell^\#))(x, \dot{x}), & |\alpha^{-1}(\ell^\#)| = 1 \end{cases}$$

  where $F_\ell = \exists x : (Flow(\ell)(x, \dot{x}) \wedge I(\ell)(x))$.
- $Trans^\# = \{(\ell^\#, g, \xi, \hat{\ell}^\#) | \exists \ell \in \alpha^{-1}(\ell^\#), \hat{\ell} \in \alpha^{-1}(\hat{\ell}^\#)$ *s.t.* $(\ell, g, \xi, \hat{\ell}) \in Trans\}$*, i.e., an abstract transition between* $\ell^\#$ *and* $\hat{\ell}^\#$ *is added when a transition in the concrete state space connecting the corresponding locations exists.*
- $\forall \ell^\# \in Loc^\# : I^\#(\ell^\#) = \mathcal{CH}(\bigcup_{\ell \in \alpha^{-1}(\ell^\#)} I(\ell))$*, i.e., similarly to the initial regions, the invariants are merged and over-approximated by a convex hull.*

In other words, we merge the dynamics of multiple locations in two steps. We first over-approximate the original dynamics in every concrete location by quantifying away unprimed variables, i.e., we obtain a constraint reasoning only about derivatives (see Fig. 2). Secondly, we define abstract dynamics by constructing a convex hull of the constraints computed in the first step. If an abstract location is a singleton, i.e., $|\alpha^{-1}(\ell^\#)| = 1$, we just keep its original dynamics.

We observe that by construction the set of reachable states of the abstract automaton $\mathcal{H}^\#$ leads to an over-approximation compared to the states reachable by the concrete automaton $\mathcal{H}$. Therefore, the following proposition holds:

Proposition 1. *Let* $\mathcal{H}^\#$ *be a location-merging abstraction of the concrete hybrid automaton* $\mathcal{H}$*. Then the non-reachability of the bad location* ℓ_{bad} *in* $\mathcal{H}^\#$ *implies its non-reachability also in the concrete automaton* $\mathcal{H}$*.*

3.2 Compositional Analysis

Our compositional analysis is illustrated in Algorithm 1. In order to simplify the presentation we consider a case of a system consisting of two components

Dynamics:
$\dot{x} = 2x + 3y$
$\dot{y} = 4x - 5y$

Invariant:
$0 \leq x \leq 1$
$\wedge\ 0 \leq y \leq 1$

F_1:
$-5\dot{x} - 3\dot{y} \leq 0$
$\wedge\ -22 + 5\dot{x} + 3\dot{y} \leq 0$
$\wedge\ -2\dot{x} + \dot{y} \leq 0$
$\wedge\ -11 + 2\dot{x} - \dot{y} \leq 0$

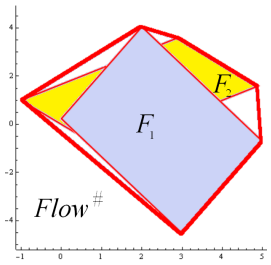
(a) Location ℓ_1.

Dynamics:
$\dot{x} = -x + 3y + 5$
$\dot{y} = \quad x + 2y$

Invariant:
$1 \leq x \leq 3$
$\wedge\ -1 \leq y \leq 0.3$

F_2:
$-5 + 2\dot{x} - 3\dot{y} \leq 0$
$\wedge\ -5 - 2\dot{x} + 3\dot{y} \leq 0$
$\wedge\ -\dot{x} - \dot{y} \leq 0$
$\wedge\ -6.5 + \dot{x} + \dot{y} \leq 0$

(b) Location ℓ_2.



(c) Convex Hull.

Fig. 2: Elimination of unprimed variables before merging of the locations.

$\mathcal{H}_1$ and $\mathcal{H}_2$, where $\mathcal{H}_1$ is a plant and $\mathcal{H}_2$ is a controller. However, the scheme is applicable to systems with more than two components [5].

In the following we provide a conceptual description of the algorithm. The algorithm checks whether the bad state S_{bad} can be reached by the system $\mathcal{H}_1||\mathcal{H}_2$.The algorithm starts by computing an abstraction of $\mathcal{H}_2$ in the function ConstructAbstraction (line 1). For more details on the abstraction construction see Sec. 3.1. The algorithm iteratively refines the original abstraction (lines 2–14). Note that in the worst case we will end up with the original system. However, in many cases we will need to refine only a part of the system (see Sec. 5 for the detailed discussion). In every refinement iteration the algorithm proceeds as follows. First, the state space of the abstract system $\mathcal{H}_1||\mathcal{H}_2^\#$ is analyzed in the function Analysis (line 3). This function returns an abstract bad path or "empty" if no such path has been found. If no abstract bad path has been found, we can conclude that also the original system is safe as we consider only over-approximations (line 5). Otherwise, the algorithm proceeds in the function SpuriousnessAnalysis (line 7) with the spuriousness analysis of the found abstract bad path $\pi^\#$. The function SpuriousnessAnalysis returns the information on how to refine $\mathcal{H}_2^\#$ or "empty" if the abstract path $\pi^\#$ can be concretized. In the latter case, we exit with status "System is unsafe" (line 9). Otherwise, $\mathcal{H}_2^\#$ is refined in the function Refinement based on the structure of the abstract bad path gained during the spuriousness analysis.

3.3 Spuriousness Check

In this section, we consider the function SpuriousnessAnalysis (see Algorithm 2) in more detail. Given an abstract bad path $\pi^\# = s_0^\#, \ldots, s_{m-1}^\#$, the function enumerates concrete paths corresponding to $\pi^\#$ and looks for the ones which end up in a bad state. The enumeration of concrete paths of the composed automaton $\mathcal{H}_1||\mathcal{H}_2$ along the abstract path $\pi^\#$ is organized in a breadth-first

Algorithm 1 Compositional analysis of $\mathcal{H}_1||\mathcal{H}_2$

Input: Hybrid automata $\mathcal{H}_1$ and $\mathcal{H}_2$
Output: Is the composed system $\mathcal{H}_1||\mathcal{H}_2$ safe?

```
 1: H_2^# := ConstructAbstraction (H_2)
 2: while true do
 3:    π^# := Analysis (H_1||H_2^#)
 4:    if π^# is empty then
 5:       return "System is safe"
 6:    else
 7:       SP := SpuriousnessAnalysis (H_1, H_2, H_2^#, π^#)
 8:       if SP is empty then
 9:          return "System is unsafe"
10:       else
11:          H_2^# := Refinement (H_2^#, SP)
12:       end if
13:    end if
14: end while
```

fashion. In particular, we make use of two lists: $\mathcal{L}_{waiting}$ and $\mathcal{L}_{passed}$. $\mathcal{L}_{waiting}$ stores symbolic states which still have to be considered and $\mathcal{L}_{passed}$ stores symbolic states which have already been considered and thus do not have to be visited again. The data structure $\mathcal{SP}$ stores information relevant for the refinement step. In particular, tuples $(\pi^{\#}, \pi)$, where π is a path in the concrete state space which does not belong to $\alpha^{-1}(\pi^{\#})$, are kept in $\mathcal{SP}$. In other words, in the last symbolic state $s_{|\pi|-1}$ of π we cannot take any discrete transition which would lead to some concrete state represented by an abstract state $s^{\#}_{|\pi|}$. Therefore, a tuple $(\pi^{\#}, \pi)$ essentially provides a possible *reason* for the spuriousness of π with respect to $\pi^{\#}$. We will use this information to refine the abstract component $\mathcal{H}_2^{\#}$ (see Sec. 3.4).

The algorithm starts by pushing the concrete initial states which correspond to the first abstract symbolic state $s_0^{\#}$ in $\mathcal{L}_{waiting}$ (line 2). It is important to mention that α^{-1} concretizes only the part of the symbolic state relevant to $\mathcal{H}_2^{\#}$. This property also holds for the algorithm described in Sec. 3.4. Note that we furthermore store the position of the abstract state which corresponds to the considered concrete symbolic state in the waiting list (we start with $s_0^{\#}$ and thus the position is 0). We will consequently use this information to compute the discrete symbolic successors of a given symbolic state which correspond to the analyzed bad path $\pi^{\#}$. In lines 3–20 the concrete state space is iteratively explored in a breadth-first manner. Every iteration consists of the following steps. First, the next tuple (s_{curr}, i) is picked from the waiting list $\mathcal{L}_{waiting}$ (line 4), where s_{curr} is a symbolic state and i shows its position with respect to the abstract path. Afterwards, the continuous successor, i.e., a symbolic state reflecting the states reachable according to the continuous dynamics, is computed and added to the passed list $\mathcal{L}_{passed}$ (lines 5–6). If the end of the abstract path is reached then the intersection with the bad state is checked (lines 8–10). If the

Algorithm 2 Spuriousness analysis

Input: Concrete automaton $\mathcal{H}_1$, concrete automaton $\mathcal{H}_2$ and its abstract version $\mathcal{H}_2^{\#}$ and abstract bad path $\pi^{\#} = s_0^{\#}, \ldots, s_{m-1}^{\#}$ in the state space of $\mathcal{H}_1 || \mathcal{H}_2^{\#}$.

Output: Information about the possible splitting points store or empty set if the abstract bad path $\pi^{\#}$ is concretizable

1: $\mathcal{SP} := \emptyset$
2: PUSH $(\mathcal{L}_{waiting}, (\alpha^{-1}(s_0^{\#}) \cap S_{init}(\mathcal{H}_1 || \mathcal{H}_2), 0))$
3: **while** $\mathcal{L}_{waiting} \neq \emptyset$ **do**
4: $(s_{curr}, i) :=$ GETNEXT $(\mathcal{L}_{waiting})$
5: $s'_{curr} :=$ CONTSUCCESSORS (s_{curr})
6: PUSH $(\mathcal{L}_{passed}, s'_{curr})$
7: **if** $i = m - 1$ **then**
8: **if** s'_{curr} is a symbolic error state **then**
9: **return** empty set, i.e., concrete bad state found
10: **else**
11: Store the abstract bad path $\pi^{\#}$ and the corresponding concrete path π ending in s'_{curr} into $\mathcal{SP}$
12: **end if**
13: **end if**
14: $S' :=$ DISCRETESUCCESSORS $(s'_{curr}) \cap \alpha^{-1}(s_{i+1}^{\#})$
15: **if** S' is empty **then**
16: Store the abstract bad path $\pi^{\#}$ and the corresponding concrete path π ending in s'_{curr} into $\mathcal{SP}$
17: **else**
18: PUSH $(\mathcal{L}_{waiting}, S' \setminus \mathcal{L}_{passed}, i + 1)$
19: **end if**
20: **end while**
21: **return** $\mathcal{SP}$

end of the abstract path is reached, but no intersection with the bad state is detected, we store both the abstract and concrete paths in $\mathcal{SP}$ in order to use this information in the refinement step. If the algorithm is still in the middle of the abstract bad path, it moves on to the computation of the concrete symbolic states which correspond to the abstract bad path (line 14). We achieve this by computing discrete successors and intersecting them with the concrete states represented by the next symbolic state on the abstract path. Note that the position i allows the algorithm to easily find the next abstract symbolic state on the path with respect to the currently considered concrete state.

If the set of discrete successors is empty, we say that a possible *splitting point* has been found. In other words, we could refine the abstract location $\ell_i^{\#}$ of $s_i^{\#} = (\ell_i^{\#}, R_i^{\#})$ by splitting it (see Sec. 3.4). We store the abstract bad path and the concrete path we have considered up to now into $\mathcal{SP}$ (line 16). Otherwise, we add the discrete state into the waiting list $\mathcal{L}_{waiting}$ (line 18). After having analyzed all concrete paths corresponding to $\pi^{\#}$, the function SPURIOUSNESS-ANALYSIS returns $\mathcal{SP}$. It is only possible to report that the considered abstract bad path is not concretizable after having considered all possible concrete paths

corresponding to it. Thus, the algorithm does not stop after discovering a particular splitting point, but just stores it for the later reuse during the refinement.

While mapping an abstract bad path to a concrete one, Algorithm 2 refers to the functions ContSuccessors and DiscreteSuccessors which are applied to concrete symbolic states. Thus, if the function SpuriousnessAnalysis declares some abstract bad path $\pi^{\#}$ to be genuine by finding its concrete counterpart π, then we can automatically conclude that the standard SpaceEx reachability algorithm would also have reported π to be a bad path. Therefore, our framework provides the same level of precision as the standard SpaceEx reachability algorithm. Finally, we note that the full concretization of a symbolic path is known to be a highly nontrivial problem. Once a concrete symbolic bad path is found with our approach, further concretization to hybrid automaton trajectories can be achieved using techniques from optimal control such as the one proposed in the work by Zutshi et al. [17].

3.4 Refinement Algorithm

The refinement algorithm Refinement uses $\mathcal{SP}$ in order to appropriately refine the abstraction $\mathcal{H}_2^{\#}$ in a compositional way. The data structure $\mathcal{SP}$ contains information about multiple possible splitting points. For the refinement we choose a tuple $(\pi^{\#}, \pi_{max}) \in \mathcal{SP}$ which *maximizes* the length of the concrete path π over all the elements of $\mathcal{SP}$. Intuitively, by choosing a tuple with this property, we ensure that π_{max} cannot be extended for all concrete paths which correspond to $\pi^{\#}$. Let the abstract bad path $\pi^{\#} = s_0^{\#}, \ldots, s_i^{\#}, \ldots, s_n^{\#}$ and the concrete path $\pi_{max} = s_0, \ldots, s_i, \ldots, s_m$ $(m \leq n)$, where $s_i = (\ell_i, R_i)$ and $s_i^{\#} = (\ell_i^{\#}, R_i^{\#})$. Furthermore, $\ell_i = (\ell_i^{(1)}, \ell_i^{(2)})$, where $\ell_i^{(1)}$ and $\ell_i^{(2)}$ are locations of $\mathcal{H}_1$ and $\mathcal{H}_2$, respectively. The location of the abstracted composed automaton $\mathcal{H}_1 || \mathcal{H}_2^{\#}$ is given by the tuple $\ell_i^{\#} = (\ell_i^{(1)}, \ell_i^{\#(2)})$. Depending on the location partitioning of $\mathcal{H}_2^{\#}$ the refinement algorithm distinguishes three cases:

1. $|\alpha^{-1}(\ell_m^{\#(2)})| > 1$, i.e., the abstract location corresponding to the last concrete location can be split:
   The refinement algorithm proceeds by splitting the abstract location $\ell_m^{\#(2)}$ of $\mathcal{H}_2^{\#}$ into two locations: $\alpha^{-1}(\ell_m^{\#(2)}) \setminus \ell_m^{(2)}$ and $\ell_m^{(2)}$, where $\ell_m^{(2)}$ is a location of $\mathcal{H}_2$ corresponding to the concrete symbolic state $s_m = ((\ell_m^{(1)}, \ell_m^{(2)}), R_m)$.
2. $|\alpha^{-1}(\ell_m^{\#(2)})| = 1$ and $|\alpha^{-1}(\ell_{m+1}^{\#(2)})| > 1$, i.e., the abstract location of $\mathcal{H}_2^{\#}$ corresponding to the last concrete location cannot be split, whereas the successor abstract location still comprises multiple locations:
   The refinement algorithm splits $\ell_{m+1}^{\#(2)}$ into $\alpha^{-1}(\ell_{m+1}^{\#(2)}) \setminus \ell'$ and ℓ', where $\ell' = \{\ell | \ell \in \ell_{m+1}^{\#(2)}$ and ℓ is a target location of discrete transition from $\ell_m^{\#(2)}\ \}$.
   In other words, we look for locations in $\ell_{m+1}^{\#(2)}$ which have incoming transitions from $\ell_m^{\#(2)}$ and split them apart. Note that in this case we do not look at the transition guard and any other continuous artifacts.

3. $|\alpha^{-1}(\ell_m^{\#(2)})| = 1$ and $|\alpha^{-1}(\ell_{m+1}^{\#(2)})| = 1$, i.e., neither the abstract location corresponding to the last concrete location nor its successor can be split: The algorithm iterates over the abstract path and looks for a abstract state in $\mathcal{H}_2^{\#}$ with a location which still can be split, i.e., we look for i s.t. $i < m \wedge |\alpha^{-1}(\ell_i^{\#(2)})| > 1$. The location $\ell_i^{\#(2)}$ is split into locations $\alpha^{-1}(\ell_i^{\#(2)}) \setminus \ell_i^{(2)}$ and $\ell_i^{(2)}$, where $\ell_i^{(2)}$ is a location of $\mathcal{H}_2$ corresponding to $s_i = ((\ell_i^{(1)}, \ell_i^{(2)}), R_i)$.

Therefore, during the refinement process, we only refer to the locations of the abstracted component $\mathcal{H}_2^{\#}$, i.e., we consider the projection of the found path to $\mathcal{H}_2^{\#}$. The refinement algorithm as described above also has a progress property:

Proposition 2 (Progress property). *The size of the location partitioning increases by one location after every application of the refinement algorithm over cases 1–3.*

Proof. By construction, the number of locations in $\mathcal{H}_2^{\#}$ increases by one in cases 1 and 2 after every refinement iteration. In case 3 the refinement can be only done under the assumption that there exists an index i s.t. $i < m \wedge |\alpha^{-1}(\ell_i^{\#(2)})| > 1$ holds. This statement is true as the opposite would mean that the whole abstract bad path $\pi^{\#}$ only consists of *concrete* states. This in turn would lead to the fact that $\pi^{\#}$ is already a concrete path to the bad state. The function Refinement is, however, called only for abstract bad paths which were found to be spurious. □

This proposition lets us conclude that Algorithm 1 terminates after a finite number of iterations after having considered the original system in the worst case. By combining this result with Proposition 1 and rule Asym, we can derive the following soundness and relative completeness results:

Theorem 1 (Soundness). *If our compositional framework is able to prove that $\mathcal{H}_1||A$ cannot reach the (abstract) error states, then the composition $\mathcal{H}_1||\mathcal{H}_2$ is safe, that is, it cannot reach the (concrete) error states.*

Theorem 2 (Relative Completeness). *If our compositional framework is able to find a symbolic error path in $\mathcal{H}_1||A$ which is not spurious, then there exists a concrete symbolic error path in the composition $\mathcal{H}_1||\mathcal{H}_2$, too.*

The existence of a symbolic error path does not necessarily imply the existence of an error trajectory (due to the undecidability of the reachability problem for affine hybrid automata). This is why we call the above result (for symbolic paths) relative completeness.

4 Related Work

The framework developed by Pasareanu et al. [13] enables automated compositional verification using rule ASym. In that work, both assumptions and properties are expressed as finite state automata. The framework uses the L* [4] automata-learning algorithm to iteratively compute assumptions in the form of deterministic finite-state automata. Other learning-based approaches for automating assumption generation for rule ASym have been suggested as well [3]. All these approaches were done in the context of transition systems, not for hybrid systems as we do here.

Several ways to compute abstractions of hybrid automata have been proposed. Alur et al. [2] propose to use a variant of predicate abstraction to construct a hybrid automaton abstraction. In a slightly different setting, Tiwari [16] suggests to use Lie derivatives to generate useful predicates. Both mentioned approaches essentially reduce the analysis of a hybrid automaton to the level of a discrete transition system. Jha et al. [12] partially eliminate continuous variables in the system under consideration. Prabhakar et al. [15] propose the use of CEGAR for initialized rectangular automata (IRA), where the abstractions reduce the complexity of both the continuous and the discrete dynamics. In this paper, we use a similar idea, but apply it to the more general class of affine hybrid automata, and even more importantly, we extend it to a compositional verification framework. Finally, Doyen et al. [8] take an affine automaton, and, through hybridization, obtain its abstraction in the form of a rectangular automaton with larger discrete space. We do the opposite: we take an affine automaton, and construct a much smaller linear hybrid automaton.

5 Evaluation

5.1 Benchmarks

For the evaluation of our approach we have extended the switched buffer network benchmark [10]. The system under consideration consists of multiple tanks connected by channels. The channels are used to transport the liquid stored in the tanks. There are two special tanks: the liquid enters the network through the *initial* tank and is transported towards the *sink* tank. We consider properties reasoning about the fill level of the sink tank.

The rate of change of the fill level f_T of a tank T, depends on the rates of inflow $v_{in\,i}$ and the rates of outflow $v_{out\,j}$ of the liquid, where $v_{in\,i}$ is the velocity at which the liquid flows into the tank of the i-th input channel, and $v_{out\,j}$ is the velocity at which the liquid flows out of the tank for the j-th output channel. Therefore, the evolution of the fill level of the tank T is described by the differential equation $\dot{f_T} = \sum_i v_{in\,i} - \sum_j v_{out\,j}$, where i and j range over incoming and outgoing channels of T, respectively. Note that due to fine-granular modelling of tanks and channels this benchmark class exhibits a large number of continuous variables. In particular, in our benchmark suite the number of continuous variables is in the range from 17 to 21 for the buffer networks with up to 4 tanks,

whereas it is well-known that the analysis complexity of hybrid automata rapidly grows with the number of variables in the system under consideration.

We extend the switched buffer network [10] by the model of a complex *stratified* controller. The controller is organized in a number of phases of some given length, where multiple options (governing the modes of particular channels) are available in every phase. After having finished the last phase the controller returns to the first one. The controller can open/close channels and adjust the throughput values at every step. We consider the following modes of controller operations:

1. Throughput provided by an interval ("No Dynamics"): when the channel is activated, its throughput v is constrained by the inequality $v_{min} \leq v \leq v_{max}$.
2. Throughput evolving at a constant rate ("Constant Dynamics"): the throughput is defined by the differential equation of the form $\dot{v} = c$ for some constant c.
3. Throughput evolving according to affine dynamics $\dot{v} = c(v_{target} - v)$ ("Affine Dynamics"): the controller provides a target throughput velocity v_{target} and some constant factor c. According to this dynamics the channel opens gradually with the opening speed decaying towards the target velocity.

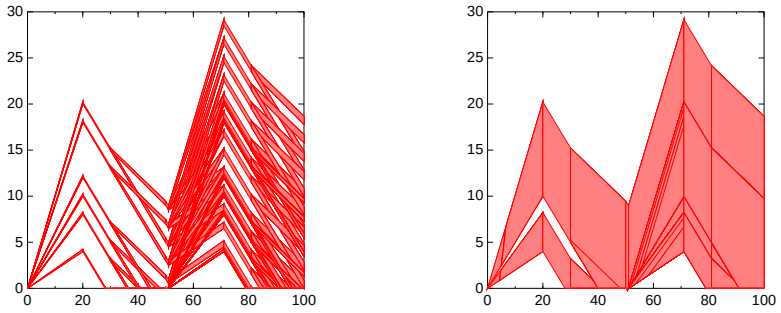
5.2 Experiments

We have implemented our approach in SpaceEx [9]. The implementation and the benchmarks are available at `http://swt.informatik.uni-freiburg.de/tool/spaceex/agar`. The experiments were conducted on a machine with an Intel Core i7 3.4 GHz processor and with 16 GB of memory. In the following, we report the results for our compositional analysis implemented in SpaceEx. We compare the analysis results of the original concrete system and the compositional analysis. For both settings, we compare the number of iterations of SpaceEx and the whole analysis run-time in seconds (see Table 1). The best results are highlighted in bold. We analyze 12 structurally different benchmark instances. For each of them we vary forbidden states and in this way end up with 36 different benchmark settings. We also vary controller dynamics. In particular, we provide 12 instances for each of the modes "No Dynamics", "Constant Dynamics" and "Affine Dynamics". The number of continuous variables varies in the considered benchmark instances from 17 to 21 variables. The initial abstraction is generated by merging some of the strata in the controller.

We observe that our compositional reasoning algorithm generally boosts the run time compared to the analysis of the original system. For example, in instance 4 (system is safe) the analysis of the concrete system takes around 609 seconds compared to around 158 seconds with the compositional analysis. The speed-up is justified by the smaller branching factor due to location merging. In Fig. 3a and Fig. 3b the fill level of sink tank vs. time for the original system and the initial abstraction are plotted. Fig. 3b particularly shows that multiple "thin" flow-pipes are merged into a couple of "thick" ones, i.e., the system stops differentiating between some options in the controller.

#	Res.	Tanks	Vars.	Phases	Refs.	It. (u)	It. (m)	Time (u)	Time (m)
No Dynamics									
1	safe	3	17	2 (5,1)	0	4640	**253**	779.754	**14.692**
2	unsafe	3	17	2 (5,1)	0	2555	**191**	299.437	**35.370**
3	safe	3	17	2 (5,1)	1	4640	**1744**	796.218	**191.841**
4	safe	3	17	4 (6,1,2,1)	0	3242	**1115**	608.796	**157.924**
5	unsafe	3	17	4 (6,1,2,1)	0	2410	**756**	196.461	**66.740**
6	safe	3	17	4 (6,1,2,1)	2	3242	**1648**	639.838	**254.653**
7	safe	4	21	2 (5,1)	0	2345	**690**	2162.273	**621.137**
8	unsafe	4	21	2 (5,1)	0	1348	**483**	1139.365	**479.811**
9	safe	4	21	2 (5,1)	1	2345	**1001**	2164.069	**937.064**
10	safe	4	21	4 (4,1,2,1)	0	1361	**394**	1327.062	**406.592**
11	unsafe	4	21	4 (4,1,2,1)	0	1070	**316**	502.992	**303.988**
12	safe	4	21	4 (4,1,2,1)	1	1361	**684**	1174.735	**700.072**
Constant Dynamics									
13	safe	3	17	4 (2,1,5,1)	0	1386	**424**	90.457	**21.484**
14	unsafe	3	17	4 (2,1,5,1)	0	461	**232**	18.773	**10.807**
15	safe	3	17	4 (2,1,5,1)	2	1386	**1261**	81.076	**77.938**
16	safe	3	17	6 (2,1,6,1,2,1)	0	1989	**1027**	146.726	**63.878**
17	unsafe	3	17	6 (2,1,6,1,2,1)	0	809	**352**	32.961	**14.279**
18	safe	3	17	6 (2,1,6,1,2,1)	2	**1989**	2041	**142.385**	250.451
19	safe	4	21	4 (2,1,4,1)	0	1293	**787**	1350.973	**1318.623**
20	unsafe	4	21	4 (2,1,4,1)	0	1080	**682**	1429.120	**1298.147**
21	safe	4	21	4 (2,1,4,1)	1	1293	**814**	1579.792	**1197.098**
22	safe	4	21	6 (2,1,4,1,2,1)	0	903	**563**	1255.978	**1140.114**
23	unsafe	4	21	6 (2,1,4,1,2,1)	0	798	**510**	1230.193	**1141.791**
24	safe	4	21	6 (2,1,4,1,2,1)	1	903	**581**	1365.629	**1318.049**
Affine Dynamics									
25	safe	3	17	4 (2,1,5,1)	0	7747	**1168**	1544.363	**86.046**
26	unsafe	3	17	4 (2,1,5,1)	0	5103	**1042**	939.430	**100.871**
27	safe	3	17	4 (2,1,5,1)	1	7747	**6214**	1669.268	**1240.215**
28	safe	3	17	6 (2,1,6,1,2,1)	0	6129	**2760**	717.462	**231.727**
29	unsafe	3	17	6 (2,1,6,1,2,1)	0	5382	**2397**	639.342	**203.143**
30	safe	3	17	6 (2,1,6,1,2,1)	7	**6129**	15068	**706.960**	2158.671
31	safe	4	21	4 (2,1,4,1)	0	1718	**1451**	3603.238	**3125.016**
32	unsafe	4	21	4 (2,1,4,1)	0	1692	**1392**	3776.840	**3247.464**
33	safe	4	21	4 (2,1,4,1)	1	**1718**	2559	4372.284	**3805.045**
34	safe	4	21	6 (2,1,4,1,2,1)	0	983	**642**	1382.567	**1078.893**
35	unsafe	4	21	6 (2,1,4,1,2,1)	0	922	**611**	1206.011	**1213.798**
36	safe	4	21	6 (2,1,4,1,2,1)	1	983	**755**	1442.506	**1321.658**

Table 1: Experimental results for the switched buffer benchmark. Abbreviations: #: benchmark instance number, Res.: result of the system analysis, i.e., whether the bad state can be reached, Tanks: number of tanks in the instance, Vars.: number of continuous variables in the system, Phases: number of phases in the controller and number of options in every phase, Refs.: number of refinement steps, It. (u): number of SpaceEx iterations when analyzing the concrete (unmerged) system, It. (m): number of SpaceEx iterations in scope of the compositional analysis, Time (u): total time in seconds of the analysis of the concrete system, Time (m): total time in seconds of the compositional analysis.



(a) Original system. (b) Initial abstraction.

Fig. 3: Fill level of the sink tank for instance 4 vs. time

Furthermore, we remark that our compositional algorithm shows promising results also in the falsification setting, i.e., when the bad state is reachable. In instance 5, our approach reduces the run-time from around 196 seconds for the concrete system to only 67 seconds in scope of the compositional framework.

The necessity to refine the abstraction, in case a spurious abstract bad path has been discovered, can generally be handled efficiently by our framework, e.g., in instance 6 our approach takes around 254 seconds (including two refinement steps) compared to 640 seconds for the concrete system. However, due to an unfortunate choice of the abstract bad path, we might need to refine an excessive number of times (instance 30) which in turn decreases the overall performance.

6 Conclusion

In this paper, we have adapted the idea of compositional analysis to the domain of hybrid systems. We have presented an abstraction based on location merging. The abstract location invariant is computed by taking a convex hull of the concrete locations to be merged. The abstract continuous dynamics are computed by eliminating the state variables and computing a convex hull.

Acknowledgments

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, `http://www.avacs.org/`). We thank Jannik Rebmann and Simon Ganz for their help with the benchmark suite preparation.

References

1. R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicolin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.
2. R. Alur, T. Dang, and F. Ivančić. Reachability analysis of hybrid systems via predicate abstraction. In *Hybrid Systems: Computation and Control (HSCC)*, pages 35–48. 2002.
3. R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *Computer Aided Verification*, pages 548–562, 2005.
4. D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.
5. M. G. Bobaru, C. S. Pasareanu, and D. Giannakopoulou. Automated assume-guarantee reasoning by abstraction refinement. In *Computer Aided Verification (CAV)*, pages 135–148, 2008.
6. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification (CAV)*, pages 154–169, 2000.
7. A. Donzé and G. Frehse. Modular, hierarchical models of control systems in SpaceEx. In *European Control Conference (ECC)*, 2013.
8. L. Doyen, T. A. Henzinger, and J.-F. Raskin. Automatic rectangular refinement of affine hybrid systems. In *Formal Modelling and Analysis of Timed Systems (FORMATS)*, pages 144–161. 2005.
9. G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SpaceEx: Scalable verification of hybrid systems. In *Computer Aided Verification (CAV)*, pages 379–395, 2011.
10. G. Frehse and O. Maler. Reachability analysis of a switched buffer network. In *Hybrid Systems: Computation and Control (HSCC)*, pages 698–701, 2007.
11. T. A. Henzinger. The theory of hybrid automata. In *LICS*, pages 278–292, 1996.
12. S. K. Jha, B. H. Krogh, J. E. Weimer, and E. M. Clarke. Reachability for linear hybrid automata using iterative relaxation abstraction. In *Hybrid Systems: Computation and Control (HSCC)*, pages 287–300. 2007.
13. C. S. Pasareanu, D. Giannakopoulou, M. G. Bobaru, J. M. Cobleigh, and H. Barringer. Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design (FMSD)*, 32(3):175–205, 2008.
14. A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems, NATO ASI Series*, 1985.
15. P. Prabhakar, P. S. Duggirala, S. Mitra, and M. Viswanathan. Hybrid automata-based CEGAR for rectangular hybrid systems. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 48–67, 2013.
16. A. Tiwari. Abstractions for hybrid systems. *Formal Methods in System Design (FMSD)*, 32(1):57–83, 2008.
17. A. Zutshi, S. Sankaranarayanan, J. Deshmukh, and J. Kapinski. A trajectory splicing approach to concretizing counterexamples for hybrid systems. In *Conference on Decision and Control (CDC)*, pages 3918–3925, 2013.